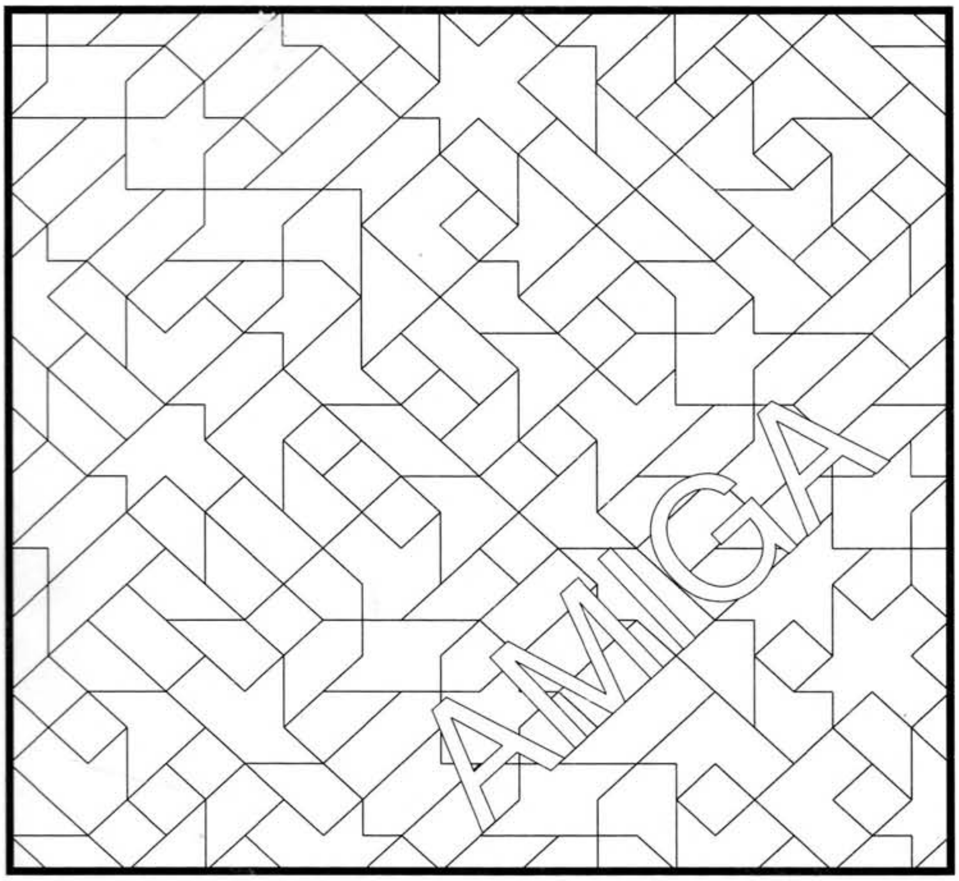
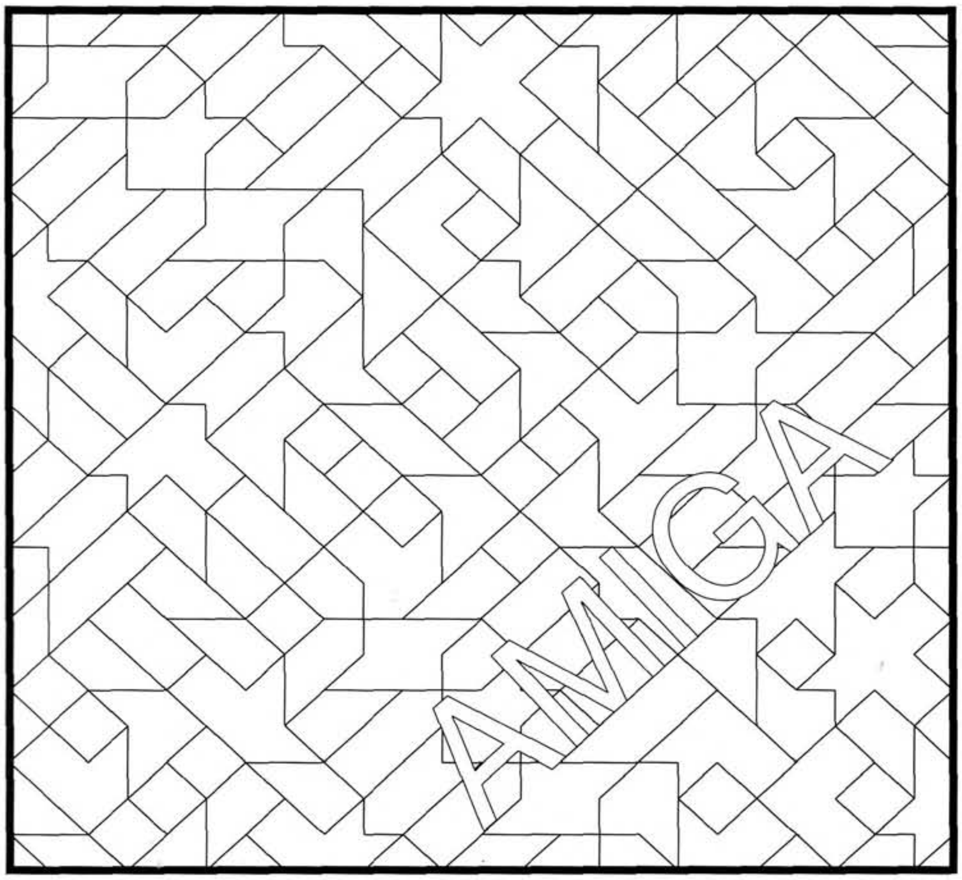
THE ARexx COOKBOOK



By Merrill Callaway

A Tutorial Guide to the ARexx Language on the Commodore Amiga® Personal Computer

THE ARexx COOKBOOK



By Merrill Callaway

Printed in the USA

ISBN 0-9632773-0-8
Library of Congress Catalog Card Number 92-081371

Distribution
The ARexx Cookbook and its optional *Companion Disk* (ISBN 0-9632773-1-6) are available from:

WHITESTONE
511-A Girard Blvd. SE
Albuquerque, NM 87106
(505) 268-0678

Dealer and Distributor inquiries invited. The ARexx Cookbook is $30 including postage and handling in the USA, and $35 outside the USA. The optional Companion Disk is $12 postage paid in the USA and $17 outside the USA. Send check, money order, or VISA/MC information. US dollars only.

About
This book was written and published using Wordperfect, TurboText, ARexx, Snap, Proper Grammar, Electric Thesaurus, and PageStream2. The draft was output in PostScript to an Epson EPL 7500 laser printer. It is a 100% Amiga product. Many thanks to my Dad who financed this project. I am grateful to Adrian Vyner-Brooks who encouraged me along the way. Most of all I am grateful to the Lord Jesus Christ, the giver of all perfect gifts and inspirations.

Table of Contents

Table of Contents

List of Illustrations

Companion Disk

An optional companion disk with all the programs in this book and several additional programs may be ordered from the publisher. See the copyright page under ***Distribution*** *for details.*

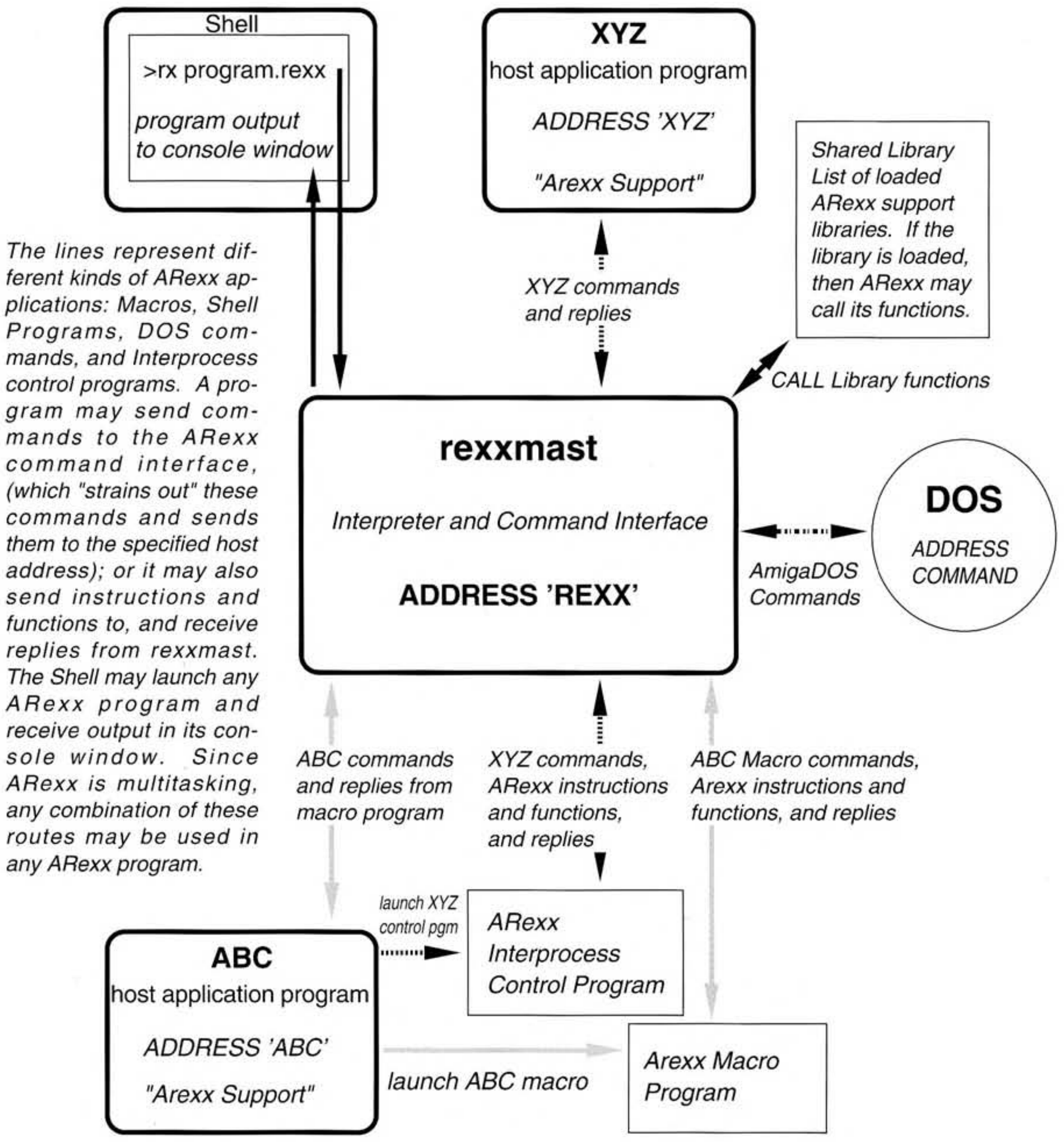
Preface

Not long after you first bought an Amiga computer, you began to hear about something called ARexx. In the software reviews and magazine ads you may have noticed the buzz words *ARexx Support*. The implication was that this ARexx support somehow added value and power to an application program, and was therefore worth having. Perhaps you asked yourself the question, "Just what is ARexx, anyway?" You soon found that ARexx is a *programming language*. Because the price was very reasonable, you may have bought ARexx sight unseen because everyone said it was wonderful. You may have bought a new Amiga and discovered that ARexx was already installed in the system software. Maybe you aren't a programmer and the documentation you received seems too hard to understand; or you may have some experience in another language, and you are really wanting to start programming in ARexx, but haven't found the time to learn it, so you've put it off time and again. Perhaps you simply need some creative ideas and some usable code examples to apply to your own system. *The ARexx Cookbook* is for you!

The purpose of *The ARexx Cookbook* is twofold: To bring less experienced Amiga users quickly up to speed in ARexx; and to provide some really useful ARexx programs that can be easily adapted or used as is for real world applications. This book provides some interesting, useful, and fun things to do with ARexx that make learning it fast and painless. ARexx truly *is* wonderful, and with a minimum of study, is easy to use. Everyone who uses an Amiga computer deserves to become proficient at ARexx, because it opens the real power of the Amiga multi-tasking system. I wrote this book because I went through some of the same frustrations you may be experiencing trying to start to use ARexx. Wading through a language reference manual is not the fun way to learn ARexx, but I persevered because I sensed it would be worth it. Much sooner than it looked like at first, I reached a level of competence that now allows me to customize my Amiga operating environment in ways I only dreamed of before. *The ARexx Cookbook* was written to help some of you get over your initial fear of programming and to accelerate your climb up the first hump of the learning curve, and to lead you to the level where you can begin to use the full power of your Amiga. For others, at a more intermediate stage, the book is an immediate source of useful code (and inspiration) that you can use to make your own programs in ARexx to customize your Amiga applications.

Merrill Callaway
December 10, 1991

The lines represent different kinds of ARexx applications: Macros, Shell Programs, DOS commands, and Interprocess control programs. A program may send commands to the ARexx command interface, (which "strains out" these commands and sends them to the specified host address); or it may also send instructions and functions to, and receive replies from rexxmast. The Shell may launch any ARexx program and receive output in its console window. Since ARexx is multitasking, any combination of these routes may be used in any ARexx program.



Shell
>rx program.rexx
program output to console window
XYZ
host application program
ADDRESS 'XYZ'
"Arexx Support"
Shared Library List of loaded ARexx support libraries. If the library is loaded, then ARexx may call its functions.
XYZ commands and replies
CALL Library functions
rexxmast
Interpreter and Command Interface
ADDRESS 'REXX'
DOS
ADDRESS COMMAND
AmigaDOS Commands
ABC commands and replies from macro program
XYZ commands, ARexx instructions and functions, and replies
ABC Macro commands, Arexx instructions and functions, and replies
launch XYZ control pgm
ARexx Interprocess Control Program
ABC
host application program
ADDRESS 'ABC'
"Arexx Support"
launch ABC macro
Arexx Macro Program

Chapter 1: Introduction to the ARexx Language

About This Book

The ARexx Cookbook is not designed to take the place of the excellent ARexx language description contained in the manual by William S. Hawes, the author of ARexx. It is expected that the reader will study this book with their copy of the ARexx *Hawes* or *Commodore* documentation close at hand, because there will be a need to reference one or the other of those texts frequently. The spiral binding and the index referencing not only this book, but *Hawes* and *Commodore*, too, are designed for easy use, and the handy size is meant for the desktop, not the bookshelf. Rather than presenting yet another list of instructions and function definitions that would essentially duplicate the ARexx user manual (which is already adequate in this respect), *The ARexx Cookbook* explains how to *program* your own applications in ARexx, by demonstrating good techniques such as pseudo-coding and modular thinking, along with specific examples of useful code which may be adapted to match your own needs. The ARexx instructions and functions are not the basis of organization of the book. Rather, the proper use of ARexx instructions occurs implicitly in the broader context of specific application examples, in much the same way that a tutorial in a foreign language presents *literature* instead of a steady diet of *grammatical rules*. This book refers to the ARexx manual when the context calls for it in order to save weight and bulk as much as to avoid duplication.

A good understanding of jargon, rigorous (formal) definitions, semantics, and syntax are all essential to programming properly in any computer language. The *Cookbook* attempts to de-mystify ARexx jargon and definitions that may seem unfamiliar to readers who have little programming experience, but does not provide an ordered or complete set of definitions as these are covered in *Hawes* or *Commodore* ARexx documentation. The book contains explanations which shed light upon the use and inter-relationships of various ARexx features as they apply to actual examples; but it leaves it up to the reader to study the ARexx manual for the complete formal definitions of things. This book uses terms throughout the text according to their formal ARexx definitions and *introduces* them to the reader even if not completely defining them. The

Hawes and *Commodore* references are noted, however.

Typography, Syntax, and Semantics

New terms appear in **bold** type. If there are several equivalent terms for something, the most intuitive term comes first, with the more rigorously defined term(s) following (in parentheses), to provide a sort of lexicon. This way, if one term doesn't seem familiar, a synonymous term will be alongside to assist the reader to learn a working vocabulary of ARexx. ARexx terminology is not always as intuitive as some older computer programming terminology. *The ARexx Cookbook* initially includes several equivalent words for things, but gradually, as the book progresses, the more rigorously defined or formally correct ARexx terms will have supplanted the more intuitive but looser terms. By then, the reader will be familiar with them. In the same way, semantics (the meaning of instructions) and syntax (the physical arrangement of symbols, words, and punctuation) will be explained in great detail at first and later less so. The emphasis shifts to programming later on, just as the lessons become more conversational and literary when you learn a foreign language.

Margin References

Cross references:
William S. Hawes and Commodore Manual
* (Part No. 363313-05)
are cross referenced by

Subject and

H = *Hawes*
C = *Commodore*

followed by page number; for example:

VERIFY() function

H 68
C 10-122

* Older *Commodore* pages will not match.

References

If you don't have the original ARexx manual, but have the Commodore version of ARexx installed, you should realize that the *Commodore* documentation largely duplicates William S. Hawes' manual, but leaves out some of the original material. If the *Commodore* documentation covers the subject under discussion, this book will reference it as well. If you want to go deeper into ARexx you really should get the original manual, too, just for the extra documentation. The *Hawes* manual, by the way, is much handier to use because it's spiral bound, and the pages are easier to turn. You will have a well thumbed manual before you become an expert! You will also receive many useful utilities and ARexx programs on disk which make the purchase even more worthwhile.

Wherever possible, regarding teaching method, the *Cookbook* uses the concrete example over the abstract definition; attempting, however, to clarify some of the definitions, commands and functions that seem a little opaque in the manual. Overall, I wrote the book to infuse you with the enthusiasm and downright affection I feel for this wonderful and indispensible tool for the Amiga. After answering a few general questions some less experienced readers may have about ARexx, *The ARexx Cookbook* gets down to the applications examples.

Is ARexx Hard to Learn?

You may think, "Yes, it is!", based upon a first reading of the manual, but let's think about a foreign language for a minute. No one learns a foreign language by studying a dictionary or grammar book only, in spite of their essential importance. One learns *to communicate* by mimicking a native speaker. Total immersion is the quickest way to learn: One is forced to try speaking or else not communicate at all. Total immersion has a tendency to make us overcome our fear and embarrassment at seeming stupid, and it isn't long before we are talking away! The situation with you, the Amiga, and ARexx is similar. You really have no idea what you are missing by being wary of learning ARexx. You do not have to be extremely fluent in any language before you can communicate, and neither must you memorize every command and function before you can do satisfying things in ARexx, because it is easy to learn. After only a short time in this book and doing some work on your own, you will be appreciating the clarity and readability of ARexx code as well as its ingenious and powerful elegance.

What Kind of Language is ARexx?

ARexx comes from a parent language called **Rexx**. Rexx was developed by Michael F. Cowlishaw at the IBM UK Laboratories, Ltd. in the mid 1980's. The design goals of Rexx were all aimed at one simple objective: To make programming easier than ever before. In every design decision, people were considered first, but language power was not sacrificed. People's perception of things like "Readability" and "Ease of Use" were actual design criteria that took on a high priority. Since people who deal with computers work with things such as words, numbers, names, and so on, Mr. Cowlishaw made Rexx especially adept at handling these symbolic objects, and therefore very accessible to the casual user. At the same time he made it easy to write small "quick and dirty" programs, he included in the language advanced facilities that make it possible (and feasible) to write large, well behaved (robust) programs as well.

Rexx was intended not only for personal use, but also as a command program interpreter. This is a relatively new technique where a general purpose language (Rexx) is also used to tailor the command environment of the operating system of the host computer. In other words you can program your own custom commands in your particular operating system. These custom commands are really Rexx programs which can run in a command shell environment. Since Rexx is so good at manipulating strings (those symbolic objects mentioned before), it lends itself to becoming a general use macro language to control application software running in the system.

Finally, Rexx was designed to make prototyping easy. The overall design prototype of any software system can be tested and modified readily, before coding into a more difficult compiled language. It is no wonder that millions of lines of Rexx code do service every day in prestigious mainframe installations around the world! ARexx is the implementation of Rexx on the Amiga. Until recently, when IBM PC Rexx came out, ARexx was the only implementation of this wonderfully easy and powerful language on a personal computer platform. We all have William S. Hawes to thank for his brilliant implementation of ARexx. ARexx inherits all the power and ease of use of its parent language. We already know that ARexx is a programming language. ARexx is what is known as a high level, interpreted language. By **high level**, we mean that the code is readable, intuitive, and accomplishes complex tasks with a minimum of coded instructions. For instance, a single ARexx instruction, if translated into a low level language such as Assembler Language might take up dozens of lines or even several pages of **source code** (the original instructions typed into the editor by the programmer), expressed as cryptic and difficult to read instructions, each of which accomplishes only a small part of the complex task handled by the one ARexx instruction.

An **interpreted language** is one which uses a single program, called oddly enough an **interpreter**, to execute your list of coded instructions one at a time and in the sequence they occur. Where does your list of instructions come from? Any computer program (a list of coded instructions) starts life within an editor as lines of text and numbers representing instructions which the programmer types in and saves in ASCII text format. No one writes program code in a word processor unless they remember to save their file as an ASCII text file. Word processors add their own formatting codes that render the program code useless. This is the reason that programmers use an *editor* rather than a *word processor* to write their programs: They save all their files in ASCII and they appreciate the features in a good programmer's editor that make entering program code easier.

The other type of programming language is a **compiled language**. Although its program code starts out in an editor, too, this type of language depends on at least two programs, one called a **compiler** which translates the code you typed into your editor (called **source code**) into a transition code (called **object code**). Yet another program (called a **linker**) takes the object code and transforms it into its final form: machine readable, stand alone binary code. After you finish compiling and linking your program, it can be run without further dependence on any other program.

What Happens When I Run an ARexx Program?

Although you do not need to know all about what happens when you run any program on the Amiga, it certainly helps to have a basic understanding. ARexx, as an interpreted language, depends on a program running in the background which is, among other things, the **ARexx interpreter**, or the program that takes care of launching all ARexx programs by interpreting their coded instructions and then running each program as a separate task (really a DOS process) in the multi-tasking environment of your Amiga. The name of this background program is **rexxmast**. It is also properly called the ARexx **resident process**. Without its presence, no ARexx programs will work.

Rexxmast
Resident Process

H 6, 83, 89
C 10-7

Make sure that in addition to running **rexxmast** in your startup-sequence, you also ASSIGN REXX: to the REXX or REXXC directory where you keep your ARexx programs. This assignment is best made at startup, too.

Rexxmast can be (and usually is) run from within the startup-sequence, so that any ARexx program you want it to launch will be interpreted and executed right away. **Rexxmast** takes up little memory and no system resources until needed. Remember we are in a multi-tasking environment on the Amiga! **Rexxmast** just **sleeps**, waiting until you need it. The way you request this resident process to run your ARexx program from the CLI or shell is to type the command **rx** followed by your program name (and any of its arguments) at the CLI prompt. It feels just like issuing an AmigaDOS command except that you precede your program name with the command **rx** to signal the resident process to launch your ARexx program. Under AmigaDOS, the command **rx**

returns a Usage:rx filename [arguments] message if you do not supply a filename. The [*arguments*] are optional, and represent arguments, if any, to your ARexx program. An ARexx program must always be **launched**, (or started, or run) as any other program must be run. You should keep your ARexx program files in the **Rexx** directory in your **sys:** device. In system 2.0 of the Amiga the directory is called **Rexxc**. The **rexxmast** program searches first for your program in this **Rexx** or **Rexxc** directory. It is a good idea to give your general use ARexx programs a filename extension **.rexx** and ARexx programs launched from an application program a different extension.

Rexx Directory

H 6
C 10-15

For example, if you have a program called **TurboText**® with ARexx support, it has several ARexx programs with names like **AddChars.ttx** to remind you that they are programs that you use in your TurboText editor program. All programs with an extension **.rexx** will run without your typing in the extension. The **rexxmast** interpreter will recognize and run your program without it. This does not apply to extensions different from **.rexx,** however, unless the application program has implemented this feature.

Naming Conventions

H 6
C 10-13

T = TurboText Manual see page 1-12

AddChars.ttx Macro

T 10-4

ARexx has some powerful features that set it apart from other languages. It has the ability to control outside programs! It can control programs (which have ARexx support) running as separate tasks within the Amiga multi-tasking environment. It can perform this remote control of other programs by means of the **command interface**, which is composed of two parts: the **rexxmast** resident process, and the implementation of the ARexx command interface in some outside application program running at the same time. Any Amiga *application program* capable of communication with an *ARexx program* (receiving commands, sending replies) is called a **host application**, and is also said to provide an **ARexx command interface**. Any host application, once started, makes its presence known to the Amiga operating system by opening one (or more) of what is called a **public message port** through which it can receive its own commands from ARexx programs, and send back replies.

Command Interface

H 3, 43, 89
C 10-15, 10-78

This is exactly what is meant when you see the term *ARexx support* in application software advertising. The **rexxmast** resident process is a communications center for all of ARexx. It does more than just interpret and launch ARexx programs. It allocates memory, keeps track of libraries and global system resources, and processes **commands** (which we will define later) through the command interface, sending them to an outside program to do things within that program using the internal set of commands unique to that program. The *resident process itself* has a public message port called **'REXX'** (Public message ports are always case sensitive), through which it sends and receives commands and replies, respectively. The resident process implicitly determines the destination for any commands it processes through the command interface by maintaining a **current host address** (for convenience, it also maintains a **previous host address**).

Public Message Port
Host Address

H 44
C 10-5, 10-75 f

A host address is the same name as the public message port maintained by a host application program, and until your ARexx program tells **rexxmast** to change to another host address, commands are sent to the *current* host address. Whenever an ARexx program is launched from a host application, the current host address automatically becomes the address of that host application (for that ARexx program). The default host address is **'REXX'**, so until a host application program overrides this default, the resident process will send itself any commands it encounters, and attempt to execute them. The resident process itself is therefore a host application, because it is able to receive ARexx commands. As we saw above, the **rx** command used in the CLI or Shell invokes the resident process to interpret your program's statements and launch them as a DOS process. This recursive quality of ARexx is a little confusing at first, but it is worth your while to contemplate its ramifications, because it presents a very powerful control mechanism, not only for your Amiga operating system, but also for any software having an ARexx command interface.

Commands

H 43 thru 46
C 10-74 thru 10-82

What Exactly Are Commands? ARexx has the unusual feature that it reserves a whole **syntactical class** of program statements called **commands** which are actual ARexx executable statements, but which are *not* required to have any *meaning* (in programming, *meaning* is sometimes called **semantics**) within the ARexx language itself! What do we mean by a syntactical class? This means commands are known by their *syntax* or the overall arrangement or aspect of the **command statement** within the ARexx code statements. Commands are usually surrounded by quotes (punctuation is a part of syntax), if the commands are to be sent to some program outside the host program (assuming your ARexx program changes the address appropriately). If these commands are to be sent *back* to the host program as its *own commands* in a macro, then it is safe to leave off the quotes (but including them will not hurt).

Clauses

H 14
C 10-31 ff

Also, the *position* in the ARexx program statement itself can determine if a statement is a command statement. Simply put, a command statement is any expression that **rexxmast** *cannot* identify and classify as one of the three other types of ARexx statements that *do* have meaning within ARexx itself (statements are also called **clauses**). The resident process uses the unique and powerful command interface to send these meaningless commands to outside programs, through their public message ports, where they finally take on meaning, because they are part of that program's internal command set; and where they will actually control that outside program exactly as if the commands *had been issued internally* in that program!

The primary purpose of the **rexxmast** resident process is to launch ARexx programs, but since it also serves as the communications center between ARexx and all other entities in the Amiga multi-tasking environment, **rexxmast** is much more than simply a host application. General use ARexx programs are usually run from your Amiga shell or the CLI. In case you didn't know, the shell and the CLI are programs themselves which launch AmigaDOS commands and start up other programs. The CLI is capable of receiving messages from the system in its console window, and when you send an ARexx program from a CLI, the resident process inherits the input and output streams from that CLI and communicates its replies to the output stream of the console window, so you can see your program's replies. When you send your ARexx program from the CLI by hitting the [Rtn] key, your command line enters the input stream from the console window, to be found and processed by the resident process. Thankfully, coding in ARexx is much more simple than understanding the Amiga system environment, and you need not worry excessively if you do not learn everything about it. What you do learn, however, will benefit you.

What Can I Do With ARexx?

General Programs

ARexx is a very good way to make general use programs or utilities.

Suppose we want to make a program to solve a word or number puzzle. Write a quick ARexx program to analyze and solve the puzzle. We'll see an example later in the famous *Coconut Problem*. As for creating and launching an ARexx program, simply type it into your favorite editor and save the resulting ASCII text file in the **Rexx** directory in your **Sys:** device. Let's say you named your program file **coconut.rexx**. To run your ARexx program, from the shell (or CLI) you type in at the prompt>`Rx coconut` hit **[Rtn]**, and your program runs! Output is written to your CLI window, unless you redirected it. Remember you don't need to type in the **.rexx** extension, but it won't hurt if you do. It's *very* simple to write and run an ARexx program! ARexx is an interpreted language as we discussed above, so you have very little overhead in preparing a program to run.

Graphical User Interface

Note: Shared library names are case sensitive!

To continue: Maybe you want to open a window on your work bench with some gadgets for your favorite programs. You want to be able to click on these and start up your programs without opening the disk partitions and drawers. By way of something called the **rexxarplib.library**, you have a great deal of control over windows and screens and menus. Arexx itself is *not* capable of accessing the **Amiga Intuition** graphical user interface, but it *is* capable of using **shared libraries**, so you have the full power of libraries written by expert programmers. **rexxarplib.library** is a shared library that provides a way to access the **Intuition** (graphical interface) features of your Amiga from an ARexx program. This very useful library was written by Willy Langeveld of the Stanford Linear Accelerator Center. An example of the use of functions from this library is included in **Appendix A**. It constructs a graphic interface to perform complex ARexx macros in Art Department Professional (ADPro) by ASDG, Inc. It is possible to bash the code of the example program to make a customized graphic user interface for your own environment. The custom window in the example demonstrates an ARexx program which starts up other programs, creates and communicate with ARexx message ports, opens shared libraries and much more. The **rexxarplib.library** was developed with D.O.E. Federal Government funds, and since you already paid for it with your taxes, you may have a free copy of this professional tool for the asking. You may obtain **rexxarplib.library** free on most Computer Bulletin Boards that support the Amiga (e.g. BIX).

Word Processing Support

Next, let's say you have a word processor (without ARexx support) that does everything you like, except it doesn't have a way to make an index. Even so, you can write an ARexx program to make that index from an ASCII text file of your document. In this book, you will get some hands-on experience making an ARexx program that takes a text file and makes an alphabetic list of all the unique words in the file after leaving out trivial words (*a*, *and*, *the*, etc.). One could make this bit of code the core program for a larger one that also finds the page number(s) for each word from the original document and then makes and saves the index list which you can bring into your word processor for final formatting. Notice you do not necessarily need to have ARexx support in your application software in order to use ARexx to increase its power and flexibility. As in the example, any application which can save and retrieve files is a possible candidate for ARexx enhancement even if it must be done indirectly. There are many, many other things you can do with ARexx programs!

Controlling Other Programs Directly

This brings us to the second major way to use ARexx: launching an ARexx program from a program that has ARexx support. This means that the program has the ability to launch an ARexx program from within itself. A program with ARexx support has an ARexx Port which you can think of as a software back door through which any ARexx program (including those launched within *this* host application itself) can communicate with this application program and execute this program's command set. A host application program, as we saw before, can both send and receive via its ARexx public message port. The vehicle with which hosts communicate is what is called an **ARexx message packet**.

ARexx Message Packet

H 90 ff
C no reference, but see
C 10-5 f

Whenever any ARexx port receives a message the program which owns that port takes the message, deals with it and sends out a reply. A reply is always sent back in a special variable, usually but not always, called **result**. The host that sent the packet in the first place then does what it's programmed to do with this **result** variable. In the case of the self-referential or recursive use of an ARexx program launched within a host application and sending itself its own commands, the host application also replies to itself with the result variable.

Result Special Variable

H 26
C 10-53

Think of ARexx as a *very polite* society that's gone a little crazy: Everyone always gives a reply when addressed; and everyone always listens for messages as well as replies; and people who talk to themselves, always answer themselves! An ARexx program that can tell itself what to do and then reply to itself like this is sometimes called an **ARexx macro**. One can build up very complex command sequences in any program with ARexx support, and most such programs then allow you to map this macro program to some key such as a function key. When ARexx is used to go outside an application and control or communicate with another application program via its ARexx port, we call it **interprocess control**, although it's still technically a macro, too.

ARexx: The Universal Amiga Macro Language

ARexx Macro

H 45
C 10-79

One of the definite advantages to learning ARexx is that it is the standard for Amiga interprocess controls and macros. Commodore is including ARexx on all its computers now for a good reason. You don't need to learn ten different macro languages as you do on other platforms; on the Amiga, you do it all with ARexx. An increasing number of software developers are making their products with ARexx ports. The time cost of the ARexx learning curve, spread over so many programs, is actually quite reasonable. When you make the next logical leap and consider the ramifications of several application programs each with an ARexx port; each hosting its own ARexx program to send a set of commands to another application program (via its ARexx port), then bringing back and processing the result from the execution of those commands inside the other program, ... you soon find the possibilities mind boggling, but exciting, and undoubtedly powerful.

What Features Does ARexx Have?

Some of the following terms may be unfamiliar to you but you'll soon know them well, as we get into some tutorials. People who have programmed in other languages already, or even a little in ARexx, will appreciate the fact that they do not have to **declare variables**, as integers, floating point numbers, arrays, etc. nor do they have to declare precision (unless they want to). ARexx is smart. It **types** (declares) your variables from their context, on the fly! You can initialize your arrays with one line of code. There is no better language for performing string manipulations, thanks to the ARexx **PARSE** instruction.

Recursive calls are easy in ARexx. You can **CALL** an internal function and **protect** the variables so the calling program cannot confuse its own variables with those inside the internal function, or you can **expose** any variables you choose and let the results of changing them in the interior function filter back to the main calling program. You can make **multi-dimensional arrays** with almost anything as a **node** (qualifier); not just integer numbers but strings or the values of any variable (but not a direct expression. This you must assign first.). There are so many other things you can do, but rather than continue to sing the praises of ARexx, let's just get down to some concrete examples!

The ARexx Tool Kit

Perhaps some specific examples of the tools you need will help, so before we get into coding ARexx examples, please allow some specific recommendations as to what you need in order to make an ideal environment for creating your ARexx programs.

Documentation and Utilities

The original ARexx package distributed by William S. Hawes contains more complete and easier to use documentation including some valuable material left out of the free ARexx implementation provided by Commodore. Even though it is not strictly necessary, Hawes' version includes some very useful utilities and the **rexxarplib.library**, too.

Your Text Editor

TurboText
by
Oxxi, Inc.
P.O. Box 90309
Long Beach, CA
90809-0390
(310) 427-1227

Although you can get by using any editor to write ARexx code, you will benefit greatly by getting a good one. There is no better ARexx editor than **TurboText**® by Oxxi. It not only has a full implementation of ARexx support, but it has a special ARexx window and a console window which allow you to code and test your programs without leaving the editor itself. It has the added feature of emulating all the other popular editors, so if you are used to a particular older editor, you can use its emulation if you like. We will look at some example programs that control TurboText in the tutorials. TurboText is a wonderful editor for any programming language and it contains some specific features for C, Modula 2, and Assembly Language. It has a stand-alone calculator for programmers. It converts among all computer number bases from binary to hex, and has logical operators, too. TurboText is fully configurable which means you can customize your keys to do whatever you want, even launch ARexx programs. It allows you to fold (hide) sections of the text. As you finish a block and you know it works, you can make it disappear so you can see the main program more easily. There are many more useful features in TurboText. It has a superb manual. Finally, it contains a wealth of ARexx examples that do useful things, and which provide some great examples to study.

Your Command Shell

WShell 2.0
by
Wishful Thinking
Development Corp.
P.O. Box 308
Maynard, MA 01754
(508) 568-8695

WShell by William S. Hawes is the best shell environment for ARexx. You do not strictly need a special shell to run ARexx, but if you do a lot of things in ARexx, and particularly if you want to customize your AmigaDOS environment, WShell is wonderful. ARexx is transparent to WShell and runs just like a DOS command, without the rx command, because WShell is a *bonafide* host application program with its own message port and address. Also, there are some powerful utilities that come with WShell such as a file completer called FComp that saves your typing in path names to files. After you type only a few characters of the path, FComp finds the rest of the name for you and types it in for you. WShell features the ability to send itself resident ARexx macros, create complex or simple aliases, to push and pop current directories, and to perform concurrent piping. WShell contains a set of resident commands, maintains a command history, allows programmable prompts and custom window title bars, and is compatible with system 1.3 and 2.0. You may of course keep your Amiga shells as well. Frankly, it's difficult to understand why Commodore chose to keep using their Amiga Shell: They went so far as to recognize the indispensability of ARexx, but stopped short of adopting WShell, the *one shell made for ARexx*, and by the same author! It isn't logical, so it must be marketing. For a modest cost, you can put WShell into your system.

Notes

Chapter 2: ARexx Basics: Files, Strings, and Arrays

Modular Coding: Building a Foundation

In this section we start the tutorials and show you how to make ARexx programs that depend solely on rexxmast and are to be launched from your shell or CLI. Each tutorial is a complete program, but the code can and will be used in other larger programs quite easily. Feel free to use this code in *your* applications. That's what it's there for. The complexity will build as we learn new techniques and improve our code. Each tutorial may build upon whatever material has gone before both as a means to review, and to suggest to you how to make your code **modular** (in pieces which can stand alone). This modularity will teach you something about good programming habits and techniques. We will learn to write pseudo-code to outline what the program is to do, so that when things get more complicated, you'll have the tools to sort out what you want to do before you get down to coding ARexx. In all the tutorials, the ARexx instructions are in CAPITALS, and the variables and strings that are supplied by the programmer are in lower case. This is to make the code easier to read. Note, however, that ARexx *does not require this. The only case sensitive expressions in ARexx are the names and addresses of public message ports; and library and logical file names.* Instructions, commands, and assignments can be mixed case.

Case Sensitivity

Only **names** of *public message ports*, *addresses* of host applications and names of *shared libraries* and *logical files* are case sensitive in ARexx.

Commands and instructions and functions are *not*.

How to Open and Read A Text File

Assume that you have a text file somewhere that you wish to manipulate with ARexx. Write down, or at least study the pseudo-code of what we want to our program to do:

Pseudo-Code:

Step 1: Start the program with a comment /* All ARexx programs start with a comment **delimited** (begun and ended) as you see here */

Step 2: Solicit user input of filename and path to the file, by putting up a prompt on the screen to ask for this information.

Step 3: Take the user input from the screen and assign it to a **string variable** (a group of ASCII characters, a **symbol**) that ARexx can deal with.

Step 4: Open a **logical file** (a **name** to represent the file in the program) for reading. (logical file names are 'quoted') Use the string variable from Step 3 to represent the file name and path to this logical file. If open was successful, go on. If not, do Step 9 (Exit).

Step 5: In a loop, test for the end of file (EOF) marker in your text file. If the program is not at the EOF, do Step 6. If at end of file, do Step 9 (exit).

Step 6: Read (next) line of the logical file (reading starts at the beginning, and each *read line* reads next line in turn, **sequentially**).

Step 7: Write the line to the console window (your shell window).

Step 8: Go back to Step 5. This is the end of the loop.

Step 9: Exit the program with a return code of 0 to say all went well.

An Alternate to Steps 5, 6, 7, 8, and 9:

Step 5: By means of a loop, read each line and write it to the console window (your shell window). Exit when you reach EOF (end of file).

At this point, our pseudo-code tells us pretty much what we want to do. All we have to do now is make a text file to test and write some ARexx code to accomplish the work. Note that the pseudo-code can be as simple or as complex as we want. The alternate to Steps 5 through 9 is one complex step that takes the place of four steps. The "Dick and Jane" Steps 5, 6, 7, 8, and 9 explain what our program does in greater detail, however. We will come to a discussion of the subtle differences between attending to details and ignoring details after we have coded our little program, but for now, keep in mind that these seemingly trivial things are frequently the source of headaches and mysteries, so do not take details lightly.

Start ARexx, Create a Test Text File, Make the Program

If you haven't done so, start rexxmast from its icon (double click on it), or start it from the CLI, by changing directory to where rexxmast resides and type `rexxmast`[Rtn] at the prompt. Take time now to make a text file in your editor of the three lines below. Make sure to put a carriage return at the end of each line by pressing the [Rtn] key. Enter the following in an ASCII editor such as TurboText (or Ed or other editor):

```
An example of a text file.  This is line 1.
Line two of our file looks like this.
Three lines to read from our file, and we are done!
```

Save it to `RAM:Text.file` (and also somewhere permanent, because we will use it again later on.), and then enter the following in your editor (again in ASCII):

```
/* OpenRead.rexx Start with a comment! */
SAY 'Input filename and path.'
PARSE UPPER PULL infile
IF OPEN('textfile',infile,'READ') THEN DO
   DO WHILE ~EOF('textfile')
      line=READLN('textfile')
      SAY line
      END /* end of do while... */
   END /* end of if open(... */
EXIT 0
```

SAY instruction
H 38
C 10-70

PARSE UPPER PULL instruction
H 33
C 10-64 ff

IF instruction
H 29
C 10-58 ff

OPEN() function
H 60, 61
C 10-109

DO WHILE instruct.
H 27
C 10-53 ff

EOF() function
H 57
C 10-103

READLN() function
H 63
C 10-113

END instruction
H 29
C 10-57

EXIT instruction
H 29
C 10-57

Choose a Form for Readability

Take the time and effort to indent as you see the program here. ARexx doesn't care if you indent, but *you* will on longer programs when you need to go back and change them later. A systematic way of indenting your code blocks increases the readability and helps to minimize mistakes, so try to start now to develop good coding habits. Also think of comments as lifesavers later. It is always a good idea to identify all your END instructions with comments as you see here. Notice that they do not interfere with the executable code on the same line as long as they are **delimited** (set apart) properly with **/*** and ***/**. Save this program text you have just typed in as **OpenRead.rexx** (put it in your Rexx or your Rexxc directory). From now on, Rexx and Rexxc will be the interchangeable; Rexxc is in system 2.0 and Rexx is in system 1.3. We'll always refer to the **Rexx directory** but you will know what we mean. Open a shell or a CLI. We will denote what the system or ARexx writes on the screen in **bold**; what *you* type and send is in `courier` type. [Rtn] means hit the return key. Notes about other things are in *italics.* Type in:

```
prompt>CD sys:rexx[Rtn]
```

This command changes the current directory to the Rexx directory. At the next prompt you type:

```
prompt>rx OpenRead.rexx[Rtn]
```

What Happens? The console takes our input to rexxmast which interprets it and returns a reply. I want you to be clear exactly what you are putting in and what the system is sending back to you. The messages will appear on your screen without a prompt, and the cursor will appear just below it all the way to the left, like this:

```
prompt>rx OpenRead.rexx
Input filename and path.
<--cursor will be here.
```

You now type in the path information (no need for a prompt; you are in your first ARexx program!). You type in the last line:

```
prompt>rx OpenRead.rexx
Input filename and path.
RAM:Text.file[Rtn]
```

The system comes back, and the screen now looks like:

```
prompt>rx OpenRead.rexx
Input filename and path.
RAM:Text.file
An example of a text file.  This is line 1.
Line two of our file looks like this.
Three lines to read from our file, and we are done!
<--a mysterious blank line appears here.  Do you know why?
prompt><--cursor will be here, ready for your next command.
```

You are no longer a beginner! You have written and run an ARexx program, and that's about all there is to it (except the *details).* Opening and reading files will be something you use over and over, so we will go over our code line by line and show you why it works. This is about the simplest way to open a file and read it. You will always have one version or another of this routine at the start of most general use ARexx programs. We will soon see how to improve on our original code, but first, the simple, and later the complex.

```
/* OpenRead.rexx Always start with a comment! */
```

The first line of our ARexx program is a comment line. *It accomplishes Step 1 of the pseudo-code.* Comments may appear anywhere in the code and do not affect anything, but there *must* be a comment at the very start of every ARexx program. The **/*** is how we begin a comment and the ***/** is how we end it. These are called comment **delimiters** in computer programmer's jargon. Also, you are allowed to put in blank lines, as they do not affect anything either, but can make complex code much easier to read later. Lines consisting only of comments or entirely of blanks are called **null clauses** in the ARexx definitions. You see the other four types of clauses defined in the margin reference. A **clause** or a **statement** is the smallest unit of ARexx that can be executed by ARexx. We mentioned before that these other four types of clauses may be interchangeably called **statements**, but null clauses are always *clauses*. Three of the five types of clauses appear in our first program.

Five Clause Types

H 14, 15
C 10-31 thru 10-35

```
SAY 'Input filename and path.'
```

The second line is known as a *key word* **instruction clause** (or statement), *and performs Step 2 of the pseudo-code.* SAY is an instruction clause, but if we were merely to put SAY on a line by itself, the program will return a blank line, or skip down one line before writing other output to your shell window. We use the **key word** SAY here followed by a string in quotes (single or double). SAY is not a function; it is an instruction; but it is somewhat similar in that it signifies doing a specific action upon information following it. That is why we call it a **key word instruction**: The key word SAY completes an action using the rest of its own line. Instructions are defined to be clauses (statements) with an initial keyword symbol that is *not* followed by a colon **:** or an equals **=**. The latter **tokens** (entities : and =) identify **label clauses** and **assignment clauses**, respectively. A keyword instruction may contain subkeywords, expressions, or other information specific to the instruction.

Keyword Instructions

H 25
C 10-50 ff

Quotes are used to delimit a **'string token'**. SAY returns a reply to the screen exactly like what is between the quotes, but without returning the quotes themselves. If you want to include the quotes themselves in a string token, you must double them. A **token** is the smallest unit corresponding to a word in an ARexx clause. So, loosely speaking, clauses are "sentences" made up of "words" which we call tokens. In the syntax of ARexx, tokens can also be **comments**; **symbols** (what we think of as *variables*); **operators** (+, -, *, etc.); and **special characters**: *parentheses* (); the *colon* (:); the *semi-colon* (;); and the *comma* (,). Re-read pages 11 through 16 and the top of page 25 of the *Hawes* ARexx manual (pages 10-26 through 10-37 and 10-44 through 10-50 of the *Commodore* documentation) unless this is all perfectly clear to you. Realize the trade off between easy, no-declaring-of-variables ARexx and boring, "Dick and Jane" declaration of variables in some different language. *You are responsible* to know "with which and to whom" when you code in ARexx! Your initial confusion between and among the different elements of ARexx will cause you headaches until you sort out the differences between an *instruction* clause and a *command* clause, between a *symbol* token and a *string* token, and so on. Now back to SAY... If we put a variable (a *symbol* token) which has previously been assigned a value, or an **expression** (a mixture of different kinds of tokens which can be evaluated), after SAY, then SAY replies with the **value** of the variable or of the expression. This is more powerful than it seems as we shall soon see. In our simple case SAY is used to prompt the user to enter some information into the console (shell) window.

Review

H 11 thru 16, 25
C 10-26 thru 10-37
C 10-44 thru 10-50

Expressions

H 15 thru 17
C 10-35 thru 10-37

```
PARSE UPPER PULL infile
```

The PARSE UPPER PULL instruction is used to take input from the user from the screen , *and is the ARexx code equivalent of Step 3 of the pseudo-code.* PULL waits until you type something in and hit return. The PULL instruction used alone without the PARSE UPPER in front of it accomplishes the same task with less typing on your part. I include the PARSE instruction here for you to make its acquaintance, because it is the single most powerful instruction in ARexx when it comes to dividing up (parsing) strings. In our example, the entire line you enter is first converted to UPPER CASE (that's what the UPPER option is for, and UPPER must immediately precede all other options). Then it is parsed (a way of assigning a string or part of a string to a variable or several variables). In this case the entire line you type in is put into the

Symbol Tokens

H 11, 12, 21, 78
C 10-28 ff, 10-44
C 10-148

string variable (a **simple symbol token**) *infile*. Now, whenever we refer to infile, the ARexx resident process will evaluate it as the exact line or string we entered after the SAY instruction above it. The only way it will change, is if we decide to assign it another value in an assignment statement. PARSE UPPER PULL or PULL will take any input from the user. If you sent

```
Mary had a little lamb.[Rtn]
```

the variable *infile* would end up as a string variable evaluated to: **MARY HAD A LITTLE LAMB**. If you had entered 1234[Rtn] after the SAY instruction, the value of infile would be the integer **1234**. This is why we say ARexx **types** our variables on the fly from context. (Now if I could only get ARexx to *type my correspondence* as well...) Variables are therefore what you make them and you can change them any time at all, you just have to be aware that this can get you into trouble at times. For instance if your program had tried to open a filename and path called **MARY HAD A LITTLE LAMB** or **1234**, your program wouldn't get very far!

```
IF OPEN('textfile',infile,'READ') THEN DO
```

The next line of our program, *Step 4 of the pseudo-code translated to ARexx*, is a bit more complicated. It demonstrates the wonderful compactness of ARexx in that it combines an instruction and a **function** in one line: an IF instruction and an OPEN() function. Functions are easy for you to identify. If it has the parentheses (), it is a function; in our case here, we have an example of one of the built-in functions of ARexx, OPEN(). Later, we will see how you can "roll your own" functions in ARexx with a minimum of effort. Remember that a function is a special kind of **expression** (because it can be *evaluated*). Functions have the following syntax: A function is a string followed immediately by an open parenthesis, then arguments separated by **special characters** (commas), then a closing parenthesis. Are you beginning to see how everything is inter-related and neatly arranged and defined? Not yet? You will soon!

Special Characters

H 13
C 10-30 f

Let's take the OPEN() function first. All functions take arguments, the stuff inside the parentheses. Arguments are always separated (delimited) by a comma ,. Here we have three: **'textfile'**, **infile**, and

'READ'. The quotes are important. **'Textfile'** is the **logical name** (a *string* token) of the file we will open. This is a name you make up. Here we call it **'textfile'** to remind us that we are reading a text file. The second argument is **infile** (*no quotes*). We don't use quotes because infile is a *string variable* (a simple symbol token) which when evaluated is the exact string you entered at the console window, translated to UPPER case. That's what the PARSE UPPER instruction did for you. You could leave out the SAY and the PARSE lines if you knew you would always read a certain file with a certain path. Let's pretend you have a text file called **MyNotes.doc** in a directory called **Docs** in your hard drive partition **DH0:**. If this example program were intended to read only the lines in **MyNotes.doc** every time you ran this program, you could leave out the lines with SAY and PARSE, and make the statement into:

```
IF OPEN('textfile','DH0:Docs/MyNotes.doc','READ') THEN DO
```

Note that the second argument is now a name (a string token) and not a variable (a symbol token), so you use quotes. The SAY and the PARSE lines are merely a way to vary the files our program 'reads'. The third argument is **'READ'** one of three **options** which tells the OPEN() function that we wish to **'READ'** the file instead of **'WRITE'** to it or **'APPEND'** (add to the end of an existing file). We may use **'R'**, **'W'**, or **'A'** (upper or lower case) to denote these options, also, but leave in the quotes (**'options'** when they are arguments to a function are string tokens, too. Remember, by definition, string tokens need quotes around them.

Now we need to discuss something the manual leaves a little unclear. Functions (unlike instructions) always, *always* return something, in the way of a reply, so *you must never forget to do something with the returned value*, or your ARexx program in the host application will attempt to execute what is returned, and it probably won't make sense as a command! Remember, if it doesn't make a positive ID on anything it gets sent, rexxmast will call it a command and send it to the current host address. Now, the function OPEN(), if successful returns the number 1 (boolean or logical true); and if it fails (i.e. you've let MARY HAD A LITTLE LAMB stand for infile), then it returns (you guessed it!) the number zero (0) to stand for a logical or boolean false. Imagine my chagrin when I finally found out why my programs were hanging up with messages saying *unknown command: 1*. The resident process was simply trying to execute the replies from my OPEN() functions! It thought they were commands, so it sent them back to the WShell as commands.

I was using WShell which is a host application (unlike the Amiga Shell which won't quibble about this), so it can send and receive messages. The WShell makes the current host address to be **'WSH_6'** or whatever number of shell was running the program. When the OPEN() was successful, it returned a 1 to the WShell which WShell did not know as a command, so I got an error message. If you try this with and ordinary Amiga Shell, nothing will seem to go wrong because the current host address is **'REXX'**, but you will not be doing it correctly, and you will get into trouble when you start working with *bonafide* host applications.

There are three ways to do something with the reply from your OPEN() function (if you are in a host application) and prevent an error from happening. Note: you can deal with *any function* by doing one of the following. First you can CALL it:

CALL instruction

```
CALL OPEN()
```

H 26
C 10-53

Refer to page 26 of the *Hawes* ARexx manual (page 10-53 of the *Commodore* documentation) for the CALL definition. Notice the fine print. The value returned by the function is assigned to the special variable RESULT, so CALL automatically does something with the returned value. It assigns it, and so rexxmast doesn't think it's a command. The second way to prevent trouble is to assign it yourself to anything at all. For example:

```
reply=OPEN()
```

If OPEN() succeeds, **reply** will have the value 1 assigned to it, and it will be 0 if OPEN() fails.
The third way is more subtle. You make the result mean something to a key word instruction that uses an expression (our function) in performing its work. If we write:

```
SAY OPEN()
```

then a 1 will appear on our screen if OPEN() worked, and a 0 will appear if not. Remember SAY can write the value of something like a variable (a symbol token). In our example, the IF instruction is one that tests for a 0 or a 1 directly, so it is quite natural and ingenious that ARexx allows the IF instruction to do something directly with the returned value of OPEN(), or with any other function that returns a boolean value. No matter how complicated you may think logic is, it all boils down to a true or a false; a 1 or a 0 in the end. Every IF statement in every computer language evaluates some **conditional expression** (no matter how convoluted) into a 1 (true) or a 0 (false) and branches accordingly! In our example, IF OPEN() THEN DO boils down to: If the returned value of OPEN() is 1, then do the following block of instructions down to the END instruction for this DO block of instructions. If the value is 0, then skip to the first clause after the END instructicn. In our example this would be the EXIT 0 instruction clause. Naturally, we can use IF and other logical instructions to deal directly only with functions (and expressions) that return a boolean value (1 or 0). If a function returns a string, for example, we have to use a less picky instruction like SAY, or assign it to a variable of our own, or CALL it.

```
DO WHILE ~EOF('textfile')
```

The next line is the start of a DO WHILE loop, *which encodes to ARexx Step 5 of the pseudo-code*. A DO WHILE loop tests a conditional expression and proceeds as long as it is true. The expression here is **~EOF('textfile')** which is pretty readable to start with, don't you think? Translated, the whole line reads, While we are not past the end-of-file marker of our logical file, **'textfile'**, then we will continue to **iterate** (do over and over) the instructions contained in the block delimited by the DO WHILE instruction and its END instruction. The tilde (~) is used to mean **not** in ARexx and may be compounded with any expression (or function that returns a boolean value (0 or 1) to negate it. Here, we want to continue something as long as we are **not** somewhere: the EOF. EOF() is a function, too, and like OPEN(), it returns a boolean value, so we can use it in a test expression, in this case negated so that not EOF will return a 1 and EOF will return a 0. You may have noticed by now that we have two instruction blocks in our program, one connected to the IF OPEN() THEN DO line, and one nested inside that block (a DO WHILE instruction block). Block instructions always have an END instruction to delimit the block. If our END instructions do not match the ends of the blocks we intended, or if we leave one out (easy to do in longer programs), we get some strange and unwelcome errors.

Quotes must match up as well. If you are at all experienced at programming in any language, you will not find it strange to see me stress small, obvious points, because you will have undoubtedly experienced firsthand the old proverb, It's the *little* foxes that spoil the whole vineyard! Syntax can be taxing, in other words.

```
line=READLN('textfile')
SAY line
```

OK, so now we have two lines of instructions to execute over and over until we get to the end-of-file marker. The first is an example of an **assignment clause** where we set something equal or equivalent to something else. We assign the value of the return from the function READLN() (which means to read one line from a file) to a variable (a simple symbol token) we call **line**. Notice that this time our function returns *something else* besides a boolean 0 or 1. It returns an entire line from our file, but we still must do something with it or else expect rexxmast to try and execute that line, fail, and send it out as a command. We assign the value of this line (the evaluated string expression) to our variable **line**. Note that we don't need to type or classify **line** beforehand, we just let ARexx do its thing and classify it for us as a simple symbol token. The assignment clause then causes rexxmast to evaluate the tokens to the right of the = as an expression and the result becomes the value of **line**. Note that **line** has no quotes; its not a name. It's a **string variable** (a simple symbol token). The following line is the familiar SAY instruction to illustrate its utility in printing out the value of a variable to the screen. We could have combined two lines into one:

```
SAY READLN('textfile')
```

would accomplish the same output to the console, but usually you'll need to do something more with what you read in from a file and the assignment clause is therefore essential to know.

```
      END /* end of do while... */
   END /* end of if open(... */
EXIT 0
```

We now come to the END of the DO WHILE loop (Step 8 of the pseudo-code) and go back to test for EOF. When the DO WHILE loop is satisfied, it passes control to the next instruction past its END instruction, which is another END instruction, this time the one that finishes up the original IF OPEN() THEN DO block. Since this isn't a loop, we keep going to find an EXIT 0 instruction to exit the program, replying with a Return Code of 0 to the shell or CLI. Although it's not necessary to use it here, it's a good practice to get into, and a return code of 0 means everything went OK. In Shell, you can program its prompts, and displaying the return code is one option. If you do a lot with ARexx, program the Shell prompt to display the return code.

You now understand a lot more than when you started. As we progress, we will not discuss old topics in as much detail as in this tutorial, but from time to time we shall repeat important things as a review. If you want to do some exercises to gain confidence before you go on, try modifying this program to do different things, and think about how you would answer the following questions.

1. How will I deal with input mistakes from the user in my ARexx code?

2. What if I want to deal with only some of the words in lines I read from the file? How do I deal with individual letters of those words?

3. What can I do to warn myself if the file is wrong or doesn't exist?

4. Where did that blank line in the output come from?

Improving the Opening of a File

Now that we know how to open a file, let's improve our first program and try to answer some of the questions at the end of the last section. Type in the following into your editor and save as **OpenReadImp.rexx** as always in your **rexx** directory:

```
/* OpenReadImp.rexx Improved file reading */
PARSE UPPER ARG infile
IF infile = '' THEN DO
   SAY 'Input text filename and path: '
   PULL infile
   END
IF ~OPEN('textfile',infile,'READ') THEN DO
   SAY 'File cannot be opened. '
   EXIT 20
   END
DO UNTIL EOF('textfile')
```

```
    line=READLN('textfile')
    SAY line
    END
EXIT 0
```

This program illustrates the way we can *implicitly* ask for the file and path as an argument to an ARexx program. We have met the PARSE instruction before, but this time we are parsing an ARGument (the file name and path) on the same input line as our program. ARG by itself is shorthand for PARSE UPPER ARG, which is similar to the way PULL operates. Let's say our text file is still in **RAM:text**, so at our prompt we input (remember the **.rexx** qualifier is optional):

prompt>`rx OpenReadImp.rexx RAM:text[Rtn]`

and our program runs in one step. The output is the same as for the first example. Notice that we do not need to put quotes around **RAM:text**. Why? Because as before, the program is parsing it directly into the variable **infile** which gets assigned the value of the argument. If we put quotes they would be evaluated because we are dealing with a *simple symbol* token and not a *string* token. Quotes can be a pain in ARexx until you understand how they work. The program checks to see if you have supplied an argument, **infile**, and if not (it's the *null string* '') then ARexx prompts you to enter the file name and path as before, and then PULLs it from the screen.

Error Checking

Notice how the program begins to check for errors. If we put **1234** after our program name, then the OPEN() function will fail and return a 0 and the IF statement will be true because OPEN() is negated by the tilde ~ which you recall means *not*. So if *not* open is true (an error) then the program puts a message on the screen and sends back a return value of 20 to show it failed. The rest of the program is exactly like the first one, except we've used UNTIL instead of WHILE and not negated EOF() to show you one more way to do the same thing. WHILE is evaluated at the start of each iteration, and UNTIL is evaluated at the end of each iteration. This is useful if you need to leave the loop in different ways, but here these two instructions are interchangeable. Now let's find out some details and learn to count lines, and later on words and letters.

Counting Lines

Make this program just like before. Call it **CountLines.rexx**:

```
/* CountLines.rexx */
/* Opening & Reading a File. Finding the EOF. How
many lines? */

SAY 'Input filename and path.'
PULL infile
rcode=20
IF OPEN('textfile',infile,'READ') THEN DO
   rcode=0
   i=1
   DO WHILE ~EOF('textfile')
      line.i=READLN('textfile')
      /*IF line.i ~= '' THEN*/ SAY 'Line#'i'='line.i
      i=i+1
      END
   i=i-1
   /*IF line.i-1 = '' THEN i=i-1 */
   SAY 'There were 'i' lines in this file.'
   END
ELSE SAY 'Could not open your file!'
EXIT rcode
```

Notice the way we have **commented out** two sections of the code by putting **/*** in front and ***/** behind the sections. This is a good way to change a program temporarily and leave out code you aren't sure you want to use or delete just yet. Our purpose here is to test the number of lines in our file.

Some Fine Points About the End Of File

Run the program as it stands on your file **RAM:text** and see if it comes out correctly. What? We have the wrong number of lines? No? Everything is fine? Why expect to get different answers? It depends on how you entered your text, where the **EOF marker** ends up. If you entered the text *exactly* as instructed before, you hit [Rtn] after *every line* and our program says we have more lines than we put in. But wait! Text editors do a Carriage Return and a Line Feed at every [Rtn] so that last [Rtn] we put in added an extra blank (or null) line which now consists of only the EOF marker. If you use an editor like TurboText, you can make the end-of-lines and end-of-files visible as a special patterned character, and you can then see two ways of having the "same" file. So our program is "right" after all, and we are "wrong". The parts of the program **/* commented out */** if put back in, will take care of all possibilities, so that no matter how we save the lines into a file, the lines will be counted properly and null lines left out.

Returning the Value of an Expression

Other new things we are using here are the IF THEN ELSE construction to return a code (we call it **rcode**) to the caller (the shell) depending on whether we were successful or not in opening our file. Note that we make our own **rcode** here. Do not confuse our return code with the special variable Return Code **RC** in ARexx, which changes after every function call, and is primarily used to carry error messages at interrupts which we will look at later. Here, we just want to signal the shell with a **0** if all went OK, and a **20** if we couldn't open the file. We just assign **rcode** a value according to how everything came out and the instruction **EXIT rcode** carries it back to the shell. We are developing good habits here. Our **rcode** is not necessary to run the program, but use it anyway. When you must write complex programs you'll be glad you took the time to learn how to do it right.

Arrays: Stems with Nodes are Compound Symbols

Array elements and **Compound Symbol Tokens** are equivalent terms.

H 11, 21ff
C 10-28, 10-44 ff

We are introducing **Arrays** (**Compound Symbols**) for the first time here, too. We want to count lines. So what better way than to set (assign) a counter (i=1), and use this number as the **node** of our **stem symbol** (the *name* of the array). Reading pages 11 and 21ff of *Hawes*, (or 10-28 and 10-44 ff of *Commodore*), we notice that a stem symbol has exactly one period (.) at the end of its name (here, **line.** is our stem symbol). A **node** is what comes after the period(s) in a stem symbol, and we are allowed to have multiple nodes: **node1**, **node2**,... etc. up to as many dimensions as we need. We only need one dimension to count lines. In ARexx, a **node** is defined to be a **fixed** or a **simple symbol**, so you can't put in a node that is an *expression* such as **i+1** without *first assigning* it to a fixed or a simple symbol. We could write **n=i+1** and only then could we use **n** as a node. When the node is combined with its stem symbol, the whole thing is called an **array element** in general terms, and a **compound symbol** in ARexx jargon. A node which is itself a compound symbol is illegal, and must be assigned to a simple symbol before becoming a valid node in an array.

Each node, evaluated and combined with its stem makes up a *new name* which is used in place of the compound symbol (**stem.node1.node2.node3** ... is the compound symbol). In English this means each of our lines from our file will be given a different name. Our three lines will be represented by three compound symbols: **line.1**, **line.2**, and **line.3** ...simple enough!

Initializing and Counting with Arrays

Initializing Arrays (assigning Stem Symbols)

H 21
C 10-44

Don't let arrays scare you. They are a "bear" in other languages, but this is ARexx and they're fun and easy here. Want to initialize a whole, gigantic array to make every element = 0? Let's call our gigantic array **GAR**. and our elements go from **GAR.1.1** to **GAR.1000.2000** ...but one simple instruction in ARexx initializes every element: **GAR.=0** (For *any* array; *any* dimension: *same* instruction!).

Now make those *commented out* sections of your **CountLines.rexx** program active, and re-save the file. Each time we go through the DO WHILE loop, we are reading a line from our file, only this time we assign a different compound symbol (**line.i**) to the value of the line, because **i** starts with **i=1** outside the loop and we increment **i** (add **1** to **i**) each time through.

Expressions Can Mix and Match Symbols

The IF instruction takes care of SAYing only those lines which are *not* null (''). The expression after SAY is worth looking at, however. It is an **expression** because it mixes literal **string symbols** (**'Line #'** and **'='**); a **simple symbol** (**i**); and a **compound symbol** (**line.i**). This expression is evaluated during each loop iteration to one of the three strings that you see on your screen:

```
Line #1=An example of a text file.  This is line 1.
Line #2="Line two" of our file looks like this.
Line #3=Three lines to read from our file, and we are done!
```

Count from Zero or One?

After the loop finishes, we need to **decrement** (subtract 1 from) **i** because it got incremented once too often. We could have started with **i=0** and avoided this. Inside computers most things are counted starting from 0 instead of 1 for the reason that it is easier to reset things in a binary (1 or 0) environment by initializing to 0 (low or off) instead of 1 (high or on). We will do this sometimes, too, but it seems more natural to start counting things at 1 and not 0, so we do that here.

Exhausting All Possible Outcomes

The next line takes care of the situation if we put in a [Rtn] at the end of the last line of our file. If we did, then there is an extra line containing only an EOF marker, and **i** is still one too many, so we need to decrement it again. We output a message to check our final line count and come to the end of the loop.

Now we see an example of the ELSE instruction. It goes with the IF OPEN() instruction, and takes care of the other possibility: we didn't open our file, in which case the program tells us it couldn't open the file. Finally, we exit with the appropriate **rcode** of **0** (success) or **20** (failure).

You've seen three variations on the way to open a file and read in the lines of that file, and have had a taste of the power of **compound symbols** which are *arrays of related variables*. In the next section we'll use some of these techniques to start taking apart lines into words, so you can witness the unique power of ARexx when it comes to string manipulations. You will find if you experiment on your own that even taking apart arrays of words into arrays of letters is merely a simple extension of this technique. It is easy to make a stem **let.** with three nodes that represent for instance the value of the second letter of the second word ('example') of the first line or our sample data file: **let.1.2.2='x'**. An example program (**UNIarray.rexx**) using multi-dimensional compound symbols is in the Program Listings in the Appendix. Compare it to the program discussed in the next section.

How to Take Apart Lines of Text Into Words

Breaking Lines into Words

Now we want to break up our lines into words. The best way to do this is by using compound symbols (arrays). Since we've learned to put our lines into arrays, we have the necessary tool already. Let's combine one of the programs from the previous section with a custom made **internal function** (a string followed by () containing arguments) to give you a taste of how easy it is to make modular programs in ARexx.

Format for Future Readability

In your editor, open the previous tutorial's program file called **OpenReadImp.rexx** as a start and type in the changes shown and save as **GetWords.rexx** Note the use of comment blocks. Comments never affect the program. Also note the use of null clauses (blank lines) to make the readability better. We do this to make future reference easier.

```
/* GetWords.rexx Getting a WORD List from file.*/

SAY 'Input filename and path.'
PULL infile
rcode=20
IF OPEN('textfile',infile,'READ') THEN DO
   rcode=0
   i=1
   DO WHILE ~EOF('textfile')
      line.i=READLN('textfile')

   /***********************************/
   /* modify code into a new block    */
   /***********************************/

      IF line.i ~= '' THEN DO
         SAY 'Line #'i'='line.i

         /*******************************/
         /* call an internal function   */
         /* named Stripword (see below) */
         /*******************************/

         CALL Stripword(i,line.i)
         END

/***********************************/
/* end of new block                */
/***********************************/
      i=i+1
      END
   i=i-1
   IF line.i = '' THEN i=i-1
   SAY 'There were 'i' lines in this file.'
   END
```

```
ELSE SAY 'Could not open your file!'
EXIT rcode

/*******************************/
/* end of main program         */
/*******************************/

/*******************************/
/* add a new internal function */
/*******************************/
Stripword: PROCEDURE    /*label*/

/* get the argument list from caller */
PARSE ARG j,list

/* how to strip off words from a line*/
k=0
DO WHILE list ~= ''
   PARSE VAR list kthword.k list
   k=k+1
   END
SAY 'There are 'k' words in line.'j

/* write a list of words        */
DO n=0 to k-1
   SAY 'WORD#'n+1' is:'kthword.n
   END

/* RETURN to caller in main program */
RETURN
/****************************/
/* end of internal function */
/****************************/
```

RETURN instruction

H 37
C 10-70

RETURN is similar to **EXIT** and is used as the final instruction of an internal function.

The *value* of an expression may also be returned as in:

RETURN *expression*

See margin note on page **2-21** of this book. The *value of **Fun**(args)* becomes the value of its returned *expression*.

As before, run **GetWords.rexx** from your shell or CLI. Unless you use **WShell** don't forget **rx** in front of it! When asked for the filename and path use **Text.file** from the previous sections and your screen output will look like this:

```
Line #1=An example of a text file.  This is line 1.
There are 10 words in line.1
WORD#1 is:An
WORD#2 is:example
WORD#3 is:of
WORD#4 is:a
WORD#5 is:text
WORD#6 is:file.
WORD#7 is:This
WORD#8 is:is
```

```
WORD#9 is:line
WORD#10 is:1.
Line #2="Line two" of our file looks like this.
There are 8 words in line.2
WORD#1 is:"Line
WORD#2 is:two"
WORD#3 is:of
WORD#4 is:our
WORD#5 is:file
WORD#6 is:looks
WORD#7 is:like
WORD#8 is:this.
Line #3=Three lines to read from our file, and we are done!
There are 11 words in line.3
WORD#1 is:Three
WORD#2 is:lines
WORD#3 is:to
WORD#4 is:read
WORD#5 is:from
WORD#6 is:our
WORD#7 is:file,
WORD#8 is:and
WORD#9 is:we
WORD#10 is:are
WORD#11 is:done!
There were 3 lines in this file.
```

Notes on the Code

We will explain the sections of code we have modified.

```
   /**********************************/
   /* modify code into a new block   */
   /**********************************/

      IF line.i ~= '' THEN DO
         SAY 'Line #'i'='line.i

         /*****************************/
         /* call an internal function  */
         /* named Stripword (see below)*/
         /*****************************/
```

```
CALL Stripword(i,line.i)
END
```

A Key Word Instruction

Keyword Instructions

H 25
C 10-50

First, we put in a DO block so that we can do several commands instead of simply one SAY instruction. The comments show our new block clearly. We've kept the old SAY instruction, but after it we put in a CALL instruction. To review: Remember instructions which have something on the same line on which to operate, are called **key word instructions**. SAY is a key word instruction, an instruction clause that starts with a keyword, *not* followed by a colon (:) or an equals (=), that identifies the instruction. It is followed by: one or more **subkeywords**; expressions upon which the keyword(s) operate; or any other information specific to that instruction. Refer to page 25 of *Hawes* or page 10-50 ff of *Commodore* to get the entire formal definition. This will help us not to confuse a keyword instruction with a *function* which *operates* upon *arguments*.

Calling an Interior Function

An **Internal Function** may be invoked with the **CALL** instruction, or by implicitly calling it by name. If the function is named **Fun()**, then you may
CALL Fun(*arguments***)**
or assign something to it as in
variable=Fun(*args***)**
In any case, the function **Fun()** is identified by the label clause **Fun:**

So, our newest keyword instruction is a CALL instruction which *calls a function* (a type of *expression*) named **Stripword()**. This is a custom made **interior function**. We identify this interior function by putting in a **label clause**. This is the only type of clause (or instruction) we have not met until now. A label clause is a name with a colon (:) at the end of it. It acts like a *place marker* in our program, so that the interpreter **rexxmast** can find our function. An interior function is actually *the code between its label clause and a RETURN keyword instruction (which may return the value of an expression).*

Look at the code below the main program and you will see our function **Stripword()**. In our case here RETURN does not return an expression value, because we do not need it to, and using the CALL instruction insures that the RESULT will be dealt with, remember? If not, refer to page 26 of *Hawes* or page 10-53 of *Commodore.*

Functions

H 8, 14, 15, 26, 35
C 10-21, -35, - 47, - 48
C 10-82 thru 10-91

```
/******************************/
/* call an internal function  */
/* named Stripword (see below)*/
/******************************/
```

```
CALL Stripword(i,line.i)
```

CALL instruction

H 26
C 10-53

Arguments

H 8, 22, 26
C 10-21, 10-48, 10-53

We CALL a function, the function does something and it returns. What does it do and to what? To answer "to what?" is why we need to include **arguments** to our function (the stuff inside the parentheses). **Stripword(i, line.i)** passes two arguments: the value of **i** (which depends upon which iteration we are in), and the value of **line.i** (also dependent upon which iteration we are in). The arguments are delimited (separated by) a comma (,). The *parentheses are optional* when using CALL, but they match the syntax of ARexx built-in functions (with which they must be used), so we include them here for better syntactical consistency.

The PROCEDURE Key Word Instruction

PROCEDURE instr.
EXPOSE option

H 35
C 10-68 f

Symbol Table

H 23
C 10-48

Rexxmast, when it encounters the call, looks for and finds a label called **Stripword:** and *the program branches to that label.* On the same line with our label is a PROCEDURE keyword instruction. This is used within an internal function to build a new **symbol table**. This is how ARexx **protects** the variables in the *calling program* (the main program we just left) from being altered. In a big program, you may well forget that you had a variable named **furbz** in your main program, and once you are coding an internal function, you might create a different variable that represents different things from your original **furbz**. If there were no *protection* available by way of a *separate symbol table* in the interior function, then you (or at least your program) would get into deep guacamole soon enough.

Exposing Symbols

Just in case you want to *expose a variable* (symbol) in the main program to the effects of operations in the interior function, there is an optional **subkeyword** to PROCEDURE called EXPOSE, following which you put in the list of variables you want to expose to the machinations of your interior function. Variables in an EXPOSE list are processed from left to right. If you are trying to pass a variable that is a compound symbol to be exposed in a function, make sure to list the symbol(s) of its node(s) before (to the left of) the compound symbol itself, or it won't be evaluated fully. Take time now to read section 4-20, page 35 of *Hawes*, or pages 10-68 and 10-69 of *Commodore* for the definitions of the PROCEDURE instruction.

Later, we will make a program to show how EXPOSE works. Unless you put in the EXPOSE subkeyword explicitly, the default is to *protect* all of your main program's symbols by building a *new* symbol table. The program has now branched to our internal function in which all symbols are protected, and we follow it.

The Internal Function

```
/******************************/
/* add a new internal function */
/******************************/

Stripword: PROCEDURE    /*label*/

/* get the argument list from caller */

PARSE ARG j,list

/* how to strip off words from a line*/

k=0
DO WHILE list ~= ''
   PARSE VAR list kthword.k list
   k=k+1
   END
SAY 'There are 'k' words in line.'j

/* write a list of words        */

DO n=0 to k-1
   SAY 'WORD#'n+1' is:'kthword.n
   END

/* RETURN to caller in main program */

RETURN
/****************************/
/* end of internal function */
/****************************/
```

The next instruction after we get to the label clause

```
Stripword: PROCEDURE
```

is our old friend PARSE. We PARSE ARG this time, because we do not want to change everything to UPPER case, and neither PARSE UPPER ARG nor its equivalent ARG will avoid this. We simply parse the arguments in the same order they were sent: **i** first, then **line.i**.

Notice *we do not need to name them the same in our function procedure*. The originals are *protected* because we didn't EXPOSE them, and we could have used their exact names with complete impunity, but we've used *different* symbols to emphasize that fact.

Parsing Multiple Arguments

Now when we use PARSE ARG (or use ARG as a shorthand for PARSE UPPER ARG), ARexx provides the capability of parsing **multiple strings** at one time (*only* for the **ARG** option to the PARSE instruction, however), which is what we are doing here. We have two strings arguments sent from the caller (the main program) to parse: **i** and **line.i**.

PARSE instruction

H 33 thru 35
H 77 thru 81
C 10-64 thru 10-68
C 10-146 thru 10-154

See also the next chapter of this book.

Parse Templates

We introduce here a new term used to discuss parsing: the **template**. A template is simply a pattern. When a furniture maker is cutting out scroll work for use on all four sides of a table, he makes a template first, and then cuts the wood according to the template to insure uniformity. Parsing is similar except it cuts out a pattern from a string or strings, and *multiple patterns may be cut from the same string*. Think of our woodworker being able to use the same piece of wood for all four sides of the table. Every time he cuts it, the entire piece of wood is available for another cut using the same or a different pattern. The woodworker can build these cut pieces into anything he wants: table sides, bookshelves, etc. Even though with wood it is *physically* impossible, this is exactly analogous to how parsing can assign any pieces of the string to any kind or number of variables, over and over again if necessary.

Parsing uses several kinds of templates. In our examples up to now the template, as such, was one variable name to which an entire line is assigned; simple, but not too interesting. The ARexx PARSE instruction is very powerful and it is definitely worth studying how the PARSE instruction works until you learn how to use it. The next chapter discusses parsing in detail, but for now, we will discuss only one kind of template at a time and concentrate on *using* PARSE in actual examples. It will accelerate your progress and increase your options if you always review *Hawes*, section 4-19 on pages 33 to 35, and also chapter 8 on pages 77 to 81, whenever we discuss parsing. The equivalent *Commodore* documentation is on pages 10-64 to 10-68 and 10-146 to 10-154. You will begin to see the powerful ways strings (*and therefore commands for application software*) can be manipulated easily. Now back to the example...

The Comma is a Special Template Pattern

```
PARSE ARG j,list
```

In our example, the template is a special one used only within ARG or PARSE ARG. Parsing multiple strings is accomplished by a **special pattern**, the comma (**,**) in the **template** (our pattern for assigning multiple strings to variables or symbols). The template itself is always the information following the PARSE instruction itself, in this case **j,list** is the template.

In our first Tutorial, in the **PARSE UPPER ARG infile** line, the template was simply **infile** a single variable name which was also a template in disguise. This is some of that self-referential or recursive power mentioned before. Think about how clever and how ingenious it is to name your variables, create a pattern (template) with which to assign values to these variables, and make the assignments all in *one line* of code! Try doing this in some other language and you will immediately appreciate the beauty and compactness of ARexx. ARexx can get slightly abstract at times because so little code can stand for so much power, but once you get to a certain level of prowess, you'll never look back.

Templates Assign Variables

Our two **argument strings**, **i** and **line.i** were sent by the main program. The **template** says, "Take the *first* **argument string** and assign its value to **j,** then take the *second* **argument string** and assign its value to **list**. The comma in the template says we have *two separate strings.* We are allowed to use *multiple commas* to PARSE ARG *multiple argument strings.* The comma is very important; don't leave it out or you will see strange results. The result of all this is that the original values are put into variables like this:

```
     i --> j
line.i --> list
```

You may be more comfortable with the equivalent notation:

```
   j = i
list = line.i
```

The DO Loop

```
/* how to strip off words from a line*/

k=0
DO WHILE list ~= ''
   PARSE VAR list kthword.k list
   k=k+1
   END
SAY 'There are 'k' words in line.'j
```

Now we come to a DO loop right after we initialize a counter, **k.** This should look familiar as it is the same way we counted lines. We are starting from **0** this time however as a variation on our theme. In this familiar loop, we come to another of those PARSE instructions, except it has a subkeyword of VAR. This time the template is

```
list kthword.k list
```

What pattern is this template? First notice that we are using a new type of PARSE instruction itself, a PARSE VAR instruction. This means to parse a VARiable, and you tell it which variable to parse by naming the variable immediately following the VAR option. So **PARSE VAR list** means we are to cut up the variable list somehow and assign the parts according to the rest of the template.

Parsing: One Word, One Variable

The next symbol is a compound symbol **kthword.k** which stands for the kth word of our line. The **k** counter just as we did for lines, counts up the words in our line one at a time and each word in the line gets a unique name indexed by its *node* **k**. So **kthword.k** is assigned to some part of the variable list, but which part? Our template operates *implicitly. Unless we say otherwise*, the first variable *named in the template* gets assigned the value of the **first word** of the VARiable, in this case **list**. A **word** is defined to be the first part of the string *delimited by one or more blanks*. Since very few people enter text without putting spaces between their words, it is a safe bet that you'll see the first word of the original line (now represented by **list**) assigned to the compound symbol **kthword.k**.

The Last Variable Gets All the Rest of the Parse String

The other implicit operation of a template is that when the *last* variable in our template is to be assigned a value, it gets assigned *the value of all that is left over* from the original **parse string** (the input string to the parse instruction). In our case, in the first iteration of our DO loop, we have only put the first word of **list** into a variable called **kthword.0**; we still have the rest of **list** (*minus its first word*) to go.

Using Self-Reference to Advantage

By using a clever trick of letting *the very symbol* which represents **list** get the assigned *value of all the leftovers*, we set it up for the next time through the loop, all in one shot! Next time the

```
PARSE VAR list kthword.k list
```

is executed, **list** will be shorter by one word (its original first word), and will be ready to get *its* first word (this time the original list's *second* word) stripped off and assigned to a new **kthword.k** because **k** will have been incremented to the next higher number. Then the remainder of **list** (now shorter by *two* words is assigned *to itself*, and we go on.

Each time through this loop, one word gets nibbled off and assigned to the element of an array. Since we are in the DO loop only as long as our list is not the null string (''), we get all the words assigned to a unique name in our array, which is now ready for anything.

The Effects of EXPOSING Variables

Before moving on, we need to look at some of the subtle things that happen if we EXPOSE our variables to an internal function. Let's do an experiment. Is there a way we can get the same results as the previous example and EXPOSE **i** and **line.i** to our function **Stripword()**? Try substituting the following code for the function in our **GetWords.rexx** program and save it under another name such as **Tst.rexx**.

```
/* Same main program as before ....*/

/* an experiment with exposing variables to an
internal function */
Stripword: PROCEDURE EXPOSE i line.i
k=0
DO WHILE line.i ~= ''
   PARSE VAR line.i kthword.k line.i
   k=k+1
   END
SAY 'There are 'k' words in line.'i
DO n=0 to k-1
   SAY 'WORD#'n+1' is:'kthword.n
   END
RETURN
```

Now try running it. Do you get exactly the same results? You will if the EOF marker is on its own separate line, but the last message in the screen will say we have *one line too few* if the EOF marker was at the *end of the last line* of our text file.

Try creating two text files, one in which you save it after backspacing until your cursor is at the end of the last line of the text (and nothing is after it) and another file that you save with the cursor on a "blank" line. Using the original **GetWords.rexx** program, it will make no difference which file you use, both outputs will be correct. But using our **Tst.rexx** program, the former text file will display an incorrect number of lines in the final screen message. Why? Because we have *exposed our variables* to the function and the effects filter back to the main program.

Since we stripped off the words one at a time from **line.i** until it was the null string (''), this *final* **line.i** is picked up by the main program as a *null line* and the **IF line.i = '' THEN i=i-1** statement in the main program that decrements again if necessary is *always* true, so we decrement when we really don't want to (in the case of the EOF at the end of the last line of the text file and not on a separate line).

We can fix this by making two assignments in the internal function: **line=line.i** as the first instruction after the label clause, and a **line.i=line** assignment clause as the last instruction before the RETURN instruction. This fix stores the value of the *original* **line.i** in **line** and restores its value when we leave, but this is *de facto* what *protecting* our variables does (by not using EXPOSE in the *first* place). This rather subtle glitch demonstrates why we have to be very careful what we do with variables which we EXPOSE to an internal function after that function executes.

Notes

Chapter 3: Parsing Made Easy

Parsing is Central to Most ARexx Programs

Parsing

H 33 thru 35
H 77 thru 81
C 10-64 thru 10-68
C 10-146 thru 10-154

Keep your ARexx documentation open to the above sections as you read this section.

If possible, enter the programs **ParseTest.rexx** and **PU.rexx** (function) and run **ParseTest.rexx** as you read this section. It will help you to see the results of a parse instruction immediately.

Since parsing is central to almost anything you do in ARexx, it is essential that you become fluent in this instruction at the earliest possible opportunity. We have been discussing the PARSE instruction frequently in the preceding sections, and this would be a good place to enter into a more rigorous tutorial concentrating solely on parsing and templates. Remember the general definition of parsing: an operation that extracts **substrings** (pieces) from a larger string and assigns them to variables called **targets**. The way in which these substrings are extracted is determined by a pattern called a **template**. The **source string** on which PARSE operates is specified by the **option subkeywords**. So far, we have met the options, (sometimes modified by the elective subkeyword UPPER which must come immediately after PARSE): VAR, ARG, and PULL. In this chapter, we will also cover the remaining options, NUMERIC, SOURCE, VALUE *expression* WITH, and VERSION. You will want to turn in your *Hawes* ARexx manual to pages 33 through 35; and the whole of Chapter 8, pp 77 through 81 as we study some examples. In the *Commodore* documentation, use pages 10-64 through 10-68; and pp 10-146 through 10-154.

We will code a program that will help us to experiment with various ways to parse a source string with different templates. It will display the strings assigned to template targets. Copy the following two programs, **PU.rexx** and **ParseTest.rexx** and save in your **rexx** directory. You need not pay too much attention to understanding how the **ParseTest.rexx** program works in order to learn to parse. The program is more complex than our fare so far. The program is described, however. You may wish to review it later if you find it difficult, and skip over its description for now.

A Useful PARSE Utility

Let's make a utility function to help visualize the scan positions in a parse source string. We will write it as an **exterior function**, which we may **CALL** from within our parse emulation program:

This small function will help us to see the column positions of any expression we send it for purposes of parsing the string without having to count the characters every time. It is a simple matter to send this function any string variable and see the column positions. The function returns the value of the expression in the EXIT instruction, in case we want to use it. In returning from an exterior function, we use EXIT instead of RETURN, but both accomplish the same thing. We have used the **SAY expression** instruction to verify that **PU.rexx** is passing back the value of our expression.

```
/* PU.rexx PARSE function */
OPTIONS RESULTS
PARSE ARG expression
pos1='1234567890123456789012345678901234567890123456789012345678901234567890'
pos2='         1111111111222222222233333333334444444444555555555566666666667'
SAY
SAY pos2
SAY pos1
SAY expression
EXIT expression
```

Making a Parse Tester Program

The following program uses ARexx to modify its own code. Self-reference is not easy to understand at first glance, but this program illustrates just how powerful ARexx can be. What we want to do is simple enough, but its implementation is subtle.

Pseudo Code

We want to choose a PARSE option, and then display (using the function above) the appropriate parse string. Next we want to enter single or multiple templates with targets, markers, and so on. Then we want the program to display the actual evaluation of the substrings assigned to each target, with markers at each end to reveal leading and trailing blanks if any. (The program will therefore have to modify its own code and then execute a PARSE instruction.) The program should allow us to begin again with a new PARSE option; enter a new template for the same source string; input a new value for a position or numeric pattern variable; or exit. It should be a useful tester to try out various parse templates and options while we are learning to parse.

```
/* ParseTest.rexx Parse experiments */
start: /* a label clause for use by SIGNAL start*/

/* default test line */
line='This is a line of text. No.123456 #98765. The last sentence!'

SAY 'Enter PARSE OPTION: 1=NUMERIC 2=SOURCE 3=VAR 4=VERSION'
PARSE PULL option
SAY 'Parse UPPER? Y/N'
PULL uo
IF uo='Y' THEN uo='UPPER';ELSE uo=''
name=' '
SELECT
   WHEN option=1 THEN DO
      PARSE NUMERIC line
      option='NUMERIC'
      END
   WHEN option=2 THEN DO
      PARSE SOURCE line
      option='SOURCE'
      END
   WHEN option=4 THEN DO
      PARSE VERSION line
      option='VERSION'
      END
   OTHERWISE DO
      option='VAR'
      SAY 'Enter test parse source string or [Rtn] to use built-in test.'
      PARSE PULL input
      IF input='' THEN input=line; ELSE line=input
      name=' line '
      END
   END

CALL PU.rexx line

mid: /* label for SIGNAL mid */
position=''
SAY
SAY 'Enter PARSE Template (only template string).'
PARSE PULL template

/* get only alphabetic targets and not markers */
temp=template
n=1
DO WHILE temp ~=''
   PARSE VAR temp tar.n temp

   /* take off the commas, keep the targets */
   IF RIGHT(tar.n,1)=',' THEN tar.n=LEFT(tar.n,LENGTH(tar.n)-1)
   IF LEFT(tar.n,1)=',' THEN tar.n=RIGHT(tar.n,LENGTH(tar.n)-1)

   /* get rid of non alpha stuff */
   IF ~DATATYPE(tar.n,MIXED) THEN ITERATE

   n=n+1
   END
n=n-1
pat=0 /* set default flag for column number */
```

```
/* the expression we wish to execute */
template='PARSE' uo option||name||template

pos: /* label for SIGNAL pos */
/* if we find "position" used in the template string, ask for value */
/* depending upon whether it is a pattern or a column number        */
IF FIND(template,'=position')~=0|FIND(template,'(position)')~=0,
THEN DO
   /* pattern */
   IF FIND(template,'(position)')~=0 THEN DO
      pat=1 /* set flag for pattern */
      SAY 'Enter pattern variable'
      PARSE PULL position
      SIGNAL out /* get out of here! skip the rest. */
      END

   /* a column number */
   SAY 'Enter position number'
   PARSE PULL position
   /* check for valid column number */
   IF ~DATATYPE(position,WHOLE) THEN DO
      SAY 'Not a valid column position.  Try again.'
      SIGNAL pos /* do over again */
      END
   END

out:
/* display stuff */
SAY
CALL PU.rexx line
SAY
SAY template
SAY
/* the heart of the matter. execute the code we built */
INTERPRET template

/* display the targets and their values */
DO i=1 TO n
  SAY tar.i'==>'VALUE(tar.i)'<'
  SAY
  END
/* if we use position, display its value depending on flag pat...*/
IF position~='' THEN DO
   IF pat=0 THEN SAY 'Position='position' =Position is at column 'position
   IF pat=1 THEN SAY,
      "Position="position" (Position) matches pattern '"position"'."
   SAY
   END
/* control options */
SAY 'Enter S=start over; N=new template; P=new position; Q=quit.'
PARSE UPPER PULL in
SELECT
   WHEN in='Q' THEN EXIT
   WHEN in='S' THEN SIGNAL start
   WHEN in='N' THEN SIGNAL mid
   WHEN in='P' THEN SIGNAL pos
   OTHERWISE SIGNAL START
   END

EXIT 0
```

Running the ParseTest Program

Launch this program from a Shell or WShell. Enter the number for the option you want to parse. If you select VAR, then you may enter your own parse string (called **line** by the program), or use the program's default test string. The other options will display the correct parse string for that option. Choose whether to PARSE with the UPPER subkeyword next. When prompted, enter a *template only*; the PARSE OPTION part is automatic. Templates using patterns, markers, absolute and relative positions, and targets of any *alphabetic* characters name may be entered. Make sure to separate template elements by a space. Commas separating templates may be next to a target, however. While ARexx will support just about *any* character string for a variable name, even numbers, our emulator will not. Your targets must be alphabetic or they will not display, even though correctly parsed. You may also start over, make another template, or enter a new variable **position** by entering **S**, **N**, or **P** respectively. Entering **Q** quits.

Testing a Variable Position Marker

To use a *variable* as a scan position, then use the *lower case* name **=position** or **(position)** in your template (**position** acts as a variable name). You will be prompted to supply the position pattern or its value; a whole number. If you wish to use **position** as a *position marker*, use **=position** as the template entry; if you want to use **position** as a *variable pattern to match*, use **(position)** as your template target. Patterns not represented by a variable should be entered normally inside single quotes in the template. You may also experiment with multiple templates separated by commas. The options ARG, PULL, and VALUE expression WITH are not included as their templates work exactly the same as the VAR option.

The output shows the names of your targets and the exact strings assigned to them by the PARSE template: **target**<==>*string*<== on separate lines. You can then see leading and trailing blanks if present. If you are using the variable **position**, the program will display the value you entered for **position**, and whether it is a pattern to match or a column position. You can learn much about parsing by experimenting with this utility. It is handy when you are developing a program to check to see if you are parsing your strings the way you intended.

INTERPRET Instr.

H 30
C 10-59

A related subject is
Inline Arexx Program
W = *WShell 2.0* Manual

W 78, 79

An inline ARexx program may run within a shell or another ARexx program. It is delimited by double quotes (") and program steps are sepatated by semicolons (;) thus: "*step1*;*step2*;*step3*" will run as a **separate process**.

SELECT instruction
H 38
C 10-70

SIGNAL instruction
H 38
C 10-71 ff

VALUE() function
H 68
C 10-122

FIND() function
H disk notes only
C 10-104

LEFT() function
LENGTH() function
H 60
C 10-108

RIGHT() function
H 63
C 10-114

Notes About the Theory of Operation of ParseTest.rexx

We list the program here, because it will help you to explore the PARSE instruction. You may wish to skip ahead to the **Parsing by Tokenization** section to continue to learn to parse. This program may seem a bit out of sequence until you explore a few later sections.

Self-modifying Code

Most of the program is straightforward but it does depend upon self-modifying code. The INTERPRET instruction is used to execute an expression. This is the part of our program that after modifying its own code, it actually executes a PARSE instruction according to the template you have entered; first building up an expression containing the modified code. This expression is a valid ARexx statement and the INTERPRET instruction causes it to execute. To make the program as readable as possible, we assign the expression to the variable **template** and then INTERPRET it. An INTERPRETED statement or program is *not a separate process*, but the instruction itself activates a control range for it as though it were a DO...END block of instructions. *Hawes*, page 30; and *Commodore*, page 10-59 document the INTERPRET instruction.

Most of the code is to take care of the string handling necessary to build up our expression to be executed. We use an array for the target names, and check them for commas (separating templates) and remove them if necessary from the target name. Then we check for alphabetic characters to distinguish target names from markers. The SELECT and SIGNAL instructions are covered later in this book. SELECT is used for multiple choice options. SIGNAL is used here as a simple GO TO some *label* instruction, to handle our choices, and to handle the possibilities for a pattern as opposed to a numeric variable. Note the use of the VALUE() function to distinguish between a *name* for a target **tar.i** and its *actual evaluation* **VALUE(tar.i)**. We must *evaluate* **tar.i** with VALUE() because the variable whose name is represented by **tar.i** first acquires a value when we INTERPRET **template**. *Hawes*, page 68; and *Commodore*, page 10-122 document VALUE().

For now, use the program to learn what PARSE does, and do not be overly concerned if you do not fully understand this program. After you have studied the later chapters, you may want to review this program.

Tokenization, Forced Tokenization

H 34, 79
C 10-147, 10-150

Note: These exercises assume you have NOT chosen to PARSE UPPER at the second prompt, but you may try them with the UPPER subkeyword if you like.

Parsing by Tokenization: Words into Targets

We have studied PARSE VAR before, but not where we had fewer variables than words in the line. Notice what happens. Run the program **ParseTest.rexx** and choose VAR option by entering the number **3**. Press [Rtn] twice again to use the default string. At the prompt to enter the template, enter (Enter always means to type in the line and hit [Rtn]):

```
a b c d rest
```

Now look at each target symbol in the template. The variables (symbols) **a b c d** all get assigned only one word, but the variable **rest** gets the rest of the line assigned as its value. Note also that the value of **rest** includes the leading blank space as well. Blanks, both leading and trailing are *not* included in the value of any variable which is assigned a *word in the string* as its value. In other words, when PARSE matches up *words* (tokens) in a string to *variables* (targets,symbols), it doesn't matter how many blanks separated the words (tokens); they will *not* be part of the variable's value. The last variable always gets what is left over in the line, that has not been assigned. What happens if there are *more* variables than words? The extra variables are assigned the value of the *null* string ('').

Targets

H 77
C 10-146 f

Targets are Variables in the Template

In ARexx the variables (symbols) in the template line are called **targets.** In this example, our five targets are **a b c d** and **rest**. The tokens (words) assigned to them are the words from **line**. This type of parsing is called **parsing by tokenization**. Genuine parsing by tokenization always insures that there are no leading or trailing blanks in the variables assigned in this manner. In order to make sure that you are parsing by tokenization, *always include an extra period* (**.**) at the end of the parse template. If you have exactly as many variables to assign as you have words in the input string, ARexx will leave a leading blank on the last target value in the template. The period (**.**) acts like a placeholder for a valid target except it is assigned no value.

Forced Tokenization

An extra placeholder period is essential to prevent strange errors from happening should the leading blank interfere with your intentions. Including an extra period (**.**) is called **forced tokenization**. In the **ParseTest.rexx** program, enter **S** to return to the start. Choose option **3** again (or [Rtn] because option 3 is the default). Now enter this line:

```
one two three
```

Supply a template **a b c** and then look at the strings assigned to these targets. Did you notice **c** has a leading blank? Enter **N** to use another template and this time supply the template **a b c .** (note the extra period in this template). Now forced tokenization has eliminated the leading blank.

Pattern Markers and Scan Position

Scanning

H 78, 79
C 10-149 f

Our second example, is how to PARSE VAR using a pattern **marker** instead of words. The other general type of template object besides the **target** is what ARexx calls a **marker** which determines the **scan position** within the parse string. The scan position ranges from 1, the *start* of the parse string, to the length of the string plus 1, the *end* of the string. There are three types of *marker objects*: a**bsolute**, **relative** and **pattern**. The parse string is scanned from left to right and the template determines, implicitly by *tokenization*, or explicitly with *markers*, just how the variables (targets) will be assigned substrings from the parse string. Notice the differences between the way the PARSE instruction works with a *pattern marker* and the way it works with tokenized *words*, as in the first example. Begin again (enter **S**). Use the default test string in the program, and supply this template:

```
a b '#' c d rest
```

With the program, verify that this template assigns the value of the first word (token) "This" to the variable **a**. The pattern marker **#**, however, which occurs at the start of the second number **98765** in our string, tells the PARSE instruction to assign to the second variable **b** the part of the string running from the **current scan position** up to but not including the first character in the pattern **#**. Look at the value of **b** with the program.

Since PARSE always scans the PARSE string (the variable **line**) from left to right, the current scan position is where this scan of the string left off after the last target was assigned the value of some substring. In the example, when PARSE gets finished with assigning the value of the first word (token) **This** to the target **a**, the current scan position is at the end of **This** in the string **line**.

The assignment of **b**, therefore is the value of the substring that goes from the first character (a blank) following **This** to the last character before '#' (also a blank), which we see printed to the screen as the value of **b**.

How Parsing with a Pattern Marker Divides the Parse String

Pattern Parsing
Positional Parsing

H 80
C 10-151

The next variable is **c** which takes the value of the first word in what is left of the string *to the right* of the **#** character. The pattern marker in the template effectively *divides the string* into two pieces which may be parsed with their own templates just like one string may be parsed: that familiar self-reference once again! Wait! What happened to the **#** character? The ARexx parser *removed* it from the PARSE string.

The Pattern is Removed During a Re-Scan

Whenever a match with a pattern marker is found by the scanner, then the *entire pattern* is removed from the PARSE string. If you try to scan the string again from the beginning, you will find the pattern gone. Enter **S**, choose the default string, and supply a new template **a b '#' c d 1 e** and you will find # missing when you look at the target **e**. The positional pattern **1** started the scan over at position 1 and assigned the entire string to **e** (see below).

The Source String Remains the Same

The original *source* string is never altered by PARSE pattern matching, however. The PARSE instruction makes a *copy* of the source string for actual parsing. For example, our variable **line**, if displayed after the above PARSE pattern matching instruction would have the character "#" in its proper place.

Try the above template with a comma after **d**. This signifies a **second template** using the original source string and **e** will now include the **#** character. *A re-scan removes the patterns previously matched and a new template uses a new copy of the original source string.*

Parsing Using Positional Patterns

(Absolute and Relative Markers)

The next way we are going to parse is by using a position pattern in the template. This **marker** may be an explicit number, designating the **absolute character column** in the parse string; a **relative marker**, specifying a certain positive or negative offset from the current scan position; or a **variable marker**, where the *position or pattern* is determined by the value of a variable.

The Scan Position and Multiple Scans

Start over (enter **S**) and use the default string and this template:

```
=1 wA +1 wB =position wC 19 wD 'No.' -4 rest 1 all
```

Choose the number **5** when prompted for **position**. The template starts with a *number*, **=1**. This tells the PARSE instruction to start scanning at the start of the PARSE string, which you can see is position 1. We mentioned our ability to scan more than once. The last target (variable) **all** is preceded by a **1** also. The = sign is optional with integer position markers. The last **1** tells the PARSE scanner to start at the beginning (absolute position 1) of the parse string. Notice also, as before, the *patterns* are removed from the parse string as the pattern marker matches them, so by the time we scan for the assignment of the entire parse string to **all**, the pattern **No.** is *missing*.

Absolute, Relative, and Variable Positional Markers

In between the first **absolute positional marker**, (**1**) and the second **relative positional marker**, (**+1**), lies the target (variable) **wA**. Exactly as before, the PARSE scanner assigns the value of a substring of the parse string to our variable based upon the template. Translated, the template says, "Assign to the target **wA** the value of the substring lying between absolute column **1** and the column determined by *addition* (**+1**). The number 1 is to be added to the **last scan position** (column 1 in this example)." The template fragments **1 wA +1** and **1 wA 2** therefore produce identical results.

Point of Reference for Relative Positional Markers

The Scanning Process

H 78 f
C 10-149 f

Relative markers may also use subtraction to *back up* a specified number of places in the position of the scanner, as we find in the assignment of the variable **rest.** The *point of reference* of a relative positional marker is the index of the *first character of the last matched pattern*, or it is the *previous absolute positional marker*. Relative positional markers preceded by a minus sign *subtract* from the *same* point of reference. Remember that patterns are *removed* from the parse string. If no match was found for a pattern marker, then the index is placed at the end of the parse string. The scan proceeds from left to right from index 1. The scan position is updated whenever PARSE finds a marker object, according to the type and value of the marker.

Variable positional markers may use the value of a variable previously assigned a value, as in the case here of the variable **position**, which we assigned a *value* of **5**. The template fragment **+1 wB =position** therefore assigns the value of the substring from position **2** (**1 +1**) up to position **5** (**=position**) to the target (variable) **wB**. As before, the *left column position is included* in the string, and the *right column position is not.*

Continuing the scan, from left to right, the target **wC** takes its value from the variable **position** (**=5**) up to but not including position **19** designated by an absolute marker. Note the substring assigned to **wC** has a leading and a trailing blank.

The next target is **wD**, which has a trailing blank, because parse scans from position **19** up to the last character before the pattern **No.** as we saw before.

We now tell the parse scanner to back up 4 places (a relative marker **-4**) and assign the rest of the line to the variable **rest**. The scanner counts backwards: **space**, **period**, **t**, **x**, and therefore will begin the new substring with **x**.

What If PARSE Cannot Find a Match?

Since the next marker is an absolute marker **1**, *the beginning of the line*, the scanner cannot get a match by continuing from left to right, so by default, it matches *the end* of the parse string, **line**. **This is a general**

principle of the PARSE instruction: When it *cannot find* a pattern or a positional match in the current scan, it *matches automatically the end of the parse string.* Therefore, our variable **rest** is the rest of the parse string. Note again that the pattern **No.** is now *missing* because it was used as a pattern marker before.

Multiple Scans and Multiple Templates

Multiple Templates

H 80 f
C 10-153 f

There is still one more variable left: **all**. The **1** tells the scanner to make another pass starting with position **1** and since there are not any more targets (variables) to assign, the entire parse string is put into the target **all**. The pattern **No.** is missing as noted above. If we wanted to use this pattern in another assignment, we could either assign it to a target with the template *before* using it as a pattern marker, or put in a *second* parse template after this one to scan the source string **line** anew. Each new template is separated by a comma from the previous template as we did in our experiment above, putting a comma after **d** to start a new template. Try a new template (enter **N**), and put a comma after **rest**. Did the pattern 'No.' come back when **all** displayed?

In order to distinguish between multiple templates we are using a **special character**, the comma (**,**) to separate templates, just as we did with the internal function call to separate its arguments. Even though *only one* parse source string may be used at a time with PARSE VAR (as compared with PARSE ARG, which, you remember, can parse up to 15 *multiple* source strings), you may use **multiple templates** with *all* options of the PARSE instruction. Remember, the *parse source* is never altered[1]; **line** dstill contains the pattern!

1 If you PARSE VAR with multiple templates and one of the templates uses the VARiable itself as an assigned target, then the value of the source string in this case could change since the *current value* of the source string is used.

Patterns as Variables

The next example illustrates how small changes in *syntax* can mean big changes in results. Enter **N** to make a new template, and change the **=position** variable marker to **(position)** in the parse template. It makes a big difference! This syntax, putting parentheses around the variable in the template, tells the PARSE instruction we are specifying a *pattern*, not a *position* with our variable. This time, the assignment, **position=5**, means something entirely different when interpreted by the template. It means match the **pattern '5'** and not the *position* **5**, so both **wB** and **wC** are different this time, because the parser finds a match on the number

5 *in the parse string* and divides the parse string at the position of that *pattern*, rather than at *position* 5.

Notice that the *character* **5** is left out of the parse string when scanned later, because it is classified as a *pattern* marker now, and not any kind of numeric position marker. Note: you may leave off the parentheses around **position** if it is to be a *pattern* variable, and it will work, but syntactically, and for ease of readability later, using the parentheses is better. *Always* use the parentheses around **position** as a pattern variable when using **ParseTest.rexx** or it will think **position** is an ordinary target. This will reinforce your using the better syntax.

More Ways to Parse

We will now look at some various other ways to parse, some new, some variations on the above techniques. Replace the *source strings* and the *parse templates* with the ones we show now, if you want to try out various input strings and templates. Display and study the targets until you see what is happening during the PARSE instruction. Practice and experimentation are good ideas until you feel thoroughly at home with parsing. We will now list some source strings and templates.

In-line Variable Patterns

Choose option 3: VAR
The test line is...
This&is*a line of text. No.123456 #98765. The last sentence!
Use this template:
```
wA =position delim +1 wB (delim) wC (delim) rest
```
position=4

Study all the targets. Note the way we can make an *in-line* call to a variable pattern **delim**; in this case the pattern is the letter **s**, and we use it to *delimit* our targets. The first instance of **delim** gets assigned the value **s** and the subsequent variable patterns **(delim)** are evaluated to the same pattern value during parsing.

The Period as a Placeholder

The use of a period (.) as a placeholder in a template is an important feature of the PARSE instruction we have discussed before. You may use a period (**.**) as a placeholder for *any* target in the parse string to which you *do not* wish to assign a value. This is useful if the source string has more words in it than you have variables (targets) to assign and you want to avoid assigning the last variable in your template the value of the rest of the parse string.

Choose option 3: VAR
The test line is the default...
This is a line of text. No.123456 #98765. The last sentence!
Use this template:

```
. '#' num '.' .
```

Look at the target **num**. Since we are only interested in the number coming after the # character, we use two periods (**.**) to represent parts of the string which we do not want to assign. They act exactly like any *bonafide* target in the parse template, except they never get assigned a value. We also use two pattern markers, a **'#'** and a period **'.'** to assign *only* the number, and not the **#** character or the **period** at the end.

Using the Relative Marker +0

We now illustrate the use of the **relative marker +0** to include the *previous* literal or variable pattern. The following shows how to assign the same substring to two different targets (variables) without re-scanning the line.

Choose option 3: VAR
The test line is the default...
This is a line of text. No.123456 #98765. The last sentence!
Use this template:

```
. 'No.' num . +0 j .
```

Study **num** and **j**. The relative marker **+0** tells the parse scanner to use the *last pattern matched* (**No.**) as the *start* of the next variable. The period after both variables **num** and **j** prevents the scan from including anything except the number, by acting as a placeholder for the *rest of the string.*

Using Parsing by Forced Tokenization for Screen Data Input

Sometimes you will need to input data from the screen to fill data fields in an ARexx program. The PARSE instruction used to tokenize the input makes this a simple process. Suppose you are entering expense account data. You have the *item name*, the *amount* you spent, the *date*, and the *category*, for instance. You want to place these data into variables for calculations and eventual entry into your data base file. You may use the PARSE (or the PULL) instruction to take the data from the screen to your variables in one step. PARSE PULL will preserve the case of your entries, and PULL will convert everything to upper case, but otherwise they operate identically.

Let's assume you will enter your data in this order:
date, **item**, **category**, **amount**. The following program illustrates how to use PULL to parse these into targets by tokenizing them.

```
/* TokenScreen.rexx Example program to tokenize
screen input */
SAY 'Enter: date item category amount'
PULL date item cat amt .
SAY date
SAY item
SAY cat
SAY amt
EXIT 0
```

The period as a placeholder is used to prevent the last item from having a preceding blank. Remember tokenizing removes blanks from the target, but assigning the rest of the line to a target does *not*. Experiment with this program and try it with different inputs. See how the program assigns your data neatly to the variables. What would you do to prevent mistakes on entry? Can you write some *error checking* routines to improve the program? Can you write some routines to make each entry format consistent; for example the date entry?

Parsing Multiple Input Lines from the Screen

Suppose you need to ask ARexx to parse multiple input lines from the screen. The way to do this is to use multiple templates separated by commas, as we have discussed. Use the PARSE PULL instruction, or PULL by itself which will translate all the strings to UPPER case. In the preceding example, change lines *two* and *three* as follows:

```
SAY 'Enter: date item category [Rtn] amount'
parse pull date item cat .,amt .
```

We have changed the prompt to remind the user to put the *amount* on a separate line by hitting the [Rtn] key. Then, we have put a comma (**,**) in line three to tell the PARSE PULL instruction to wait for two lines of input: the *first* template **date item cat .** will tokenize *date*, *item name*, and *category* from the *first* screen input line; and after you hit [Rtn], and type in the amount, and hit [Rtn] again, the *second* template will tokenize your entry into the target **amt**. Slick! You may use multiple templates to parse different strings in this way from the screen.

Other PARSE OPTIONS

The remaining four PARSE options are EXTERNAL, NUMERIC, SOURCE, and VALUE *expression* WITH. They allow you to parse some special information from the ARexx program environment, and you may use templates exactly like the ones we have discussed already. The VALUE *expression* WITH option evaluates *and* parses an expression on the same line, which frequently proves useful.

EXTERNAL: Not Recommended!

The option EXTERNAL is supposed to be equivalent to PULL, but it does some unexplainable things once in a while, like putting in a mysterious extra line in the output of the above example of multiple line screen inputs. The extra line appears between the **SAY cat** and the **SAY amt** output instructions. Apparently, the extra line is generated somehow if we use a PULL EXTERNAL instruction instead of PULL or PARSE PULL. Since EXTERNAL may not be used alone (like PULL), and since the above example doesn't work using a PARSE EXTERNAL instruction, there doesn't seem to be much reason left to use it at all! You're better off if you forget all about the EXTERNAL option.

The PARSE expression VALUE WITH template Option

If you do not have a VARiable, but an *expression* to parse, use this option. It operates exactly like the PARSE VAR option, except that the expression is first *evaluated*, and then its *value* parsed. If you use multiple templates with this option, the expression is evaluated only once, at the start of the instruction. The subkeyword, WITH, must always *immediately precede* the template. Here is what this option produces if we use the default **line** from our emulator program as part of a larger expression.

```
The test line is...
This is a line of text. No.123456 #98765. The last sentence!
PARSE VALUE 'Example #1:'line WITH . pre ':' all, nm . '#' '#' num +5
The targets evaluate to:
pre = #1
all =This is a line of text. No.123456 #98765. The last sentence!
nm  =Example
num =98765
```

The format of this instruction is always:
PARSE *expression* WITH *template*[*,template*][*,template*]...

We have shown the instruction using *two* templates, separated by the required comma (,), but any number of templates may be used. In the first template, **. pre ':' all** the scan starts with a placeholder (**.**) target because we do not want the first word in the parse string (the *value* of the *expression* **'Example #1'line**) to be assigned to a target. Note that the *parse string* is just the string 'Example #1' appended to the start of our variable **line**, so expanded, the parse string is

```
Example #1:This is a line of text. No.123456 #98765.
The last sentence!
```

You may enter this *evaluated* string into **ParseTest.rexx** to experiment, and verify that PARSE VALUE expression WITH acts exactly like PARSE VAR once the expression has been evaluated.

The second target in the first template is the variable **pre**, lying between the end of the first word of the string and the pattern marker **':'**. It has a preceding blank, because it isn't a *tokenized target*. How would you make it a tokenized target and get rid of the blank? Putting a period placeholder target **.** immediately after **pre** and before **':'** will do it. Try it! The use of the colon (**:**) as a pattern marker insures that it will be removed from the parse string and will not appear as part of our variable **pre**. The variable (target) **all** now takes the entire left over string which, thanks to the use of the pattern marker **':'**, is the entire original string variable **line**. The comma after **all** ends template *one*.

Updating the Scan Position with Multiple Sequential Patterns

Template *two* is the pattern **nm . '#' '#' num +5** which ought to be somewhat easy for you to read by now. It tokenizes the first word "Example" of the parse string into **nm**, skips the substring from the end of the first word until the first instance of the pattern **'#'**, skips to the second instance of **'#'**, and assigns the five digit number found there to the target **num** by means of a *relative position marker*, **+5**. The only thing new here is to observe the use of *two pattern markers* in a row *without* any targets (variables) in between. This is legal.

Direct and Indirect Control of the Scan Position

You may use multiple patterns in a template without needing to assign anything to a target; even a placeholder period (.) is not necessary. This is because the scan position is updated every time a marker object is encountered. Therefore, a pattern marker object, without an associated target, explicitly and directly updates the scan position without demanding that the parser assign a string to a target. A template may therefore control the current scan position *explicitly and directly* or *implicitly and indirectly*. Notice that since we have ended up by controlling the scan position directly with the **+5** relative marker, it is not necessary to use a placeholder or any other targets in the second template, even though part of the parse string is left over.

NUMERIC instruction

H 31
C 10-61 f

SOURCE option

H 33
C 10-65

VERSION option

H 34
C 10-66

Parsing with the NUMERIC, SOURCE and VERSION Options

The only options we have not yet met are the NUMERIC, SOURCE and VERSION options of the PARSE instruction. You may look at their source strings by selecting the other options in **ParseTest.rexx**, and you may also experiment with various templates applied to these options. The only real difference between these options and VAR is that they *supply their own source strings*. To learn more about NUMERIC settings which have to do with precision and numeric display formatting, refer to *Hawes*, page 31; or *Commodore* page 10-61 f.

The SOURCE option returns information about the program you are running: whether it is a COMMAND or a FUNCTION; a boolean 0 or 1 to reflect whether a RESULT string was requested by the caller; its name; its path; its file extension; and the name of the host application port. Use this option when you are doing interprocess control with ARexx and want to check on the status of a function or a command. *Hawes*, page 33; and *Commodore*, page 10-65 document the SOURCE option.

The VERSION option returns information about the hardware, and the version of ARexx we have installed. These values are explained on page 34 of *Hawes* and page 10-66 of *Commodore.*

Summary of the PARSE Instruction

Perhaps it seems we have beaten parsing to death by now. When you realize how much you'll use it, you will appreciate these lessons as much as you appreciate your third grade teacher for drilling those multiplication tables into you when you were little! Parsing is *essential* to almost every ARexx program. To review:

PARSE provides a way to extract one or more substrings from a string and assign them to variables called **targets**.

PARSE uses a pattern called a **template** to describe how substrings of the parse string will be assigned to the **targets**, which are variables (symbols) written into the template. Besides targets there are **markers**, the other type of template object. The parse string is scanned from left to right. The markers serve to keep track of the **current scan position**, and to determine the endpoints of substrings assigned to targets. There are three types of marker objects: **absolute**, **relative**, and **pattern** (which may be variables) *explicitly* determine the scan position of the parse string. Tokenization *implicitly* determines the scan position by assigning words from the parse string to targets one-for-one.

PARSE operates on an *input* string, called the **source string**. PARSE *never* alters the source string. It makes a *copy* called the **parse string**. **Pattern markers** alter the parse string. Patterns are removed from the parse string as they are matched.

Markers may be explicitly represented as absolute numbers; relative numbers with an accompanying arithmetic operator (+, -, =); patterns in quotes; or they may be implicitly represented by a variable name, preceded by an operator (+, -, =) for **+positional** markers, or enclosed in parentheses to denote a variable ***(pattern)***.

Targets are either a *variable* name or a *placeholder* period (.) which represents a target not assigned a value.

The special character comma (,) serves to separate **multiple templates** and **multiple source strings** passed as ARGuments to a function.

If a template is used to parse a string word-for-word to its targets, it is called **parsing by tokenization** and blanks are automatically removed from the beginning and the end of the target values. If an additional period (.) is placed at the end of a template, it **forces tokenization**.

The source strings can come from various places, and we use different **PARSE OPTIONS** to handle them:
For function or command *arguments* use PARSE ARG
For *numeric* settings use PARSE NUMERIC
For program *source* information use PARSE SOURCE
For *version* information use PARSE VERSION
For *expressions* use PARSE VALUE *expression* WITH
For v*ariables* use PARSE VAR
For *console* input use PARSE PULL, PULL, and PARSE UPPER PULLThe optional subkeyword UPPER must follow the keyword PARSE immediately. It translates everything in the parse string into UPPERCASE.

Chapter 4: Numbers, Logic, and Recursion

Logical and Mathemetical Operators

Operators

H 12, 13, 18 thru 21
C 10-29 thru 10-31
10-39 thru 10-44

Arexx offers a full complement of mathematical and logical operators to enable you to program mathematical solutions to various problems. Refer to *Hawes*, pages 12 and 13; or *Commodore*, pages 10-29 through 10-31 for the definitions and examples of operators. There are in general four types of **operators** in ARexx:

Arithmetic Operators
H 18 ff
C 10-39 ff

Arithmetic operators use one or two numeric operands and return a numeric result.

Concatenation Oper.
H 20
C 10-42

Concatenation operators join two strings into one string.

Comparison Operators
H 21
C 10-43

Comparison operators compare two expressions and return a 0 or a 1 (boolean) result.

Logical Operators
H 21
C 10-44

Logical operators require one or two boolean operands and return a boolean result.

To demonstrate the power of ARexx to solve mathematical problems, let's solve one about probability, using a few of these ARexx operators. The problem is an interesting one, because it caused a storm of publicity and controversy when it first appeared in the Sunday newspaper's *Parade Magazine* in the "Ask Marilyn" column. The "Ask Marilyn" column frequently exhibits intriguing puzzles whose solutions are by no means intuitive. When she proposed this puzzle, and offered her proof, Marilyn aroused so much controversy, much of it from prestigious university professors who "proved" that she was "wrong", that she published several follow-up articles both to hold her ground and to invite people to convince themselves of the correctness of her proof, by conducting a series of experiments. The program we will write here is a computer simulation of an empirical proof of Marilyn's puzzle.

The Door Problem

The problem goes like this. A man is a contestant on a game show. There are three doors, behind one of which is a new sports car. If the contestant picks this door, he of course wins the car. He is invited to pick a door. He does, but now the game show host introduces a twist. He opens one of remaining two doors that the contestant has not picked. It is empty. He now asks the contestant, "Do you want to stay with the door you have picked, or will you switch your choice to the remaining door?" The question is this: Is the contestant better off to *stay* with his original choice, or to *switch* to the remaining door? In other words is the *probability* (the contestant's chances) of a win higher if he stays or if he switches?

A Brief Definition of Probability

Probability uses pretty simple arithmetic. In an experiment, the total number of equally possible outcomes is represented by an integer **n**. The number of ways one particular event **E** can happen is represented by another integer, call it **h**. There are **h** ways of **E** happening out of a total of **n** possibilities. The probability of **E** is represented by the number (a fraction, a rational number) **h/n**.

The convention is to write the probability of success **Pr{E}** (event E happens) as **p = Pr{E} = h/n**. So, in a flip of a coin, the probability of *heads* is 1/2, because *heads* can occur in one of two possible ways: *heads* or *tails*. Since an event either happens or it doesn't happen, the probability of failure **Pr{not E}** may be represented by

$$q = \Pr\{\text{not } E\} = (n-h)/n = 1 - (n/h) = 1 - p = 1 - \Pr\{E\}$$

Thus p + q = 1. All probabilities are fractions which lie between 0 and 1. If an event is certain to happen, it takes on a probability of 1. Conversely, if it can never happen, it has a probability of 0.

The Question

Our "door problem" is a bit more tricky. What do you think? Are the contestant's chances equal if he stays with his original choice or are they better if he switches? What is his probability of success (winning the car)?

Many people thought that he had a 50-50 chance of winning and it did not matter if he switched or not; Marilyn maintained he had a 2/3 chance of winning if he switched! Let's not spoil the fun of ARexx by giving a rigorous proof right away, but let's proceed as if we know nothing. Here is a good opportunity to write some *pseudo code* to organize our thoughts and list the steps in our empirical (experimental) solution.

Pseudo Code to Solve the Door Problem

1. Assign a variable **pat** to count successes in staying or *standing pat.*

2. Assign a variable **switch** to count successes in *switching* doors.

3. Get screen input for number of games or number of experimental trials to calculate: **limit**. Assign a variable **prt** a value of **limit** minus 3 to be used in choosing only the *last three* trials to output to screen.

4. Loop for **i to limit**, the number of games input in step 3. Loop ends at step 16. Note: all integer *nodes* **n**, **k**, **l**, **j** range from **1** to **3**, and index a particular *door* number.

5. Initialize to value **0** an array: **door.1**, **door.2**, and **door.3**.

6. Pick a random number **n** between **1** and **3** to represent the door behind which is the prize sports car.

7. Assign the value **1** to **door.n.** Now we have the prize door flagged.

8. Get a random guess between **1** and **3** from the contestant, and assign it to a variable **j**.

9. If the car is behind **door.1**, make the host open a *losing* **door.k**, one which does *not* hide the car, *and* one which has *not* been chosen by the contestant. If the host *has a choice*, make him open a door at *random*.

10. If the car is behind **door.2**, make the host open a *losing* **door.k**, one which does *not* hide the car, *and* one which has *not* been chosen by the contestant. If the host *has a choice*, make him open a door at *random*.

11. If the car is behind **door.3**, make the host open a *losing* **door.k**, one which does *not* hide the car, *and* one which has *not* been chosen by the contestant. If the host *has a choice*, make him open a door at *random*.

12. If the contestant was **correct** in his original guess (**door.j = door.n**), increment the **pat** counter for standing pat.

13. Calculate the **cumulative success probability** of **standing pat**; assign it to the variable **stand**.

14. Calculate *directly* (*without relying* on the probability definitions), the **cumulative success probability** of **switching** to **door.l**: Make a logic table; determine success of switching; as appropriate, increment switching success counter **switch**, and assign the cumulative probability to variable **change**.

15. If **i>prt**, arrange for the program to output data and results from the last three iterations of the loop. Do loop again.
16. End of experimental loop.

17. Exit.

The *pseudo code* is more detailed this time. Notice that we have named our variables and defined loops and tests explicitly in order to clarify the process. You may make your pseudo code detailed or not as your needs dictate. In step 14, we could have calculated the success of switching by using the identity **p+q=1**, but we choose not to, in order to demonstrate the truth of the identity empirically, and more important, to introduce you to some useful logic operators in ARexx. Here is one way to code the problem in ARexx:

```
/* Door.rexx Three door game show */
pat=0
switch=0
SAY 'Input total number of trials.'
PARSE PULL limit
prt=limit-3
/* Loop for repeating experimental trials */

DO i=1 to limit
   door.=0                        /* initialize array to 0          */
   n=RANDOM(1,3)                  /* pick a door at random          */
   door.n=1                       /* flag it as a win               */
   j=RANDOM(1,3)                  /* contestant guesses at random   */
   IF door.1=1 THEN DO            /* In case car is behind door 1   */
      IF j=1 THEN k=RANDOM(2,3)   /* Host opens random loser door   */
      IF j=2 THEN k=3             /* Host must open door 3          */
      IF j=3 THEN k=2             /* Host must open door 2          */
      END
   IF door.2=1 THEN DO
      IF j=2 THEN DO
         h=RANDU()                /* How to pick either 1 or 3      */
         IF h>0.5 THEN k=3        /* at random, based upon a        */
         ELSE k=1                 /* random fraction h              */
         END
      IF j=1 THEN k=3
      IF j=3 THEN k=1
      END
   IF door.3=1 THEN DO
      IF j=3 THEN k= RANDOM(1,2) /* Similar to first block IF      */
      IF j=2 THEN k=1
      IF j=1 THEN k=2
      END
   IF door.j=1 THEN pat=pat+1     /* increment win by standing pat  */
   stand=pat/i                    /* cumulative wins by standing pat*/
   /* Calculate the losing probabilities. The situation if */
   /* the player stands pat and loses if he doesn't switch */

   IF j=1 & k=2 ^ j=2 & k=1 THEN l=3
   IF j=1 & k=3 ^ j=3 & k=1 THEN l=2
   IF j=2 & k=3 ^ j=3 & k=2 THEN l=1
   IF door.l=1 THEN switch=switch+1    /* increment win by switching  */
   change=switch/i                     /* cumulative wins by switching*/
```

```
 /* output controls write to screen */

   IF i>prt then do
      SAY,
     'trial #'i': car in door #'n'; guess #'j'; host opens door #'k'.'
      IF door.j=1 THEN SAY 'staying wins.    '
      IF door.l=1 THEN SAY 'switching to #'l 'wins. '
      SAY 'cumulative wins:'
      SAY 'switching: 'change'='switch' wins out of 'i' tries.'
      SAY 'standing:  'stand'='pat' wins out of 'i' tries.'
      SAY ''
      END
   END /* i loop.  Do another test */
EXIT 0
```

Explanation of the ARexx Code

Thanks to our pseudo code, it is pretty easy to read this program without needing a lot of explanation. The first three pseudo code steps are completed between the first and second comments in our ARexx listing. They present nothing new. The pseudo code step 4 loop begins in ARexx with the familiar DO instruction. The DO *instruction-specific information* is **i=1 to limit** which means to count up from **1** to the number of trials **limit** we specified and use the index **i** to keep track. This information is called the **iteration specifier**. Iteration specifiers cause a DO instruction to *execute repeatedly* until a **termination condition** occurs which makes the loop stop. In our problem, this termination condition is that **i** reaches the number **limit**. Step 5 of the pseudo code is to initialize an array **door.** which is accomplished in one step as we have seen in a previous example.

DO instruction
H 27 f
C 10-53 ff

Getting a Random Number

RANDOM() function
H 62
C 10-112

Step 6 introduces a new ARexx built in function called RANDOM(). The arguments for this function are integers which define a range from which to select a pseudo-random number (an integer). The function returns an integer which we assign to the variable **n** according to our pseudo code in step 6.

Now that we have a *random node* with which to identify the door with the car behind it, we assign the value of **1** to **door.n** and flag that door as the winner. Since we initialized all doors to **0** before, we have only one **door.n** in the array with a value of **1**. We let the RANDOM() function serve as an independent trial and stand for the contestant's guess (a random number **j** between 1 and 3) as well.

The Possibilities: Where is the Car?

IF instruction
H 29 f
C 10-58 f

THEN instruction/keyword
H 39 f
C 10-58 f, 10-73

DO instruction
H 27 f
C 10-53 ff

Next we come to three **IF THEN DO** blocks which correspond to steps 9 through 11 in the pseudo code. These blocks model the situation when the car is actually behind door 1, door 2 and door 3, respectively. Notice the way the game show host is *constrained to pick* his open door in two out of three cases, in each block. He must pick a door both *not* picked already by the contestant and *not* hiding the car. If the contestant picks a loser for a door, the game host has *only one choice* for his pick! For you analytical types, this should be a clue as to where the *red herring* in this little problem lies! Anyway, we will assume complete innocence and proceed.

Modelling Choices

In each IF block, if the contestant has indeed picked the winner, then the game show host has two choices and we model his choice by randomizing it. In blocks one and three he must pick between two numbers in a row: 2 and 3; or 1 and 2, respectively, so we can use the same function RANDOM() to return a number for us. However in the second IF block, what do we do when we need to choose at random between two numbers 1 and 3, *not* in sequence?

The RANDU() Function

RANDU() function

H 62
C 10-112 f

We use an ARexx function not yet introduced, the RANDU() built-in function. RANDU() returns a value at random between 0 and 1. Note that this function returns a decimal fraction and not an integer. RANDU() may take an argument called a **seed**. This is to make it choose a different first number, when required, by initializing the internal state of the ARexx pseudo-random number generator. The **seed** argument is optional and we did not need it here. For the fine points of

ELSE instruction
H 28 f
C 10-56 f

IF instruction
H 29 f
C 10-58 f

the functions RANDOM() and RANDU(), refer to *Hawes*, page 62; and *Commodore* page 10-112. Here, we simply let RANDU() choose a random fraction and after assigning its value to the variable **h**, we test **h** to see if it is greater than 0.5 and if it is, we let **k=3**; and if not, we use an ELSE clause to assign the other value **k=1**. All three IF blocks are similar except that each takes care of a different door hiding the prize.

The Cumulative Probability of Winning by Standing Pat

The fourth IF instruction takes care of step 12 of the pseudo code. All we do here is increment the counter **pat** which represents the wins when the contestant guesses correctly and stands pat rather than switching. Step 13 of the pseudo code is the next ARexx assignment. We calculate the *expression* **pat/i** and assign its value to the symbol (variable) **stand**. This is the cumulative probability of wins by standing pat. Each time **pat** wins, it is incremented and the ratio of this **pat** total to the total of all the trials so far is the **cumulative probability** of winning by *standing pat*. The variable **stand** thus represents the ultimate number we are searching for, because the number it *converges to* as the number of trials *increases*, will be the *theoretical probability* we are trying to find. It is easy to see that **stand** will lie between 0 and 1 because **pat** will always be less than or equal to **i**, *and* greater than or equal to 0.

Calculating the Probabilities of Winning by Switching

As we mentioned before, we could easily let the probability of failure in standing pat (which is the probability of winning by switching) be represented by the value of **1-stand**, but we are going to be obtuse on purpose and assume nothing in our model. Therefore, we come to the next group of three IF instructions. Simply stated they are the ARexx encoding of all the possibilities that obtain if the contestant fails to guess the prize door correctly on the first try and after the host tempts him with the opened losing door.

Boolean Operators

The first IF statement translates into, "IF either [the contestant has picked door 1 and the host has opened door 2] or (the mutually exclusive event) [the contestant has picked door 2 and the host has opened door 1] THEN the car *must* be behind door 3." ARexx uses an elegant shorthand to make complicated logical comparisons and return a boolean value. In these IF instructions, **j** is the node of the door array that the contestant guesses; **k** is the node of the door the host opens and **l** is the node of the prize door.

BOOLEAN operators

H13, 21
C 10-30, 10-43 f

BOOLEAN values

H 17
C 10-38

In the next ARexx line, we test the array element **door.l** to see if it is equal to **1**. If it is, we increment the **switch** counter to show that switching wins, and we calculate the cumulative wins by switching in the same way we did for **stand** and **pat**. We use the logical operators **&** and ^ which mean logical **and** and **exclusive or** (XOR). Since **&** has priority over ^, we do not need to use parentheses as the expressions **j=1 & k=2**, and **j=2 & k=1** are evaluated *first*, and only then is the XOR operator ^ applied to the resulting boolean values which returns a single boolean value. To learn more about operator sequences and priorities, refer to *Hawes*, page 13 and *Commodore* page 10-29 ff.

Screen Output

The final section of the code is the output to the screen. We don't want to output every trial to the screen as that would slow down the program execution considerably. We are only interested in the numbers **stand** and **change** and their values for a high number **limit** of trials. We use a DO instruction to accomplish this quickly. We do a simple test to check whether **i** exceeds the value of **prt** which has an integer value of three less than **limit**. If it exceeds **limit**, we output some *expressions* to the screen to tell us the values we are interested in.

Continuing a Line of Code

Note the use of a comma after the first keyword SAY. A comma (,) is used to **continue a line** that is too long to fit on one line, but which is part of the same instruction line. If you need to split a line that contains a comma *at a comma* (for instance between two arguments), don't forget to use the continuation comma as well. You would in that case have two commas in a row, the first one for separating the arguments and one to continue the instruction line. The rest of the listing is routine and familiar to you by now and we will not discuss it. Try entering the program and running it with 3000 as the number of trials. In a minute or two, you will have the result on your shell window. What can you conclude about the probabilities of switching as opposed to standing pat? Can you make a "real" proof instead of just a computer simulation?

The Proof of the Door Problem

The notion of **independent trials** is crucial to understanding the Door Problem. An independent trial is one that has no dependence upon any other event; in other words it can be deemed to be a random event, such as the repeated flipping of a coin. Whether the coin shows heads or tails on any one particular flip is *independent* of any other flip.

In our door problem, there are only two independent trials: the placement of the prize car behind a door and the original guess by the contestant. All other events are **dependent** upon something else: the host's opening another door is *dependent* upon where the car is *and* the original guess by the contestant. Similarly, if the contestant chooses to switch, this event is *dependent* upon his original guess *and* the door which the host has opened.

If we concentrate upon only the independent trials, we can see that there are only three ways the contestant may pick a door the first time, and there are only three ways in which the car may be hidden behind one of three doors. Therefore, there are 3 x 3 = 9 different possible outcomes in first picking one of three (hiding the car) and then picking one of three things (choosing a door). If we enumerate the possible wins when the contestant *stands pat*, we see that he can win only when his guess coincides with the door behind which the car is hidden. This can happen only 3 times out of the 9 possibilities. Since the contestant is constrained to either win or lose, the probability of winning by switching is therefore 1 - 1/3 or 2/3. *Q.E.D.*

Still Not Convinced?

What the game show host does, really doesn't affect the contestant *if he stand pat.* A later event in probability has no effect on the probability of a previous independent event, it only may affect events later than itself which depend upon it. Let's make a table of all possible outcomes of placing the car and choosing a door. The first number in the pair represents the number of the door behind which the car is in fact hidden. The second number represents the first choice (guess) of door number by the contestant who doesn't know which door the car is behind. If we assume a random placement of the car and a random guess by the contestant, by the definition of randomness, each of these ordered pairs or combinations of truth and guess occur equally often. Therefore, each combination has a probability of 1/9, since the table exhausts all possible ways of hiding a car behind one of three doors followed by a contestant's guess at one of the three doors. Remember that by the definition of probability, the probabilities of all possibilities must add up to unity (1).

Table of Possibilities

```
Car in #1;Guess #1    Car: #1;Guess #2    Car:#1;Guess #3
"stay"    wins        "stay"    loses     "stay"    loses
"switch" loses        "switch" wins       "switch" wins

Car in #2;Guess #1    Car: #2;Guess #2    Car:#2;Guess #3
"stay"    loses       "stay"    wins      "stay"    loses
"switch" wins         "switch" loses      "switch" wins

Car in #3;Guess #1    Car: #3;Guess #2    Car:#3;Guess #3
"stay"    loses       "stay"    loses     "stay"    wins
"switch" wins         "switch" wins       "switch" loses
```

Clearly, there are nine (9) possibilities. Out of these nine, only in three cases underlined--those in which the guess matches the truth -- is it successful to *stand pat* with the original guess; *switching* would in these three cases guarantee failure, *no matter which* of the possible two losing doors the host opened! In the remaining six cases out of nine (6/9=2/3) it is uniformly successful to *switch* to the correct door. *There are no other possibilities!*

Note that in the six cases *not* underlined, the host is constrained to open a *specific* door the number of which is *not* either of the numbers in the table entry; e.g. "Car in #1; Guess #3" constrains the host to open door #2, which means that *switching* will inevitably be to the correct door #1. *Success always occurs* when switching *outside* the underlined entries (6/9=2/3 of the time); and *failure always occurs* when *standing pat* outside the underlined entries. It is easy to see that winning and losing are thus *mutually exclusive* with respect to *standing pat* and *switching* within and between each of the two groups: underlined and not underlined. Therefore *switching* is successful 2/3 of the time and *standing pat* is successful only 1/3 of the time. *Q.E.D.*

It is worth reiterating that the *independent trial* of the first guess, which is equivalent to standing pat in the second choice, really is independent of later events with respect to the probabilities of its success. We also should be able to see that the host directly influences only the six out of nine entries not underlined when he opens a losing door in that he guarantees the success of the second choice if it happens to be *switch*. The contestant's second choice is clearly neither entirely independent nor random (only *standing pat* remains independent and random). The *red herring* in this simple problem comes with the temptation to "mix pears and apples and oranges and nuts": independent trials, dependent revelations, doing nothing, and dependent choices. It is also worth noting that the host's action reduces the contestant's number of choices from three to two, but he *does not change the probability of winning of the original guess*!

Summary

The above discussions and proofs are meant to give examples of the ways in which a problem may be analyzed, and then solved using ARexx logical operators. If we cannot characterize (write down) a problem, how do we expect to solve it? Programming is more than simply understanding syntax and semantics. It is an art as much as a science; perhaps more so. If we practice using our imagination with ARexx, to propose and then solve puzzles, problems, or anything else that yields to a programming solution, we will soon become fluent in ARexx and will have trained our minds to think in such a way that we make our computers work *for* us instead of the other way around. Recreational programming in ARexx is a fun and painless way to learn the language! It is *never* a waste of time as perhaps computer gaming can be.

Recursive Function Calls in ARexx

Recursive Functions

H 35
C 10-68

ParseTest.rexx program

see page 3-3

A recursion is a self-referential structure in which a function or a routine calls itself during its own execution or performs some other form of self-reference such as modifying its own code. Recursions are powerful, but they can be tricky. In programming it is generally best to avoid recursions if possible, but in some cases they are convenient and useful, as we saw in the **ParseTest.rexx** program which modifies its own code. ARexx is capable of running recursive functions that call themselves from within the function procedure itself. An example of such a recursive function (that computes factorials) is in *Hawes*, page 35 and in *Commodore*, page 10-68. A function call *outside* a procedure may itself be recursive as we will demonstrate here.

Iteration

In general, whenever you can make a function use only one "kernel" computation that starts with a "seed" estimate and feeds the result of that computation into the same computation as a new "seed", it is a candidate to become either a recursive function or a to be called recursively. When many calculations must be accomplished, this method is sometimes referred to as an **iterative** technique, and proves useful in solving all sorts of number theoretic problems and finding solutions to systems of equations that do not yield to ordinary algebraic means. An entire branch of mathematics called **Numerical Analysis** concerns itself with these iteration techniques and the theory of their operation.

Algorithms

ITERATE instruction
H 30
C 10-60

Iteration (not to be confused with the ARexx ITERATE instruction) and other mathematical **computation procedures** are called **algorithms**. You may think of an algorithm as analogous to a recipe in a cookbook.

An iteration algorithm is a *set of procedures* for doing multiple calculations evaluating one formula over and over again until some sort of criterion is met such as convergence to one value with an arbitrary degree of accuracy. The number of calculations sometimes runs into the millions. The famous Mandelbrot Set is a map of the complex number plane showing the results of multiple recursions of one simple formula called the *Mandelbrot fractal* iterated thousands or millions of times to determine *only one* criterion for each coordinate: Is it *inside* or *outside* of the Mandelbrot Set? Whether the value of the expression converges or not (as each new evaluation is fed into the expression as the new seed) determines whether the point is in or out.

Computer Proofs

With a computer, you may find solutions that are difficult or even impossible to obtain otherwise. If the solution is involved in a proof of a theorem, it is called a **computer proof**, and usually sneered at by pure mathematicians. Mathematicians also call such proofs "brute force solutions", and they may have a point. Although computer proofs are not usually elegant or even ingenious, they *do* find answers and therefore have a practical side to them. Here is a famous number theoretic brain teaser that yields its answer readily to ARexx.

The Coconut Problem

It seems that there are four shipwrecked sailors on an island with a large grove of coconut palms, and upon the coconuts they are surviving. One day a typhoon sweeps the island and although the sailors survive by hiding in caves, the grove (and their only food source) is destroyed, but there are a number of coconuts left littering the ground. The sailors collect them into a large pile, and agree to divide them equally the next day as it is now dusk.

Each sailor happens to be both dishonest and greedy. During the night, the first sailor wakes up and hatches a selfish scheme. In the moonlight, he sneaks over to the pile without waking his mates, and counts the coconuts and finds that their number can be evenly divided into four equal piles with one coconut left over which he throws to a nearby monkey. He hides one of the four piles for himself as a hedge against starving before his three mates, groups the remaining three piles into one, and sneaks back to bed and sleeps.

As you may have guessed (mathematics isn't much on plot twists), the other three sailors do exactly the same thing: They find the remaining pile may be evenly divided into four parts with one coconut left over, which they throw to the monkey. Then they hide one of the four piles for themselves and regroup the other three piles into one and go back to bed. In the morning, since all four are guilty as sin, they pretend not to notice the much diminished pile of coconuts, but *once again* they find that the pile may be evenly divided into four equal piles with one coconut left over which they throw to the monkey (who ends up with five coconuts). What is the minimum number of coconuts that must have been in the original pile last night?

An Ideal Candidate for Recursion

This tricky little problem is ideal to demonstrate recursion in ARexx, because it does the same thing five times: subtract one; divide by four; multiply by three. However, it cannot be readily solved by ordinary algebraic means, because it has only *one* equation and an infinite set of unknown quantities, one of which has the minimum value.

If **N** represents the original number, then the first sailor leaves a pile containing **3*[(N-1)/4]** coconuts. The next sailor uses the *value of this expression* as *his* N and so on. By means of ARexx we can find the answer by brute force if we start with a seed **n** equal to the integer **1** (coconuts are a code word for *integers* or *whole numbers*). We then try the kernel computation above as a recursion formula nested five deep. At the end, we test if the result is an integer or not. If it is, we stop and report that we have found **N**, the minimum number of coconuts, and if not, we increment **n** to the next integer and try again. We've just written the pseudo code, so we can start on ARexx coding right away.

```
/* Coconut.rexx The coconut problem */
```

FOREVER iteration specifier for **DO** instruction
H 27 f
C 10-53 ff

DATATYPE() function
H 56
C 10-101 f

PROCEDURE instruction
H 35
C 10-68 f

ARG instruction
H 26
C 10-52

RETURN instruction
H 37
C 10-70

```
n=1      /* Start with integer 1*/
DO FOREVER
/* THE RECURSIVE CALL */
num=fun(fun(fun(fun(fun(n)))))

/* Test for whole number */
IF DATATYPE(num,whole) THEN DO
   SAY 'The number 'n 'is valid. It is the minimum.'
   EXIT 0
   END
/* Try the next iteration */
n=n+1
END

/* THE INTERNAL FUNCTION */
fun: PROCEDURE
ARG i
i=i-1    /* Throw one to the monkey */
i=i/4    /* Divide into 4 equal piles*/
i=3*i    /* Group the other 3 piles*/
RETURN i
```

The program is very simple. We use a DO FOREVER loop with only one way to exit, when we satisfy the DATATYPE() function with a whole number **n** (an integer). This would be dangerous if we never reached a solution.

HI command utility

H 83
C 10-155

If you ever get stuck in an endless loop, to stop all ARexx activity in the system, open another shell, and at the prompt enter: **HI** (for Halt Interrupt) and all ARexx programs will stop.

DATATYPE() is very useful to check strings for UPPER or miXed case, Alphanumeric data, numbers of various sorts, valid ARexx symbols, and more. The manual reference is *Hawes*, page 56 or *Commodore*, page 10-101. This is an important function to know and a little study will pay off.

The Central Recursion

Our recursion is a five-deep nest of calls to the function **fun**, an internal procedure. The only thing different here that we have not covered before about function calls is that the **recursive call** does not assign a value to **num** until the internal function **fun** has been called five times by the nested assignment expression. Next, if the value of **num** satisfies the DATATYPE() 'whole' then we exit and SAY the message. Otherwise we increment our **n** guess and do it all over again.

See if you can succeed in making **fun** itself recursive. You will soon run into trouble, as it modifies its argument **i** each time through and any attempt to keep track of five iterations will confront you with a problem in computing called **self-modification**. Any attempt on your part to implement a loop to count from 1 to 5 within **fun** will be a part of any recursive calls to **fun** and you will soon get into deep guacamole. This is why you must be very careful with recursions, should you use them. Run this **coconut.rexx** program from a shell and in a little while you will know the answer. If you were to prove this result in a more rigorous mathematical sense, you must expect to spend a lot more time! Try it!

Dealing with Number Bases and Character Codes

The Binary Number System

For proper **ARexx** notation for **binary**, **hex** numbers, see **STRING TOKENS**

H 12
C 10-28 f

Programmers frequently must deal with numbers expressed in different number bases, or translate characters into their ASCII decimal codes. A computer, as you probably know, operates with logic based on the **binary** number system composed of only two digits (properly called **bits**): 0 and 1, to represent the logical state of electronic devices as either "high" (1) or "low" (0). In binary, instead of having numbers based upon powers of ten, they are based upon powers of two.

Expanding Numbers

In our *base ten* decimal system, a number such as 103 can be expanded as: (1x100)+(0x10)+(3x1)=103. We never think about it, but the digits of our numbers are ordered so as to fit into the *units*, *tens* and *hundreds* positions. In binary, we position bits (binary digits) by powers of two, because instead of ten digits we have only two bits, counting in binary: 1, 10, 11, 100, 101, 110, 111, 1000, etc. The binary number 101 ('101'b in ARexx notation), is expanded to (1x4)+(0x2)+(1x1)=decimal number 5. The binary number 110 ('110'b in ARexx notation), is expanded to (1x4)+(1x2)+(0x1)=decimal number 6.

It is straightforward but inconvenient to convert number bases one to another, so ARexx includes a complement of functions to do these conversions easily.

Hexadecimal Numbers

Since binary number strings can become very long, programmers frequently use the **hexadecimal** system of numbering to make binary codes more compact and readable. Instead of ten digits or two, hexadecimal is based upon 16 digits: 0 through 9, plus the letters A through F. Conversion is easy between a base two binary number and a base sixteen hexadecimal or "hex" number because the hex base 16 is also a power of two. Each four places of a binary number represents one place in a hex number: For instance, the binary number 1111 translates to the hex number 'F'x in ARexx notation (decimal 15).

Octal Numbers

Another popular representation of computer numbers is **octal** or base eight numbers, where the digits run from 0 to 7. It is also easy to convert to or from binary as 8 is a power of two (3 bits = 1 octal digit).

Character Codes

All computer languages represent letters and printable characters, as well as control characters by means of numeric codes. ARexx provides functions to convert characters into or from their decimal, binary, or hex representations.

A Useful Conversion Program

In this exercise, we will make a program to convert and display the translation of any number (in decimal, octal, hex or binary), or any character string, into all the other representations. This is a handy utility for when you are programming and need to convert number base or translate a character or string. ARexx has no built in functions to handle octal numbers, but by means of a couple of simple interior functions we can take care of this.

B2C() function
H 52
C 10-94

C2B() function
H 54
C 10-97

C2D() function
H 54
C 10-97

C2X() function
H 54
C 10-97

D2C() function
H 56
C 10-99

D2X() function
H disk
C 10-100

X2C() function
H 69
C 10-124

X2D() function
H disk
C 10-124

SELECT instruction
H 38
C 10-70 f

DATATYPE() function
H 56
C 10-101 f

ARexx and Custom Conversion Functions

The ARexx conversion functions appear alphabetically in the section on functions in both *Hawes* and *Commodore*. Some of the functions in *Hawes* were added after the manual was printed, and are in the update notes. We will list them here. The format is always *letter2letter*, as in **x2c()** which means to *convert a hex number to characters*. In the function set, **d** stands for decimal; **c** for character; **b** for binary; and **x** for hexadecimal. The ARexx set includes: **d2c()**, **d2x()**, **x2c()**, **x2d()**, **b2c()**, **c2d()**, **c2x()**, and **c2b()**.

If you think about it, you may decide that several possibilities are missing; such as **b2d()** for instance. We will demonstrate how these missing functions may be constructed easily by nesting the existing functions together. Also, since ARexx has no facilities for conversion to or from octal, we will build that in, too, and make for instance a **b2o()** and an **o2b()** function; **o** standing for octal.

We will also demonstrate the correct use of the SELECT instruction for times when there are a number of options we need the program to select from in its execution. Since the program code is very readable, we can dispense with all but the most rudimentary pseudo code:

Pseudo Code

1. Get the user input string and option to designate number base: **x** will denote hex; **o** octal; and **b** binary.

2. Check the option and determine if the number base is valid: i.e. for binary are all digits 1's or 0's? We will use ARexx's DATATYPE() function for all cases except octal where we will need to make a custom datatype check function to insure that all digits are less than 8, since octal is not an option of the ARexx DATATYPE() function.

3. Make a selection block to convert based upon the selection: Decimal, Characters, Hex, Octal, or Binary. Code in the appropriate conversion functions or make the missing ones with recursive calls, or

internal functions or both. If the string to convert is binary, then we will need to use (or make): **b2d()**, **b2x()**, **b2o()**, and **b2c()**. Each selection will have similar entries with appropriate conversions.

Here is the program, called **Hex.rexx** (for its poetic sound!).

Notes on the Hex.rexx Conversion Program

```
/* Hex.rexx Number & character translator */
DO FOREVER
SAY 'Input string [Rtn] option (x or b or o => hex, bin, oct).,
Quit=[Rtn][Rtn].'
PARSE PULL answer, option .
option=UPPER(LEFT(option,1))
IF option=='H' THEN option='X'
IF answer='' THEN EXIT 0
IF option='' THEN IF DATATYPE(answer)='NUM' THEN option='D'

IF option='X' THEN IF ~DATATYPE(answer,'x') THEN DO
   SAY 'Invalid hex number. Try again.'
   ITERATE
   END

IF option='B' THEN IF ~DATATYPE(answer,'binary') THEN DO
   SAY 'Invalid binary number. Try again.'
   ITERATE
   END

IF option='O' THEN IF ~octal(answer) THEN DO
   SAY 'Invalid octal number. Try again.'
   ITERATE
   END

SELECT
   WHEN option='D' THEN DO
      cha=d2c(answer)
      hex=d2x(answer)
      bin=c2b(d2c(answer))
      oct=b2o(c2b(d2c(answer)))
      SAY 'decimal number='answer 'and is equivalent to:'
      SAY
      SAY 'character='cha 'hexadecimal='hex 'octal='oct 'binary='bin
      SAY
      END

   WHEN option='X' THEN DO
      cha=x2c(answer)
      dec=x2d(answer)
      bin=c2b(x2c(answer))
      oct=b2o(c2b(x2c(answer)))
      SAY 'hex number='answer 'and is equivalent to:'
      SAY
      SAY 'character='cha 'decimal='dec 'octal='oct 'binary='bin
      SAY
      END

   WHEN option='B' THEN DO
      cha=b2c(answer)
      hex=c2x(b2c(answer))
      dec=c2d(b2c(answer))
      oct=b2o(answer)
      SAY 'binary number='answer 'and is equivalent to:'
      SAY
      SAY 'character='cha 'decimal='dec 'hexadecimal='hex 'octal='oct
      SAY
      END

   WHEN option='O' THEN DO
      bin=o2b(answer)
      cha=b2c(o2b(answer))
      hex=c2x(b2c(o2b(answer)))
      dec=c2d(b2c(o2b(answer)))
      SAY 'octal number='answer 'and is equivalent to:'
      SAY
      SAY 'character='cha 'decimal='dec 'hexadecimal='hex 'binary='bin
      SAY
      END
```

```
OTHERWISE DO  /* characters */
      dec=c2d(answer)
      hex=c2x(answer)
      bin=c2b(answer)
      oct=b2o(c2b(answer))
      SAY 'character string='answer 'and is equivalent to:'
      SAY
      SAY 'decimal='dec 'hexadecimal='hex 'octal='oct 'binary='bin
      SAY
      END
   END /* SELECT Block */
END /* DO FOREVER Block */

/* The Internal Functions */
/* check for valid octal number */
octal: PROCEDURE
PARSE ARG octnum
DO WHILE octnum~=''
   PARSE VAR octnum 1 first 2 octnum
   IF first>7 THEN RETURN 0
   END
RETURN 1

/* convert octal number to binary */
o2b: PROCEDURE
PARSE ARG octnum
k=1
DO WHILE octnum~=''
   PARSE VAR octnum 1 num.k 2 octnum
   k=k+1
   END
k=k-1
n=k-1
sum=0
DO i=1 TO k
   sum=(num.i)*(8**n)+sum
   n=n-1
   END
binnum=c2b(d2c(sum))
RETURN binnum
```

```
/* convert binary to octal */
b2o: PROCEDURE
PARSE ARG binnum
k=1
DO WHILE binnum~=''
   PARSE VAR binnum 1 num.k 2 binnum
   k=k+1
   END
k=k-1
finalsum=''
/*TRACE i*/  /* We will uncomment this in the next section */
DO i=k TO 1 BY -1
   sum=0
   DO n=0 TO 2
      IF DATATYPE(num.i)=CHAR THEN LEAVE
      sum=(num.i)*(2**n)+sum
      i=i-1
      END
   finalsum=sum||finalsum
   i=i+1
   END
   /*TRACE OFF */ /* We will uncomment this in the next section */
RETURN finalsum
```

First we put the program into a DO FOREVER loop, because we may need to convert several strings and don't want to exit until we are done. Exiting is done via a [Rtn] and the answer tested for the null string ''.

FOREVER iteration specifier for **DO** instruction

H 27 f
C 10-53 ff

If you make two input lines and parse the option separately using the ARexx capability of parsing multiple templates, then you can use embedded blanks and all characters for input. Note the comma in the parse template between answer and option. The comma is the special character that tells ARexx to parse two lines of input from the user. This multiple template assigns the entire first line of input to the string variable **answer**, and forces tokenization of the option. The comma at the end of the first line is the continuation character which is used to continue an ARexx instruction line that is too long to fit on the page.

MULTIPLE TEMPLATES in **PARSE** instruction

H 80 f
C 10-153

IF instruction
H 29 f
C 10-58 f

UPPER() function
H 67
C 10-121

LEFT() function
H 60
C 10-108

SELECT instruction
H 38
C 10-70 f

WHEN instruction
H 41
C 10-73

OTHERWISE instruction
H 32
C 10-63

DATATYPE() function
H 56
C 10-101 f

PROCEDURE instruction
H 35
C 10-68 f

END instruction
H 29
C 10-57

RETURN instruction
H 37
C 10-70

OTHERWISE instruction
H 32
C 10-63

After we parse the option, we reassign only the first letter of option in UPPER CASE to option, and next reassign option to the value of 'X' if the user forgot and entered hex or h instead of X as the option. This is a small example of how you can make ARexx user friendly.

The next two IF clauses check for valid hex and binary numbers, respectively. Notice how they use the **not** operator ~ attached to the function. The final IF block before the SELECT instruction uses a custom internal function to accomplish the same thing as the DATATYPE() tests above, using a function procedure called **octal()**.

The **octal: PROCEDURE** comes after the main program, the END of the DO FOREVER loop. It first parses the argument sent to it (**answer**) into a variable called **octnum**, which is protected. Then we do a loop: DO WHILE **octnum** is not the null string. The loop parses the variable one digit at a time and checks to see that it is not greater than 7, illegal in octal base eight. After it checks, it either finishes and RETURNs a 1 for true, or it sends back a 0 for false. The call to octal is in an IF clause, so it is looking for a boolean return, which we supply.

The SELECT block is a new structure that we have not yet discussed. Use SELECT when there are a number of possibilities to select from. We have an ideal use of SELECT here, because our string is *one* of five possibilities and cannot be two at once. Note the syntax of the SELECT block. It starts with the keyword on a line by itself and each possibility is accounted for in a WHEN *condition* THEN block constructed just like any other block of instructions, terminated by an END clause. Each specific option is enumerated by the WHEN clauses. There is an OTHERWISE clause at the last. *This clause is mandatory*, not optional, so don't leave it out of your own code! If you've covered all your possibilities in the WHEN blocks simple insert a NOP (no operation) instruction in the OTHERWISE clause and you're home free.

In our example, however, we have a use for the OTHERWISE block: to take care of the cases where we want to translate characters, and not numbers. If the input string was a valid decimal number, the option became 'D', so the only other time the option will be null will be when we

WHEN instruction

H 41
C 10-73

translate a character. The WHEN blocks are fairly straightforward, but notice the times we have nested functions of functions of functions. This is an example of the compactness and power of ARexx. In the first WHEN block, for instance, there is no ARexx function for converting decimal into binary, but we can nest two functions together as in the line:

```
bin=c2b(d2c(answer))
```

C2B() function

H 54
C 10-97

D2C() function

H 56
C 10-99

which first converts **answer** from decimal into character representation **d2c()**, and then its result, a character string, is converted by the outer function **c2b()** into binary number as we desire. Nested functions always work from the innermost parentheses outward. The next clause is nested *three* deep. The outermost function is **b2o()**, another necessary custom function which converts from binary into octal numbers.

Translating Binary to Octal

ARRAYS see **STEMS** and **COMPOUND** symbols

H 21 f
C 10-44 ff

Look at the last PROCEDURE in the program, **b2o:** It receives the argument from the caller and parses it into a variable called **binnum**. Then the DO loop parses **binnum** into an array called **num.** which ends up with k-1 elements. How do you convert binary to octal? First of all it takes a binary number three places long to describe only one digit of an octal number. Why? Because the eight octal digits run from 0 to 7, so the largest single digit of octal, 7, is the binary number 111, which you'll recall is (1x4)+(1x2)+(1x1)=7. For each *place* in an octal number, we therefore need *three* corresponding binary digits to represent it. Since the octal base 8 matches the powers of two in binary *every three binary digits*, then all we have to do is count every three binary digits from right to left and place that group's *decimal* representation into the placeholder of the octal number.

It's OK to use the decimal representation, because at most, three binary digits will equal 7. We simply need to count three binary digits from right to left, calculate the decimal representation of the three-place binary number, place that number as the least octal digit; and begin counting three more binary digits to the left of the first three; convert them; and place the result into the next place to the left in our octal string; and so on until we run out of binary digits.

For example, we start with the binary string 010110111. The first three digits starting on the right, 111 convert via **b2d()** to decimal 7, so 7 is the least digit of the octal translation. The next three binary digits are 110 and they equate to decimal 6, so our octal number is now at 67. The final step converts 010 into decimal 2. The translation of binary 010110111 is therefore octal 267.

Nested function:
b2d(*n*) =c2d(b2c(*n*))

C2D() function
H 54
C 10-97

B2C() function
H 52
C 10-94

We accomplish this algorithm in ARexx by means of two nested loops, one to count the binary digits *backwards* (since the binary digit array was built from left to right instead of from right to left); and the nested loop to evaluate every three binary digits. We don't really need to apply **b2d()** to these digits as its easier to calculate the result (never more than 7) directly. Otherwise we'd need to do something to get the binary digits the other way around again, in order to apply the function **b2d()**.

DATATYPE() function
H 56
C 10-101 f
(used for the test in the escape clause.)

Notice that we actually decrement the outer loop counter **i** inside the inner loop, too. The inner loop contains an escape clause in the case that the array element **num.i** is not a number. This is because the length of the binary number may not be an exact multiple of three and at the leftmost group to evaluate in the binary string, the inner loop may try to decrement past the end of the binary string.

LEAVE instruction
H 31
C 10-60 f
(used to get out of loop.)

RETURN instruction

H 37
C 10-70

Finally we build up the output variable **finalsum** into an octal number string using concatenation. We must increment **i** before we do another iteration of the outer loop because **i** was decremented once too often when we finished the inner loop. The function RETURNs **finalsum** to the caller.

Translating Octal to Binary

The other WHEN blocks operate in a similar manner. We find one more custom function to make in the **o2b()** function. How do we turn an octal number into binary? To use the ready made ARexx functions as much as possible, we need to turn the octal number into a decimal called

C2B() function
H 54
C 10-97

D2C() function
H 56
C 10-99

PROCEDURE instruction
H 35
C 10-68 f

sum, and then use a nested pair of ARexx conversion functions to transform sum first into a character with **d2c()**, and then from a character into binary using **c2b()**. By means of a simple loop in the **o2b: PROCEDURE**, we calculate the decimal number by computing the powers of 8 multiplied by the octal digits in each position, and then summing the results.

We now have a handy tool for looking up the equivalents for various numbers and strings. Note: In the ARexx manual, there is no mention of the limits of some of these functions, but the length of the string *is* limited and if you enter too long of a string, you will get some ARexx error messages. The intent of the conversion functions is to transform short strings and numbers. For instance, you may legally enter 12 decimal digits but only four characters to convert. Longer strings will produce an error message. Later, we will look at the ARexx facilities to debug and trace your programs, and we will look at sections of **Hex.rexx** as they are traced.

Notes

Chapter 5: Sorting and Working with Arrays and Lists

Sorting With ARexx

The subject of sorting and searching could and does fill volumes, so we will not go deeply into the general subject here; but everyone should have a good sort routine in their bag of tricks, so we will look at a particularly good one in some detail and learn a few useful ARexx and pseudo-coding methods as well. The sort routine we demonstrate here is called the Shell Sort after its inventor, Donald L. Shell. It is complicated, but very fast and efficient, and has the best performance when the list is mostly sorted to start with; so it is handy for sorting a list after you make a few more entries to it.

The Shell Sort is a modified **bubble sort**. It uses the bubble sort to sort and merge together many interleaved smaller lists. At the last, Shell Sort merges the smaller sorted lists together into one list. Its speed is much faster than a bubble sort alone. Before we can understand the Shell Sort, we need to take a look at the so-called bubble sort.

The Bubble Sort

For the sake of example, let us assume that you have a list of seven numbers to sort: 11, 33, 20, 44, 22, 60, and 31. We will make some commented pseudo-code to express the way to sort these numbers using the bubble sort:

ARRAYS see **STEMS** and **COMPOUND** symbols

H 21 f
C 10-44 ff

1. Set up an array **list.** with nodes **i** or **j**, where **i** or **j** stands for the place in the array that the number occupies: e.g. **List.1**=11; **list.2**=33; **list.3**=20; etc., is the *initial assignment*. As we sort the list, the value of any specific **list.** element, **list.i**, will change. Think of **list.** as a set of pigeon holes in a post office which we fill with the values of our numbers by assigning variables as we did above in the *initial* assignment.

2. a) Start with **i**=1 and **j**=2. In a loop, continue until all pairs **i** and **j** in the array **list.** have been compared as prescribed in the following steps. In the example, **i** runs from 1 to 6 and **j** runs from 2 to 7, so in an *outer loop*, we will compare *six pairs* of numbers in all: 1 and 2; 2 and 3; etc.
b) Compare **list.i** with **list.j** (11 with 33 initially).

3. If **list.i** is greater than **list.j** then do step 4. Otherwise increment **i** and **j** (set **i**=2 and **j**=3 in the second pass), end the iteration of the outer loop, and go back to step 2 b).

4. Store the value of **list.j** in a temporary variable (we'll call it **store**). Then assign the value of **list.i** to the variable **list.j** (**list.j**=**list.i**). Now We have now created a *bubble* at position **i** since we have reassigned the value originally in pigeon hole **i** to **list.j**, and the original **list.j**, we safely tucked away temporarily in the variable **store**. The *position of the bubble* will change dynamically, so we will call the **list.** array node which denotes the bubble position, **bubpos** instead of **i** or **j**, to distinguish it: **bubpos**=**i** at the moment.

5. Calculate a *new* **bubpos** as *old* **bubpos-1**. In other words, we are concerned with the node (position) *one before* the **list.i** (**i**=old **bubpos**) element in Step 4. Why? We need to check and see whether the variable in **store** is less than the variable **list.bubpos** (the value at the *new* **bubpos**). This number **list.bubpos** was placed by the *previous* loop iteration, so we need to look at it now to coordinate with the present iteration's results. If **store** is less than **list.bubpos**, then we must insure that the variable in **store** is put into its proper place in the array **list.**! The bubble serves as a *placeholder* to determine where (in which node) to put the number in **store**. We do this with an inner, nested loop which moves the bubble's position, counting backwards from (new) **bubpos** TO **1**, BY **-1** because we don't know exactly where **store** will fit in. It may even be the *least* of all the numbers we are sorting, and we would need to count (move the bubble) all the way back to 1 before we could place (assign) **store**. We must further control the loop execution with a **WHILE store<list.bubpos** conditional test, because we want to *stop* once we find the proper bubble position for **store**. To summarize: We will move the bubble's position to receive the variable **store**'s value. We

therefore want to do a loop, counting down from **bubpos** (*the old bubpos-1*) to 1 BY -1, but *only* WHILE **store** is *less than* **list.bubpos** (note that **bubpos** changes dynamically as it is the loop counter or index; and therefore so does **list.bubpos**). In the body of this inner loop, we want to make a variable called **nextnode** which is our new **bubpos+1**. Then we assign the value of **list.bubpos** to the position of **list.nextnode**, effectively *moving the bubble* to the position of **list.bubpos**. Note we have not created a new bubble, we have just floated the current bubble to another position; one closer to the top of our list, hence the name *bubble* sort. The inner loop repeats until **store** is no longer less than **list.bubpos** (which changes with every iteration of the inner loop), or our counter **bubpos** reaches 1. The bubble follows according to the counter **bubpos** which is also the node position. At the end of every iteration of the inner loop, the bubble position has moved up one position in the list, and the number formerly occupying its position has moved down one position in the list. After we do all the iterations of the inner loop which satisfy the conditions, we reach the end of the inner loop.

6. We pop out of the inner loop, back to a continuation of the outer loop. Now add 1 to **bubpos** to get back to the correct bubble position. As we left the loop above, we *decremented* the counter (**bubpos**) once too often. Even if we did not *actually do* the loop because of not satisfying the WHILE statement, the value of **bubpos** still was assigned a value of *old* **bubpos-1**. So whether we went through the inner loop or not, we still need to add 1 to **bubpos**. Finally we assign the value of **store** to **list.bubpos**. (**list.bubpos=store**). The number in temporary storage is now placed correctly in relation to *this iteration* of the outer loop, but it may get moved down by the sort in a later iteration if a smaller number in the list is found.

7. This is now the end of the outer loop. We return to step 2 b) and compare a new pair of numbers in the sequence: 1 and 2; 2 and 3; 3 and 4; etc. until all pairs are used up. At the finish, the **list.i** for **i=1** to **listlength** will be sorted. **Listlength** is the number of things to be sorted in our original list.

A Pseudo-Trace of our Example Using the Bubble Sort

Pseudo Code

The above pseudo code, even though extensively commented, may not be clear to you yet, so we will look at a step-by-step trace of the results of a bubble sort on our example numbers, and it should become more apparent how it works. Note the two methods open to you: a narrative and detailed pseudo-code in which you may describe what you wish to do; or a pseudo-trace of how the results should look at any one moment of program execution. Both methods can help you to clarify or understand complex routines. Although it may seem trivial and obvious, it will astonish you how much your coding can benefit if you try to write down a procedure in ordinary words. Words, after all are just one more form of code! One code can be mapped on to another, so if you start with a code familiar to you (words), you can facilitate your coding into a language less familiar to you. In the pseudo-trace, only the numbers that are relevant at the time are shown. Other numbers are represented by "--".

```
The Bubble Sort
7 entries to sort...
Outer DO loop for i = node = 1 to number of pairs.
number of pairs=6

  1  2  3  4  5  6  7   * * *  nodes of array list.
 11 33 20 44 22 60 31   * * *  example list of numbers

Outer DO loop. i=node=1
 11 33 -- -- -- -- -- >node=1 Step 2: i=1; j=2. Compare: list.1>list.2?  NO:
                              Do step 2 again. End of outer loop iteration.

Outer DO loop. i=node=2
 -- 33 20 -- -- -- -- >node=2 Step 2: i=2; j=3. Compare: list.2>list.3? YES;
                              Do step 4: store=list.3=20

 11 () 33 -- -- -- --  node=2 Step 4: list.3=list.2 = 33. bubpos=2. Step 5:
                              bubpos=2-1=1. WHILE store(20)<list.1=11? NO:
                              skip inner loop.

 -- 20 33 -- -- -- --  node=2 Step 6: bubpos=bubpos+1=1+1=2.
                              List.2=store=20. End of outer loop iteration.

Outer DO loop. i=node=3
 -- -- 33 44 -- -- -- >node=3 Step 2: i=3; j=4. Compare: list.3>list.4?  NO:
                              Do step 2 again. End of outer loop iteration.
```

```
Outer DO loop. i=node=4
 -- -- -- 44 22 -- -- >node=4 Step 2: i=4; j=5. Compare: list.4>list.5? YES;
                              Do step 4: store=list.5=22

 -- -- 33 () 44 -- --  node=4 Step 4: list.5=list.4=44. bubpos=4. Step 5:
                              (new)bubpos=4-1=3. WHILE store(22)<list.3=33?
                              YES: DO inner loop...

                              Step 5: DO bubpos(3) = node(4)-1 TO 1 BY -1
                              WHILE store(22)<list.bubpos = list.3 = 33.
 -- -- 33 -- -- -- --  node=4 Step 5: nextnode = bubpos+1 = 3+1 = 4.
 -- 20 () 33 -- -- --  node=4 Step 5: list.4 = list.3 = 33. Step 5: DO
                              bubpos-1=3-1=2 WHILE store(22)<list.2=20?  NO:
                              At end of inner bubble loop, bubpos=2.  End of
                              all inner loop iterations.

 -- 20 22 33 -- -- --  node=4 Step 6: bubpos = 2+1=3. list.3 = store = 22.

  1  2  3  4  5  6  7  * * *  nodes of array list.
 11 20 22 33 44 60 31  * * *  current positions: list of numbers

Outer DO loop. i=node=5
 -- -- -- -- 44 60 -- >node=5 Step 2: i=5; j=6. Compare: list.5>list.6?  NO:
                              Do step 2 again. End of outer loop iteration.

Outer DO loop. i=node=6 (final pass)
 -- -- -- -- -- 60 31 >node=6 Step 2: i=5; j=6. Compare: list.5>list.6? YES;
                              Do step 4: store=list.6=31

 -- -- -- -- 44 () 60  node=6 Step 4: list.6=list.5=60. bubpos=6. Step 5:
                              (new)bubpos=6-1=5. WHILE store(31)<list.5=44?
                              YES: DO inner loop...

Step 5: DO bubpos(5) = node(6)-1 TO 1 BY -1
WHILE store(31)<list.bubpos = list.5 = 44.

 -- -- -- -- 44 -- --  node=6 Step 5: nextnode = bubpos+1 = 5+1 = 6.
 -- -- -- 33 () 44 --  node=6 Step 5: list.6 = list.5 = 44. Step 5: DO
                              bubpos-1=5-1=4 WHILE store(31)<list.4=33?
                              YES: DO inner loop...
                              Step 5: DO bubpos(4) WHILE
                              store(31)<list.bubpos = list.4 = 33.

 -- -- -- 33 -- -- --  node=6 Step 5: nextnode = bubpos+1 = 4+1 = 5.
 -- -- 22 () 33 -- --  node=6 Step 5: list.5 = list.4 = 33. Step 5: DO
                              bubpos-1=4-1=3 WHILE store(31)<list.3=22?  NO:
                              At end of inner bubble loop, bubpos=3.  End of
                              all inner loop iterations.


 -- -- 22 31 33 44 --  node=6 Step 6: bubpos = 3+1=4. list.4 = store = 31.
                              End of outer DO loop.

 11 20 22 31 33 44 60         FINAL SORTED LIST!
```

Coding the Bubble Sort into ARexx

With all the pseudo-coding we have done, it is relatively easy to write the ARexx code for the Bubble Sort. Enter the following and call it **BubbleSort.rexx**:

```
/* Bubblesort.rexx */
/*Input the file for sorting */
PARSE UPPER ARG infile' 'outfile
IF infile = '' THEN DO
   SAY 'Input sort filename and path: '
   PARSE PULL infile   END

IF ~OPEN('sortfile',infile,'READ') THEN DO
   SAY 'File not opened. Separate arguments with space no comma.'
   EXIT 20
   END

k=1
DO WHILE ~EOF('sortfile')
   list.k=READLN('sortfile')
   k=k+1
   END
listlength = k-2
SAY listlength 'entries to sort...'

/* The Bubble Sort */
CALL TIME('R')

DO node = 1 TO listlength-1  /* The Bubble Sort outer loop */
   nextnode = node + 1
   IF list.node > list.nextnode THEN
      DO
      store = list.nextnode
      list.nextnode = list.node

      /* The Bubble Sort inner loop */
      DO bubpos = node-1 TO 1 BY -1 WHILE (store < list.bubpos)
         nextnode = bubpos + 1
         list.nextnode = list.bubpos
         END bubpos

      /* Continue in the outer loop */
      bubpos = bubpos + 1
      list.bubpos = store
      END

   END node

SAY 'Elapsed time='TIME('E')' seconds.'
```

```
/* output results */
IF outfile = '' THEN DO
   SAY 'Specify filename and path for sorted output.'
   PARSE PULL outfile
   IF outfile='' THEN DO
      SAY 'No output written.'
      EXIT 5
      END
   END
CALL OPEN('outfile',outfile,'WRITE')
DO i=1 TO listlength
   CALL WRITELN('outfile',list.i)
   END iSAY 'Output has been written to 'outfile
EXIT 0
```

Input and Output

READLN() function

H 63
C 10-113

WRITELN() function

H 69
C 10-124

TIME() function

H 66
C 10-119 f

The code includes some coding for input and output at the beginning and the end to facilitate your use. There is nothing new in these blocks, except that you should note the way the output block is the *inverse* of the input block and reconstructs the array in an output file of your choice. Note the use of the WRITELN() function is the same as the READLN() function, and also the way we allow for various possibilities or errors in the output. The input and output are coded so that you have the option of including your input and output files on the command line as *arguments* to the program call; or if you do not specify them, the program prompts you to enter the names of the input and output files. The WRITELN() function is documented in *Hawes* on page 69 and in *Commodore* on page 10-124. Some instructions to keep track of the elapsed time are included in order to measure the speed of this sort routine. The use of the built-in ARexx function TIME() is found in *Hawes*, page 66, and *Commodore*, page 10-119 f.

How the Shell Sort Modifies the Bubble Sort

WHILE iteration specifier for **DO** instruction

H 27 f
C 10-53 ff

We stated before that if the list is mostly sorted, then the Shell Sort is most efficient. This characteristic it inherits from the bubble sort. It is relatively easy to see that if the list is mostly sorted, then the bubble sort will often do an early exit (because of the WHILE condition) from the inner nested loop, saving a great deal of time compared with the time it takes a "brute force" sort blindly to compare *every* number with *every other* number. The Shell Sort takes advantage of this fact by creating, in a special way, many shorter sublists which sort much faster than one big list.

The Shell Sort then feeds these sublists to the bubble sort, *nested within* it, and at each iteration of the Shell Sort outer loop, merges the results together dynamically. At every step, the overall list is improved into an *almost sorted* list, so by the end, when Shell Sort sorts only one list, the work is mostly done. Let's take a look at how to define the Shell Sort.

The key to the Shell Sort is that it arranges any list into a certain number of smaller sublists by systematically skipping over a prescribed number of entries in the **list.** array at each iteration. These sublists are then sorted with the bubble sort routine nested within the Shell Sort outer loop. The next set of sublists is chosen in a way that merges the previous set of sorted sublists together, therefore improving the overall sort order as well.

Here is how Shell Sort constructs the sublists. First, we must define some variables: **listlength** is the number of items to sort in our big list. In the above example, **listlength=7**. The number of pairs to compare (with the nested bubble sort) in the sublists of Shell Sort in any one of its iterations is denoted by **numpairs**. The prescribed number of entries to skip over we will call **span**-1. **Span** is therefore the number to add to or subtract from any particular **node** in order to get to the next or previous **node** in the sublist. There is only one new variable here, **span**, and its construction is at the heart of the Shell Sort. Again, we turn first to some pseudo-code:

1. Choose **span** so that it is *both* a **power of 2** and *greater than or equal to* **listlength**. In our example, the least power of 2 greater than or equal to 7 is 8. Make a short DO loop to construct **span**. Start with span=1. Then WHILE **span** is *less than* the **listlength**, set **span** equal to **span** multiplied by 2. The progression will go: 1, 2, 4, 8, 16,... and stop once **span** is *greater than* **listlength**, which is what we wanted.

2. The outer Shell Sort loop. DO WHILE **span** > 1. (See the following.)

3. Divide **span** in half using *integer* division because we want a *counting number* as the result. In our example, therefore, span starts out as 8. In this step of the Shell Sort outer loop, **span** becomes 4 on the first iteration; 2 on the second pass; and finally 1 on the last iteration. Because of the nature of **span** (a power of 2), we always end up with **span**=1 during the last iteration, where Shell Sort collapses to an ordinary bubble sort. We will soon see that this is the reason Shell Sort sorts only one list at the end.

4. Assign the number of pairs **numpairs** = **listlength** - **span**. In the example, **numpairs** = 7 - 4 = 3 for the first iteration of the Shell Sort outer loop. In the Bubble Sort coded above, the counter **node** ran up to **listlength**-1. Notice how Shell Sort uniformly substitutes **span** instead of 1 in its modification of the Bubble Sort.

5. Now we do a bubble sort on these **numpairs** pairs of numbers. Instead of the outer bubble sort counter exhausting *all pairs* of numbers in the original list, we modify the bubble sort counter (**node**) to run only from 1 to **numpairs**. In the first iteration of the Shell Sort using our example data, **node** runs from 1 to 3 only; comparing 3 pairs of numbers in three sublists. The second time through the Shell Sort outer loop, it compares 5 pairs of numbers contained in 2 sublists. The final iteration compares six pairs of numbers in one list.

6. Modify the **nextnode** to become **node** + **span** instead of **node** + 1. In other words, we want to ignore the (**span**) numbers in between.

7. At the inner loop, we substitute **-span** for **-1** in the conditions, because we are counting by **span** and not by 1. We still count down **TO 1**, however, as that is a *limit* and does not represent a number of entries to *skip*. Inside the inner bubble loop, we do a similar thing: there, **nextnode** = **bubpos** + **span** instead of **bubpos** + 1; again effectively skipping a prescribed number of nodes (**span**-1) in between. We are merely incrementing and decrementing by **span** instead of by 1.

8. Similarly, in the continuation of the outer bubble loop, we set the corrected **bubpos** = **bubpos** + **span** instead of **bubpos** + 1, to bump the counter back to the correct position for the bubble.

Now we can use the above modifications to re-code our Bubble Sort into a *bonafide* Shell Sort. Enter and save the following lines as the **ShellSort.rexx** program. Keep your copy of the Bubble Sort, too. You can compare the elapsed time indications of each sort on the same data to convince yourself that Shell Sort is more efficient!

```
/* ShellSort.rexx */
/*Input the file for sorting */
PARSE UPPER ARG infile' 'outfile

IF infile = '' THEN DO
   SAY 'Input sort filename and path: '
   PARSE PULL infile
   END

IF ~OPEN('sortfile',infile,'READ') THEN DO
   SAY 'File not opened. Separate arguments with space no comma.'
   EXIT 20
   END

k=1
DO WHILE ~EOF('sortfile')
   list.k=READLN('sortfile')
   k=k+1
   END
listlength = k-2
SAY listlength 'entries to sort...'
/* The Shell Sort */
CALL TIME('R')

span = 1
DO WHILE (span < listlength); span = span * 2; END
DO WHILE (span > 1)
   span = span % 2
   numpairs = listlength - span
   DO node = 1 TO numpairs
      nextnode = node + span
      IF list.node > list.nextnode THEN
         DO
         store = list.nextnode
         list.nextnode = list.node
        DO bubpos = node-span TO 1 BY -span WHILE (store < list.bubpos)
            nextnode = bubpos + span
            list.nextnode = list.bubpos
            END bubpos
```

```
          bubpos = bubpos + span
          list.bubpos = store
          END

        END node
END  /* DO WHILE (span<listlength...*/

SAY 'Elapsed time='TIME('E')' seconds.'


/* output results */
IF outfile = '' THEN DO
   SAY 'Specify filename and path for sorted output.'
   PARSE PULL outfile
   IF outfile='' THEN DO
      SAY 'No output written.'
      EXIT 5
      END
   END
CALL OPEN('outfile',outfile,'WRITE')
DO i=1 TO listlength
   CALL WRITELN('outfile',list.i)
   END i
SAY 'Output has been written to 'outfile
EXIT 0
```

TIME() function
H 66
C 10-119 f

DO instruction
H 27 f
C 10-53 ff

WHILE iteration specifier for **DO** instruction
H 27 f
C 10-53 ff

We can incorporate this nifty sort routine as a stand alone program, or (as we will see later) as a function to be called by another program, and even lift out and use its central chunk of code intact as a built-in sort for a specific program. The first sections should be familiar. In them we read in a file and put it into an array called **list.** which we then sort beginning with the section commented as **/* The Shell Sort */**. In the first lines of the sort routine, we reset the time counter with CALL TIME('R'), and then we meet the variables **listlength** which is **k-2**, the count of how many items in our list. The variable called **span**, initially set to 1 and the first DO WHILE loop (a set of three instructions on one line separated by **;** construct **span**. The one-line syntax for the short loop is handy for when we don't want to indent a block of instructions.

Integer division, see **ARITHMETIC** Operators
H 18 ff
C 10-39 ff

Then we enter the Shell Sort outer loop, starting with **DO WHILE (span>1)**, to be executed as long as **span** is greater than 1. Next comes integer division, the syntax **%** indicating that we only want the integer part of **span** divided by 2 assigned to **span** and not the fractional part, if any. There aren't ever any remainders, but integer division avoids any system overhead dealing with precision.

Each element in the array **list.** at any one time through the loop is in *exactly one sublist*. The sublist's entries are **span** entries apart. The Shell Sort effectively sorts many short lists and then merges them by interleaving the elements at each pass through the loop. The easiest way to see how Shell Sort works is to refer to a trace of the results which shows the evolution of these lists when we have an array of numbers to sort. We use a specially written program using the SAY instruction and some loops to output the array at intermediate stages of the Shell Sort. The program listing is included in the optional companion disk under the name **Sortout.rexx**.

Let's look at an example. The sample list to sort is arranged so that at the last iteration of Shell Sort, when **span**=1, the last list to be sorted is the same one used in the Bubble Sort example above. This demonstrates the fact that Shell Sort collapses to an ordinary Bubble Sort at its last iteration. The format of the trace is similar to the pseudo-trace of the Bubble Sort, but not as detailed. If you have studied the pseudo-trace, however, you will have no difficulty seeing what is happening. Only the relevant list items are shown on each line as before. Note how seldom the Shell Sort needs to enter the inner Bubble Sort loop!

```
 7 entries to sort...
 1  2  3  4  5  6  7   * * *  nodes
11 33 22 44 20 60 31   * * *  list

span=4 numpairs=3
 1  2  3  4  5  6  7   * * *  nodes
11 33 22 44 20 60 31   * * *  list
11 -- -- -- 20 -- -- >node=1 no change
-- 33 -- -- -- 60 -- >node=2 no change
-- -- 22 -- -- -- 31 >node=3 no change

span=2 numpairs=5
 1  2  3  4  5  6  7   * * *  nodes
11 33 22 44 20 60 31   * * *  list
11 -- 22 -- -- -- -- >node=1 no change
-- 33 -- 44 -- -- -- >node=2 no change
-- -- 22 -- 20 -- -- >node=3 20->store
-- -- () -- 22 -- --  node=3 22->list.5
-- -- 20 -- 22 -- --  node=3 store->(20)
-- -- -- 44 -- 60 -- >node=4 no change
-- -- -- -- 22 -- 31 >node=5 no change
```

Now the Shell Sort has bubble-sorted five short lists. The final pass through the loop does a straight bubble sort on a single list of seven numbers, comparing six pairs of numbers.

```
span=1 numpairs=6
 1  2  3  4  5  6  7  * * *  nodes
11 33 20 44 22 60 31  * * *  list
11 33 -- -- -- -- -- >node=1 no change
-- 33 20 -- -- -- -- >node=2 20->store
-- () 33 -- -- -- --  node=2 33->list.3
-- 20 33 -- -- -- --  node=2 store->(20)
-- -- 33 44 -- -- -- >node=3 no change
-- -- -- 44 22 -- -- >node=4 22->store
-- -- -- () 44 -- --  node=4 44->list.5

DO bubpos(3)=node(4)-span(1)TO 1 BY-span(-1)
WHILE store(22)<list.bubpos(33) (new)nextnode=4
-- -- 33 -- -- -- --  node=4 store=22<33
-- -- () 33 -- -- --  node=4 33->list.4
end of bubble loop. (22)->(new)bubpos 3

-- -- 22 33 -- -- --  node=4 store->(22)
-- -- -- -- 44 60 -- >node=5 no change
-- -- -- -- -- 60 31 >node=6 31->store
-- -- -- -- -- () 60  node=6 60->list.7

DO bubpos(5)=node(6)-span(1)TO 1 BY-span(-1)
WHILE store(31)<list.bubpos(44) (new)nextnode=6
-- -- -- -- 44 -- --  node=6 store=31<44
-- -- -- -- () 44 --  node=6 44->list.6

DO bubpos(4)=node(6)-span(1)TO 1 BY-span(-1)
WHILE store(31)<list.bubpos(33) (new)nextnode=5
-- -- -- 33 -- -- --  node=6 store=31<33
-- -- -- () 33 -- --  node=6 33->list.5
end of bubble loop. (31)->(new)bubpos 4

-- -- -- 31 33 44 --  node=6 store->(31)

11 20 22 31 33 44 60  final sorted list.

Elapsed time=1.64 seconds.
```

Extracting a Word List without Duplicates from Text

The Proper Handling of Arrays (Compound Symbol Tokens)

The next exercise will demonstrate how to extract a list (without duplicates) of all words from a text file. In the introduction, we mentioned that it would be useful to create an index of words from a text file for possible use in a document if our word processor did not have this capability. With ARexx, using its powerful array operations, this is very easy to do. We will also meet an example of using something other than *numbers* as *nodes* in an array.

ARRAYS see **STEMS** and C**OMPOUND** symbols
H 21 f
C 10-44 ff

PROCEDURE instruction
H 35
C 10-68 f

Before we go further, however, we need to note a few important principles regarding how to manipulate arrays properly (ARexx calls arrays **Compound Symbol Tokens**, and we will use these terms interchangeably). We have already looked at an example of stripping the individual words from a line. We introduced an internal function labeled **Stripword:** which is a PROCEDURE instruction to do it. Everything about stripping off the individual words and outputting the results we did *inside* this internal function, however; and no variables were returned at the completion of the function procedure. Now we will explore the correct way to pass variables and even entire arrays to and from internal and external ARexx functions of our own design.

The Differences Between Interior and Exterior Functions

Receiving an Expression Result

RETURN instruction
H 37
C 10-70

EXIT instruction
H 29
C 10-57 f

EXPRESSIONS
H 16
C 10-35

The most important things to note are the limits upon what ARexx is capable of passing to and receiving from an interior or exterior function. ARexx lets RETURN or EXIT (the final instructions in interior and exterior functions, respectively) send back *only one thing*: A *single string or expression*. It may be a long and complicated string, to be sure, but it is still only a *single variable*; and neither a specific entire array; nor explicit multiple variables.

Interior Functions

Interior functions are the more versatile in that you may pass an entire

PROCEDURE instruction
H 35
C 10-68 f

RETURN instruction
H 37
C 10-70

array to an interior function by using the EXPOSE subkeyword in the PROCEDURE instruction. You may also EXPOSE specific variables which you wish to modify with the interior function, and after the RETURN they will reflect these modifications, but that is because you have *exposed them globally*, not because they were contained in the RETURN instruction.

Exterior Functions

EXIT instruction
H 29
C 10-57 f

Entire arrays *may not* be passed except explicitly, element by element, to an *exterior* function. Multiple arguments may be passed to either interior or exterior functions, but there is no facility to *expose* any variables to an exterior function, and you must deal with a single string or expression upon its EXIT.

Overcoming Limitations by Encoding an Array into a String

RESULT special variable, see **CALL** instruction

H 26
C 10-53

We will look at several ways to work around these limitations by implicitly encoding all the required data (such as the data in an entire array or the values of several variables) in the RESULT variable; to be decoded back into an array in the main program which called the function. We will also look at some examples of exposing arrays and variables.

PARSING, see **Chapter 3**

If you studied the section on PARSING, you may have guessed that this is the tool we will use to decode our RESULT strings! To encode an array into a single string for later parsing, we use a loop. Building up and tearing down arrays in ARexx is something that you must learn to do if you are to do very much with text of any kind whether it is a document file or a set of commands to pass to an outside host application. Let's look at some specific code fragments that do just this.

Building Up an Array From a String

We have met this technique before, but it is worth reviewing. Suppose we have a large string (called **string**) composed of **n** words each separated by one or more spaces. Then the following code fragment will construct an array composed of individual elements with *nodes* **n**=1

PARSE instruction
H 33 ff
C 10-64 thru 10-68

VAR subkeyword to **PARSE** instruction
H 33 ff
C 10-64 ff

PROCEDURE instruction
H 35
C 10-68 f

to **n**=**k**-1, where **n** and **k** are integers:

```
k=1
DO WHILE string ~= ''
   PARSE VAR string kthword.k string
   k=k+1
   END
SAY 'There are 'k-1' words in the string.'
```

You may recall we used code much like this in the interior function PROCEDURE *Stripword:* in a previous exercise, except we started **k** from 0 instead of 1. We will not explain the code again, except to say that this technique is one which you will use over and over again, so it is worth your study and even memorization. This code uses *parsing by tokenization* to build an array of words, but you will find that it may be adapted to parse words into letters; or words into *arrays* of letters.

An Array Cannot be a Node

ARRAYS see **STEMS** and **COMPOUND** symbols
H 21 f
C 10-44 ff

Compound symbol tokens are powerful tools for string manipulation, but don't make the mistake of using an array itself as a *node*. You must assign its value to a fixed or a simple symbol first before you can use the evaluation of an array element as a node. For instance, suppose we wish to make a compound symbol token (array element) to represent the 3rd letter of the fourth word of the example **string** above. Since the fourth word of **string** is **kthword.4**, it would be tempting to write our array element as **letter.3.kthword.4**, but this is **illegal** in ARexx syntax, and would not evaluate properly. We could assign the *word itself* as a new variable **newword=kthword.4** and then **letter.3.newword** would be legal. Note that the last node is *not a number*, but the *word itself*, which is quite legal. Also legal of course, are the cases of **letter.3.k**, and **letter.3.4** which use simple or fixed tokens as numeric nodes.

Tearing Down an Array and Making It into a String

The inverse of building up an array *from* a string is to tear one down and encode the results *into* a long string. The following code fragment will prove useful time and again, so make your understanding of the procedure part of your ARexx tool kit. We will assume that you have a function that has done something to an array called **list.** and that you wish to return the entire array to the calling program in a string variable called **output**. The array has **length** elements in our example. We also assume that the array elements of **list.** have no leading or trailing blanks.

DO instruction
H 27 f
C 10-53 ff

TO iteration specifier for **DO** instruction
H 27 f
C 10-53 ff

RETURN instruction
H 37
C 10-70

```
/* code fragment to tear down an array list. into a
string output */
output=''
DO i=1 TO length
   output = output||list.i' '
   END
RETURN output
```

CONCATENATION
Operators
H 20
C 10-42

This loop starts with assigning the string **output** the value of the null string. Then a DO loop from the index **i**=1 TO **length** follows. We construct the string in such a way that it has no leading blanks, but one trailing blank. The concatenation operator || we have not met before, but it simply means to *connect without spaces* the two variables **output** and **list.i**. If we had coded this assignment clause as output=output list.i then ARexx would implicitly place one blank between each variable, and we would end up with one leading blank and no trailing blanks. This is merely a matter of personal preferences most of the time, but you need to watch those blanks if later it makes a difference to whatever use the RESULT string is put to.

Stripping Blanks Left and Right

IF instruction
H 29 f
C 10-58 f

If you want to strip away all blanks before and after the output string, you might consider inserting the line

RIGHT() function
H 63
C 10-114

```
IF RIGHT(output,1)=' ' THEN,   /* note comma */
output=LEFT(output,LENGTH(output)-1)
```

LEFT() function
H 60
C 10-108

LENGTH() function
H 60
C 10-108

before the RETURN instruction. We used the RIGHT(), LEFT(), and LENGTH() ARexx functions to look at the trailing blank, and strip it off. We will discuss the use of these functions in more detail as we discover ways to strip off unwanted punctuation from words in a list. You may refer to *Hawes* page 60 or *Commodore* page 10-108 to learn the definitions of LEFT() and LENGTH(); and *Hawes* page 63 or

Commodore page 10-114 for the RIGHT() function. They are extremely useful in string manipulations. Notice the powerful way they may be nested together as we see here. The LENGTH of output minus 1 is one of the arguments to the function LEFT().

The last part of the expression assigning a new value to **output** takes the leftmost all-but-one characters in the string and assigns it to **output** as its new value. Since the LENGTH() function returns an integer, we can do arithmetic with it on the fly, which brings up a small digression.

Pay Attention to Variable Types

TYPELESS data

H 12
C 10-16

A common mistake to fall into is forgetting about the type of value returned by a function and then trying to do something illegal with it. For instance, a function returns a character string, and then you try to divide it by 5. ARexx, even though it types your variables automatically, will not save you from this type of mistake (no pun intended). In fact, this uncovers one of the *advantages* of those other languages which forces you to type your variables and declare precision, etc. It also forces you to think about what kind of results your variables carry, and you are not as susceptible to the "wrong type" kinds of errors. ARexx, if it has a weakness, tempts you to make this kind of error. Always remember to look at variable types if strange errors occur. Before you make a large program depending upon some new algorithm or routine you dreamed up, make a small test program to see if your ideas work properly.

Pseudo Coding the Program to Remove Duplicate Words from a Text File

As always, let's write down what we want to do before we code.

1. Open a file in the standard way in order to read it. Note: we could substitute all the pseudo code from our earlier examples of opening and reading a file here, but we don't have to. We plan to lift the modular code and use it as a header to our main program. We simply have to insure that variable names and so on are consistent with the main program.

2. We need to make the file into one huge line. We do not want to have individual lines as we are interested in the file as a whole: a large list of words.

3. Call an internal function to make a list of all words from the entire file and return a large string with all the words, but with no words duplicated. We will pass as an argument the large line we created in step 2. It will return another large string with the duplicate words removed.

4. Call a second function, a modified Shell Sort performing as an external function, to sort our unique word list into alphabetical order. We send and receive a string as in step 3. We send an unsorted string and receive a sorted string. We will modify the Shell Sort to take a string argument instead of read a file and to return a result instead of writing to a file. We will change only the beginning and the end input/output sections, in other words.

5. Arrange for output to the screen (or to a file). Exit.

Modular Programming

Now this pseudo code doesn't tell us much about coding, but it shows how the structure of modular programming works. We have already coded a useful sort routine and with minimum input/output modifications, we can use it for step 4. We can expand step 3 above into pseudo code describing how our internal function will work. The most important thing to learn at this stage is that you need not understand everything in a complex program or a system of programs *at one time.* You only need to understand the behavior of one module at a time at its interface to a connecting module, and to have control of the traffic flow (the input and the output to/from modules) throughout the program.

Within the modules, themselves, you may further modularize as much as you desire or need to. In this way you can create very large and robust applications which you can service or modify later. You must do more *a priori* thinking with this approach, but it is worth it. If you are the type of individual who wrote your high school book reports after you watched the movie instead of reading the book; and wrote your outline only after you wrote your paper, then this reasoning will be lost on you, but thanks anyway for buying my book! To be honest, some of *my* best work has come from inspired programming the "wrong" way where I just sit down and code an idea. Eventually, however, we all come to grips with the limitations of our own memory, and modular programming becomes the only feasible way to structure large projects.

As a final word, it is essential that you get into the habit of commenting your code even if you are the only one who will ever use it. It is always useful to make comments as if you are going to share your program with friend who doesn't know how you reasoned it out. When you go back to change the program in six months, you will be glad you wrote to a friend!

Expanding Step 3. The Internal Function

ARG subkeyword to **PARSE** instruction
H 33 ff
C 10-64 ff

3.1 First, we need to receive the string from the caller. PARSE the ARGument string.

Initializing Arrays see **STEMS and COMPOUND** symbols
H 21 f
C 10-44 ff

3.2 Make an array and initialize all elements to value 0. This will be a boolean array to answer "yes" or "no" to the question, "Is this word already in the output list (the list of nonduplicated words)?

3.3 Assign the output list variable to the value of the null string. As we have seen in the code fragment above, we will want to build up a string for output. Note: We will not need to build up our string from an *array*, however. We will consider each word based upon a *decision array* instead of an array of words, and then construct the output immediately from the input string by including the word in the output list if the decision array says "no"; and skipping the word if the decision array says "yes".

VAR subkeyword to **PARSE** instruction
H 33 ff
C 10-64 ff

ITERATE instruction
H 30
C 10-60

LEAVE instruction
H 31
C 10-60 f

3.4 Make a loop to repeat while we have input string data to PARSE. We will use the technique of nibbling off a **word** at a time as we did previously using a PARSE VAR instruction, and using the string name itself as the remainder of the string variable. Note: Leaving a loop early in ARexx is accomplished with either the ITERATE or the LEAVE instruction. ITERATE means to stop the current execution of the loop and start the loop over, incrementing any control variables to their next step or value. LEAVE means to stop the loop iteration and exit the entire loop. We will discuss details later.

3.5 In a loop, test the word nibbled off for punctuation, and strip off all leading and trailing punctuation. When all the punctuation is gone, LEAVE the loop. Also eliminate numbers and combinations of characters and numbers, as we are looking only for proper language words of alphabetic characters only. We could expand the pseudo code of this step further later on, but explaining the ARexx code directly is easier to understand, as it is very readable, so we will look at this step in more detail when we write the code.

3.6 Use the **word** variable from the PARSE instruction as a *node* of the decision array! If the array is named **listed.** and the variable for the word is called **word**, then form the array element **listed.word** and see if it is equal to 1 or 0. If it is a 1, then start the loop with a new word (ITERATE the loop), as we have already listed this word. Otherwise, assign the value of 1 to the array element **listed.word**; and then attach **word** to the end of the output string variable, as demonstrated in the code fragment for tearing down an array. Since we initialized the array to all zeros at the first, then each word will fail the IF test the first time it occurs in the input string and succeed (have a value of 1) on all subsequent occurrences, and therefore be skipped over. In this case success means to do the instruction on the same line as the IF test which will mean to end this iteration of the loop and return to the start of the loop without attaching the value of **word** to the output string.

3.7 Based upon the IF test failure, the remainder of the loop attaches the word to the output string, and continues the loop. For purposes of sorting later on we choose to insert a comma between each word in the output list. RETURN the value of the output list to the caller.

Now that we have defined what we wish to do, let's code the ARexx program to do it. Call your program **uniword.rexx** and enter the following into your editor.

```
/* Uniword.rexx This removes duplicate words from a string, and   */
/* shows the use of a compound symbol token (LISTED.) which is    */
/* indexed by arbitrary data (words).  An external function sorts */
/* the list of words alphabetically.                              */
OPTIONS RESULTS
/* Tells program to look for result variable from external calls.   */

/* Input section */
PARSE UPPER ARG infile
IF infile = '' THEN DO
   SAY 'Input text filename and path: '
   PARSE PULL infile
   END
RC=OPEN('textfile',infile,'READ')
IF ~RC THEN DO
   SAY 'File cannot be opened. '
   EXIT 20
   END

/* make a large string out of the file */
list=''
DO WHILE ~EOF('textfile')
   line=READLN('textfile')
   list=list line
   END

/* 'comment out' the following SAY list instruction if you do not want */
/* to see the original file written to the screen.                     */
SAY list
SAY

/* the internal function call to remove duplicate words. */
newlist=Unique(list)

SAY newlist
SAY

/* A CALL to an external function sortword.rexx. Its argument: newlist.*/
CALL sortword.rexx newlist
sortout=result
/* result is a special variable in which the data is returned from the */
/* external function sortword.rexx                                     */

SAY sortout

EXIT 0

/* An example of an interior function, called a procedure follows...*/

Unique: PROCEDURE
PARSE UPPER ARG wordlist
LISTED.=0          /* Shows all possible words as new.                 */
outlist=''          /* Initializes the output list                     */
DO WHILE wordlist ~='' /* Loop while we have data.                     */
  /* Split WORDLIST into first word and remainder.                     */
  PARSE VAR wordlist word wordlist
  /* get rid of numbers */
  IF DATATYPE(word)='NUM' THEN ITERATE
  /* Get rid of punctuation at end and beginning of words. */
  DO FOREVER
    IF ~DATATYPE(RIGHT(word,1),MIXED) THEN word=LEFT(word,LENGTH(word)-1)
    IF ~DATATYPE(LEFT(word,1),MIXED) THEN word=RIGHT(word,LENGTH(word)-1)
    IF DATATYPE(LEFT(word,1),MIXED)&DATATYPE(RIGHT(word,1),MIXED) THEN LEAVE
    IF LENGTH(word)=0 THEN LEAVE
    END
  IF word='' THEN ITERATE
  IF LISTED.word THEN ITERATE       /* Loop if had word before.         */
  LISTED.word=1                   /* Remember we have had this word now.  */
  IF outlist='' THEN sp=''
  ELSE sp=',' /* Put a comma between words in our list                  */
  outlist=outlist||sp||word         /* Add word to output list.         */
  END
RETURN outlist                      /* Finally return the result.       */
```

ASSIGNMENT clauses
H 14 f
C 10-32 f

EOF() function
H 57
C 10-103

READLN() function
H 63
C 10-113

SAY instruction
H 38
C 10-70

RESULT special variable, see **CALL** instruction
H 26
C 10-53

Now let's look at some specific sections of the code. First, we open and read the text file, and put some messages on the screen if the file cannot be opened properly. We have seen code like this before. Starting with the `list=''` assignment clause, we read the file into one large string to ready the file for extracting the individual words from it. We simply read a line of the text file and append the line to the end of the string **list**. We do this while we are not at the EOF of the input file 'textfile'. Then we have an option to SAY list, and add a blank line, so that we can check to see that the text file is one large string. You will want to /* comment this out */ in any final version you use as it will take a long time in a large file to print it all to the screen.

The line `newlist=Unique(list)` represents our internal function call to the routine which eliminates the duplicate words from the list. Then we **SAY newlist** to check to see that the words have been extracted properly. Again, you may want to comment this out in a final version. Finally, we call an exterior function **sortword.rexx** and provide an argument to it in the form of **newlist**. When the sort routine is finished with the list we sent to it, the RESULT variable comes back and we assign its value to the final list which we've named **sortout**. We SAY **sortout** and EXIT 0. This is the end of the main program.

You may want to attach the code we used for writing the output of Shell

RETURN instruction

H 37
C 10-70

Sort to the main program to allow you the option of saving the sorted output. You could also put the writing of the output in the sortword.rexx exterior function and not RETURN anything in which case you would not need the RESULT variable for anything. We have done it this way to demonstrate how RETURN works, but we have by no means exhausted the possible ways to accomplish the task at hand.

The Internal Function, Unique:

PROCEDURE instruction
H 35
C 10-68 f

UPPER subkeyword to **PARSE** instruction
H 33 ff
C 10-64 ff

WHILE iteration specifier for **DO** instruction
H 27 f
C 10-53 ff

VAR subkeyword to **PARSE** instruction
H 33 ff
C 10-64 ff

We have covered almost all of the things we wish to do in the pseudo code. We define a label **Unique:** as a PROCEDURE, and we do not expose any variables. The variables in the procedure are protected from anything in the calling program, and *vice versa*. We PARSE the argument from the caller and put it into a string variable named **wordlist**. Note that we parse everything into UPPER CASE. This is because when the routine checks for words in the list, the value of the node will take into account the case of each letter. Therefore, the easy way to make all words uniform is to change them all to UPPER CASE. We initialize the decision array with a single instruction, and then we set outlist equal to the null string as we have discussed in the pseudo code. Now a DO WHILE wordlist is not equal to the null string loop takes care of going until the input string is exhausted. We split the wordlist into the first word and the remainder of the string as we did stripping words from a line before. We use a PARSE VAR instruction to do it. The next section shows how we get rid of unwanted punctuation and numbers.

How to Get Rid of Punctuation

DATATYPE() function
H 56
C 10-101 f

This section of code is useful in its own right and you may want to lift it out to do other things in other programs. You know enough now to make it into a custom function that may be inserted into any program. We introduce the DATATYPE() ARexx function here. DATATYPE() is a versatile tool for testing or manipulating strings to see what kind of data they contain. You may find the definitions of DATATYPE() and its options in *Hawes* page 56; or *Commodore* page 10-101 f. Turn there now, if you like. DATATYPE() takes a string as its argument and an additional option may be specified. If you use the option, DATATYPE() always returns a boolean value (0 or 1). If you *do not* specify an option, then DATATYPE() returns one of two strings: NUM or CHAR which represent a valid NUMber or a string containing non-numeric CHARacters.

ITERATE instruction

H 30
C 10-60

The first line of our punctuation/number remover takes care of numbers: it tells the loop to ITERATE if PARSE has nibbled off a number from the input string. If the ITERATE instruction appears in the execution of a loop, then the counter is incremented and the loop starts over. No further instructions after the ITERATE instruction are executed for that pass (iteration) through the loop. In this example, all the ITERATE instructions do is to avoid attaching the variable **word** to the end of the output string **outlist**. They stop the execution of the loop early and tell the PARSE instruction to nibble off the next **word** from **wordlist**. The former **word** is discarded into limbo.

FOREVER iteration specifier for **DO** instruction
H 27 f
C 10-53 ff

LEAVE instruction
H 31
C 10-60 f

Now that we are rid of numbers, we enter a DO FOREVER loop to take off any stray punctuation marks from our **word**. We use a FOREVER loop parameter in conjunction with several possibilities of leaving the loop via a LEAVE instruction. LEAVE is similar to ITERATE, except that we don't do the loop again, we leave it for good. If you write a DO FOREVER loop, you probably don't need to be reminded that you must cover all the possibilities, or your program may go on forever. Open another shell and enter a HI (Halt Interrupt) command if you ever need to stop a rampant loop. We have included the only two possibilities in our loop so we are safe. Either the word will begin and end with a valid letter of the alphabet, or else the loop will nibble off all the characters from the word until its length is 0, in which case we have a test to LEAVE if "word" is nibbled to death. This additional code is for strings such as #123 or 56% or 8*9. They will not pass the test for valid numbers and yet they are not words, so the loop will nibble from each end until they are nothing.

Note that our program is far from perfect. Strings such as **A#5A** will be included in the output string going back to the calling function. Strings such as **Z99** will end up with the letter **Z** included in the output as a word. Non-alphabetic characters embedded in letters, and letter/character combinations will almost certainly need to be handled explicitly if you want to eliminate them from a word list, and you will have some additional overhead in having to check every character of each word. Our program is designed to work on fairly ordinary text with spaces between every word. It will correctly remove all normal punctuation even if there are several of them at each end of the word. It won't remove embedded apostrophes, and we wouldn't want it to.

MIXED option to **DATATYPE()** function
H 56
C 10-101 f

LEAVE instruction
H 31
C 10-60 f

FOREVER iteration specifier for **DO** instruction
H 27 f
C 10-53 ff

END instruction
H 29
C 10-57

ITERATE instruction
H 30
C 10-60

Let's do an explicit translation of the first line in the loop: If the *last* character of the **word** is not either an upper or a lower case alphabetic letter (MIXED), then assign the word the value of all its leftmost characters except the last character. In other words, if the last character is not a letter, strip it off the word. The second instruction is similar except it checks the first character of the word, and strips it off the word if it's not a letter. The third instruction checks to see that *both* the beginning and the end of the word are valid letters and if they are, it LEAVEs the loop. We use a compound IF instruction and a boolean **&** to denote *both conditions must be true before we do.*

The final instruction checks to see if we have by chance nibbled a character string such as **#1234** to nothing; if we have, we LEAVE, and come to the END of the DO FOREVER loop. The very next instruction belongs to this group of instructions, however. We need to ITERATE the outer loop if we end up with a null **word**. We don't want to do the rest of the instructions if that is the case, so we make the program start the loop over again at the next **word**.

The Boolean Array at the Heart of the Program

The heart of the duplicate word remover is the
`IF LISTED.word THEN ITERATE`
expression. Note that we don't even have to assign anything because the **LISTED.** array is defined as a boolean array in the first place, specifically because we need to test it conditionally in this IF statement. This is the one line of code in the program that does everything. It decides if the word is included or skipped because its already included. The array design is unusual because it uses a *large number of nodes* (our words from the list) to make variables which can be only 0 or 1.

Non Numeric Nodes

Non Numeric Nodes
see **STEMS** and
COMPOUND symbols

H 21 f
C 10-44 ff

The other unusual thing about this array is that the nodes are not numbers; they are strings. Don't miss the pure elegance of ARexx because this line of code is so compact! Try doing an equivalent program in another language and see how you like it.

The next instruction assigns a value of 1 to the array element **LISTED.word** if **word** has not occurred before in our list and the array element representing it therefore failed the above conditional test because it had a value of zero. The next few instructions take care of the initial case where **outlist** is null. We mentioned we wanted to put a comma between each word in the list (just for fun); but we don't want the list to start with a comma, so we introduce a variable **sp** to put between words: If we are at the start, **sp** is the null string, and otherwise it is a comma.

CONCATENATION
Operators

H 20
C 10-42

Building the Return String from the Array

Last, we build the output string for every **word** that made it this far by concatenating **outlistl l spl l word** and ending the loop. When the loop has exhausted all data from the input string **wordlist**, we let the function **RETURN outlist** to the caller, and we are done with the internal function.

The External Function Sortword.rexx

Because we have discussed all of its features before, we will make only a few remarks about this sort function. Study the listing and look for the code we have gone over before. Notice how easy it is to construct a fairly involved ARexx program using bits and pieces from previous programs and examples. In any language, once you begin to remember and use common phrases and build your vocabulary at the same time, you get to a point where it gets easier and even enjoyable to learn more. ARexx is like that, too. I sincerely hope that you are beginning to get excited about the possibilities of ARexx!

There is very little difference between a stand-alone program and an

PARSE instruction
H 33 ff
C 10-64 thru 10-68

EXIT instruction
H 29
C 10-57 f

RETURN instruction
H 37
C 10-70

exterior function in ARexx. Note that we PARSE the argument from the caller as we did in Shell Sort, except that this time it is a long string instead of a file name; and we make up the array in a slightly different way; but there is nothing truly new or unfamiliar about this function as opposed to the Shell Sort program. Also notice at the end we use **EXIT sortout** to return the string variable.

Both EXIT and RETURN may carry variables or expressions with them. Up to now we have been using EXIT to carry numbers as a return code, but it may carry string variables and expressions (which are evaluated before sending) as well. Thus, you may put multiple variables (including specific array elements) in an EXIT expression by name. The caller will receive only one string, however, as we stated before; the variables will be evaluated before sending and the result string will be that evaluated expression. One way to send a list of variable values back in one string is to make the EXIT expression by concatenating a particular character in between each of the variables and then to PARSE the result string in the main program using pattern matches to remove the character pattern from the string and to retrieve your variables. To review EXIT and RETURN, consult *Hawes*, pages 29 and 37; and *Commodore*, pages 10-57 f and 10-70, respectively. Here is the listing of **Sortword.rexx**.

```
/********* SECTION ONE: INPUT THE ARGUMENT MAKE THE ARRAY **********/
/* The Shell Sort sortword.rexx */
PARSE ARG stat      /* Bring in the line of data to sort */
m=1        /* Steps 3- 10 make an array of data (list.) */
DO WHILE stat ~=''
   PARSE VAR stat list.m',' stat      /* Cut up into items */
/* An example of parsing on a symbol, in this case ','.          */
/* The VARiable stat is cut up into an array list.m              */
/* Each time through the loop the next word is put into array.  */
   m=m+1        /* increment the array index m.          */
   END
   m=m-1        /* Adjust total number of unique words    */
SAY ''
SAY 'Sorting 'm 'unique words ...'

/*****************************************************************/
/********* SECTION TWO: THE SHELLSORT OF ARRAY 'LIST.' *************/
/* From here until end is a good sort routine. See text, figures.  */
listlength=m
span = 1
DO WHILE (span < listlength); span = span * 2; END
DO WHILE (span > 1)
   span = span % 2
   node = listlength - span
   DO scan = 1 TO node
      nextnode = scan + span
    IF list.scan > list.nextnode THEN
         DO
         store = list.nextnode
         list.nextnode = list.scan

         /* The inner nested bubpos loop */
         DO bubpos = scan-span TO 1 BY -span WHILE (store < list.bubpos)
            nextnode = bubpos + span
            list.nextnode = list.bubpos
            END bubpos
         bubpos = bubpos + span
         list.bubpos = store
         END
      END scan
END
/* End of sort routine, output follows...*/

/*****************************************************************/
/********* SECTION THREE: THE OUTPUT AND RETURN TO CALLER **********/
/* This block reconstructs a line to return to caller...*/
sortout=''
DO i=1 TO listlength

/* A loop to reconstruct the list, now in alphabetic order.      */
/* Each time through, the variable sortout is increased by a word */
/* until all the words are rearranged in the final sortout list   */

   sortout = sortout||list.i' '
   END
EXIT sortout  /* The string sortout returned to caller... */
/*****************************************************************/
```

Exercises

Try out this program on the text file we made for the first tutorial. Try it out on various text file you may have. See if there are things you should modify to account for special conditions (like upper or lower case) or characters in your data. Think about how you would go about constructing an index for a text file and tying every instance of each word to a specific page number. It is by no means a trivial task.

Can You Use Higher Dimensional Arrays?

See if you can rewrite the code using two dimensional arrays instead of building up and tearing down arrays from long strings. Try making your program so that there are no interior or exterior function calls. Hint:

make an array **word.** with nodes **i** and **j** to denote *line* number and *word* number, respectively. What must you look out for when dealing with the nodes of the **LISTED.** boolean array? A listing in **Appendix A** under the name **UNIarray.rexx** is one solution to this exercise. It is more compact and perhaps more elegant at the expense of being more complex.

The real elegance of ARexx, however is that you are never stuck using only one program to do what you want. You have the full power of every program in your computer which has ARexx support working for you. In later sections we will begin to look at the really sophisticated aspects of ARexx as we begin to control other programs as if by magic!

Chapter 6: Debugging, Tracing and Interrupting ARexx Programs

Handling Mistakes

Inevitably you will make a mistake and wonder why your program doesn't work. ARexx is the exception among high level languages in that it provides a powerful set of tracing, debugging, and interrupt commands to aid your program development.

Tracing

TRACING and INTERRUPTS

H 71 thru 76, 83 thru 87
C 10-134 thru 10-145
C 10-83 f, 10-89 f,
C 10-155 thru 10-157

Note: The manuals are organized differently, but contain the same information.

A **trace** is the ability of ARexx to output the results of each instruction as it executes. The TRACE instruction has a number of options with which you may select exactly which results you want to see. You have the added option of selecting **interactive tracing**, in which mode the program stops after each step to allow you to input commands, re-execute the last step, or continue. The facility for inserting and executing instructions in interactive tracing mode is sometimes referred to as a **debugging facility**, because the programmer may try different instructions and view their effects interactively and immediately.

Arexx Interrupts

ARexx **interrupts** are completely separate from AmigaDOS interrupts, and are commonly known in programmer's jargon as **error traps**. The ARexx interrupts allow you to trap errors of different kinds by making the program branch to a certain label and begin executing any routine you put there if an error of the kind specified occurs. You may simply want to write a custom error message and then exit, for instance.

Further Documentation

The relevant sections for the entire discussion of tracing, debugging, and interrupts are in *Hawes*, Chapter 7, pages 71 to 76; and Chapter 9,

BACKGROUND option of **TRACE** instruction
H disk
C 10-135 ff

OFF option of **TRACE** instruction
H 40, disk notes
C 10-135 ff

pages 83 to 87. Or you may look at *Commodore*, pages 10-134 to 10-145; 10-155 to 10-157; 10-83 f; and 10-89 f. Note: The two manuals include identical material, but Commodore chose to arrange it differently. The *Hawes* ARexx Update notes from disk include two TRACE options not included in its manual's option listing: **off** and **background**, which we will look at later. We will discuss tracing first.

Tracing an ARexx Program

Trace Options

TRACE instruction
H 40, 71
C 10-134 thru 10-145

TRACE() function
H 67
C 10-120

The TRACE keyword may be followed by several **options** called *alphabetic* options because they can be shortened to the first letter of their name. If you do not supply a letter or an option, the **normal** option is the default. The options are ALL, COMMANDS, ERRORS, INTERMEDIATES, LABELS, NORMAL, RESULTS, SCAN, OFF, and BACKGROUND. Each option may be further qualified by a **mode character**.

Trace Modes

TRACE MODES
H 73
C 10-139 f

There are three modes available: Normal, **Command Inhibition** and **Interactive Tracing**. To enter *command inhibition mode*, you precede the option by the **!** character. To enter *interactive tracing mode*, you precede the option by the **?** character. In *command inhibition mode*, any time rexxmast encounters a command (a syntactical line it cannot interpret as a meaningful ARexx instruction), it sets the return code **RC** to zero (success), but doesn't actually send the command to the host application. This is obviously useful for times when you need to see if the right values are to be sent to a host application, but do not want to send them, because they could potentially prove destructive as in altering a file or erasing data.

The Special Variable RC

RC special variable
H 39, 73, 75
C 10-73, 10-139, 10-145

What is **RC**? This is a special variable in ARexx like RESULT. **RC** is a variable used to denote the success or failure of an operation and is available immediately afterwards to look at. If the operation is successful, then **RC=0**; if it failed, **RC** is set to the value of the **error** or **syntax code** resulting from the *condition* that triggered the interrupt (error trap). Since all functions and many commands use **RC** to convey their success or failure, you must deal with its value *immediately after* a command or function is executed. Some host applications also use **RC** to return values in addition to the information returned in the RESULT variable.

Combining Mode and Option

If you wanted to trace intermediates (as defined in the options), but suppress command execution, you would enter the instruction **TRACE !i** in your program at the position at which you wish to start the trace.

Interactive Tracing

Interactive tracing is just what the name implies: You can trace a program one line at a time, and enter code at each line, execute the instruction as written, or re-execute the previous line. If you are tracing results, for instance, a **TRACE ?r** instruction will begin tracing each result (as defined in the options) and pause at each line for user input or control. If you are in interactive trace mode and happen to enter a wrong instruction, say one with a syntax error, rexxmast will report the error, but will not terminate the program as usually happens. If one of the instructions originally coded into the program fails however, the program will process that error as it usually does and the program will terminate.

TRACE OFF

This simply turns off the trace instruction at some point, assuming you have turned it on at some other point. Suppose, for instance you have a complicated set of calculations and you want to trace all the intermediate results of each calculation, for only a few instructions. You would insert the line **TRACE intermediates** (or **TRACE i**) one line before your area of interest and **TRACE OFF** just after the last instruction you are interested in.

Example

Let's do a specific example in line with our philosophy of hands-on experience. Open your editor and load the **Hex.rexx** program from the last chapter. Then go to the last PROCEDURE in the program and uncomment the two trace instructions. This will effectively trace all the intermediate results of all these instructions and turn off the trace at the end.

Looking at a Trace

Where can you see the trace? There are two possibilities. If you simply run the program from your shell, the trace results will be **interleaved** (written as they occur in the program) with the program's regular output to the console window of the shell. Since there is no program output to the shell window during this part of the program, there will be no confusion in this example. Try running the program now and watch the trace fly by. Remember, as always in the Amiga shells, you may stop the action by pressing the space bar. When you are finished looking, press backspace to continue. You may wish to resize the window as large as possible to view lengthy trace output. Note the display output format codes (**tracing prefix codes**) that tell you everything about the result. Refer to the manual to learn their meanings.

TRACING PREFIX CODES

H 72
C 10-137

Now try the program again after you have specified another option, say **results**. Observe how the trace output changes its format. As an exercise, try the program (or any other ARexx program) and try different trace options in different places until you feel comfortable using the different options, and can read their output. I like the TRACE RESULTS option best, but TRACE INTERMEDIATES is great for debugging sections heavy with number crunching instructions. You may use the TRACE instruction in several places using several options if you wish to vary the kind of tracing you do. You may turn off the tracing anywhere with a TRACE OFF instruction. ARexx provides a very flexible and easy to use debugging and tracing facility. There are several more variations open to you as we will now discuss.

ARexx Command Utilities

COMMAND UTILITIES
H 83 f
C 155 f

GLOBAL TRACING CONSOLE
H 72 f
C 10-137 f

TCO command utility
H 84
C 10-156

TCC command utility
H 84
C 10-156

TRACING OUTPUT
H 72 f
C 10-138 f

The Global Tracing Console

If you wish to trace a part or all of a program that has lots of output to the console window, or if you have a host application program that *does not provide an output stream* of its own, you may open the ARexx **global tracing console**. This is a window just like a shell window which may be resized and moved as you wish. It is opened and closed from a shell window. You cannot close it from within itself like you can a shell window. To open the global console window, issue the **tco** command from a shell (**tco** stands for *trace console open*). To close the trace console window, issue its sister command **tcc**. It is a good idea to open a separate shell from which to issue these commands.

The advantage of the trace console is that only the *trace output* from the program is directed there. The ARexx interpreter accomplishes this feat by looking for a **logical stream** called **stderr**. A logical stream is a *name* for an *actual* stream of data just as a logical file is a name for an actual file in the system. Rexxmast sends all *tracing* output to one of only two logical streams: **stdout** or **stderr**. Opening a global trace console with **tco** *automatically* opens **stderr** for each ARexx program which has not previously defined **stderr**. The tracing output for each program then goes into this new stream. Note that all ARexx programs currently running will share the global trace console: That's what the word *global* means. If you are tracing a program, it is a good idea to shut down all other ARexx programs that contain a trace instruction to insure that you are seeing only one program's trace output at a time.

Interactive Inputs

If your program is tracing in interactive mode, you input your interactive commands and control from the *global console window* and not from the original window.

Setting Trace Flags

External Trace Flag

TS command utility
H 84
C 10-156

TE command utility
H 84
C 10-156

There are two other global commands that affect tracing: **ts** and **te**, which are used in a similar way to **tcc** and **tco** as a shell command. The command, **ts** sets what is called the **external trace flag**, which forces all active ARexx programs into **interactive tracing mode**, whether they contain the interactive mode character **?** or not. These programs are forced into an interactive (**TRACE ?RESULTS**) option unless they are already set to **TRACE intermediates** or **TRACE scan**, in which cases they continue to trace with these options.

Global Halt Interrupt (HI) Flag

HI command utility
H 83
C 10-155

We have mentioned the **hi** or halt interrupt command before. It is also a global command that sets the **global halt flag** which issues all active ARexx programs a request to halt. Rather than pressing this panic button, you may use **ts** for those times when your program gets stuck in an endless loop and you are not sure why. The **ts** command will put the program into interactive tracing mode and allow you to step through the loop and see where you went wrong, and try various commands interactively to see if they will fix the problem. To turn off the interactive mode externally, you issue the **te** command. The commands **ts** and **te** set and clear the **global external trace flag**, so remember that all ARexx programs launched subsequent to the **ts** command will enter interactive trace mode until you turn the mode off with **te**. That's why it's less confusing to have a separate shell open to do *global commands*, properly called **command utilities** in ARexx jargon.

RXLIB command utility
H disk
C no reference
see page 6-14

RX command utility
H 84
C 10-155

The ARexx rx, rxc, rxlib and rxset Commands

RXC command utility
H 84
C 10-156

RXSET command utility
H 84
C 10-155

CLIPLIST
H 86
C 10-89 f

Note that the **rx** command used to start your ARexx programs is such a command utility. Its inverse is **rxc** which closed the rexxmast program. The final ARexx command utility is **rxset** which adds a (name, value) pair to the global ARexx **Clip List**. This is a structure similar to the AmigaDOS clipboard device where you may put up constant variables or other information to be shared by all ARexx programs. An entry in

GETCLIP() function
H 58
C 10-105

INTERPRET instruction
H 30
C 10-59

the Clip List may be retrieved by name using the GETCLIP() ARexx function and either parsed or interpreted by the INTERPRET instruction. In this way you may store code hunks for later interpretation and execution. **Rxlib** acts the same as the ADDLIB() function (see pg 6-14).

The Trace Background Option

BACKGROUND option of **TRACE** instruction
H disk
C 10-135 ff

One of the options not mentioned in either *Hawes* or *Commodore*, is the TRACE BACKGROUND option. This option is covered in the disk update notes for ARexx v1.10 and v1.15. If you have a tried and true ARexx program that you wish to leave running while you use **ts** to trace another program, you may put a TRACE BACKGROUND instruction in the well behaved program. It will then prevent that program from being forced into interactive tracing if the global tracing flag is set by **ts**. Obviously, you would not want to use a mode character with this option.

Exercises

1. Open a shell (No. 1). Enter the command **tco** to open the global console. Open another shell (No. 2). Run the **Hex.rexx** program with a trace instruction in it, but *not* in interactive mode. Watch the output in the global console. You enter your character string from the second shell.

2. Do 1. above, except during the time that the output is whizzing past in the global console, enter the command in the first shell: **ts**. Watch the output stop in the global console and enter interactive mode. Step through a few commands by pressing [Rtn]. Go back to the first shell and enter **te**. Go to the global console and press [Rtn] and watch the output resume.

3. Put an *interactive* trace instruction (TRACE ?R) inside the **Hex.rexx** program where we uncommented the TRACE i instruction. Save the program and start it from shell No. 2. Note that you must enter all *normal input* to the program from shell No. 2, but you must step through the program from the global console. Try entering an = instead of a [Rtn]. Did the last instruction re-execute? Note how easy it is to get lost

if you do not input from the correct window! If you really mess up use an **hi** instruction from shell No. 1. to stop everything. You may have to step through a few instructions before the program gets the halt instruction flag information.

4. Now run a normal ARexx program *without* any TRACE instructions in it from shell No. 2. After it starts, use shell No. 1 and enter **ts**. Go back to shell No. 2 and begin to enter normal input if called for. Does the program enter interactive mode? If you have a global console open, does the trace output go there? How does it work if you do not have a global console opened? If you try to close a global console before an ARexx program using it finishes, what happens? (It should not close until all programs using it exit!)

5. Remember: Turn the **exterior interactive tracing** on and off from shell No. 1. Open and close the global tracing console from shell No. 1 also. Start the program and enter any *normally requested* input from shell No. 2. If the global console is open, step through the program in interactive mode from *that* window. You can make up lots of experiments and soon you will be a pro at the ARexx tracing and debugging facility!

ARexx Interrupts

INTERRUPTS (ARexx)

H 74 ff
C 10-143 ff

We have all heard on the radio, "We interrupt this program to bring you a special message..." Computer interrupts are functionally the same. Whenever something occurs that needs immediate attention, such as an error, or a signal from the user, all operating systems and programming languages provide a means to trap errors or deal with the situation dynamically before the program terminates. ARexx is no exception.

When certain conditions occur within ARexx its **internal interrupt system** allows you to trap errors or detect events such as pressing the **[Ctrl]-C** keys. This syntax means you press the **control** key and the **C** key at the same time. We say a **[Ctrl]-C** *break* is an **asynchronous event**. **Synchronous events** occur when the program detects something like a syntax error. ARexx can handle both types of events.

The internal interrupt system transfers program control to a **label** specific to the event or condition, only if the interrupt itself is **enabled** (turned on with a SIGNAL ON *name-of-interrupt* instruction).

Name the Labels After the Interrupts

SIGNAL instruction
H 38 f
C 10-71 ff

The interrupts themselves are named according to the label to which program control transfers in the event of the interrupt (Self reference *again*!). For instance, the **ERROR** interrupt is also the name of the label to which the program will branch if the **ERROR** interrupt is enabled, so a **SIGNAL ON ERROR** instruction in your code will transfer program control to the label **ERROR:** in the event that an error occurs. You may put any code you want to execute after the label such as an orderly exit or a custom error message. Refer to *Hawes*, page 38 f; and *Commodore*, page 10-71 ff to learn more about the **SIGNAL** instruction and its interrupt subkeywords: **BREAK_C**, **BREAK_D**, **BREAK_E**, **BREAK_F**, **ERROR**, **HALT**, **IOERR**, **NOVALUE**, and **SYNTAX**. These are explained in the manual. The correct syntax for using SIGNAL with, for instance, a *control-C break* is to use **SIGNAL ON BREAK_C** to turn on (*enable*) the interrupt, and a SIGNAL OFF BREAK_C instruction to turn it off (*disable*) it again. All the rest follow the same pattern.

What happens during an interrupt? First, ARexx *dismantles* all active **control ranges**, which are the ranges in DO loops, IF instructions, SELECT, or INTERPRET blocks; or interactive TRACE. Then ARexx transfers control to the label specified by the enabled interrupt. Since the active control ranges are dismantled, you cannot use an interrupt to jump into a control structure such as a SELECT block. If you are inside an internal function, and an interrupt occurs, it is safe to use SIGNAL without affecting the environment of the calling program. Recall how variables are protected there as well.

Special Variables

We mentioned **RC** before, and it plays a part in interrupts, too. **RC** is set to the error code (for SYNTAX interrupts); or severity level (for ERROR interrupts) of the condition that caused the interrupt, and you may therefore check **RC** immediately after your label statement if you want to

SIGL special variable
H 39, 75
C 10-73, 10-145

know about what caused the transfer. There is another special variable called **SIGL** which returns the line number that was being executed at the time of the interrupt, and you may check after your label, too. You may thus trap and glean information about errors using the SIGNAL instruction with the appropriate interrupt subkeywords.

Yes, ARexx Has a GO TO Statement

SIGNAL instruction
H 38 f
C 10-71 ff

A second way to use the SIGNAL instruction is to **SIGNAL [value]** ***expression***, which computes the expression if supplied and jumps to a label with its value; or simply jumps to the supplied value if it is a string. This is exactly like a "GO TO" instruction in other languages. For instance if you have a need for the program to jump to a label called **Instead:** then a **SIGNAL Instead** instruction will do the trick. If the label name is the result of an expression, then the **SIGNAL VALUE** ***expression*** acts just like a *computed GO TO*, jumping to whatever label the evaluated expression indicates. Note: Use a GO TO *only when absolutely necessary*, or your code will suffer. Sometimes a GO TO or two make sense, but all too frequently, if overused, they produce weird logic flow and strange, inscrutable errors to show up in your logic. Stick to structured programming (using only IF THEN ELSE; DO WHILE; and DO UNTIL type constructs). You will be better off. It is provable that structured programs can do any logic you need.

An Example of the Use of Interrupts

Here is a little program that will demonstrate what we have just discussed. It contains an example of the use of every interrupt except the IOERR interrupt. I couldn't figure a way to make my system misbehave in order to trap this one! The program also demonstrates the use of SIGNAL properly used as a GO TO statement. Although the program does nothing except demonstrate, you could use it as a pattern to insert error traps in your own code sometime. Here is **Int.rexx** a demonstration of Interrupts in ARexx:

```
/* Int.rexx Demo of interrupts */
/* turn on the interrupts */
SIGNAL ON BREAK_C
SIGNAL ON BREAK_D
SIGNAL ON BREAK_E
SIGNAL ON BREAK_F
SIGNAL ON ERROR
SIGNAL ON HALT
SIGNAL ON IOERR
SIGNAL ON NOVALUE
SIGNAL ON SYNTAX

/* error */
'FOOBAR'
ERR:

/* I/O error*/
/* If I could think of an IO error example it would
go here! */
IO:

/* uninitialized variable */
SAY 'i=v'
i=v
UNI:

/* syntax error */
SAY 'END'
END
SYN:

SAY 'PRESS [Ctrl]-C,D,E, or F to stop endless
loop...'
SAY 'OR open another shell and do a HI command...'

/* stuck in an endless loop...*/
DO FOREVER
NOP
END
```

NOP instruction

H 31
C 10-61

```
START:
SAY 'DONE!'
EXIT 0

/* INTERRUPT LABELS FOLLOW */
/* Right here is where you would program your */ /*
recovery routines */

BREAK_C:
SAY 'CONTROL C BREAK detected...'
/* display the special variables */
SAY 'Line'SIGL 'RC='RC

SAY 'Press [Rtn] to continue.'
PARSE PULL answer
SIGNAL START /* using SIGNAL as a GO TO */
```

```
BREAK_D:
SAY 'CONTROL D BREAK detected...'
SAY 'Line'SIGL 'RC='RC
SAY 'Press [Rtn] to continue.'
PARSE PULL answer
SIGNAL START

BREAK_E:
SAY 'CONTROL E BREAK detected...'
SAY 'Line'SIGL 'RC='RC
SAY 'Press [Rtn] to continue.'
PARSE PULL answer
SIGNAL START

BREAK_F:
SAY 'CONTROL F BREAK detected...'
SAY 'Line'SIGL 'RC='RC
SAY 'Press [Rtn] to continue.'
PARSE PULL answer
SIGNAL START

ERROR:
SAY 'ERROR detected...'
SAY 'Line'SIGL 'RC='RC
SAY 'Press [Rtn] to continue.'
PARSE PULL answer
branch='RR'
SIGNAL VALUE 'E'||branch /* a computed GO TO */

HALT:
SAY 'EXTERNAL HALT detected...'
SAY 'Line'SIGL 'RC='RC
SAY 'Press [Rtn] to continue.'
PARSE PULL answer
SIGNAL START

NOVALUE:
SAY 'UNINITIALIZED VARIABLE detected...'
SAY 'Line'SIGL 'RC='RC
SAY 'Press [Rtn] to continue.'
PARSE PULL answer
SIGNAL UNI

SYNTAX:
SAY 'SYNTAX ERROR detected...'
SAY 'Line'SIGL 'Error:'RC 'Refer to documentation.'
SAY 'Press [Rtn] to continue.'
PARSE PULL answer
SIGNAL SYN

IOERR:
SAY 'Line'SIGL 'RC='RC
SAY 'I/O ERROR detected...'
SAY 'Press [Rtn] to continue.'
PARSE PULL answer
SIGNAL IO
```

Now you know how to use the ARexx Tracing, Debugging and Interrupts! With these tools, you need not be afraid to tackle large projects, because you'll have the confidence to know that you can find out why things don't work the first time. We have almost come to the most difficult but the most powerful way to use ARexx: Controlling other programs. You will be glad you took the time to learn to trace and debug when you start dealing with multiple host applications. There is one last thing we need to do before we take on the control of other programs, and that is to load the ARexx support libraries.

The ARexx Support Libraries

We have mentioned that ARexx is capable of using what is called a **shared library**. ARexx cannot use just any Amiga shared library, however, because there must be a public message port of the required type and certain other technical considerations must be satisfied before a shared library may be used by ARexx. A shared library is a collection of programs which serve as external functions which a program may call. For instance, a file requester may be structured as a library rather than being made a part of the main program, and whenever the program needs to open a file requester, it summons the requester library. Many of your applications programs have undoubtedly used shared libraries in processing your results, even if you didn't know about it.

Shared libraries reside in the **Libs:** directory of your system disk, and often, when you install a new piece of software, part of the installation process is to copy a library necessary to the main program into the **Libs:** directory, so that your program can find it and load it at the appropriate time. ARexx provides several libraries for use by ARexx programs, and there are a few ARexx libraries programmed by third parties. Before you may use them in an ARexx program they must be loaded into the **library list** maintained by ARexx.

Loading ARexx Libraries

ADDLIB() function

H 51
C 10-92 f

RXLIB command utility
H disk
C no reference

RXLIB functions the same as ADDLIB(). The command is followed by the library name, the priority, the offset, and the version, but there are no parentheses used as in the function ADDLIB().

You may load your ARexx libraries only when you need them in each individual program by means of the **rxlib** command utility or the ADDLIB() built in ARexx function, or you may do what is far simpler and load them all in your startup-sequence. Since they sleep and do not take up any system resources until they are called, there is not really any reason *not* to load them at your computer startup and be done with it. If you plan to export your ARexx programs to others, however, who may not have had the foresight to pre-load all the ARexx libraries, then you would do well to include some code to check for the library in the ARexx library list and to load it if it's missing.

In either case the following little program will take care of loading some available ARexx libraries. You can put it into your startup-sequence *after* the start of **rexxmast**, or you can include appropriate pieces of the program in the code you export to others. Some of the support functions in these libraries will be necessary to our programs later. If you do not have one or more of the following libraries, then leave out the code relevant to that library! Check your **libs** directory. You should at least have **rexxsupport.library**.

SHOW() function

H 64, disk
C 10-115

Note: Library names are case sensitive!

```
/* LAL.rexx Loads ARexx Libraries */

L.1='rexxsupport.library'
/* extended functions (DOS,etc.) */

L.2='rexxarplib.library'
/* intuition, windows, gadgets */

L.3='rexxmathlib.library'
/* sin, tan, cos, and other math functions */

L.4='rexxutil.library'
/*  rexxutils  */

DO i=1 TO 4
   IF ~SHOW('L',L.i) THEN CALL ADDLIB(L.i,0,-30,0)
   IF ~SHOW('L',L.i) THEN SAY L.i 'failed to open.'
   END
EXIT 0
```

You can add this code (minus the EXIT instruction) at the start of your programs that need ARexx libraries, or you can include the line

```
sys:rx LAL.rexx
```

in your startup-sequence *after* it starts **rexxmast**. The documentation for the syntax and arguments of the ADDLIB() and SHOW() functions are in *Hawes*, pages 51 and 64; and *Commodore*, pages 10-92 f and 10-115. Note: The SHOW() function is documented *further* in *Hawes* v1.15 Update Notes on disk to include the option **SHOW('Internal')** or **SHOW('I')** to show the **internal ports list** (the ports that are open *internal* to the current program).

What Do These Libraries Do?

ARexx Support Library

H 127 thru 130, disk
C 10-129 thru 134

Starting at the top, the **rexxsupport.library**, by William Hawes, is a collection of *AmigaDOS specific* functions that supplement the built in ARexx functions (which adhere solely to the Rexx language standards). In other words they *add* functionality that works *only* in the AmigaDOS environment, so your code containing references to these library functions will not **port** (transfer) to another kind of computer (as if any of the others have Rexx!).

The second entry, **rexxarplib.library**, by Willy Langeveld, is a collection of functions which allow you to use the Amiga Intuition Graphical User Interface (or GUI) to open gadgets and menus to do things from an ARexx program. ARexx has no graphics capabilities whatsoever on its own, but thanks to the Amiga shared library environment, graphics are possible.

The third, **rexxmathlib.library**, also by Mr. Langeveld, is a collection of advanced math functions such as SIN, COS, TAN, etc.

The final entry, **rexxutil.library**, by David Junod, is a collection of functions to add standard data sharing techniques to ARexx such as reading and writing from and to the AmigaDOS clipboard device (which is essential for some programs we will code later); and storing and retrieving environment variables.

The Companion Disk for *The ARexx Cookbook* contains a free copy of all libraries used by its programs and full documentation by the libraries' authors. See the copyright page to order.

Obtaining Libraries

You may or may not have all of these libraries but those you *do* have, you may load at startup using an adaptation of this code. The first library, **rexxsupport.library** comes standard on new Amigas with ARexx, but the others do not. The two by Mr. Langeveld are available with the purchase of the *Hawes* version of ARexx, or free from BIX BBS (Bulletin Board Service). The library by Mr. Junod is available free on BIX. Undoubtedly as the popularity of ARexx soars, there will be others.

Library Documentation

Since each of these libraries comes with its own documentation, and are often the subject of updates, I will only point out certain ways to use these libraries and leave it to the reader to explore the details and documentation of each library. The **rexxsupport.library** is documented in *Hawes*, Appendix D pages 127 through 130; and in *Commodore*, pages 10-129 through 10-134. However, it has many *new* functions not documented except in the *Hawes* ARexx v1.15 Update Notes on disk. If we need any of these, we will mention how to use them. Now we are prepared to explore the complex and fascinating world of controlling your application programs with ARexx.

Chapter 7: Controlling Host Applications with ARexx

A Model for ARexx Remote Control

Host Address and **Host Application**
H 25, 43 f
C 10-50, 10-75 thru 10-78

Message Packets
H 89 ff
C no reference

We are about to enter the realm of *remote control* which can become somewhat abstract. We would do well to visualize a *model* of how our ARexx programs will work in a multi-tasking environment, in order to keep track of where our program *is* and what it is *doing* and to *what*.

In the introduction, we said that ARexx controls other programs by means of message packets sent to or received from a host application sporting a message port, with a HOST ADDRESS to mark it. Since the term *address* is a part of ARexx nomenclature, we can incorporate this word into our model directly, and use the image of a postal service delivering mail to visualize how ARexx communicates with other programs.

Addresses and Programs

Let's think of each ARexx host application program on our disk as an address in a city (your computer). In order to *read* the mail, or to *send* mail, any particular resident must be *at home* (Let's say things are a little crazy in our utopia. When you're not home, you take your mailbox with its address and lock it inside your house, so your address is no longer accessible!). We will pretend that you send mail *only* by leaving a letter in your mailbox for the postman to pick up, and that any time you receive a letter, you immediately reply to the sender with an answering letter left in your mailbox. If a *person* is not at home at his or her address, it is equivalent to a *program* not running in our system, so all the mail with that address will be returned to sender with a message (by the postmaster -rexxmast) saying "HOST ADDRESS not found".

The Public Message Port

ARexx Commands
H 43 thru 46
C 10-74 thru 10-82

Obviously, the mailbox is the public message port, and the address itself is the host address. The ARexx program itself is represented by the *actions* of the postman, as directed by the Postmaster -rexxmast, and ARexx *commands* are the content of each letter delivered by the postman. Remember that *commands* are "none of ARexx's business" any more than the content of your letters are the concern of the postman who delivered them, so our analogy has a good deal of relevance. Any letter not specifically for the Postmaster is sent on to the addressee.

Asking for Results

RESULT special variable, see **CALL** instruction
H 26
C 10-53

RESULTS subkeyword of **OPTIONS** instruction
H 32
C 10-62 f

Let us suppose that each letter sent by a resident (host application) contains a request to some other resident to perform some research and write down an answer and reply by return mail in a message titled RESULT. Now our little utopia takes a bizarre twist, because occasionally, residents (host applications) write notes to themselves to remember to do some research themselves at home, but they never seem to remember to do *anything* unless they get a letter at the mailbox (slavish creatures of habit!), so they post a letter to themselves to remember to perform the research.

COMMAND special address for **ADDRESS** instruction
H 44
C 10-76

The postman dutifully delivers the letter, (it takes three weeks to arrive); the resident does the research, and writes the reply; and so that he can find it again, puts it in his own mailbox! (These nutcases actually represent how ARexx *macros* operate.) Since the Post Office is a branch of the government (you can tell because of the External Flag flying from the rexxmast outside), each resident of the city has access to all government files and records. They have only to post a letter to the central government with ADDRESS COMMAND on the outside in order to access these government services.

DOS Commands

If they want to do any detailed research, such as finding a particular line in a file, then they use the services of the Post Office which maintains close contact with the government processes. They can, and often do, send requests to their Postmaster to do such work. What is even more bizarre than in the real world, they actually get a reply *every* time! The government, of course represents AmigaDOS. Notwithstanding the silliness, our model is a pretty accurate map of how ARexx works in the Amiga environment.

A Word About the Examples of ARexx Control

To paraphrase the Apostle John: "If every way to control programs with ARexx were written down, I suppose that even the world would not have room for the books that would be written."[1] It is impossible to choose examples of ARexx macros or interprocess control which will be explicitly relevant to everyone. Keep in mind, however, that in the abstract, every ARexx program which controls another application shares certain characteristics. It is possible, therefore, to learn how to control your own programs by studying ARexx programs that control programs you may not even own[2]. The hardest part of making any ARexx interprocess control program work is keeping up with the sections of the program *outside* the actual host appliction. Once your letter (your command) safely arrives[3] at its proper destination, it is relatively easy to execute it inside the program. It is dealing with replies (what is the content of each reply as expressed in the RESULT variable?), interleaved with issuing commands (how can I use the internal command set to best effect?); and keeping up with addresses for those commands, that are most difficult for the beginner.

1 See:
John 21:25 (NIV)

2 Other good sources of ARexx programs to study are magazine articles, and macros and examples from your applications which support ARexx.

3 The proper way to surround with quotes the commands sent to the ARexx command parser is guaranteed to cause you trouble until you understand the way ARexx builds these command strings. The **TRACE intermediates** instruction is a good way to learn about this.

See also:
Symbol Resolution
H 16
C 10-36

In the following examples, every effort has been make to select host application programs (controllable by ARexx) which are of general use and proven popularity. Also, the ARexx control programs themselves have been carefully chosen actually to do something useful for any owner of that application. By means of studying specific examples, you will soon be capable of writing your own custom programs that do exactly what you want done.

The ADDRESS Instruction vs the ADDRESS() Function

Our first order of business is to understand how to deal with addresses in ARexx. There are an ARexx *instruction* and an ARexx *function* to

ADDRESS instruction
H 25
C 10-50 f

ADDRESS() function
H 52
C 10-93

manipulate addresses, and they are not to be confused with each other. Refer to *Hawes*, pages 25 and 52; and *Commodore*, pages 10-50ff and 10-93 for the documentation of the ADDRESS instruction and the ADDRESS() function, respectively. Simply put, the ADDRESS instruction is used to **assign** an address as the current host address; and the ADDRESS() function is used to RETURN the current host address string (to allow you to see which address is the current host address).

The SHOW() Function

SHOW() function
H 64, disk
C 10-115

WAITFORPORT
command utility
H disk
C 10-157

The previously mentioned SHOW() function may be used to see if the desired host address is actually available (i.e. is the program running?) There are also other ways to check for an available address using the WAITFORPORT AmigaDOS utility, as we shall see. Note that the ADDRESS() function takes *no arguments* inside the parentheses. The ADDRESS instruction is a keyword instruction and the syntax of its use determines how it behaves. There are four ways to use the ADDRESS instruction:

1. Recall that rexxmast maintains two addresses: the **current host address** and the **previous host address**. The ADDRESS instruction by itself acts like a toggle to switch back and forth between the current and the previous host address.

Host Address
H 25, 43 f
C 10-50
C 10-75 thru 10-78

2. The ADDRESS instruction followed by a literal string (in quotes) or symbol (no quotes) specifies the new current host address as the *literal* string or *literal* symbol; and places the old current host address to be the *previous* host address. *Remember that host addresses are case sensitive.* If the host address is of mixed case, then you must use quotes around the address if you use this form of the ADDRESS instruction. Rexxmast will interpret any *mixed case* symbols into upper case! For example: If your host address is literally **PortName**, then ADDRESS 'PortName' will access that port, but ADDRESS PortName will not, because rexxmast will interpret the symbol **PortName** as **PORTNAME**, and will not be able to find the address. Note that if you use a symbol, it is *not evaluated* as a variable unless you use the

VALUE subkeyword to **ADDRESS** instruction
H 25
C 10-50 f

(Same reference for other options below)

subkeyword VALUE before it (see No. 4, below).

In the case of TurboText, it names its global address as TURBOTEXT, so either ADDRESS 'TURBOTEXT' or ADDRESS TURBOTEXT will correctly find the address (as will ADDRESS *turbotext,* since rexxmast will change the undefined symbol to uppercase). We belabor this point, because many applications do not inform the user correctly about the *real name* of its message port (address). You will do well to verify the actual host addresses by running a short ARexx program *while the application of which you wish to verify the address is running.*

```
/* A.rexx Check the addresses of currently running
programs */
SAY SHOW('P')
EXIT 0
```

3. If you have but one single command to issue to an address different from the current host address, you may issue it with the form: ADDRESS {*string* | *symbol*} *expression*, which is taken to mean that the string or symbol specifies the address (exactly as in 2. above) to which the *command* evaluated from the *expression* is sent. Neither the current nor the previous host addresses change if you issue a command with this syntax. For example, if you are in one window opened by TurboText, with an address 'TURBOTEXT3' (the current address), and you want to execute the single TurboText command MOVESOF to move the cursor to the start of file in another open TurboText window with an address 'TURBOTEXT2', and you do not want to change the current or the previous host address, then you issue the instruction: ADDRESS 'TURBOTEXT2' 'MOVESOF' in you program, and that single command would be executed at that address. Suppose, however, that the command is dependent upon the value of a variable, call it **comnd**. Suppose **comnd** were assigned previously as **comnd='movesof'**. Then the line **ADDRESS 'TURBOTEXT2' comnd** would accomplish the same thing, as the *expression* (variable) would be evaluated before sending. Complex command strings may be built using this technique.

4. The fourth way is to follow the ADDRESS instruction with the

subkeyword VALUE followed by an *expression* to be evaluated which specifies the new current host address. This is useful in cases where a host application generates multiple host addresses and ports. A good example is a text editor like TurboText which opens a new port with its own unique address 'TURBOTEXT0', 'TURBOTEXT1', etc., every time it opens a new edit window. In order to access a particular document in an open window with ARexx, you may need to compute the host address of that specific window 'TURBOTEXTn' where n is some integer, rather than use the TurboText global port name 'TURBOTEXT'. Using this syntax reassigns the current host address to the result of the evaluated expression, and places the former current host address into the previous host address. You can store the name of the port address in some variable, call it **docname**, and return to this window later on by means of the instruction **ADDRESS VALUE docname**. Do not confuse variables or expressions with literals! The instruction ADDRESS docname *will not work*, because rexxmast will look for an address called 'DOCNAME'.

Tokens
H 11 ff
C 10-27 ff

Symbol Resolution
H 16
C 10-36

The Current, Previous, and COMMAND Host Addresses

ADDRESS instruction
H 25
C 10-50 f

Note that the current and the previous host addresses are preserved in the ARexx storage environment. This means that they will not change if you call an internal function. Whenever the previous address is replaced by the current address as in cases 2 and 4 above, the former previous address is lost. There is one special host address: COMMAND. ADDRESS COMMAND sends the AmigaDOS command which follows it to the underlying DOS environment ('always put your AmigaDOS command strings in quotes; "nested" quotes are allowed'). For example, the line

```
ADDRESS COMMAND 'copy RAM:myfile to WORK:Stuff'
```

will copy the file **myfile** in **RAM:** to the **WORK:Stuff** drawer of your hard disk from inside an ARexx program. You may not see it for how powerful it is, yet, but ARexx can evaluate an *expression* to become an address or a command. Since any part of the final command string may be computed within the ARexx program, you have the power to create some very sophisticated command strings and addresses! For instance, you may want to split up the above command to *compute* the command string instead of specifying it absolutely. Maybe which file you copy is dependent upon something else in your ARexx program, so you assign the string value of the file to a variable called **copyfile**. If **copyfile='myfile'**, then the expression: 'copy RAM:'**copyfile** 'to work:stuff' (note quotes) is *first* evaluated by ARexx into the final command string you see in the example above, and *then* sent to the COMMAND host address. Addresses as expressions work the same way: first they are evaluated, then sent.

Writing a TurboText ARexx Macro

Whenever we write an ARexx macro for any host application, we need to determine the answers to the questions: Which programs will be up and running? Do we need to start another program? Do we need support libraries? What do we want the program or the macro to do? Which host application do we want to do which operations on our data? Since there is a great deal of variation among ARexx host applications, each having its own idiosyncracies in the implementation of its internal command set, our greatest challenge is figuring out in which programs to manipulate our data, and how best to program our procedures in the command sets of those host application programs.

TurboText
by
Oxxi, Inc.
P.O. Box 90309
Long Beach, CA
90809-0390
(310) 427-1227

Uniword.rexx
See Page 5-22

Let's write a program to do something in TurboText. We have already written a program, **Uniword.rexx** to remove and alphabetize the word list from a text file. We will now write an ARexx macro to do the same thing from within TurboText. We will observe some useful ways to adapt code from a previous exercise, and how to use the internal command set of a host application to its best advantage, rather than blindly porting a former program over to a host application. The comparison between our previous "stand-alone" program and this macro running from a host application will provide some useful insights into ARexx coding techniques, and will assist us later when we get into true interprocess control programming, the next level of complexity. Each level builds upon the previous level.

To answer some of the main questions: Let's say we have a text file loaded in TurboText with a window open. We want to list and alphabetize all the words in this file in a new TurboText document window, ready to edit or save. Therefore, we will design the program to run as a TurboText ARexx macro, to be run from the TurboText window host address. Here is an outline of what we want to do in pseudo code:

Stems and **Compound** symbols

H 21 f
C 10-44 ff

1. Move to the beginning of the document. Use a similar boolean array **LISTED.** as we did before to test whether we have listed the word already. Qualify the array (stem symbol) with nodes composed of words, to make boolean compound stem symbols. Get each word in turn and write it into an array of words with integer nodes if the boolean array says OK. Determine some way to detect the end-of-file in our document so that our array writing loop will terminate.

Shell Sort
see page 5-10 f

2. Sort the array of words alphabetically using the core code of the Shell sort directly on the array.

Note: In the following program, **TurboText** ARexx commands and replies are in **bold face** type to aid in readablilty.

3. Open a new document window. Do a loop to write each word in the sorted word array on a separate line. Exit.

Here is the program listing. Note the file extension .ttx to identify our program as a TurboText ARexx macro:

```
/* uniword.ttx Get an alphabetized list of words from a document */
/* This block accomplishes step 1. of the pseudo code */
OPTIONS RESULTS

/* use to find end of file of doc */
SIGNAL ON ERROR

MOVESOF
ICONIFYWINDOW

/* extract word list */
LISTED.=0 /* initialize boolean array */
n=1
DO FOREVER
   GETWORD      /* this is the TurboText command and...   */
   word=RESULT /* RESULT is the reply to the command.    */
   word=UPPER(word)
   IF LISTED.word | ~DATATYPE(word,UPPER) THEN DO
      MOVENEXTWORD /* jump out if we get error if at end of document */
      ITERATE
      END
   LISTED.word=1
   list.n=word
   n=n+1
   MOVENEXTWORD /* jump out if we get error if at end of document */
   END

/* An error is generated when we hit the end of document;   */
/* program control jumps to this label at end of document   */
ERROR:


/* This block accomplishes step 2. of the pseudo code */
/* The Shell Sort. Note: this is lifted intact from former pgm */
listlength = n-1
span = 1
DO WHILE (span < listlength); span = span * 2; END
DO WHILE (span > 1)
   span = span % 2
   numpairs = listlength - span
   DO node = 1 TO numpairs
      nextnode = node + span
      IF list.node > list.nextnode THEN
        DO
        store = list.nextnode
        list.nextnode = list.node
        DO bubpos = node-span TO 1 BY -span WHILE (store < list.bubpos)
            nextnode = bubpos + span
            list.nextnode = list.bubpos
            END bubpos
        bubpos = bubpos + span
        list.bubpos = store
        END
      END node
END
/* the end of the shell sort of the word list array */

/* This block accomplishes step 3. of the pseudo code */
/* output to a new document window */
j=listlength
OPENDOC
newdoc=RESULT /* the new address is contained in the RESULT variable */
ADDRESS VALUE newdoc   /* Note use of ADDRESS VALUE for new address   */
MOVESOF
DO i=1 TO j
   INSERT list.i
   INSERTLINE
   END
EXIT 0
```

ERROR subkeyword for **SIGNAL** instruction
H 38 f
C 10-71 ff

TurboText Manual References are marked with a **T** and page number.

MOVENEXTWORD TurboText command
T A-34

Our code is commented to reflect the three pseudo code steps. The TurboText internal command set has *no specific command* to detect the end-of-file. Within the TurboText command set, we can move to the start of the document and get each word in turn within an endless loop, but how will our program know when we are finished getting words? We combine the fact that TurboText generates an error message if we try to move the cursor past the end of the document, with the ARexx interrupt capability of branching by means of the SIGNAL ON ERROR interrupt instruction, to pass the program control out of the endless loop when all the words are exhausted. The instances of where these errors will happen (the MOVENEXTWORD commands) are commented, as is the label (ERROR:) to which the program will branch once an error occurs.

The RC Special Variable

RC special variable
H 39, 73, 75
C 10-73, 10-139
C 10-145

WHILE iteration specifier for **DO** instruction
H 27 f
C 10-53 ff

Note that any error codes are contained in the special ARexx variable **RC**. In some cases, you may want to test the variable **RC** directly, rather than using the SIGNAL ON ERROR instruction. **RC** is non-zero when an error occurs, and the exact value of **RC** is usually is the severity level of the error. If you merely wanted to exit a loop when you hit the document boundary, a **DO WHILE RC=0** loop would work fine, because **RC=5** if you hit the document boundary, and **RC=0** for all successful commands. Since we want to jump farther than just the end of a loop, the SIGNAL ON ERROR is a better choice here.

The Current Host Address is Where the Program Starts

TURBOTEXT Global and **Document** ports
T 10-1 f

Since we are running this program as a macro launched from TurboText, then the current host address will automatically become, for instance, 'TURBOTEXT3' or whatever the sequential number of the open document window is. TurboText maintains a *global* ARexx port address, 'TURBOTEXT', as well as individually numbered ARexx ports for every open TurboText window. These numbers make each port name unique within an operating session. Even if you close a window, its port name is never used again until you reboot. In this case, we are dealing with a specific window, which has a unique, numbered port. In another exercise, we will demonstrate the use of the global port 'TURBOTEXT'. At any rate, we know that the program starts at the address of the TurboText window containing our text file. Since we are at that address, we remember that *commands* will make sense at that host address, so it is *not* necessary, to put quotes around these TurboText-specific commands. We may use them transparently as we would any other ARexx instruction. (In a later example, we will see that sometimes we *do need to use quotes* around certain commands).

MOVESOF TurboText command
T A-35

ICONIFYWINDOW TurboText command
T A-27

MOVESOF moves our cursor to the Start Of File, and ICONIFYWINDOW makes the document window small so that the program will run faster, not having to redraw the screen at every cursor move.

Using TurboText Commands Saves Some Steps

RESULT special variable, see **CALL** instruction
H 26
C 10-53

RESULTS subkeyword of **OPTIONS** instruction
H 32
C 10-62 f

RESULTS processing
H 29, 37
C 10-58, 10-70

To extract words from the document, we will not have to read a whole line and then parse it into words. TurboText has its own parser with the built-in capability to get each word in the document directly and separately. An added bonus is that we don't have to worry about punctuation, as TurboText automatically strips off all punctuation and returns only the pure word. *That* saves a few steps! Note the syntax of the TurboText commands: First the command, then the RESULT variable comes back containing the information returned by the command, a little like the way an internal function returns a value in the RETURN instruction, except that you must always remember to assign a variable to equal RESULT explicitly and immediately, as RESULT will change with the next command. In other words, don't forget to read your mail!

Arrays see **Stems** and **Compound** symbols
H 21 f
C 10-44 ff

GETWORD TurboText command
T A-26

UPPER() function
H 67
C 10-121

A Boolean Array We Have Seen Before

We start the section on extracting the words (pseudo code step 1.) by initializing a boolean array (LISTED.=0) so that every member is 0 or false, then we set our word array counter (n=1), and enter the DO FOREVER loop to get the words. First, we GETWORD (a TurboText command) and assign the RESULT to **word**, our temporary word variable. Then we transform it to UPPER case, because our Shell sort is case sensitive, but we desire a uniform list.

IF instruction
H 29 f
C 10-58 f

Boolean operators
H 13, 21
C 10-30, 10-43 f

DATATYPE() function
H 56
C 10-101 f

ITERATE instruction
H 30
C 10-60

Hit Document Boundary TurboText error 7
T B-1

The first IF block warrants some discussion. We do two things at once: Test to see if EITHER the word has been listed before (as we did in the **Uniword.rexx** program); OR the DATATYPE() of the word is *not* UPPER case (we don't want non-alphabetic "words"). In either case, we do not want the word to become part of the word array, so the instruction block directs the program to move to the next word (MOVENEXTWORD) and then to ITERATE the endless loop. Because we have turned on the ARexx ERROR interrupt by means of the SIGNAL ON ERROR instruction, then if the MOVENEXTWORD instruction generates an error by hitting the document boundary, then program control will jump to the ERROR: label and remain running, rather than terminating. This is an example of the power of ARexx interrupts to do more than simply debug your programs. In the event **word** safely fails the IF test block, we assign the boolean element LISTED.word=1 to signify that we have now listed this word and all its subsequent appearances will be trapped by the IF test.

Building the Array of Words

END instruction
H 29
C 10-57

Initializer Expression in **DO** instruction
H 27 f
C 10-53 ff

The next step assigns a new word list array element **list.n** the value of **word**. Then we increment **n**, and move to the next word (MOVENEXTWORD), where, again, it is possible for the program to jump out of the loop if that command tries to move the cursor past the end of the document. The END instruction finishes up step 1. of our pseudo code intentions. We now have an array **list.** containing **n-1** elements, representing the values of all the words (in upper case; not including numbers) in our document. We are ready to sort this array.

Sorting the Array Directly

Since we have a handy, ready made array, we have only to lift intact the core code of the Shell sort routine and insert it here. Now you know why we went to such pains to make the input and the output of the Shell sort modular. We simply pull out the central routine of the Shell sort and use it directly. Because we chose in this macro to name our array **list.**, we do not even have to rename any variables! All we have to do is assign the variable **listlength=n-1** to get the correct number of elements, because as usual, the counter was incremented once too often. We omitted the SAY screen remarks from the old program as well, as there is no console in this macro program to display remarks. Finally, we need to output the sorted **list.** array to a new window, and accomplish step 3. of the pseudo code.

OPENDOC TurboText command
T A-37

VALUE subkeyword to **ADDRESS** instruction
H 25
C 10-50 f

Start with **j=listlength**. Then we OPENDOC which is a TurboText command which returns the new window's host address in the RESULT variable, which value we assign to the variable **newdoc**. Now, for the first time since we started the program, we change to another host address with the ARexx instruction ADDRESS VALUE newdoc. Note that we need ARexx to *evaluate* the variable newdoc, so we must use the VALUE subkeyword, or we will get a "host address not found" error message. This is because the ADDRESS instruction is looking for a literal 'surrounded by quotes' as an address. The VALUE subkeyword makes the variable **newdoc** into a literal string representing the address of the new document window.

MOVESOF TurboText command
T A-35

INSERT TurboText command
T A-28

INSERTLINE TurboText command
T A-28

Inside the New Window

Now at the new address, the new open TurboText window, the program moves to start of file (MOVESOF), then enters a loop for **j** iterations, in which TurboText uses INSERT list.i and INSERTLINE to insert each array element in turn and then add a new line after it for the next entry. We have accomplished all the goals of our pseudo code, and now have a way to extract an alphabetic word list from our TurboText editor. It turns out that it was easier to use the power of TurboText *with* ARexx to accomplish our task than to use ARexx alone!

An Example of ARexx Interprocess Control

Electric Thesaurus
by
SoftWood, Inc.
P.O. Box 50178
Phoenex, AZ 85076
(602) 431-9151

Souping Up TurboText with the Electric Thesaurus

In the last section, we made an ARexx macro to run within only one host application program. Now we will increase the complexity just a little and write a true interprocess control ARexx program. Since the last example used two addresses, it effectively contained all the aspects of an interprocess control program, so it will not be too difficult to explain how to make an ARexx program that accesses two different host applications. In this example, we will continue to use TurboText, but we will add power to it by incorporating the Electric Thesaurus (ET) by Softwood, Inc. By means of an ARexx program, we can add to our editor an on-line *Roget's Thesaurus* to assist us when we are composing text in TurboText.

Working with a Thesaurus

Word processors frequently have an on-line thesaurus that allows you to substitute an alternate word whenever you want. They work upon the word under the cursor at the time you call up their thesaurus, and open a window with a list of alternate words from which you can select. They give you the choice to cancel and retain the original word, or to substitute some alternate word for the one under the cursor in your document. When you close their thesaurus window, your document is updated to reflect your choice, or left the same if you cancelled.

Pseudo Code

This ARexx program is designed to emulate the above procedure. We will let the above discussion mold our pseudo code, as it defines adequately what we wish the ARexx program to do. We simply need a way to pass the word under the cursor in a TurboText document to ET (first opening ET if it isn't running already); and a way to return our choice from ET to our document and make the substitution. This little program is somewhat challenging because ET has such a limited ARexx command set, that you cannot do much of anything except get the word in the ET string gadget, or look up another word. There are no commands to get the port name, for instance, or to open a console or a requester.

Electric Thesaurus Manual Referencesare marked with an **E** and page number.

Electric Thesaurus ARexx commands **E** 2-8

This is not that inconvenient, however, as ET is a window based, mouse and menu driven program, and we will implement our interprocess control so that the ARexx program just waits for you to finish what you are doing in the ET window, and when you quit by closing the window, the ARexx program will get what is in the ET string gadget and substitute it for the word in the TurboText document. The

REQUESTBOOL
TurboText command
T A-41

Note: In the following program, **TurboText** commands and replies are in **bold**, and *Electric Thesaurus* commands and replies are in *italics*.

only caveat is to make sure that the replacement word you want is in that string gadget before you close the ET window, because that is the word that you may substitute. Our emulation of cancelling is to open a boolean requester in TurboText asking if we want to make the replacement. It is a small window with a OK and a Cancel gadget, and we will program the requester strings to show our old and replacement words.

Here is the program listing for **Th.ttx**, a Thesaurus for TurboText:

```
/* Th.ttx Thesaurus for TurboText */
SIGNAL ON ERROR

/* note: you must have rexxsupport.library loaded! */
Lib='rexxsupport.library'
IF ~SHOW('L',Lib) THEN CALL ADDLIB(Lib,0,-30,0)

OPTIONS RESULTS

/* inside TurboText */
GETWORD
word=RESULT
GETPORT
docaddress=RESULT
SCREEN2BACK
IF ~SHOW('P',ETHES_1) THEN DO
   /* run Electric Thesaurus. Change path to YOUR PATH! */
   ADDRESS COMMAND "run work:thesaurus/ET WB"

   /* wait for ET window port */
   ADDRESS COMMAND "WAITFORPORT ETHES_1"
   END

/* current address ET window */
ADDRESS ETHES_1

/* bring WB screen, window to front */
SCRTOFRONT
WINTOFRONT

/* look up word from TurboText in ET */
LOOKUPND word

/* wait for user to close ET window or quit */
DO FOREVER
   IF ~SHOWLIST('P','ETHES_1') THEN LEAVE /* if user quits */
   IF SHOWLIST('P','ETHES_1') THEN DO
      GETWORD
      newword=RESULT
      CALL DELAY(30) /* delay so program doesn't hog CPU cycles */
      END
   END

/* something went wrong? program branches here. */
ERROR:
/* back to TurboText */
ADDRESS VALUE docaddress
SCREEN2FRONT

/* keep the replacement string and make a "quoted string", too */
replacement=newword
newword='"'newword'"'

/* make a "requester string" from an expression */
/* the " quotes are for TurboText and the ' quotes are for ARexx */
insertstring='"<'word'> with <'replacement'>?"'

/* Ask if OK to insert word(s) */
'REQUESTBOOL "Thesaurus: Replace..."' insertstring
answer=RESULT

/* substitute word in text */
IF answer = 'YES' THEN REPLACEWORD newword
EXIT 0
```

rexxsupport.library
H 127 ff, disk
C 10-129 ff

Note: Library names are case sensitive!

ERROR subkeyword for **SIGNAL** instruction
H 38 f
C 10-71 ff

TURBOTEXT.LASTERROR
TurboText error variable
T 10-4

Notes on the ARexx Thesaurus Program

Because of the limitations of the ARexx interface in ET, we need to use ARexx itself to take care of some of these shortcomings. We will need to use the ARexx **rexxsupport.library** in an important way. Even though we recommended that you load this library in your startup sequence, we code a section to load it if you haven't, just in case. We use another example of the ARexx ERROR interrupt to make the program do an orderly exit if in fact this library cannot be loaded. First we turn on the error interrupt as before (SIGNAL ON ERROR). Then we include some familiar code to load the support library if it's not there. If the program cannot load the library for some reason, then it signals the ERROR: label near the end of the program and then you have the opportunity to CANCEL in the requester box. If anything goes wrong and returns an error, then its message will display in the title bar of TurboText. TurboText also maintains a special variable called TURBOTEXT.LASTERROR, which you may access like any other variable for further information.

RESULTS subkeyword of **OPTIONS** instruction
H 32
C 10-62 f

GETWORD TurboText command
T A-26

GETPORT TurboText command
T A-22

SCREEN2BACK TurboText command
T A-45

Get the Word Under the Cursor

After the request to ARexx to ask for a return of RESULTS, we enter the main program, started from an open TurboText document window with its own unique host address. Following our model, we get the word under the cursor with GETWORD, and assign **word** to the RESULT, just as we did in the last program. Next we GETPORT and store the address name of our document window in the variable **docaddress**. Then we put the document window to the back with a SCREEN2BACK command (and this command doesn't return any RESULT, of course). We now have our word safely tucked away in an ARexx variable, and are ready to launch ET.

COMMAND special address for **ADDRESS** instruction
H 44
C 10-76

1 This is sometimes called **asynchronous** launching of a program.

Details of Launching ET

If ARexx doesn't show the existence of the first ET window port, the program enters a block of instructions where we use the ADDRESS COMMAND special address to launch ET. *Note that you must change the path name in the program listing to reflect where you keep your ET program!* Also you must use the run command to launch ET, or it will *take over* the process that launched it, and although it will open a window, it will not let the ARexx process continue until you close the window. Run allows the parent process to continue[1]. These are small details but necessary to our success.

Electric Thesaurus ARexx commands
E 2-8

WAITFORPORT command utility
H disk
C 10-157

ET Options and Finding Its Port

The WB option means that ET opens on the WorkBench screen. There are several options in ET which you may use instead, such as opening a custom screen rather than a WB window. After we have launched a program, we cannot guarantee that if we next try to access its public message port, that we will succeed. Sometimes a program takes several seconds to load and open a window, a much slower process than executing the next ARexx instruction. We therefore must *find* the port before we try to access it! The ARexx implementation comes with what is called a **DOS utility** named **WAITFORPORT** (put it in your C: directory) which is a *DOS command* which does what its name implies: It waits for a port of some given name (for a maximum of 10 seconds).

In our program, after we launch ET, we know that the first window it opens is named **ETHES_1**, so by means of another ADDRESS COMMAND, we tell the program to **waitforport 'ETHES_1'** which it does. As soon as **ETHES_1** comes up, the program continues. If **ETHES_1** was found in the first place (ET was already running), this whole block is ignored.

Is the Port Name Really the One?

A short digression: We mentioned before that sometimes a host application program manual does not correctly inform you of the real name of its public message port, and ET is such a program. The ET manual, on page 2-9 states that the port is named **"EThes_1"** in the first window ET opens. We must conclude that the writer of that manual did not understand how ARexx interprets an address *string* as opposed to an address *symbol*, because in the text, the name **"EThes_1"** is inconsistent with the examples a little further down the page in which they use an address instruction as **Address EThes_1 "LOOKUP esteem"**. Since they failed to use quotes around the address *symbol*, **rexxmast** correctly interprets the *wrong* name as the *true name*, **ETHES_1**, an example of two wrongs making a right. This is meant to warn you to be aware of the pitfalls and confusion that quotes, strings and interpreted symbols can present. *The first thing, always check a new program for the actual port name no matter what the manual says!*

To check the **port names** of your running programs, use **A.rexx**. See page 7-5.

Look Up the Word in ET

The next thing, we set the current host address to **ETHES_1**. Then we make sure that the WB screen and the ET window come to the front using two ET commands SCRTOFRONT and WINTOFRONT. Now we are ready for the program to look up our word imported from TurboText. The ET command LOOKUPND looks up the word in the current window. Soon all its definitions and synonyms are displayed. Clicking the mouse pointer on one of the words in the window writes it to the string gadget. Clicking on the Find gadget pulls up *that* word's definitions and synonyms, and so on. Move around in ET until you get a word you'd rather have into the string gadget, or else get the old word back into the string gadget.

Electric Thesaurus
ARexx commands:
SCRTOFRONT
WINTOFRONT
LOOKUPND
E 2-8

SCREEN2FRONT
TurboText command
T A-46

REPLACEWORD
TurboText command
T A-41

evaluate the address string **docaddress**. We move the screen to the front again. Now we store the entire replacement string **newword** in a variable **replacement** for later use, and put some double quotes around the variable **newword** for use by TurboText. Why? In some cases, a synonym is two or more words instead of just one; for example the word "selected" has a synonym "singled out". In order to use the TurboText command REPLACEWORD correctly, we can use only one subkeyword or else if there is more than one word, they must be a "string contained in double quotes" ('single quotes' won't do!).

The OK/Cancel Requester

REQUESTBOOL
TurboText command
T A-41

TurboText has the capability of opening a boolean requester to ask if it is OK or not to do something. We use a REQUESTBOOL command to accomplish this. The REQUESTBOOL command takes two subkeywords only: a title string and an insert string. In our case, we want to put the value of an expression into this requester, so we build up the **insertstring** from its component parts, making sure that in its final form it is a "string surrounded by double quotes". Again, note that TurboText likes "double quotes" and won't work with 'single' quotes, which *are* used by ARexx, however. *This is not covered in the TurboText Manual.*

Building Up the Insert String

Expressions
H 16
C 10-35

Symbol Resolution
H 16
C 10-36

Command Console in
TurboText
T 10-2

The insertstring expression is built up a piece at a time and it contains **word** and **replacement** as values of variables. The rest of the expression is taken as string literals surrounded by the ARexx single quotes. Quoting is one of the areas that produces frustration until you get a handle on how the ARexx interpreter works as well as what kind of strings a host application like TurboText is looking for. The compact and simultaneous combination of ARexx with an internal command set is powerful, but it can be a little confusing at times. When you try the program, it will become clear how these strings are implemented. Also you may experiment with the TurboText command console which is handy for trying out one command at a time. It displays the RESULT variable. Much of this example program was developed using the TurboText command console to test ideas and command string syntax.

Quotes: Powerful but Tricky

The next command is the REQUESTBOOL command; note the way we use quotes. This is the example we mentioned before in which you **must use quotes** around your command. The ARexx interpreter, **rexxmast**, reads the entire line as a command and sends it, including the double quotes (") to the TurboText window port. If we did *not* put single quotes around the first part of this command, then rexxmast would send the wrong number of subkeywords to TurboText and *not include* those essential double quotes as literals. Once again, the single quotes serve to tell rexxmast how to build up the expression into a literal string (to send as a command) by delimiting when to start and when to stop treating character symbols as literals.

Tokens
H 11 ff
C 10-27 ff

When a character is not part of a literal it is of course part of some symbol token, so you may think of the single quotes as toggles which turn on string when they turn off symbol and vice versa. The double quotes are *necessary within TurboText* so they are part of the literal strings. Without single quotes around the first part of the command, TurboText would think that **Thesaurus:** was the first subkeyword and **Replace...** was the second; and the presence of **insertstring**, too, would *not work* in TurboText, causing an "unknown command" error! The reason for this is that ARexx recognizes and honors *both* single and double quotes. Whichever kind is outermost, will determine their treatment as a delimiter or part of a literal.

INTERMEDIATES
option of **TRACE**
instruction
H 40
C 10-135 ff

If you are in doubt about whether a command works with or without quotes, use the quotes carefully. An interactive trace of the intermediate results is an especially good way to check to see if something is getting interpreted correctly or not, if the host application does not have a command console. In this case, if **word='specified'** and **newword='pointed out'**, then the exact string sent to TurboText by REQUESTBOOL is

```
REQUESTBOOL "Thesaurus: Replace..." "<specified> with <pointed out>?"
```

Thus, there are only two "subkeywords" to the command, each surrounded by double quotes. Notice how the single quotes disappear once the string is constructed. We chose the < and > characters to delimit our choices. The boolean requester returns a 'YES' if we click on the OK gadget and a 'NO' if we click on the Cancel gadget. If we get a 'YES', we replace the original word with the "quoted evaluated string" of the synonym **newword** using the REPLACEWORD command. Finally, the program EXITS and we are finished!

REPLACEWORD
TurboText command
T A-41

Your editor, TurboText is all ready, thanks to the Electric Thesaurus for you to write the great American novel, and you are also solidly at the intermediate stage of ARexx expertise. You may even have entered the relatively advanced stage if none of your relatives understand what you are talking about now.

A More Complex Interprocess ARexx Program

Proper Grammar (PG)
by
SoftWood, Inc.
P.O. Box 50178
Phoenix, AZ 85076
(602) 431-9151

Manual References
marked by
P and page.

TurboText Controlling Proper Grammar

We have stated before that one of the hardest challenges to the ARexx programmer is to work with and around the idiosyncracies of a host application ARexx interface, but what do you do when there are too few commands to work with? In the case of Proper Grammar, also by SoftWood, Inc., we must overcome severe limitations in order to do anything at all with it and ARexx. If Proper Grammar had any fewer ARexx commands in its internal set, then it would be absolutely useless with respect to controlling it via ARexx. Nevertheless, the challenge of making a useful ARexx controller for Proper Grammar (PG) is an excellent learning experience, and we will end up with a viable and automatic ARexx routine to add the power of PG's grammar and spell checking to the already awesome flexibility of TurboText.

rexxsupport.library
H 127 ff, disk
C 10-129 ff

rexxutil.library
by David Junod
from **BIX** bbs

We will need to make use of two ARexx libraries (sometimes called **function hosts** in ARexx jargon): **rexxsupport.library** and **rexxutil.library**, both mentioned earlier. This example is rather complex, as it makes use of two ARexx host application interfaces and two function hosts, and needs to call an exterior ARexx program as well. What we want to do is simple, but the implementation is difficult, or at least tricky. Here is the pseudo code.

Pseudo Code for PG from a TurboText Document

1. Assumptions: A TurboText document window is open and a text document is loaded. The program, called **PG.ttx** is launched from the host address of the TurboText document window. Proper Grammar is not running, but has certain specified LOAD and SAVE preferences preset. In case PG happens to be running, the program will replace the text in PG's *first* open window: with address **PGRAM_1** (As in the case of ET above, the PG ports are also mis-named in the PG manual).

2. If a block is selected in the TurboText window at the time of launching **PG.ttx**, then the selected block is cut and loaded into PG for analysis and editing. When the user quits PG, the program control returns to TurboText where a requester gives the option to replace the old text with the new, or to cancel and put back the old text. In PG, we may answer "NO" to the "Save Changes made to document?" requester and safely return to TurboText with the new text, or we can save the new text into another file and then exit. **PG.ttx** should allow use of all edit functions in PG including the clipboard cut and paste without affecting the final results.

3. If a block is *not* selected when the program is launched, then the entire document is cut and loaded into PG for analysis and editing, and passed back with exactly the options mentioned above in step 2.

4. The original format of the document in TurboText will be preserved as much as possible, by use of TurboText commands, PG startup preferences (for ASCII loading and saving formats); and a TurboText routine to reformat all paragraphs after the text is returned in the format of *one long line per paragraph*. The text inside PG, although readable, will by necessity *not be identical* to the format in TurboText, and the user should not reformat line lengths inside PG, but only edit and analyze the text. The program will restore the original line lengths to the TurboText document. New paragraphs may be added successfully inside PG, however.

5. The clipboard device is used to communicate to and from PG. Since

the PG ARexx command set is devoid of commands to cut or paste from the clipboard, or to save or load files by name, then we must use the **rexxutil.library** *function host* to read and write from/to the clipboard. PG's only ARexx commands that are useful to us are those that replace the entire text of an open window with some other text, and another command to get the entire text from a PG document window in the preset ASCII format, and that exhausts about half of the usable PG ARexx command set!

We have made our pseudo code very general this time, almost as though it were a *software specification* instead of pseudo code. This is because we want to keep our goals in sight, but we are not yet sure just how we will code the program. At this point we know only that we will use the clipboard as the holding device for text passed from one program to the other, and that we want to preserve the original format of the document.

A useful programming technique is to construct the steps that you are *sure* you know how to do and go from there. This is the actual way that this program was developed, and to illustrate the technique, we will follow the original thinking, and build up the code a piece at a time to end up with the final listing, rather than first list the code and then explain it. This will replicate the programmer's reality better. Let's first get an open document in TurboText over to Proper Grammar, and then we will worry about other things.

Note: Library names are case sensitive!

```
/* PG.ttx Text block to Proper Grammar */
SIGNAL ON ERROR

/* note: you must have rexxsupport.library loaded! */
Lib='rexxsupport.library'
IF ~SHOW('L',Lib) THEN CALL ADDLIB(Lib,0,-30,0)

/* note: you must have rexxutil.library loaded! */
Lib='rexxutil.library'
IF ~SHOW('L',Lib) THEN CALL ADDLIB(Lib,0,-30,0)

OPTIONS RESULTS

/* the following commands were added later for format
purposes to return the cursor to its former position,
convert tabs to spaces, and set wordwrap on. */
```

GETCURSORPOS TurboText command **T** A-19

CONV2SPACES TurboText command **T** A-8

SETPREFS WORDWRAP ON TurboText command **T** A-51

GETPORT TurboText command **T** A-22

GETBLKINFO TurboText command **T** A-17 f

MOVESOF TurboText command **T** A-35

MARKBLK TurboText command **T** A-30

MOVEEOF TurboText command **T** A-32

CUTBLK TurboText command **T** A-10

READCLIP() rexxutil.library function See which docs.

EXECTOOL NAME TurboText command **T** A-14

REPLACETEXT Proper Grammar command **P** 8-1

```
GETCURSORPOS
pos=RESULT
PARSE VAR pos line column .
CONV2SPACES
SETPREFS WORDWRAP ON

GETPORT
portname=RESULT
GETBLKINFO
info=RESULT
PARSE VAR info blk .
IF blk='OFF' THEN DO
   MOVESOF
   MARKBLK
   MOVEEOF
   END
CUTBLK
clip=READCLIP(,VAR,,0)

/* STOP!!!! NOTE!!!! */
/* Modify the following instruction to reflect your
actual Proper Grammar path! */
EXECTOOL NAME "Run Work:Grammar/Proper_Grammar"
ADDRESS COMMAND "WAITFORPORT PGRAM_1"
ADDRESS 'PGRAM_1'
REPLACETEXT clip
```

If we run this code, we will successfully open PG with the selected block or the entire document loaded into PG depending upon whether a block was selected or not in the original TurboText window. We model much of this starting code upon the last example for ET. We check for two libraries and load them if they are not already loaded, after turning on the SIGNAL ON ERROR interrupt which we will need in the same way as in the **Th.ttx** program. We of course want to get results so we specify that OPTION. The next commands were added later for format purposes and will be discussed later.

Next we get the port name of our open window and assign it to the variable **portname**. The next instruction, GETBLKINFO is the way we use the TurboText ARexx command set to determine whether there is a block selected or not. The RESULT variable carries a string which may be parsed. The first word of the RESULT string (which we have assigned to the variable **info**), is either **ON** or **OFF** to indicate whether a block is selected or not, so we simple parse (by forced tokenization) that string variable and assign the first word to **blk**. Then a simple IF block selects the entire document if there was not a block already selected. Inside the IF block, the TurboText commands move to the start of the document (MOVESOF), mark the block (MARKBLK), and move to the end (MOVEEOF), effectively selecting the entire document. The next command (CUTBLK), cuts the block and places it into the clipboard.

A Support Library Function Reads the Clipboard

Then we use one of the **rexxutil.library** functions: **READCLIP()**. Notice the use of the commas. Each argument for **READCLIP()** has its place and since we do not use all of the arguments, we let the commas serve as their place holders. The argument options we *do* use are **VAR** to specify that we wish to read the entire clipboard into a simple symbol token (a variable); and the other, **0**, specifies the clipboard unit we wish to read (unit 0, the primary clipboard unit which TurboText uses). The clipboard unit was found out by experimentation with an example program that illustrates the use of **rexxutil.library**, and comes with this library file (downloaded from BIX). Refer to the documentation for **rexxutil.library** for more information about how to use its many other features.

rexxutil.library by David Junod from **BIX** bbs.

The **rexxutil.library** and documentation is included free on the Companion Disk to *The ARexx Cookbook*. See the copyright page to order.

Launch Proper Grammar

EXECTOOL TurboText command
T A-14

COMMAND special address for **ADDRESS** instruction
H 44
C 10-76

WAITFORPORT command utility
H disk
C 10-157

REPLACETEXT Proper Grammar command
P 8-1

Now we are ready to open Proper Grammar. Make sure you change the EXECTOOL instruction to specify *your* Proper Grammar path name! We use the TurboText EXECTOOL command, but you could also use an ADDRESS COMMAND instruction. We RUN the PG program to allow detach PG to run separately. Otherwise the ARexx program will not continue to execute. Then, exactly as in the ET example, we use WAITFORPORT to wait until the port name is available before we continue, and we then pass program control to the address of the *first* window of PG. We use one of the precious few PG ARexx commands REPLACETEXT to replace any text that might be in the PG window with the text we read from the clipboard and assigned to the variable **clip**. Note that we have freed up the clipboard for use by PG. The old text from TurboText is safely stored in the variable **clip**, and later we can use a **rexxutil.library** function to copy **clip** back to the clipboard for pasting into the TurboText window if we decide to cancel the replacement.

Inside Proper Grammar: Set the Preferences

Proper Grammar I/O preferences
P 5-7

Click on the *second* box in Input and Output Options.

Once we are in PG with our block or document loaded, we begin to notice problems if we have not pre-set the PG preferences a certain way. There is no discussion in the PG manual about what the ASCII input/output preferences actually do, so experimentation was necessary to find the best settings. There are two settings in input and two settings in output. *Only the second setting in each should be selected.* This makes a paragraph at each blank line for input, and inserts a blank line at each paragraph in the output. The first settings (which should be "off") have to do with line feeds. If they are selected, the text will look better in PG, but you will find serious problems with the grammar checker, as it will think that each line is in a separate paragraph, and will incorrectly find all sorts of non-sentence fragments and capitalization "errors".

With the settings specified as above, the text will look different in PG from its appearance in TurboText, but we will find a way to reformat it correctly when it returns. In the meantime, we don't worry about line feeds; we merely analyze and edit the text. PG has too many shortcomings to overcome this little annoyance, but it is better to have analysis capability than (temporarily) formatted text. We will now code a loop similar to the one we used for ET to insure that we capture any changes we make to the text during our time inside PG. Here is the loop which we append to the above code.

LEAVE instruction
H 31
C 10-60 f

SHOWLIST() support library function
H 129
C 10-133

GETTEXTPARA Proper Grammar command
P 8-1

DELAY() support library function
H disk
C no reference

```
/* wait for user to close PG window or quit */
DO FOREVER
   /* IF user quits, then the port name goes away. */
   IF ~SHOWLIST('P','PGRAM_1') THEN LEAVE
   /* IF user doesn't quit, port name is there...   */
   IF SHOWLIST('P','PGRAM_1') THEN DO
      GETTEXTPARA
      text=RESULT
      /* Delay so program doesn't hog CPU cycles */
      CALL DELAY(30)
      END
   END

ERROR:
```

This loop has much more than a slight chance of an error during its execution. Should you exit during the time it is executing the GETTEXTPARA command, which takes some time if a large document is loaded, then an error will occur, and we need to branch to the **ERROR:** label to retain control. The poor PG command set prevents us from making a better way to get the text from the window, as there is no facility for a requester or for saving or loading or writing *automatically* to the clipboard, yet our specifications were for an automatic program. This solution *works*, however inelegantly, and if you make sure to wait a second or two before you exit once your text is complete, the error interrupt in ARexx will prevent any hang ups of the program, and you will be sure that the program has grabbed your latest changes. All this loop does is grab the latest text from the PG window (every 3/5ths of a second), and store it into a variable **text**.

Return Text Formatting

WRITECLIP()
rexxutil.library function

This text is formatted by the PG pre-set preferences so that we return each paragraph of text as one long line; and paragraphs are separated by a blank line. We plan to reformat in TurboText which has a complete command set. Now we want to write **text** to the clipboard using a **rexxutil.library** function, WRITECLIP(). This is done by

```
CALL WRITECLIP(text,VAR,,0,,)
```

Search Order for
functions
H 47 f
C 10-85 f

which again uses commas to place-hold for arguments we don't need. The arguments specify that we are writing the variable **text** to the primary clipboard unit 0. Note that we can call a function from a third party *function host* exactly like we call ARexx official built in functions. If the relevant library is loaded, then the operation is *transparent* and we have effectively added some new functions to our function set. This demonstrates the *modularity* of ARexx. Now our old text is in the variable **clip** and our new text is in **text** as well as in the clipboard. We are ready to return to TurboText, and our last bit of code.

```
/* back to TurboText */
ADDRESS VALUE portname
SCREEN2FRONT

/* Ask if OK to insert word(s) */
'REQUESTBOOL "Proper Grammar:" "Replace old text with new?"'
answer=RESULT
/* substitute word in text */
IF answer = 'YES' THEN DO
   PASTECLIP
   CALL ForPar.ttx
   END
IF answer = 'NO' THEN DO
   CALL WRITECLIP(clip,VAR,,0,,)
   PASTECLIP
   END
/* restore the cursor postion from the original document */
MOVESOF
line=line-1
column=column-1
MOVEDOWN line
MOVERIGHT column
EXIT 0
```

PASTECLIP TurboText command
T A-39

MOVERIGHT TurboText command
T A-35

MOVEDOWN TurboText command
T A-32

SCREEN2FRONT TurboText command
T A-46

Back in TurboText

Now we change the address back to the TurboText window we left, and bring its screen to the front and open a requester to ask should the program replace our old text or not. If the answer is **YES**, then we paste the clipboard and reformat using an ARexx exterior function: **ForPar.ttx**. Speaking of reformatting, on several occasions, depending on the document, I found that the format was not preserved correctly when tabs were involved. The CONV2SPACES command fixed this. The other command in the first section that was added later was the SETPREFS WORDWRAP ON command which is necessary if we want the paragraphs brought back from PG as one long line to wrap properly when we reformat. You may of course, put in a command to turn off word wrap again at the end if you desire.

If the answer to the requester is **NO**, then we use the WRITECLIP() function from the rexxutil.library to write clip (the old text) to the clipboard and then paste to the TurboText window from the clipboard.

Since this was the original text, we need not reformat it. The final group of commands returns the cursor to its original position in the document before we called PG.

The Exterior ARexx Function to Reformat the TurboText Document

We want to use the ARexx interrupt SIGNAL ON ERROR to detect the end of the TurboText document in order to reformat it, and since we have already determined a use for the ERROR label already, we just make a new section as a separate ARexx program (and therefore a separate task) which will not interfere with the main program's use of the ERROR: label. Isn't multi-tasking *great*? Here is the short program which will reformat a TurboText document paragraph-by-paragraph. It, by the way, may be used by itself as a TurboText macro, and is the inverse of the ARexx macro **Documentize.ttx** that comes standard in the TurboText package. That's the main reason we made it a stand alone program. We call it **ForPar.ttx**:

Documentize.ttx
See **T** 10-5

MOVESOF TurboText command
T A-35

FORMATPARAGRAPH TurboText command
T A-17

MOVEDOWN TurboText command
T A-32

```
/* ForPar.ttx Format paragraphs in TT from PG */
OPTIONS RESULTS
SIGNAL ON ERROR
MOVESOF
DO FOREVER
   FORMATPARAGRAPH
   MOVEDOWN
   END
ERROR:
EXIT 0
```

This is a simple program which first moves to the start of the document, and then enters an endless loop terminated when we get an error by trying to move past the end of the document. This time, we could have used a DO WHILE RC=0 loop instead of the DO FOREVER with the SIGNAL ON ERROR instruction, and kept the routine inside the main program, but since this program is useful by itself, we code it consistent with the main program for clarity, and keep it as an exterior function.

Inside the loop a FORMATPARAGRAPH TurboText command formats the paragraph so that the line feeds are correct for the margins of the window. It doesn't matter whether the paragraph is one long line or not: the command will reformat it. Since there is a blank line between paragraphs, the MOVEDOWN command moves the cursor (left at the end of the last paragraph) down one line to the start of the next paragraph.

Note: In the listing, **TurboText** commands and replies are in **bold**, and *Proper Grammar* commands and replies are in *italics*.

The Complete Listing

When the program hits the document boundary, it generates an error and exits. Our document back from PG is now reformatted just as it was. Here is the complete listing for the program which adds the power of a grammar/spell checker to TurboText.

```
/* PG.ttx Text block to Proper Grammar */
SIGNAL ON ERROR

/* note: you must have rexxsupport.library loaded! */
Lib='rexxsupport.library'
IF ~SHOW('L',Lib) THEN CALL ADDLIB(Lib,0,-30,0)

/* note: you must have rexxutil.library loaded! */
Lib='rexxutil.library'
IF ~SHOW('L',Lib) THEN CALL ADDLIB(Lib,0,-30,0)

OPTIONS RESULTS
GETCURSORPOS
pos=RESULT
PARSE VAR pos line column .
CONV2SPACES
GETPORT
portname=RESULT
SETPREFS WORDWRAP ON
GETBLKINFO
info=RESULT
PARSE VAR info blk .
IF blk='OFF' THEN DO
   MOVESOF
   MARKBLK
   MOVEEOF
   END
CUTBLK
clip=READCLIP(,VAR,,0)  /* rexxutil.library function */
EXECTOOL NAME "Run Work:Grammar/Proper_Grammar"
ADDRESS COMMAND "WAITFORPORT PGRAM_1"
ADDRESS 'PGRAM_1'
REPLACETEXT clip
```

```
/* wait for user to close PG window or quit */
DO FOREVER
   IF ~SHOWLIST('P','PGRAM_1') THEN LEAVE /* user quits */
   IF SHOWLIST('P','PGRAM_1') THEN DO
      GETTEXTPARA
      text=RESULT
      /* delay so program doesn't hog CPU cycles */
      CALL DELAY(30) /* rexxsupport.library function */
      END
   END

ERROR:
CALL WRITECLIP(text,VAR,,0,,)  /* rexxutil.library function */
/* back to TurboText */
ADDRESS VALUE portname
SCREEN2FRONT

/* Ask if OK to insert word(s) */
'REQUESTBOOL "Proper Grammar:" "Replace old text with new?"'
answer=RESULT

/* substitute word in text */
IF answer = 'YES' THEN DO
   PASTECLIP
   CALL ForPar.ttx
   END
IF answer = 'NO' THEN DO
   CALL WRITECLIP(clip,VAR,,0,,)  /* rexxutil.library function */
   PASTECLIP
   END
MOVESOF
line=line-1
column=column-1
MOVEDOWN line
MOVERIGHT column
EXIT 0
```

TurboText Keyboard Commands
T 9-5 ff

EXECAREXXMACRO
TurboText command
T A-13

Now you can call your TurboText SuperCharged! If you wish, you may assign the ET program and the PG program to suitable keys in your TurboText definintion file. I assigned **Th.rexx** to Alt-T and **PG.ttx** to the Alt-G key . Open your **startup.dfn** file and change the keyboard command equivalents to:

ALT-T	EXECAREXXMACRO Th.ttx
ALT-G	EXECAREXXMACRO PG.ttx

Chapter 8: Using ARexx and PostScript Together

Use ARexx to Make a PostScript Driver for a Text Editor

An increasing number of Amiga users are discovering the benefits of using a PostScript capable printer, particularly since the price of such a printer is now below $2000. With PostScript at your disposal, you can print anything imaginable, except, that is, a simple text file from your favorite text editor! When you try to do this, you immediately realize that in one of their more inscrutable decisions, Commodore failed to include any sort of PostScript printer driver in the Preferences printer choices. To use a Preferences printer (your only choice on most text editors), you must select a Hewlett Packard LaserJet printer driver, and change the settings on your printer to LaserJet II emulation, a much less than satisfactory solution. It is easy to forget in which mode your printer is set. Attempt to print a file and blank pages start to spew out, or, worse, the printer writes garbage on your expensive laser paper. I soon tired of trying to keep track of two settings, and started thinking of a workaround. I wanted to print out program listings and documentation files from my editor without resorting to switch flipping and other such kludges. "There ought to be a simpler way", I thought to myself.

One of the reasons to buy a PostScript printer is that they print using software instead of hardware to make the type faces and fonts. This means you can export your files to other platforms for printing. One day someone demonstrated to me that PostScript is, after all, an interpreted script language. This means that programs that control every aspect of the printer originate in a text file made in an editor as ASCII text files; in other words, you can read them, just like ARexx programs. The printer may be controlled simply by copying a suitable PostScript text file to the Parallel (PAR:) device of your Amiga. Since ARexx is an interpreted, script language as well, I thought that perhaps PostScript would be as much fun.

The Evolution of a Good Idea

Sometimes the history of a programming idea provides creative insight. Here is the story about one such idea, at the end of which you will have some very useful print utilities if you use a PostScript printer. While playing around with PostScript programs in my TurboText editor, I soon came across the above mentioned problem. Here I was, making PostScript programs and having to *print* their listings, for Pete's sake, in LaserJet II mode! Most editors have the most rudimentary means of printing: they just send the file to the PRT: device, which contains your preferences printer choice. On the other hand, DTP programs and word processors usually have a PostScript driver on board, but you generally can't access them from an editor, and their drivers don't work if you simply load them into Preferences. If your word processor has ARexx and so does your editor, then you can send the file to the word processor, and print it, but that can be time consuming, to wait to start your word processor package up and then close it down each time you print. I also rejected the idea of giving up the handy programmer-oriented features of my editor to use my word processor and save in ASCII text format. I use WordPerfect (WordPerfect Corporation) which doesn't support ARexx, but I started my experiment making a startup macro within WordPerfect and used ARexx to start WP from within my TurboText editor.

A Flash of Insight

The WP macro loaded and printed a file in RAM with a certain name. In TurboText, I made the ARexx macro so that it saved its current document to that specific file name in RAM before it started WordPerfect. It worked, but the system overhead was not to my liking and I had to exit WordPerfect manually since it wouldn't allow a macro that shuts it down. WP macros are irksome in the extreme as they are only a clumsily implemented record of keystrokes. While I was finishing this test, the answer suddenly flashed to me: **Make an ARexx macro to write PostScript commands directly to the PAR: device!** Yes! Then I could use the *considerable* string handling power of ARexx to parse the lines of my document file and also make the PostScript commands. ARexx could put together a PostScript program to send to the PostScript laser printer connected to my parallel port. PostScript is clumsy when it comes to file and system manipulations but ARexx is not. ARexx isn't so hot at formatting and page layout, but PostScript is, so together the two are dynamite! The following program and tutorial will guide you through the makings of a PostScript line printer for your text editor.

How Does PostScript Work?

We will be juggling and combining, not just apples and oranges, but apples, oranges and bowling balls. I'll leave it to the reader to figure out which represents which in the following. Before we get into the nitty gritty of a program listing, it would be good to explain a little about how PostScript works. PostScript uses an interpreter, a program that takes instructions one at a time and executes them. This program isn't in your **C** directory or anywhere else in your Amiga. It resides inside your printer aboard a hardware chip called a ROM (Read Only Memory), usually in the form of an EPROM (Erasable Programmable ROM) so that the latest version of the PostScript language may be installed at the factory, or so that certain printer-specific parameters may be changed (by experts only!). It waits for an instruction it recognizes and then executes it.

The LIFO Stack or Postfix Notation

PostScript executes instructions on a **stack** which is a series of registers which may contain data objects arranged in vertical order. One easy way to visualize how PostScript executes an instruction is to think of operating an RPN (Reverse Polish Notation) calculator such as the popular Hewlett Packard series. PostScript functions the same way: on a stack. This operation mode is sometimes called **postfix** notation using a **LIFO** (last in first out) **stack**. This apparently is how the *Post* part of the PostScript name came about, and we've already shown how the *Script* part of the name came about.

In postfix computing, the **operand** (the data) is specified (**pushed on to the stack**) *before* the **operator**. The operator then takes the data off the stack, operates on it, and returns the result of the operation to the top of the stack. Confused? OK, think of your local cafeteria and the stacks of trays at the head of the line. They are on a spring-loaded device so that the uppermost tray is at a constant level and accessible to the customers. The last tray put on the stack by the dishwasher is the first one to go out and get used by a customer. That's exactly how to look at a postfix stack. The dishwasher represents the program putting objects (data) on the stack, and the customers are the operators taking trays off the stack and doing things with them. Sometimes a family of four comes in and needs four trays to eat on. Sometimes a single person comes in and needs only one tray.

PostScript Operators

Operators in PostScript are the same way: Some of them need several data objects pushed on to the stack in a certain order, and some operators need only one object. In PostScript the things on the stack are called **objects** and they don't have to be numbers; they can be entire dictionaries of fonts, definitions of functions, just about anything, even (roast beef). This is how you would put a **literal text string** object (roast beef) on the stack: Enclose it in parentheses. Parentheses are special characters to the PostScript language. All PostScript language objects may be represented by ordinary ASCII text and numbers; in other words all printable characters. Therefore they are prime candidates for string manipulation leading to PostScript program construction in ARexx!

Mixing PostScript Objects, ARexx Instructions, and ARexx Commands

Our apples, oranges, and bowling balls therefore correspond to PostScript language objects, ARexx statements, and ARexx commands. The only one of these three that is not universal or standard, is the internal command set of your editor. In the following, we look at a specific program to control the PostScript printer from TurboText, but from the context, you can easily change the code to match your favorite editor's ARexx command set. If your editor doesn't have ARexx support, then you can still do the printing, but you will need to modify the program to leave out the TurboText-specific formatting commands and launch the ARexx program from a shell after formatting and saving your file manually. The program here could be done entirely in TurboText commands without resorting to saving the entire file to **RAM:** first, but to make this application more universal, we will minimized the use of the TurboText command set. Using ARexx directly also proves easier to implement than using only the internal TurboText commands.

Load the Necessary Library

Make sure you have the library **rexxsupport.library** loaded before you run the program! You may want to borrow some code from a previous example to make sure its loaded at runtime. Name the following listing **PStextprint.ttx** or something mnemonic (I forget what the definition of *mnemonic* is, but I'll think of it in a minute):

```
/* PStextprint.ttx */
OPTIONS RESULTS

/* rexxsupport.library must be loaded!!!! */
/* postscript commands and parameters */

font='/Courier findfont 10 scalefont setfont' /* 10=pt. size of font */
coordx=68                                     /* left margin         */
coordy=720                                    /* top margin          */
pscommand='moveto show'
pshow='showpage'

/* TurboText-specific commands */
/*SAVEFILE*/ /* uncomment if you wish to save before printing */
GETFILEPATH            /* keep our file open */
doc=RESULT             /* remember the path  */
CONV2SPACES            /* make tabs into spaces to keep format     */
MOVESOF
FINDCHANGE ALL FIND '\' CHANGE '\\'     /* unbalanced parentheses   */
MOVESOF                                 /* and backslashes are      */
FINDCHANGE ALL FIND ')' CHANGE '\)'     /* specials characters in Ps*/
MOVESOF                                 /* these commands change    */
FINDCHANGE ALL FIND '(' CHANGE '\('     /* the text to print them OK*/
MOVESOF                                 /* literal strings in Ps:   */
GETCHAR
ch=RESULT
IF ch='('^ch=')'^ch='\'THEN INSERT '\'
```

```
SAVEFILEAS 'ram:text'
OPENFILE doc            /* put back the old file before formatting */

/* Main program */

IF OPEN('output','PAR:','w') THEN DO
   CALL WRITELN('output',font)
   IF OPEN('input','ram:text','read') THEN DO
      DO WHILE ~EOF('input')

         DO count=1 TO 66                 /* line count=66 */
            line='('READLN('input')')'
            CALL WRITELN('output',line)
            CALL WRITELN('output',coordx coordy pscommand)
            coordy=coordy-10              /* line spacing in pts. */
            END


         CALL WRITELN('output',pshow)
         coordy=720
         END /* DO */

      END /* input */
   END /* output */
CLOSE('input')
CALL DELETE('ram:text')
EXIT 0
```

That's the whole thing! All you have to do is launch this ARexx program from TurboText and it will print the file you are in to the PostScript printer attached to the parallel port.

POSTSCRIPT commands, see

PostScript Language Reference Manual, Second Edition by Adobe Systems, Inc.

Published by Addison Wesley, 1990

How the Program Works

If you study the way ARexx puts the program together and keep in mind we are using a postfix stack, you will see the simplicity of the way it works. First ARexx constructs a prologue string to later send to the PostScript interpreter, telling it to put **/Courier** font on the stack (a '/' *backslash* means a *literal* string and not an operator to PS). Then the operator **findfont** finds the Courier font in the dictionary inside the printer's memory of resident fonts. Next, a number **10** is pushed on the stack, and a **scalefont** operator takes the *two* operands, the font and the scale number and scales the entire font. Remember, the result of findfont was shoved on the stack and is underneath the number 10.

Finally, the operator **setfont** sets up the font dictionary using the information from the stack (the scaled font) for our program to use. In the case of font dictionaries, the actual PostScript object on the stack is simply an **address pointer**, which names the address in memory of what is called an **encoding vector**. The encoding vector is a table of codes and corresponding printable characters much like you find at the end of programming books. By means of this lookup table, PostScript automatically builds each character you need, internally in software, when you set the font.

Constructing Other Stack Objects

The other assigned variables are ARexx constructs of things we need in PostScript such as the starting coordinates on the page and two other commands we'll look at later.

Formatting the Document for PostScript

Meanwhile, ARexx uses TurboText commands to format the file and then save it to a temporary file called **RAM:text**. The TurboText commands in this section **escape** the special PostScript characters to insure that they are sent along as literals and not as PostScript commands. PostScript uses a backslash (\) as an escape character. If a \ occurs in front of any special character, then it is treated as a literal. There are three special characters to PostScript: \, **(**, and **)**. Actually, **A)** PostScript doesn't mind *balanced parentheses* () and will treat them as literals, but **B)** since we never know whether we will have balanced parentheses or not, we escape them all to be safe. (The above sentence would need the parentheses *escaped*, but not this one). The last IF instruction block in the re-format section is necessary in cases where there is a special character as the very first character of the document. In that case, TurboText cannot *find* this character with a FINDCHANGE command, so we have to escape it specifically after testing to see if it's there, using the GETCHAR command.

GETCHAR TurboText command
T A-18

FINDCHANGE TurboText command
T A-15

GETFILEPATH TurboText command
T A-21

SAVEFILEAS TurboText command
T A-44

OPENFILE TurboText command
T A-38

Notice that TurboText commands also get the file path and re-open the document afterwards, to retain the exact document you were working on. If you wish to print documents in progress, you may want to

SAVEFILE TurboText command
T A-44

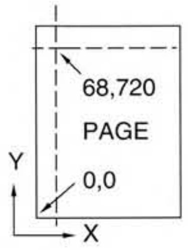
OPEN() function
H 60 f
C 10-109

WRITELN() function
H 69
C 10-124

uncomment the SAVEFILE instruction to insure that the latest version is saved before you print.

The Main Program

Now in the main program, ARexx opens the PAR: port for output. ARexx writes the PS prologue string assigned to the ARexx variable **font** and then grabs a line, puts ()'s around it, so that PostScript will know its a literal string, and pushes it on to the stack (with the font dictionary of 10pt Courier underneath). Note that "pushing onto the stack" is equivalent to writing a string to the Parallel port, since the PostScript interpreter is waiting at the other end.



X and Y Coordinates

On top of the above strings, ARexx puts (writes) first the X coordinate in 1/72nds of an inch: **68**, followed by the Y coordinate: **720**, measured from the lower left corner of the paper. The coordinates represent the upper left hand of a letter size paper with a top margin of one inch and a left margin of almost one inch. This is where the first line will print. Feel free to change these to your satisfaction.

Moveto and Show Set the Text Line

Finally the string 'moveto show' is appended to our PostScript program line, and these two commands are pushed in their order on to the stack. **Moveto** moves to the *current point* on the page determined by the two underlying x,y coordinates on the stack. **Show** commits to print the text underneath it on the stack at the current point on the page. Notice how each operator in turn *uses up* the stack entries much like the cafeteria customers use up their trays, or a Hewlett Packard calculator uses up the numbers you enter when you press the operator keys. The objects underneath keep on popping up, each in their turn.

Emulating a Line Feed/Carriage Return and Pagination

Next, we let ARexx handle the assignment of the Y coordinate to a point 10pts less than before to simulate what we refer to as a line feed/carriage return, with X held constant because it is the left margin, essentially. Then our program loops back and does it again until we've printed 66 lines. Then we let ARexx write the command **showpage** to **PAR:** which when it arrives at your printer, actually makes the printer print the page in real life. We reset all the variables to their new page settings and do it all again, until we hit the end of file. Then we clean up by closing the temporary file and deleting it with a call to the **rexxsupport.library** function DELETE().

DELETE() support
library function
H disk
C no reference

The Simplified Structure

Without all the housekeeping handled by ARexx, a PostScript program to print the string (roast beast) would look like this:

```
/Courier findfont 10 scalefont setfont
(roast beast)
68 720 moveto show
showpage
```

We could copy this file to the PAR: device and it would print. We've just done the housekeeping and loop counting and editor file controlling in ARexx because its easier. The structure of the PostScript part of our ARexx program is just like that of the above program, however, performed over and over.

Installing the Program in TurboText

EXECAREXXMACRO
TurboText command
T A-13

You may choose to open the **Startup.dfn** file from the TurboText support drawer and assign the normal menu/key sequence to **EXECAREXXMACRO PStextprint.ttx** (instead of the TurboText print function) so that you may always print a file with this macro. You can change your definition file in TurboText to the following line in the definitions wired to the menu and keys under the MENU section:

```
ITEM "Print"  "P" ExecARexxMacro sys:rexxc/PStextprint.ttx
```

Now every time you select the menu or press right-Amiga-[P], your file is printed, because the former command to **PrintFile** has been replaced by an **ExecARexxMacro** command referencing the program file. See what I mean about the power of TurboText? I can hot wire an ARexx macro to any key to do all sorts of things (even *sorts*)!

Adapting a Different Editor

If you use another editor with ARexx support, you will want to change the above program to use the commands peculiar to your brand of editor. All else will remain the same. You will want to do the following in your editor, either within the ARexx macro or manually:

1. Remember the path to your file and store it in ARexx variable "doc".

2. Convert all tab characters to spaces. Otherwise, format can suffer.

3. Take care of the special characters in PostScript: **(**, **)**, and \ to make them print as *literals*. Put the PostScript *escape character* \ in front of all parentheses and backslashes to make them into *literals*. The fastest way to do this is in your editor with a find and replace operator, although it is possible in ARexx to find and replace any string. Order is important. Escape (precede with a \\) the escape character first; then the parentheses. Balanced parentheses are not a problem, but you *may not know* beforehand whether they are balanced or not, so it is safest to change all parentheses to a parenthesis preceded by a backslash. Don't forget to move to the start of file (SOF) before each find and replace operation.

4. Save current file as 'RAM:text'

5. Re-open the file by its original file name.

Gotchas to Consider

There are a couple of Gotchas: If you haven't saved the file you were working on lately, it may be that you print your latest version and then an older one appears after printing. You may put in a command in the editor command section of the ARexx program to save the file before printing. In the case of TurboText, you would uncomment the command:

```
SAVEFILE
```

before the other commands. *You may also find a Gotcha here.* If you were working on a *new* version of a file and weren't sure of it, and you wanted to study a printout before you saved it...well you see what I mean. You may also use as a backup any automatic backups you may have programmed into your editor. The **RAM:text** file will have had its tabs converted to spaces; and any parentheses and backslashes preceded by a backslash. It is up to you to keep tabs on which file you want, in a manner of speaking.

Other Ideas

You may have gathered that this little program is only the beginning. You can of course make custom logos or drawings appear as a background to your text without resorting to a DTP program or your word processor. You may also choose to put dates, filenames or page numbers as headers or footers using ARexx to write the PostScript strings directly. For more complex things such as logos, it is simply a matter of writing the PostScript commands and saving them to a file *filename*. Then in your ARexx routine, you just put in the following:

```
ADDRESS COMMAND 'copy [filename] to PAR:'
```

Then your ARexx program will copy the program to the printer through the parallel port. The page will *not print* until you send a **showpage** command to PostScript, so you can build up complex pages easily. In this way you can unlock many creative possibilities.

References for PostScript

For further reading, I recommend the Addison Wesley series which includes *PostScript Language Reference Manual* and the *PostScript Language Tutorial and Cookbook*. Remember that PostScript is a universal language that allows you to take your files to any platform that supports PostScript. So next time you are in your bookstore, browse the section for other computers besides the Amiga and feel right at home, knowing that everything in these PostScript manuals applies to your Amiga. Enjoy the possibilities that the PostScript language coupled with ARexx offers. Remember, the other guys don't have ARexx!

Printing Envelopes with ARexx and PostScript

Here's another such ARexx/PostScript example. Let's take care of the annoying absence of any good way to print envelopes with the addresses filled in properly. It's not very satisfactory to have to use a DTP program just to print an envelope when you've written a letter in your word processor. Not only is it time consuming to fire up your DTP program to print a single envelope, but you may not even own a DTP program. We will use ARexx and PostScript to make a dandy envelope print utility that runs from the shell, and allows you to save a data base of addresses, as well as print envelopes.

Some Useful Utilities

If you have a word processor such as WordPerfect, you can open a WShell over your WP window and input the address from the letter directly, using a freely distributable program called **snap v1.62** by Mikael Karlsson (available as shareware from BIX, or included with WShell). This is an extremely handy little utility that allows you to copy (snap) text from *anywhere* and then paste it *anywhere*, even from custom screens. When running in the background, snap allows you to copy text to the clipboard by holding down the left Amiga key while you click the left mouse button. Snap allows you to copy a whole block of text, such as an address, by clicking and dragging out a box around the text. Snap pastes the copied text again at your cursor position *in any active screen or window* when you again hold down the left Amiga key and click the right mouse button.

Another handy utility included with WShell (freely distributable on BIX) is **popcli**, by The Software Distillery, a hot key program to open a shell window at any time when you press the Esc key and the left Amiga key together. It includes a screen blanker, too. Since WordPerfect does not cut or copy to the clipboard, **snap** proves invaluable for using the clipboard in spite of this drawback, and **popcli** allows you to open a shell window quickly, using hot-keys. Even if you do not have these two utilities the PostScript envelope printer will be usable, but until you make a data base of addresses, you will have to manually type in each new address.

The Envelope Print Program

The program is called simply **E.rexx** and is designed to work with any PostScript printer capable of feeding standard business sized envelopes of 9 1/2 x 4 1/8 inches. Use envelopes with a squared off flap to prevent jamming, and select a paper with no tooth to it (as smooth as possible). Here is the listing:

```
/* E.Rexx Envelope PS Printer */

start:
print='no'
savefile=''
SAY 'Start: Enter filename. [Rtn]=Enter address. Q=Quit. L=List Addresses.'
PARSE UPPER PULL input
IF input == 'Q' THEN EXIT 0
IF input == 'L' THEN DO
   ADDRESS COMMAND 'DIR data:wp/addresses'
   SIGNAL start
   END

k=1
IF input == '' THEN DO
   SAY 'Enter address: line 1 [Rtn], line 2 [Rtn], etc. @ = finished.'
   DO FOREVER
      PARSE PULL line.k
      IF line.k='@' THEN DO;line.k='';SIGNAL Decide;END
      k=k+1
      END /* FOREVER */
   k=k-1
   END /* input==''*/

IF input ~= '' THEN DO
   IF OPEN('file','data:wp/addresses/'input,'R') THEN DO
      l=1
      DO WHILE ~EOF('file')
         line.l=READLN('file')
         SAY line.l
         l=l+1
         END /* DO WHILE */
      k=l-1
      CALL CLOSE('file')
      END /* IF OPEN */
      ELSE DO
         SAY 'Could not open your address file. Try again.'
         SIGNAL start
         END /* ELSE */
   END /* input ~='' */
```

```
Decide:
SAY '[P]=print; [S]=save to file; [B]=do both. [Q]=quit.'
PARSE UPPER PULL answer
IF answer == 'Q' THEN EXIT 0
IF answer == 'S' | answer == 'B' THEN DO
   SAY 'Enter filename. Default path is Data:WP/addresses/'
   PARSE PULL savefile
   savefile = 'Data:WP/addresses/'savefile
   IF OPEN('outfile',savefile,'W') THEN SAY 'Saving 'savefile
   DO n=1 TO k-1
      CALL WRITELN('outfile',line.n)
      END /* DO */
   END /* S  | B */
IF answer == 'P' | answer == 'B' THEN print='yes'
IF print= 'yes' THEN CALL printaddress(k,line.)
CALL CLOSE('outfile')
SIGNAL start
EXIT 0

printaddress: PROCEDURE EXPOSE k line.

/* postscript commands and parameters */
font='/NewCenturySchlbk-Roman findfont 12 scalefont setfont'
coordx=410                                   /* left margin          */
coordy=300                                   /* top margin           */
pscommand='moveto show'
pshow='showpage'
tran='0 -612 translate'
rotate='90 rotate'

IF OPEN('output','PAR:','W') THEN DO
   SAY 'PRINTING...'
   CALL WRITELN('output',font rotate tran)
   DO i=1 TO k
      CALL WRITELN('output','('line.i')')
      CALL WRITELN('output',coordx coordy pscommand)
      coordy=coordy-12
      END
   coordx=135
   coordy=418   /* Change the following to YOUR RETURN ADDRESS */
   CALL WRITELN('output','(Merrill Callaway)')
   CALL WRITELN('output', coordx coordy pscommand)
   coordy=coordy-12
   CALL WRITELN('output','(511-A Girard Blvd. SE)')
   CALL WRITELN('output', coordx coordy pscommand)
   coordy=coordy-12
   CALL WRITELN('output','(Albuquerque, NM 87106)')
   CALL WRITELN('output', coordx coordy pscommand)
   coordy=coordy-12
   CALL WRITELN('output','()')   /* For foreign letters put USA here */
   CALL WRITELN('output', coordx coordy pscommand)
   coordy=coordy-12
   CALL WRITELN('output',pshow)
   END
```

```
CALL CLOSE('output')
RETURN
```

Program Design

SIGNAL instruction
H 38 f
C 10-71 ff

PROCEDURE instruction
H 35
C 10-68 f

The program is divided into three sections. The first section takes care of our address data base and entering new addresses. It is labeled **start:** and we use a **SIGNAL start** instruction (a GO TO) to jump to the start when required. The second section is labeled **decide:** and this is where the program lets you decide whether to save the address file, merely print it, or save *and* print the address. The final section is a PROCEDURE labeled **printaddress:** which the program calls when it needs to print the file on an envelope.

Getting the Data

PARSE instruction
H 33 ff
C 10-64 thru 10-68

COMMAND special address for **ADDRESS** instruction
H 44
C 10-76

FOREVER iteration specifier for **DO** instruction
H 27 f
C 10-53 ff

Arrays see **Stems** and **Compound** symbols
H 21 f
C 10-44 ff

First **E.rexx** asks for the file name, a **Q** to quit, an **L** to list the files in our data base of addresses, or simply a **[Rtn]** if we want to enter a new address. Then it parses our answer into **input** and acts accordingly. If we answer **Q** then it exits. If we answer L then it uses an ADDRESS COMMAND to perform an AmigaDOS DIR instruction on our chosen path to our address files, in case we need to jog our memory. It jumps back to the start after this choice by means of the **SIGNAL start** instruction. It initializes a node **k** for an address line array, and if we have entered a [Rtn] to signify that we want to enter a new address, it enters a now familiar DO FOREVER block to allow us to input all the lines we need for the address. We escape from the loop by entering a blank line with only a [Rtn]. Since you may need to use blank lines in your addresses, the escape to test is for some character you will never use in someone's address such as @. We build an array **line.** with nodes **k** to contain all the address information.

Using the Address Data Base

If the initial input is not the string @, then the program attempts to

OPEN() function
H 60 f
C 10-109

SAY instruction
H 38
C 10-70

ELSE instruction
H 28 f
C 10-56 f

OPEN() the file for us using the path hard coded into the program. *Make sure to change this part to match your system directories!* The program opens the file and reads it into the **line.** array using **l** as nodes for use by the printing section. While we are at it we let the program SAY each line so we can see if we have the right address from our data base. We close 'file' our logical name for the read file, so that if we access it again, we can open it successfully. If it is already open, we cannot open it again! The ELSE clause takes care of the situation in case we make a mistake and enter a non-existent file name.

Print, Save, or Both?

WRITE argument to **OPEN()** function
H 60 f
C 10-109

CLOSE() function
H 55
C 10-98

The second section, at the **decide:** label, asks us about our choice to quit, save, print, or print and save the address. If we want to print and/or save, the program OPENs a file name 'outfile' as a write file and a loop writes the array to the file. It also sets a "flag" variable **print** to 'yes' if we choose to print at all. Note that we could have accomplished this test and print procedure call in only one step, but the extra code is there to remind you that you can make your own flags this way. In many programs, the function call will not be adjacent to the set flag test, warranting the use of this technique. Variables which serve as flags may be passed to a function by the call itself and then control decisions within the function. If you like more compact code substitute

```
IF answer == 'P' | answer == 'B' THEN CALL printaddress(k,line.)
```

for the two lines. The rest of the main program cleans up by closing the write file and SIGNALing **start** to allow us to print multiple envelopes.

How to Pass an Entire Array to an Internal Function

Procedure

EXPOSE subkeyword for **PROCEDURE** instruction
H 35
C 10-68 f

Meanwhile, if we wanted to print, the program passes the largest node **k** and an *entire array* **line.** to the PROCEDURE printaddress. Notice how we EXPOSE **k** and **line.** to the PROCEDURE. This way we don't worry about parsing arguments or any other chores, and most important, this is the correct way to pass an entire array with all its elements to a function: *We expose the array by name in the procedure label line.*

Note carefully the syntax of passing an entire array. The order in which we pass the arguments: **k**, and then **line.** makes a difference. Also note the comma in the call, and that there is no comma in the EXPOSE list of variables. Also note that the period, but no nodes are attached to **line.** (the array stem). The syntax is critical here and it is easy to overlook typos and mistakes. See also: *Hawes*, page 35; and *Commodore*, page 10-68 f.

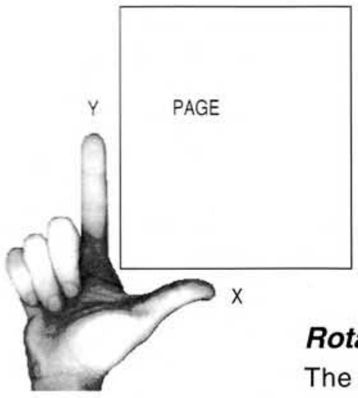
The PostScript/ARexx Print Function

Transforming the Page Coordinates for Envelopes

There are many similarities between this internal function and the program to print files from TurboText, discussed earlier. The only real difference is that we need to rotate the axes of the printing 90 degrees and then translate along the Y axis, as envelopes feed into the printer lengthwise.

The PostScript Coordinate System

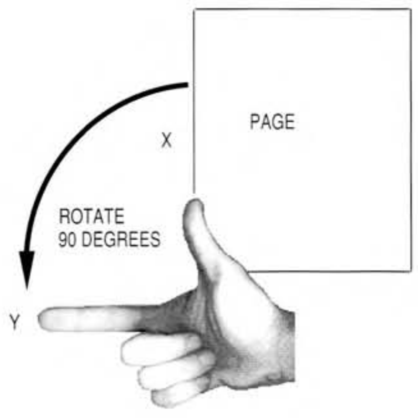
PostScript describes any page with an X and Y coordinate system with ordered pairs (X,Y) in units of points or 1/72nds of an inch. The **default origin** (0,0) is at the lower left corner of the page, but we may **transform** the coordinates, if we desire, by *rotating around the origin* or *translating the origin.*



You may think of a piece of paper on your desk upon which you place your right hand with your index finger along the left long side (the Y axis) and your thumb along the lower short side (the X axis). Your hand represents the default coordinate system, called a **right hand coordinate system**.

Rotate

The PostScript operator **rotate** rotates the entire coordinate system (your hand with the thumb and index finger at 90 degrees to each other) around the present origin (the lower left corner of the paper) *counterclockwise* the number of degrees specified by the number placed on the stack just beneath the operator (remember this is a LIFO stack!).

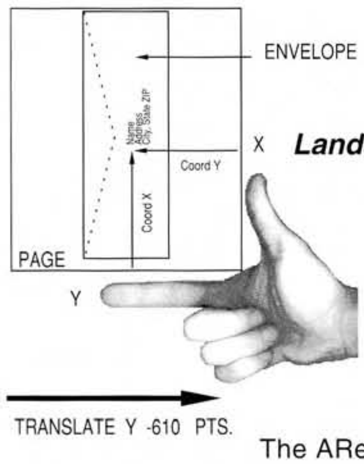


90 Rotate

Note that all the positive X and Y values rotate off the page if we do *only* a **90 rotate** operation. We therefore need to complete a **coordinate translation** before our paper is lined up correctly for **landscape printing** (print is rotated 90 degrees).

0 -612 Translate

PostScript accomplishes shifts of the coordinate system using the **translate** operator which uses the values of X and Y on the stack underneath the operator. The sequence of PostScript objects: **0 -612 translate** moves minus 612 points (8.5 inches) along the Y axis and none along the X axis. Note that the shift is **negative** along the newly rotated Y axis. The PostScript measurements are still along the index finger for the Y coordinates and along the thumb for X coordinates, the positive direction being toward the finger tips.



Landscape Printing

The **landscape transformation** (PostScript: **90 rotate** followed by **0 -610 translate**) effectively prints everything at 90 degrees to the original orientation of the paper. Order matters; the correct sequence being **rotate**, then **translate**. Refer to the illustrations.

The Output

The ARexx program builds the strings it needs to send to PostScript, and then opens the logical name 'output' for writing to the parallel port. It then writes the prologue which shoves the **font**, **rotate**, and **tran** strings on the stack. By means of a loop it puts parentheses around the lines of address information to make them literals, and puts them on the stack. The coordinates for the text position were determined by measuring a regular sheet of paper with a business envelope centered on it lengthwise.

Determining the (X,Y) Coordinates in the Transformed System

The coordinate system measurements are for a standard sheet of paper, so allowances have to be made for the smaller envelope centered on a standard page (see illustration). In other words, any margins around the envelope centered on a standard piece of paper must be added to the coordinates. Even though we are sending through only an envelope, the printer still thinks it's printing on a standard sized page, so we must make sure to print in the area covered by the envelope! **It is always a good idea to test print using a full sheet of paper, or you can damage your print drum!** After testing on a full sheet of paper, hold up an envelope centered lengthwise on the page to see if the printing lines up in the correct position.

Measurements:
72 Points = One Inch.

After the measurements for the text placement were made in inches they were converted to points (multiplied by 72) and became the coordinates. The **moveto** and **show** PostScript operators accomplish the setting of the text position as in the previous example.

The Return Address

Note:
Change the code to *your* return address!

The final section of the print function is to print the return address in the upper left corner of the envelope. Be careful not to print too close to the edge of the envelope. Since this is fixed information, it was hard coded into the program. Note the way that the program moves down one line by subtracting 12 points from the Y coordinate for each line. The final command is the PostScript **showpage** command which does the actual printing.

The Companion Disk to *The ARexx Cookbook* includes a second version of E.rexx that will print special accented characters used in international correspondence. It also prints an "Air Mail" message on the envelope. See copyright page to order.

This little program is very useful and handier than using anything else to print envelopes quickly, *and* it stores a data base of addresses for you. PostScript and ARexx together are unbeatable!

Conclusion of the ARexx Tutorials

We have now toured a lot of ground. If you have followed the exercises, you have an adequate knowledge of ARexx to start making it work for you in all your applications whether they be in text, graphics, video or music. Commands are commands, and *any* program may be controlled using identical *text manipulation* and *interprocess control* techniques covered in the previous sections. Commands, being simply strings, are best manipulated when you are good at programming text tools. Text processing is the most universal task on any computer, and that is why we studied it in such depth. Remember, you are now literate in ARexx. You can read, understand and modify any ARexx code you may find in specialist publications. Your imagination is the only limit!

The remainder of this book is devoted to listings of useful or interesting ARexx programs with only a few remarks about them. Use them directly, or modify them, or simply study them and you will learn more about ARexx.

ARexx and Art Department Professional (ADPro)

Aspects of Pixels and Images

Art Department Professional (ADPro)
by
ASDG, Inc.
925 Stewart St.
Madison, WI 53713
(608) 273-6585

ADPro ARexx Interface described starting on page 277 of *ADPro Manual.*

Four situations recur frequently in Amiga graphics:

1. How do you deal with images imported from a device that uses square pixels once they are inside your Amiga? Suppose you have a picture to scan on the Epson ES300C with 1:1 pixel aspect, and you want to fill a 640 x 400 screen. If you scan it according to that pixel count in the ADPro driver window, it will fill your screen all right, but it will appear too "tall and skinny" compared to the way the original looked, even though the number of pixels remains at 640x400. This is because the Amiga screen pixels are not square. We need to scale the height down a bit to get the proportions back, but now the image is too short and no longer fills the screen. We need both to scan a taller height in pixels and also scale the height down afterwards to fill the screen with an image of the proper aspect. How do we do this? We need to do what is called "scale to pixel aspect" while simultaneously scaling to fit the screen. Pixel aspect is the ratio of the pixel width dx to its height dy (dx:dy). Aspects generally may refer to pixels, images, or the screen format itself. We will use different variables to distinguish which one we mean. Some commonly accepted pixel aspects for popular programs are 10:11 for DeluxePaint; 69:80 for DCTV (in overscan); and 11:13 for ADPro. The Epson ES300C scanner uses 1:1 pixels.

2. Assuming we have a correct pixel aspect, the second problem is more common: How do you scale the data so that if you render it in another Amiga screen format, it will look right? Screen format pixels have an aspect of width w to height h (w:h). In an *image* with a constant aspect W:H rendered in different Amiga screen formats, notice that it takes different numbers of pixels in the W and H directions to render the "same" image in different screen formats. The "pixel aspect" to do with screen formats is *not* always identical in meaning to the "pixel aspect" in 1. above. This is why people sometimes get confused. "Screen format" pixel aspects are determined by doubling, dividing in half, or leaving alone the width or height "image pixel counts". ADPro distinguishes

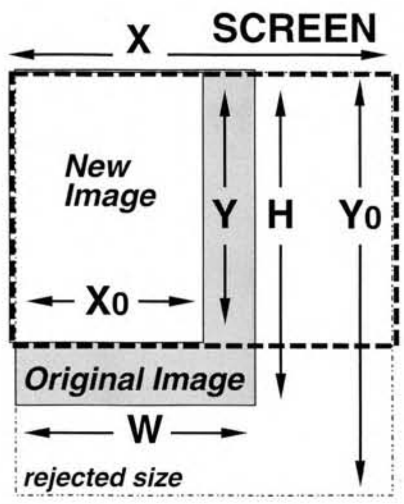
Pick one or the other of these equations:

$$\frac{W}{H} = \frac{X}{Y_0}$$

$$\frac{W}{H} = \frac{X_0}{Y}$$

$$Y_0 = \frac{H}{W} X$$

$$X_0 = \frac{W}{H} Y$$



IF $X_0 >$ X THEN USE X and Y_0
IF $Y_0 >$ Y THEN USE X_0 and Y

Here is an example of the latter case: We use X_0 and Y for our new image dimensions. Note: the above are mutually exclusive possibilities. The Screen is the heavy dotted line. Rejected size as too large is the light dotted line.

Fig 1
How to Scale an Image to Fit the Screen.

these screen formats accordingly:

1) High Res Interlace	= 22:26
2) Low Res Non-Interlace	= 44:52
3) High Res Non-Interlace	= 22:52
4) Low Res Interlace	= 44:26

These four values represent the basic four Amiga screen formats, even though 1) and 2) reduce to the same "pixel aspect" of 11:13 as defined in 1. above.

3. A related problem is how do you scale the data so that you will get as near a full screen as possible, and still keep the image aspect correct? **Fig 1** shows the relationship between an IMAGE with an arbitrary aspect W:H and the Amiga SCREEN aspect (X:Y). Solving the two equations for the two unknowns, X0 and Y0, we then compare X0 to X and Y0 to Y. We use these inequalities to decide whether to use (X and Y0); or (X0 and Y) as our new dimensions. Only one of the inequalities can hold true for a given image aspect. In the case of an exact fit, then neither inequality is true.

4. A further complication comes when you consider using Overscan. The Amiga Overscan increases the W pixel count by 15% and the H count by 20%. There are 128 possible transformations: 4 screen formats, plus the same four in Overscan, transformed both to and from each other in both the X and Y directions: 128=(4+4)*(4+4)*2.

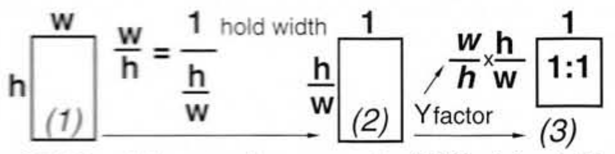
All the resolutions, colors, and screen modes on the stock Amiga 3000 add up to 208 possible combinations. Finally, there is the vertical adjustment on your monitor to consider.

ARexx to the Rescue

With these programs, you can scale from any screen resolution with or without overscan, to any other with ease, *and* you can make the pixel aspects just right for your application.

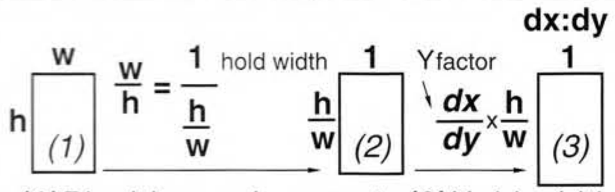
Monitor Image Aspect Adjustment

The first thing to make sure of is the correct image aspect adjustment of your monitor. In some monitors this is a small screw inside a hole in the back; in a 1950 it's a knob behind a door in the side. I like to use DeluxePaintIV in "Be Square" mode, and draw a light colored solid square (hold down the shift key), and a smaller square inside the first. Then, with a clear plastic ruler (not metal--it will mess up the magnetic field of your screen), measure the squares for accuracy. If they are off, then adjust the monitor's vertical height to make them truly square. Now we are ready.



(1)Pixel has w:h aspect. ***(2)****Hold width a constant value 1.* ***(3)****Transform to a 1:1 pixel aspect using Y factor w/h.*

(A) Scaling to a 1:1 pixel aspect.



(1)Pixel has w:h aspect. ***(2)****Hold width a constant value 1.* ***(3)****Transform to a dx:dy pixel aspect using Y factor dx/dy.*

(B) Scaling to a pixel aspect dx:dy.
(C) Image W by H pixels with aspect w:h will thus scale to X by Y pixels with aspect dx:dy where:
(1) X=W
(2) Y=(dx/dy)(h/w)H

Fig 2 How to Scale to a New Pixel Aspect.

Scaling to Pixel Aspect

Fig 2 illustrates the arithmetic to do with pixel aspects and scaling between them.

Building a GUI

The **rexxarplib.library** is handy for our purposes. ADPro has an ARexx macro capability, but its requesters are limited to just a few characters, and we have many interconnected decisions to make. A window with logically interlocked gadgets is ultimately the best solution.

Press Function Key **F2** while in ADPro. The **F2.adpro** program loads libraries if necessary, checks for a large enough Workbench screen and opens a custom screen if necessary. Then it creates a host application complete with I/O port names, opens a window, and builds some gadgets. It also calls **guiPostMsg.rexx** to put up our image information in its own window. All **rexxarplib.library** gadgets do is send a *command string* to a host application port name of your choice. That's why we program our system as several smaller ARexx programs which some gadget "calls" when clicked on. The **F2.adpro** program does its thing and exits, leaving behind a window wired to send particular strings whenever gadgets are clicked. Notice how we can send arguments along with the string; for instance the position of the mouse pointer in the %x %y arguments. Also notice the finicky way in which we must quote long strings to insure that the ARexx command parser interprets things like line continuation commas or string tokens correctly. The rat's nest of quotation marks is one small price we must pay for using typeless tokens (ARexx variables).

Environment Variables

Environment Variables are set with **rexxarplib.library** functions.
Refer to library docs.

The programs use environment variables rather than passing arguments all over the place. The program **guiTerm.rexx** terminates all processes and gets rid of these environment variables. The main screen calls either **guiSPO.rexx** (from the top gadget) or **guiSPD.rexx** (lower gadget) to set up and interlock other gadgets for further operations. The program **guiAddons.rexx** makes the gadgets that set the environment variables of our target screen format and the Y Overscan, etc. It also makes the "EXECUTE" gadget that calls the program that does the scaling work in ADPro: **guiEX.rexx**. The upper bank of four gadgets are created by **guiSPO.rexx**. These all pass their arguments to **guiStartScOnly.rexx** which takes care of more interlocks, launching **guiScalePixAsp.rexx** with appropriate arguments to scale to pixel aspect in ADPro. In **guiEX.rexx**, we use two, two-dimensional transformation matrices **WX.** and **HY.** and absolute scaling instead of percent scaling to insure exact pixel counts.

Gadgets that are not used are "removed" and/or turned off by the programs' "interlocks". The environment variables **G**, **H**, and **gad** are used to tell the programs if the top, the middle or the bottom gadgets have yet been drawn. The interlocks at each level use these to make choices.

Operation

Note: The companion disk contains an advanced version of the ADPro Scale Utility which has functions that support the Firecracker24 display board formats, as well as other refinements. See the copyright page to order this disk.

In ADPro, press function key **F2** to launch the programs. If you already have another **F2.adpro** program, then you may safely rename this one. If you use System1.3, and a non-interlaced Workbench, then a custom screen will open with the window. If you use System2.0, then you'll get a full sized Workbench window. Select the top gadget to scale to *pixel aspect only*, and click on the lower gadget to scale to screen size *and* optionally change pixel aspects, too. Each of the original gadgets opens another group of gadgets from which to select.

The lower gadgets look for rendered data. If you forgot to render, a requester asks if you want to "Smart Render" and will attempt to guess a screen format based on current pixel aspect of your data. To render manually, exit. The gadgets change their highlights indicating which data is in the environment variables. They also lock out gadgets not to be used by their interlocks. The "Y Overscan" gadget lets you choose +15% or +20% Y overscan. To keep image aspect, the default, 15% is the best choice. To fill the screen with a 640x400 image in overscan at 736x480, choose "Fill Screen" 20% Y Overscan. The image stretches 5% in the Y direction, filling the screen. In video, the stretch will not be noticeable, but a bottom border would be.

The "Help" gadget calls up some useful image information for Amiga screens. The logo code is commented out, but you may include an IFF file (from a brush) in this way.

The Change Pixel Aspect button allows you to reset the pixel aspect without scaling the data. Use it if you've previously scaled data manually and need to reset the pixel aspect to a meaningful value before you scale to another screen format.

The Listings

To aid in readability, the listings show **rexxarplib.library** functions in *italics*, and ADPro commands in **bold**. ARexx commands and instructions are in normal type face.

```
/* F2.adpro GUI for guiEX.rexx */
/* see note in guiEX.rexx      */

OPTIONS RESULTS

/* flag for message window */
flag=0

/* if our window is there, post the latest info */
IF SHOW('P','SCALEHOST') THEN DO
   CALL guiPostMsg.rexx
   flag=1
   END

/* setup Loads libraries */

libs.1='rexxsupport.library'
/* extended functions (DOS,etc.) */

libs.2='rexxarplib.library'
/* intuition, windows, gadgets */

DO i=1 TO 2
   IF ~SHOW('L',libs.i) THEN CALL ADDLIB(libs.i,0,-30,0)
   IF ~SHOW('L',libs.i) THEN EXIT 20
   END

/* set a default env variable */
CALL SETENV(YOSET,1.15)

/* put ADPro to back */
ADDRESS "ADPro" ADPRO_TO_BACK

/*
** Check for big enough screen;
** if screen too small (i.e. WB1.3) then
** we will open a custom public screen "APS".
** Otherwise we'll use the interlaced WB
** for our SCALEHOST (the port name) window.
*/
```

```
row=SCREENROWS('Workbench')
col=SCREENCOLS('Workbench')
lace=SCREENLACE('Workbench')
IF (row<400|col<640|lace=0) THEN DO
   RESULT=OPENSCREEN(,,"HIRES" "LACE",,
   "ADPro Scale Utility V1.0","APS",,640,,)
   END
/*
** Create our very own host application.
** We should use the asynchronous "AREXX" port.
** This window GETS its messages thru "SCALEHOST".
** This window SENDS its messages to "REXX".
** This window tries to open on public screen "APS".
** If it cannot, it opens on the "Workbench" screen.
*/
ADDRESS AREXX "'CALL CreateHost("SCALEHOST","REXX",APS)'"

/* wait for our new port to come on line */
WAITFORPORT "SCALEHOST"

/* Amiga Intuition parameters for the window and gads */
/* for gads */

idcmp="CLOSEWINDOW GADGETUP MENUPICK"

/* for window */

flags="NOCAREREFRESH WINDOWCLOSE WINDOWDRAG",
" WINDOWDEPTH WINDOWSIZING SIZEBOTTOM ACTIVATE"

/* open the window with the parameters we want */

CALL OPENWINDOW("SCALEHOST",0,11,640,389,idcmp,flags,,
"F2.ADPro: Select a Scaling Operation...")

/* what to do if we click on the closewindow gad */

CALL MODIFYHOST(SCALEHOST,CLOSEWINDOW,"'CALL guiTerm.rexx'")

/*
** Add the primary gadgets to the window.
** If the gad is clicked, the last string is sent to "REXX".
** Here, they are calls to the other programs.
*/

CALL ADDGADGET("SCALEHOST",10,370,"EXIT",,
" EXIT ","'CALL guiTerm.rexx'")

CALL ADDGADGET("SCALEHOST",275,125,"REN",,
" IMAGE INFO ","'CALL guiPostMsg.rexx'")

CALL ADDGADGET("SCALEHOST",400,125,"HELP",,
"    HELP    ","'CALL guiHELP.rexx'")
```

```
CALL ADDGADGET("SCALEHOST",275,105,"CPA",,
" SET PixAsp ","'CALL guiCPA.rexx %x %y'")

/* post the image info on the window */
IF flag=0 THEN CALL guiPostMsg.rexx

CALL ADDGADGET("SCALEHOST",10,20,"SPIX",,
" SCALE: To Pixel Aspect Only ",,
"'CALL guiSPO.rexx'")

CALL ADDGADGET("SCALEHOST",10,125,"SPAD",,
" SCALE: Raw & Rendered Data  ",,
"'CALL guiSPD.rexx'")

/* set the defaults for the environment vars */
CALL SETENV(gad,0)
CALL SETENV(G,0)
CALL SETENV(H,0)

EXIT 0
```

```
/* guiSPO.rexx scale pixels only GUI F2.adpro */
/* makes the four scale gads for the top screen gad */

OPTIONS RESULTS

/* interlock */
CALL SETGADGET(SCALEHOST,SPIX,ON)

/* make gad and pass args in the pgm call */
CALL ADDGADGET("SCALEHOST",10,40,"SDOC",,
" SCALE: Pixel aspect only      (pick)  X:Y  ",,
"'CALL guiStartScOnly.rexx ,%x %y'")

CALL ADDGADGET("SCALEHOST",10,55,"SDOP",,
" SCALE: Pixel aspect only (DPaintIV) 10:11 ",,
"'CALL guiStartScOnly.rexx 10 11,%x %y'")

CALL ADDGADGET("SCALEHOST",10,70,"SDOA",,
" SCALE: Pixel aspect only    (ADPro) 22:26 ",,
"'CALL guiStartScOnly.rexx 22 26,%x %y'")

CALL ADDGADGET("SCALEHOST",10,85,"SDOD",,
" SCALE: Pixel aspect only     (DCTV) 69:80 ",,
"'CALL guiStartScOnly.rexx 69 80,%x %y'")

/* tell the environment we have made these gads */
CALL SETENV(G,1)
CALL SETENV(gad,0)

/* interlock & disable gads we should not use */
IF GETENV(H)=1 THEN DO
   CALL SETGADGET(SCALEHOST,SPAD,OFF)
```

```
/* Interlocks for gadgets */
   CALL SETGADGET(SCALEHOST,PC,OFF)
   CALL SETGADGET(SCALEHOST,PP,OFF)
   CALL SETGADGET(SCALEHOST,PA,OFF)
   CALL SETGADGET(SCALEHOST,PD,OFF)
   CALL SETGADGET(SCALEHOST,PO,OFF)

   CALL REMOVEGADGET(SCALEHOST,"PC")
   CALL REMOVEGADGET(SCALEHOST,"PP")
   CALL REMOVEGADGET(SCALEHOST,"PA")
   CALL REMOVEGADGET(SCALEHOST,"PD")
   CALL REMOVEGADGET(SCALEHOST,"PO")

   CALL SETGADGET(SCALEHOST,HR,OFF)
   CALL SETGADGET(SCALEHOST,LR,OFF)
   CALL SETGADGET(SCALEHOST,IL,OFF)
   CALL SETGADGET(SCALEHOST,NL,OFF)
   CALL SETGADGET(SCALEHOST,OS,OFF)
   CALL SETGADGET(SCALEHOST,ST,OFF)

   CALL REMOVEGADGET(SCALEHOST,"HR")
   CALL REMOVEGADGET(SCALEHOST,"LR")
   CALL REMOVEGADGET(SCALEHOST,"IL")
   CALL REMOVEGADGET(SCALEHOST,"NL")
   CALL REMOVEGADGET(SCALEHOST,"OS")
   CALL REMOVEGADGET(SCALEHOST,"ST")
   CALL REMOVEGADGET(SCALEHOST,"EX")
   CALL REMOVEGADGET(SCALEHOST,"YOSET")
   END

EXIT 0
```

```
/* guiSPD.rexx scale pixels and display gui for F2.adpro */
OPTIONS RESULTS

/* interlock to turn gads on and off */
CALL SETGADGET(SCALEHOST,SPAD,ON)
CALL SETGADGET(SCALEHOST,SPIX,OFF)

/* MAKE H GADGETS */
/* note the way we pass arguments to the */
/* pgm that makes the variables          */

CALL ADDGADGET("SCALEHOST",10,145,"PC",,
" SCALE:     (pick)  X:Y  ",,
"'CALL guiMakeVar.rexx "C",%x %y'")

CALL ADDGADGET("SCALEHOST",10,160,"PP",,
" SCALE:(DPaintIV) 10:11 ",,
"'CALL guiMakeVar.rexx "P" 10 11'")

CALL ADDGADGET("SCALEHOST",10,175,"PA",,
" SCALE:   (ADPro) 22:26 ",,
"'CALL guiMakeVar.rexx "A" 22 26'")

CALL ADDGADGET("SCALEHOST",10,190,"PD",,
" SCALE:     (DCTV) 69:80 ",,
"'CALL guiMakeVar.rexx "D" 69 80'")

CALL ADDGADGET("SCALEHOST",10,205,"PO",,
" SCALE:    Display Only ",,
"'CALL guiMakeVar.rexx "O" GETENV(xaspect) GETENV(yaspect)'")

CALL ADDGADGET("SCALEHOST",120,370,"EX",,
" EXECUTE ",,
"'CALL guiEX.rexx %x %y'")

"'CALL guiMakeVar.rexx "O" GETENV(xaspect) GETENV(yaspect)'"

/* tell the environment we have made these gads */
CALL SETENV(H,1)

/* interlock */
/* removing a gad makes it inoperative */
IF GETENV(G)=1 THEN DO

   CALL SETGADGET(SCALEHOST,SDOC,OFF)
   CALL SETGADGET(SCALEHOST,SDOP,OFF)
   CALL SETGADGET(SCALEHOST,SDOA,OFF)
   CALL SETGADGET(SCALEHOST,SDOD,OFF)

   CALL REMOVEGADGET(SCALEHOST,"SDOC")
   CALL REMOVEGADGET(SCALEHOST,"SDOP")
   CALL REMOVEGADGET(SCALEHOST,"SDOA")
   CALL REMOVEGADGET(SCALEHOST,"SDOD")
   END

/* make the rest of the gads */
CALL guiAddons.rexx

EXIT 0
```

```
/* guiPostMsg.rexx posts adpro image info */
/* in its own window on SCALEHOST window  */

OPTIONS RESULTS

/* to add a logo, make one as a brush and then uncomment the next */
/* command and put the path name in here where Rexx:guiPIC.iff is */
/*
CALL IFFImage("SCALEHOST",,
"Rexx:guiPIC.iff",400,20,,,)
*/
CALL POSTMSG()
CALL SETGADGET(SCALEHOST,REN,ON)
CALL SETGADGET(SCALEHOST,HELP,OFF)

IF ~SHOW('P','ADPro') THEN,
CALL POSTMSG(275,155,"ADPro is not running",APS)

ADDRESS "ADPro"
/* find out what kind of image we have...*/
IMAGE_TYPE
itype=ADPRO_RESULT
IF (ADPRO_RESULT = "NONE") | (ADPRO_RESULT = "BITPLANE") THEN DO
   RESULT=REQUEST(10,50,"No Image Data to Scale!",,"Resume",,APS)
   CALL guiTerm.rexx
   EXIT 0
   END

/* FIND() is in the rexxsupport.library */
IF FIND(itype,BITPLANE)=0 THEN rend=0

LAST_LOADED_IMAGE
IF RC=0 THEN lli=ADPRO_RESULT;ELSE lli='none'
IF LENGTH(lli)>34 THEN lli=LEFT(lli,34)
LAST_SAVED_IMAGE
IF RC=0 THEN lsi=ADPRO_RESULT;ELSE lsi='none'
IF LENGTH(lsi)>34 THEN lsi=LEFT(lsi,34)
SCREEN_TYPE
IF RC=0 & rend~=0 THEN styp=ADPRO_RESULT;ELSE styp='no rendered data'

/* make string to display in message window */
SELECT
   WHEN styp=0 THEN stext='LoRes, Non-Interlace '
   WHEN styp=1 THEN stext='HiRes, Non-Interlace '
   WHEN styp=2 THEN stext='LoRes, Interlace '
   WHEN styp=3 THEN stext='HiRes, Interlace '
   WHEN styp=24 THEN stext='Ovrscan X&Y, LoRes, Non-Interlace'
   WHEN styp=25 THEN stext='Ovrscan X&Y, HiRes, Non-Interlace'
   WHEN styp=26 THEN stext='Ovrscan X&Y, LoRes, Interlace'
   WHEN styp=27 THEN stext='Ovrscan X&Y, HiRes, Interlace'
   OTHERWISE stext='(see page 369)'
   END

RENDER_TYPE
IF RC=0 & rend~=0 THEN ren=ADPRO_RESULT;ELSE ren='no rendered data'

OPERATOR "DEFINE_PXL_ASPECT"
IF RC=0 THEN dpaline=ADPRO_RESULT;ELSE dpaline='none'

PARSE VAR dpaline Xasp Yasp Xres Yres width height

ADDRESS "SCALEHOST"
/* set environment variables for use by other pgms */
RESULT=SETENV(Xaspect,Xasp)
RESULT=SETENV(Yaspect,Yasp)
RESULT=SETENV(xwide,width)
RESULT=SETENV(yhigh,height)
```

```
RESULT=SETENV(stype,styp)
YOSET=GETENV(YOSET)
IF YOSET=1.15 THEN t='15% Y Overscan: Keep Image Aspect'
IF YOSET=1.2 THEN t='20% Y Overscan: Fill Screen'

/* post the message in the window */
/* get the quotes exactly right!  */

CALL POSTMSG(275,155,"Last Loaded Image:                    \"lli,
"\ \Last Saved Image:\"lsi,
"\ \Image Type:\"itype,
"\ \Screen Type:\"styp": "stext,
"\ \Render Type:\"ren"-color",
"\ \Pixel Aspect X:Y    = "Xasp":"Yasp,
"\Resolution (X by Y): "Xres" by "Yres,
"\Width = "width" Height ="height,
"\ \"t,APS)

EXIT 0
```

```
/* guiHELP.rexx posts adpro image info gui F2.adpro */
OPTIONS RESULTS

/* to add a logo, make one as a brush and then uncomment instruction */
/* and put the path name in here where Rexx:guiPIC.iff is */
/*
CALL IFFImage("SCALEHOST",,
"Rexx:guiPIC.iff",400,20,,,)
*/

CALL POSTMSG()
/* set the gadgets ON and OFF */
CALL SETGADGET(SCALEHOST,REN,OFF)
CALL SETGADGET(SCALEHOST,HELP,ON)

/* put the message in our window */
CALL POSTMSG(275,155,,
   "SCREEN FORMAT (pixels)  wide high val",
  "\----------------------  ---- ---- ---",
"\ \Low Resolution:       X = 320         0",
  "\Overscan:          X+15% = 368         8",
"\ \High Resolution:      X = 640         1",
  "\Overscan:          X+15% = 736         8",
"\ \Non-Interlace:        Y =      200    0",
  "\Full Overscan:     Y+20% =      240   16",
  "\Keep Asp Oscan:    Y+15% =      230   16",
"\ \Interlace:            Y =      400    2",
  "\Full Overscan:     Y+20% =      480   16",
  "\Keep Asp Oscan:    Y+15% =      460   16",
  "\-------------------------------------",
  "\Top gadget: scale to pix asp only.    ",
  "\Mid gadget: pix asp AND/OR screen.   ",
  "\-----------------------------------",
  "\EXECUTE button scales according to   ",
  "\lower highlighted gads. Sum of vals  ",
  "\equals screen type.                  ",APS)

EXIT 0
```

```
/* guiMakeVar.rexx make up variables for gui F2.adpro */
OPTIONS RESULTS
PARSE ARG N X Y, mx my
SELECT
   WHEN N = 'C' THEN str= 'C P A D O'
   WHEN N = 'P' THEN str= 'P C A D O'
   WHEN N = 'A' THEN str= 'A P C D O'
   WHEN N = 'D' THEN str= 'D C P A O'
   OTHERWISE str= 'O D C A P'
   END

PARSE VAR str C P A D O .

IF C='C' THEN X=REQUEST(mx,my,"Enter X-Aspect",1,OKAY,CANCEL,APS)
IF C='C' & X~=''THEN Y=REQUEST(mx,my,,
"Enter Y-Aspect",1,OKAY,CANCEL,APS)
IF X=''|Y='' THEN DO
   CALL SETGADGET(SCALEHOST,PC,OFF)
   EXIT 0
   END

"'CALL SETGADGET(SCALEHOST,P"C",ON)'"
"'CALL SETGADGET(SCALEHOST,P"P",OFF)'"
"'CALL SETGADGET(SCALEHOST,P"A",OFF)'"
"'CALL SETGADGET(SCALEHOST,P"D",OFF)'"
"'CALL SETGADGET(SCALEHOST,P"O",OFF)'"

CALL SETENV(dx,X)
CALL SETENV(dy,Y)
IF N='O' THEN R=SETENV(SCO,1);ELSE R=SETENV(SCO,0)
EXIT 0
```

```
/* StartScOnly.rexx an external function for GUI to ADPro scaling */
/* called from SPO.rexx to turn on and off gadgets */

OPTIONS RESULTS

/* get the info in two arguments: X & Y, mouse coords */
PARSE ARG X Y, mx my
```

```
/* logic to figure out what to turn on or off */
SELECT
   WHEN X = '' THEN str= 'C P A D'
   WHEN X = 10 THEN str= 'P C A D'
   WHEN X = 22 THEN str= 'A P C D'
   WHEN X = 69 THEN str= 'D C P A'
   OTHERWISE NOP
   END

PARSE VAR str C P A D .

IF C='C' THEN X=REQUEST(mx,my,"Enter X-Aspect",1,OKAY,CANCEL,APS)
IF C='C' & X~=''THEN Y=REQUEST(mx,my,,
"Enter Y-Aspect",1,OKAY,CANCEL,APS)
IF X=''|Y='' THEN DO
   CALL SETGADGET(SCALEHOST,SDOC,OFF)
   EXIT 0
   END

/* interlock based on above logic */

"'CALL SETGADGET(SCALEHOST,SDO"C",ON)'"
"'CALL SETGADGET(SCALEHOST,SDO"P",OFF)'"
"'CALL SETGADGET(SCALEHOST,SDO"A",OFF)'"
"'CALL SETGADGET(SCALEHOST,SDO"D",OFF)'"

/* watch it work in ADPro */
ADDRESS 'ADPro'
ADPRO_TO_FRONT

/* call the pgm to do the scaling in ADPro */

CALL guiScalePixAsp.rexx mx my X Y

/* post the results */

/* clear the old */
CALL POSTMSG()

/* post the new */
CALL guiPostMsg.rexx

/* back to our window */
CALL SCREENTOFRONT(APS)

EXIT 0
```

```
/*
   guiScalePixAsp.rexx This ARexx program scales according to the pixel
aspect entered by the user in the gui F2.adpro.  Its main purpose
is to properly scale images scanned by an Epson ES-300C which uses
square (1:1) pixels.

   This program allows you to re-scale at a pixel aspect
suitable for the final destination of the image: DPaintIV, DCTV,
etc.  The 1:1 images are too tall and skinny if not rescaled.

***  WARNING: THIS PROGRAM ALTERS YOUR IMAGE DATA!  ***

*/
OPTIONS RESULTS
ADDRESS "ADPro"
/* get the mouse coords, the new aspect X and Y */
PARSE ARG mx my dx dy .

/* get the rest from the environment */
xaspect=GETENV(Xaspect)
yaspect=GETENV(Yaspect)
W=GETENV(xwide)
H=GETENV(yhigh)

/* if no need to act...*/
IF (dx=xaspect & dy=yaspect) THEN DO
   ADPRO_TO_BACK
   answer=REQUEST(mx,my,,
   "Requested Pixel Aspect same as Data!",," Resume "," Exit ",APS)
   IF answer='' THEN DO
      CALL guiTerm.rexx
      EXIT 0
      END
   IF answer='OKAY' THEN EXIT 0
   END

/* scaling math */
Xm=(yaspect/xaspect)
Ym=1

IF dx > dy THEN DO
   Xm=(dx/dy)*(yaspect/xaspect)
   END
   ELSE DO
   Xm=(xaspect/yaspect)*(dy/dx)
   END
IF Xm<1 THEN DO
   Ym=1/Xm
   Xm=1
   END

X=Xm*W
Y=Ym*H
N=0
```

```
SCALE:  /* label clause for go to */
ABS_SCALE X Y
/* if not enough memory, then scale down */
/* try three times and then exit w/ error */
IF RC ~= 0 THEN DO
   IF N>3 THEN DO
      ADPRO_TO_BACK
      r=REQUEST(mx,my,,
      "Scale Failed!!",," Resume "," Exit ",APS)
      IF r='' THEN DO
         CALL guiTerm.rexx
         EXIT 0
         END
      IF r='OKAY' THEN EXIT 0
      END
   X=TRUNC((.75*X)+0.5)
   Y=TRUNC((.75*Y)+0.5)
   N=N+1
   SIGNAL SCALE /* go to SCALE: */
   END

/* if scaling went OK clean up and redefine the Pix asp */
IF RC = 0 THEN DO
   OPERATOR DEFINE_PXL_ASPECT dx dy
   END
ADPRO_TO_BACK
EXIT 0
```

```
/* guiAddons.rexx add gadgets for gui F2.adpro */
OPTIONS RESULTS

/* have we made these before? */
IF GETENV(gad) THEN EXIT 0

CALL ADDGADGET("SCALEHOST",10,250,"HR",,
" High Res   ",,
"'RESULT=SETENV(xpix,640);",
"CALL SETGADGET(SCALEHOST,LR,OFF);",
"CALL SETGADGET(SCALEHOST,HR,ON)'")

CALL ADDGADGET("SCALEHOST",120,250,"LR",,
"  Low Res   ",,
"'RESULT=SETENV(xpix,320);",
"CALL SETGADGET(SCALEHOST,LR,ON);",
"CALL SETGADGET(SCALEHOST,HR,OFF)'")


CALL ADDGADGET("SCALEHOST",10,270,"IL",,
" Interlace ",,
"'RESULT=SETENV(L,2);",
"CALL SETGADGET(SCALEHOST,IL,ON);",
"CALL SETGADGET(SCALEHOST,NL,OFF)'")
```

```
CALL ADDGADGET("SCALEHOST",120,270,"NL",,
" Non IntL  ",,
"'RESULT=SETENV(L,1);",
"CALL SETGADGET(SCALEHOST,IL,OFF);",
"CALL SETGADGET(SCALEHOST,NL,ON)'")

CALL ADDGADGET("SCALEHOST",10,290,"ST",,
" Standard  ",,
"'RESULT=SETENV(OS,0);",
"CALL SETGADGET(SCALEHOST,ST,ON);",
"CALL SETGADGET(SCALEHOST,OS,OFF)'")

CALL ADDGADGET("SCALEHOST",120,290,"OS",,
" Overscan  ",,
"'RESULT=SETENV(OS,1);",
"CALL SETGADGET(SCALEHOST,ST,OFF);",
"CALL SETGADGET(SCALEHOST,OS,ON)'")

/* note how to pass mouse coodinates %x and %y */
CALL ADDGADGET("SCALEHOST",120,310,"YOSET",,
" Y Ovrscan ",,
"'CALL guiSetYOSET.rexx %x %y'")

/* set our environment variables */
"'CALL SETENV(YOSET,1.15)'"
"'CALL SETENV(gad,1)'"
"'CALL SETENV(xpix,640)'"
"'CALL SETENV(L,2)'"
"'CALL SETENV(OS,0)'"

/* interlock to set gads on and off */
"'CALL SETGADGET(SCALEHOST,HR,ON)'"
"'CALL SETGADGET(SCALEHOST,IL,ON)'"
"'CALL SETGADGET(SCALEHOST,ST,ON)'"

EXIT 0
```

```
/* guiSetYOSET.rexx requester for setting YOSET */

OPTIONS RESULTS

/* find mouse coords */
PARSE ARG mx my .

/* put up requester */
X=REQUEST(mx,my,"Set %Y-Overscan",,,
" 15% Keep Aspect "," 20% Fill Screen ",APS)

/* deal with result string */
IF X="OKAY" THEN CALL SETENV(YOSET,1.15)
ELSE CALL SETENV(YOSET,1.2)
```

```
Y=GETENV(YOSET)
IF Y=1.15 THEN t='15% Y Overscan: Keep Image Aspect'
IF Y=1.2 THEN t='20% Y Overscan: Fill Screen'

/* put the changes in the message window */
/* notice the syntax \ for skipping lines */

CALL POSTMSG(,,"\\\\\\\\\\\\\\\\\\\\"t,APS)

EXIT 0
```

```
/*
   guiEX.rexx This ARexx program scales according to the pixel aspect
entered by the user in the gui F2.adpro.  Its main purpose is to
properly scale images scanned by an Epson ES-300C which uses square
(1:1) pixels.

   This program allows you to re-scale at a pixel aspect suitable for
the final destination of the image: DPaintIV, DCTV, etc.  The 1:1 images
are too tall and skinny if not rescaled.

   If an Amiga screen format is chosen, then the program scales to fit
the screen using the pixel aspects and Amiga Screen formats. You may
scale between screen size as well without distortion to convert between
screen formats and/or overscan.

***  WARNING: THIS PROGRAM ALTERS YOUR IMAGE DATA!  ***

*/

OPTIONS RESULTS
/* get the mouse coordinates */
PARSE ARG mx my .

ADDRESS "ADPro"
/* retrieve our environment variables */
stype=GETENV(stype) /* screen type */
xpix=GETENV(xpix)   /* Hi res or Low res? */
L=GETENV(L)         /* Interlace? */
OS=GETENV(OS)       /* Overscan?  */
dx=GETENV(dx)       /* new X Pixel Aspect */
dy=GETENV(dy)       /* new Y Pixel Aspect */
xaspect=GETENV(xaspect)  /* old pixel aspects...*/
yaspect=GETENV(yaspect)
W=GETENV(xwide)          /* image width and height */
H=GETENV(yhigh)
sco=GETENV(SCO)          /* scale display only? */
YOSET=GETENV(YOSET)      /* Y Overscan percent factor */

/* look for rendered data and act accordingly */
IF DATATYPE(stype)~='NUM' THEN DO
```

```
   answer=REQUEST(mx-15,my-45,,
   "No RENDERED data to scale!",," Smart Render/Scale "," Exit ",APS)
   IF answer='' THEN DO
      CALL guiTerm.rexx
      EXIT 0
      END
   SELECT
      WHEN (xaspect/yaspect) > 1.1 THEN DO
         xpix=320
         L=2
         CALL SETENV(xpix,320)
         CALL SETENV(L,2)
         CALL SETGADGET(SCALEHOST,LR,ON)
         CALL SETGADGET(SCALEHOST,HR,OFF)
         CALL SETGADGET(SCALEHOST,IL,ON)
         CALL SETGADGET(SCALEHOST,NL,OFF)
         END
      WHEN (xaspect/yaspect) < 0.5 THEN DO
         xpix=640
         L=1
         CALL SETENV(xpix,640)
         CALL SETENV(L,1)
         CALL SETGADGET(SCALEHOST,LR,OFF)
         CALL SETGADGET(SCALEHOST,HR,ON)
         CALL SETGADGET(SCALEHOST,IL,OFF)
         CALL SETGADGET(SCALEHOST,NL,ON)
         END
      OTHERWISE DO
         xpix=640
         L=2
         CALL SETENV(xpix,320)
         CALL SETENV(L,2)
         CALL SETGADGET(SCALEHOST,LR,OFF)
         CALL SETGADGET(SCALEHOST,HR,ON)
         CALL SETGADGET(SCALEHOST,IL,ON)
         CALL SETGADGET(SCALEHOST,NL,OFF)
         END
      END /* SELECT */
   END

/* use in test later */
Xwid=W
Yhi=H

/* dealing with overscan */
OX=1
OY=1
Y=200*L
IF OS=1 THEN DO
   OX=1.15
   OY=YOSET
   Y=Y*1.2
   END
```

```
/* scaling math & tests */
X=xpix*OX
A=(yaspect/xaspect)*(dx/dy)
IF dx/dy=xaspect/yaspect THEN A=1
IF SCO=1 THEN A=1

/* screen size goal */
SELECT
   WHEN X=320 & Y=200 THEN DO
      sto=0
      rentype=32
      END
   WHEN X=640 & Y=200 THEN DO
      sto=1
      rentype=16
      END
   WHEN X=320 & Y=400 THEN DO
      sto=2
      rentype=32
      END
   WHEN X=640 & Y=400 THEN DO
      sto=3
      rentype=16
      END
   WHEN X=368 & Y=240 THEN DO
      sto=24
      rentype=32
      END
   WHEN X=736 & Y=240 THEN DO
      sto=25
      rentype=16
      END
   WHEN X=368 & Y=480 THEN DO
      sto=26
      rentype=32
      END
   WHEN X=736 & Y=480 THEN DO
      sto=27
      rentype=16
      END
   OTHERWISE DO
      EXIT 20
      END
   END

/* how to render if there wasn't any screen type */
IF DATATYPE(stype)~='NUM' THEN stype=sto


/* transformation matrices  */
/* initialize */
WX.=1
HY.=1
```

```
WX.0.0 =1
HY.0.0 =A
WX.0.1 =2
HY.0.1 =A
WX.0.2 =1
HY.0.2 =2*A
WX.0.3 =2
HY.0.3 =2*A
WX.0.24=1.15
HY.0.24=OY*A
WX.0.25=2.3
HY.0.25=OY*A
WX.0.26=1.15
HY.0.26=2.4*A
WX.0.27=2.3
HY.0.27=2.4*A

WX.1.0 =0.5
HY.1.0 =A
WX.1.1 =1
HY.1.1 =A
WX.1.2 =0.5
HY.1.2 =2*A
WX.1.3 =1
HY.1.3 =2*A
WX.1.24=0.575
HY.1.24=OY*A
WX.1.25=1.15
HY.1.25=OY*A
WX.1.26=0.575
HY.1.26=2.4*A
WX.1.27=1.15
HY.1.27=2.4*A

WX.2.0 =1
HY.2.0 =0.5*A
WX.2.1 =2
HY.2.1 =0.5*A
WX.2.2 =1
HY.2.2 =A
WX.2.3 =2
HY.2.3 =A
WX.2.24=1.15
HY.2.24=0.6*A
WX.2.25=2.3
HY.2.25=0.6*A
WX.2.26=1.15
HY.2.26=OY*A
WX.2.27=2.3
HY.2.27=1.15*A


/* NEXT COLUMN >>>*/
```

```
WX.3.0 =0.5
HY.3.0 =0.5*A
WX.3.1 =1
HY.3.1 =0.5*A
WX.3.2 =0.5
HY.3.2 =A
WX.3.3 =1
HY.3.3 =A
WX.3.24=0.575
HY.3.24=0.6*A
WX.3.25=1.15
HY.3.25=0.6*A
WX.3.26=0.575
HY.3.26=OY*A
WX.3.27=1.15
HY.3.27=OY*A


/*
make some variables
to use...
*/

B=1/1.15
C=1/.575
D=1/OY
E=1/.6
F=1/2.3
G=1/2.4


WX.24.0 =B
HY.24.0 =D*A
WX.24.1 =C
HY.24.1 =D*A
WX.24.2 =B
HY.24.2 =E*A
WX.24.3 =B
HY.24.3 =E*A
WX.24.24=1
HY.24.24=A
WX.24.25=2
HY.24.25=A
WX.24.26=1
HY.24.26=2*A
WX.24.27=2
HY.24.27=2*A


/* NEXT COLUMN >>>*/
```

```
WX.25.0 =F
HY.25.0 =D*A
WX.25.1 =B
HY.25.1 =D*A
WX.25.2 =F
HY.25.2 =E*A
WX.25.3 =B
HY.25.3 =E*A
WX.25.24=0.5
HY.25.24=A
WX.25.25=1
HY.25.25=A
WX.25.26=0.5
HY.25.26=2*A
WX.25.27=1
HY.25.27=2*A

WX.26.0 =B
HY.26.0 =G*A
WX.26.1 =C
HY.26.1 =G*A
WX.26.2 =B
HY.26.2 =D*A
WX.26.3 =C
HY.26.3 =D*A
WX.26.24=1
HY.26.24=0.5*A
WX.26.25=2
HY.26.25=0.5*A
WX.26.26=1
HY.26.26=A
WX.26.27=2
HY.26.27=A

WX.27.0 =F
HY.27.0 =G*A
WX.27.1 =B
HY.27.1 =G*A
WX.27.2 =F
HY.27.2 =B*A
WX.27.3 =B
HY.27.3 =D*A
WX.27.24=0.5
HY.27.24=0.5*A
WX.27.24=1
HY.27.25=0.5*A
WX.27.26=0.5
HY.27.26=A
WX.27.27=1
HY.27.27=A


/* NEXT PAGE >>>*/
/* transform */
H=H*HY.stype.sto
W=W*WX.stype.sto

/* scale to screen (see Fig 2) */
Y0=(X/W)*H
X0=(W/H)*Y

IF X0 > X THEN Y=Y0
IF Y0 > Y THEN X=X0

/* see if we really need to act */
IF dx=xaspect & dy=yaspect & X=Xwid & Y=Yhi THEN DO
   CALL REQUEST(120,320,,
   "No differences to scale/aspect!",," Resume ",,APS)
   EXIT 0
   END

/* do it in ADPro */
ADPRO_TO_FRONT
ABS_SCALE X Y
IF SCO~=1 THEN OPERATOR "DEFINE_PXL_ASPECT" dx dy
ELSE OPERATOR "DEFINE_PXL_ASPECT" xaspect yaspect
SCREEN_TYPE sto
RENDER_TYPE rentype
EXECUTE
ADPRO_DISPLAY
ADPRO_UNDISPLAY
ADPRO_TO_BACK
ADDRESS SCALEHOST WINDOWTOFRONT
CALL guiPostMsg.rexx
EXIT 0
```

```
/* guiCPA.rexx change Pixel Aspect gui for F2.adpro */
OPTIONS RESULTS
PARSE ARG mx my .
start:
X=REQUEST(mx,my,"Enter New Pixel Aspect X:Y",,
"10:11"," OKAY "," CANCEL ",APS)
PARSE VAR X xa ':' ya .
IF X='' THEN EXIT 0
IF ~DATATYPE(xa,'N') THEN SIGNAL start
IF ~DATATYPE(ya,'N') THEN SIGNAL start
CALL SETENV(Xaspect,xa)
CALL SETENV(Yaspect,ya)
ADDRESS "ADPro"
OPERATOR DEFINE_PXL_ASPECT xa ya
text='Pixel Aspect X:Y   = 'xa':'ya
CALL POSTMSG(,,"\\\\\\\\\\\\\\\"text"\\\\",APS)

EXIT 0
```

```
/* guiTerm.rexx terminate PGM for F2.adpro */

OPTIONS RESULTS

/* get ADPro to front */
IF SHOW('P','ADPro') THEN DO
   ADDRESS 'ADPro'
   ADPRO_TO_FRONT
   END

/* address our host */
ADDRESS SCALEHOST
/* get rid of all those env vars */
CALL SETENV(G)
CALL SETENV(H)
CALL SETENV(gad)
CALL SETENV(OS)
CALL SETENV(xpix)
CALL SETENV(ypix)
CALL SETENV(L)
CALL SETENV(xaspect)
CALL SETENV(yaspect)
CALL SETENV(xwide)
CALL SETENV(yhigh)
CALL SETENV(dx)
CALL SETENV(dy)
CALL SETENV(stype)
CALL SETENV(SCO)
CALL SETENV(YOSET)

/* get rid of message window */
CALL POSTMSG()

/* get rid of window and quit host */
CALL QUIT(SCALEHOST)

/* get rid of screen (if any) */
CALL CLOSESCREEN(APS)

EXIT 0

/* END OF ADPRO GUI AND SCALE UTILITY PROGRAM LISTINGS */
```

UNIarray.rexx Solution to Exercise on Page 5-29

```
/*
** UNIarray.rexx Uniword.rexx re-written to use multi-dimensional arrays
*/

SAY 'Input filename and path.'
PULL infile
rcode=20
IF OPEN('textfile',infile,'READ') THEN DO
   rcode=0
   listed.=0
   m=1
   i=1
   DO WHILE ~EOF('textfile')
      line.i=READLN('textfile')

      IF line.i ~= '' THEN DO
         j=1
         DO WHILE line.i~=''
            PARSE VAR line.i word.i.j line.i  /* 2-Dimensional array */
            word=word.i.j
            /* NOTE! You cannot use an array as a NODE! */
            IF listed.word THEN ITERATE
            /* Get rid of punctuation at end and beginning of words. */
            DO FOREVER
               IF ~DATATYPE(RIGHT(word,1),MIXED) THEN,
                word=LEFT(word,LENGTH(word)-1)
               IF ~DATATYPE(LEFT(word,1),MIXED) THEN,
                word=RIGHT(word,LENGTH(word)-1)
               IF DATATYPE(LEFT(word,1),MIXED),
                &DATATYPE(RIGHT(word,1),MIXED),
                THEN LEAVE
               IF LENGTH(word)=0 THEN LEAVE
               END
            IF word='' THEN ITERATE
            listed.word=1
            list.m=word
            m=m+1
            j=j+1
            END
         END
      i=i+1
      END /* DO WHILE ~EOF */
   i=i-1
   IF line.i = '' THEN i=i-1
   END /* IF OPEN */
ELSE SAY 'Could not open your file!'
```

```
/* The Shell Sort  Feed the array to Shell Sort directly! */
listlength = m-1
span = 1
DO WHILE (span < listlength); span = span * 2; END
DO WHILE (span > 1)
   span = span % 2
   numpairs = listlength - span
   DO node = 1 TO numpairs
      nextnode = node + span
      IF list.node > list.nextnode THEN
         DO
         store = list.nextnode
         list.nextnode = list.node
         DO bubpos = node-span TO 1 BY -span WHILE (store < list.bubpos)
            nextnode = bubpos + span
            list.nextnode = list.bubpos
            END bubpos
         bubpos = bubpos + span
         list.bubpos = store
         END
      END node
END
/* the end of the shell sort of the words */
DO m=1 TO listlength
   SAY list.m
   END
EXIT rcode
```

Searching Large Text Files

Extracting Bible Verses

The Disk Bible used here is from **BibleReader!** by EasySoft! Software 10006 Covington Dr. Huntsville, AL 35803

This suite of programs demonstrates a method for searching a group of very large text files to extract specified paragraphs. Since everyone is familiar with the divisions of a Bible into books, chapters, and verses, it makes a good model. The code may be used as is if you have an ASCII Bible on disk with the format indicated, or you may easily change the code to reflect a different indexing format, and apply the program to any large sequentially indexed data base. The files are found in the **bv.rexx** listing. The filename is (with a few exceptions) composed of the first three letters of the book, a period (.) and the order found in the Bible: **GEN.1** through **REV.66**. Each verse is labeled with the book abbreviation, and a colon (:) between chapter and verse. Thus **GEN 1:1** precedes the first verse in the first book.

SEEK() function
H 63
C 10-114

There are some interesting problems to consider in performing a search. Since it would be very slow to read each line in turn to get to the last verse in Genesis, we use a modified binary search. The ARexx SEEK() function allows us to find the end of the book, and then we back up a few characters to insure that the program position is *before* the next chapter reference. Then read forward and find out how many chapters are in the book, and use that number to make a ratio with which to do another SEEK(). The program reads forward again, and finds and tests the next chapter reference, and either continues to read forward, or backs up again depending upon the outcome. Eventually it finds the first verse of our desired extraction, or tells us we have requested a bogus reference. Several interesting error handling routines are embedded in these programs, to insure that valid references are requested, since the length of books, chapters and verses varies widely, and obeys no particular pattern. The program uses logic exclusively, instead of any sort of prepared index. This logic and the extensive string handling present some good exercises in ARexx.

The four programs comprise two different "front end" programs, one to run from a shell (**bv.rexx**) and the other to launch from a hypertext

Thinker
by
Poor Person Software
3721 Starr King Circle
Palo Alto, CA 94306
(415) 493-7234

program called **Thinker** (**bv.thnkr**). The program called **bv2.rexx** actually does the searching. The program **bft.thnkr** formats the extracted verses into a suitable hypertext format for use inside Thinker. If you do not have Thinker, just copy **bv.rexx** and **bv2.rexx**. The use of front end programs to suit your different applications is an economical way to code your programs. Notice the way we pass several arguments to the exterior program. One of these, **typ**, is used to signal the way in which messages are returned, one way to display a message in the Thinker window, and another way to SAY the error message in the WShell console window. It is quite easy to use this type of flag to take care of small differences between applications using the same core program.

Bible Verse Extraction Program Listings

```
/* bv.rexx Input Bible Verses Output to ram:bcut temporary file */
/* Clear the temporary file in case it's been used */
RC=OPEN('bvout','RAM:bcut','WRITE')
RC=CLOSE('bvout')
answer = Y
flag=0
book=''
entry:
DO WHILE answer ~= N
   SAY 'Enter book aaaa or aa# (e.g. john, judg(es), jo2 = II JOHN)(X=EXIT)'
   PARSE UPPER PULL inbook range1 range2
   CALL TIME('r')
   PARSE VAR inbook inbook 4 inrest
   IF inbook='X' THEN LEAVE
   range=range1 range2
   IF (inbook='' & inrest='' & book~='' & range='') THEN
      DO
        inbook=book
        inrest=rest
        SAY 'Still in book of' book
      END
   book=inbook;rest=inrest
   IF inbook='' THEN
      DO
        SAY 'No book input...'
        SIGNAL entry
      END
   IF range='' THEN DO
   SAY 'Input <> first(f) and last(l) chapter(c) and verse(v)'
   SAY 'in the format <fc:fv> [<lc:lv>] | <fc:> whole chapter | <X> to exit'
   PARSE UPPER PULL range
   END
   IF range='X' THEN LEAVE
   typ='wsh'
   CALL bv2.rexx book, rest, range, typ
   IF result=12 THEN EXIT
   flag=1
   SAY TIME('e') 'secs.'
   SAY 'Add (append) some more verses to file? Y/N'
   PARSE UPPER PULL answer 2 .
   IF ANSWER~=N THEN answ='YES, add verses. Type [Rtn] to keep same book.'
   IF answer=N THEN answ='NO, no more verses.'
   SAY answ
END
IF flag=0 THEN SAY 'Aborting program...no results written to file.'
IF flag=1 THEN SAY 'Exiting program...results will be in file ''RAM:bcut''.'
EXIT 0
```

```
/* bv2.rexx Extract Bible Verses Output to ram:bcut temporary file */
PARSE UPPER ARG book, rest, range, typ
PARSE VAR RANGE fc ':' fv ' ' lc ':' lv
chap=0
IF fv='' THEN DO
  /*whole chapter*/
  fv=1
  chap=1
  END /*whole chapter*/
IF lc=''THEN lc=fc
IF lv='' THEN DO
  lv=fv
  range = fc':'fv' 'lc':'lv
  END
ok=0
realrange = range
endnum = LASTPOS(':',range)+1
versend = SUBSTR(range,endnum)+1
range = LEFT(range,endnum -1)||versend
IF book = GEN THEN; book = GEN.1
IF book = EXO THEN; book = EXO.2
IF book = LEV THEN; book = LEV.3
IF book = NUM THEN; book = NUM.4
IF book = DEU THEN; book = DEU.5
IF book = JOS THEN; book = JOS.6
IF book = JDG THEN; book = JDG.7
IF (book = JUD)&(left(rest,1) = G) THEN; book = JDG.7
IF book = RUT THEN; book = RTH.8
IF book = SA1 THEN; book = SA1.9
IF book = SA2 THEN; book = SA2.10
IF book = KI1 THEN; book = KI1.11
IF book = KI2 THEN; book = KI2.12
IF book = CH1 THEN; book = CH1.13
IF book = CH2 THEN; book = CH2.14
IF book = EZR THEN; book = EZR.15
IF book = NEH THEN; book = NEH.16
IF book = EST THEN; book = EST.17
IF book = JOB THEN; book = JOB.18
IF book = PSA THEN; book = PSA.19
IF book = PRO THEN; book = PRO.20
IF book = ECC THEN; book = ECC.21
IF book = SON THEN; book = SON.22
IF book = ISA THEN; book = ISA.23
IF book = JER THEN; book = JER.24
IF book = LAM THEN; book = LAM.25
IF book = EZE THEN; book = EZE.26
IF book = DAN THEN; book = DAN.27
IF book = HOS THEN; book = HOS.28
IF book = JOE THEN; book = JOE.29
IF book = AMO THEN; book = AMO.30
IF book = OBA THEN; book = OBA.31
IF book = JON THEN; book = JON.32
IF book = MIC THEN; book = MIC.33
IF book = NAH THEN; book = NAH.34
IF book = HAB THEN; book = HAB.35
IF book = ZEP THEN; book = ZEP.36
IF book = HAG THEN; book = HAG.37
IF book = ZEC THEN; book = ZEC.38
IF book = MAL THEN; book = MAL.39
IF book = MAT THEN; book = MAT.40
IF book = MAR THEN; book = MAR.41
IF book = LUK THEN; book = LUK.42
IF book = JOH THEN; book = JOH.43
IF book = ACT THEN; book = ACT.44
IF book = ROM THEN; book = ROM.45
IF book = CO1 THEN; book = CO1.46
IF book = CO2 THEN; book = CO2.47
IF book = GAL THEN; book = GAL.48
IF book = EPH THEN; book = EPH.49
IF (book = PHI)&(left(rest,2) = LI) THEN book = PHI.50
IF book = COL THEN; book = COL.51
IF book = TH1 THEN; book = TH1.52
IF book = TH2 THEN; book = TH2.53
IF book = TI1 THEN; book = TI1.54
IF book = TI2 THEN; book = TI2.55
IF book = TIT THEN; book = TIT.56
IF book = PHM THEN; book = PHM.57
IF (book = PHI)&(left(rest,2) = LE) THEN book = PHM.57
IF book = HEB THEN; book = HEB.58
IF book = JAM THEN; book = JAM.59
IF book = PE1 THEN; book = PE1.60
IF book = PE2 THEN; book = PE2.61
IF book = JO1 THEN; book = JO1.62
IF book = JO2 THEN; book = JO2.63
IF book = JO3 THEN; book = JO3.64
IF book = JUD THEN; book = JUD.65
IF book = REV THEN; book = REV.66

SAY 'File is ' book
/* Make this into YOUR path name */
RC=OPEN('bvin','work:kjbible/'book,'READ')
IF RC ~= 1 THEN

SELECT
  WHEN typ='THINK' THEN DO
    'input No such book! Any key exits.'
    EXIT 12
    END
  WHEN typ='WSH' THEN DO
    SAY 'No such book!'
    EXIT 12
    END
  OTHERWISE NOP
  END

RC=OPEN('bvout','RAM:bcut','APPEND')

IF RC = 1 THEN DO
  /*output file sucessfully opened*/
  totv = SEEK('bvin',-250,E)
  teststring = READLN('bvin')

  DO WHILE ~eof('bvin')
    teststring = READLN('bvin')
    PARSE VALUE WORD(teststring,1) WITH bk .
    IF bk=LEFT(book,3) THEN DO
      PARSE VALUE WORD(teststring,2) WITH tc':'tv
      SAY LEFT(book,3) 'has a total of' tc 'chapters.'
      IF fc>tc THEN DO
        IF typ='THINK' THEN 'input No such chapter! Any key will exit.'
        SAY 'No such chapter! Exiting...'
        EXIT 12
        END
    LEAVE
    END
  END

DO i=1 TO 30 BY 3
  /*find neighborhood*/
  est=((totv+250)%tc)*(fc-i)
  IF est<0 THEN est = 0

  IF fc = 1 THEN DO
    newpos = SEEK('bvin',0,B)
    LEAVE
    END

  newpos = SEEK('bvin',est,B)

  DO WHILE ~EOF('bvin')
    teststring = READLN('bvin')
    PARSE VALUE WORD(teststring,1) WITH bk .
    IF bk=LEFT(book,3) THEN DO
      PARSE VALUE WORD(teststring,2) WITH xtc':'xtv
      IF fc < xtc THEN LEAVE
      IF (fc = xtc & fv < xtv) THEN LEAVE
      IF fc = xtc THEN LEAVE
      END
    END

  IF (fc >= xtc & fv >= xtv) THEN LEAVE
  END /*find neighborhood*/

newpos= SEEK('bvin',est,B)
outstring = ''
blankline = ' '

DO UNTIL EOF('bvin')
  /*do until EOF input file*/
  instring = READLN('bvin')

  IF WORD(instring,2) = word(range,1) THEN DO
    /*do if 1st verse found*/
    outstring = outstring||instring

    DO UNTIL (WORD(instring,2) = WORD(range,2)&ok=1)
      /*do until 2nd verse found*/
      instring = READLN('bvin')

      IF EOF('bvin')THEN DO
        /*EOF input*/
    IF (liv < lv|lic < lc)|(tc=lc & fv~=lv) THEN DO
      /*EOF before verse*/
      RC=CLOSE('bvout')
      RC=OPEN('bvout','RAM:bcut','WRITE')
      IF typ='THINK' THEN 'input EOF reached before verse. Any key exits.'
      SAY 'EOF reached before verse. Exiting...'
      EXIT 12
      END /*EOF before verse*/

    outstring = outstring||instring
    RC=WRITELN('bvout',outstring)
    RC=WRITELN('bvout',blankline)
    LEAVE
    END /*EOF input*/

  IF WORD(instring,1) = left(book,3) THEN DO
    /*first line of verse*/
    PARSE VALUE WORD(instring,2) WITH lic':'liv
    IF (liv=lv & lic = lc)THEN ok=1 /*prevent last verse doesn't exst*/

    IF (lic>lc & lv~=fv & ok=0) THEN DO
      /*last verse ~exist*/
      RC=CLOSE('bvout')
      RC=OPEN('bvout','ram:bcut','write')
      IF typ='THINK' THEN 'input Last verse non-existant! Any key exits.'
      SAY 'Last verse non-existant. Exiting...'
      EXIT 12
      END /*last verse ~exist*/

    IF (lic=lc+1&liv=1&ok=1) THEN DO
      /*write last verse before new chapter*/
      RC=WRITELN('bvout',outstring)
      RC=WRITELN('bvout',blankline)
      outstring = ''
      LEAVE
      END /*write last verse before new chapter*/

    IF ((lic>lc|liv>lv) & fc=lc & lv=fv & chap=0) THEN DO
      /*write single verse incl end of chap*/
      RC=WRITELN('bvout',outstring)
      RC=WRITELN('bvout',blankline)
      outstring = ''
      LEAVE
      END /*write single verse incl end of chap*/

    IF (lic>lc & fc=lc & lv=fv & chap=1) THEN DO
      /*write chapter*/
      RC=WRITELN('bvout',outstring)
```

```
          RC=WRITELN('bvout',blankline)
          outstring = ''
          LEAVE
          END /*write chapter*/

        RC=WRITELN('bvout',outstring)
        RC=WRITELN('bvout',blankline)
        outstring = ''
        END /*first line of verse*/

      outstring = outstring||instring
      END /*do until 2nd verse found*/

    IF word(realrange,1)~=word(realrange,2) THEN,
    insert='through' WORD(realrange,2)||' '
    IF (WORD(realrange,1)=WORD(realrange,2)&chap=0) THEN insert=''
    IF (WORD(realrange,1)=WORD(realrange,2)&chap=1) THEN,
       insert='through end of chapter'
    SAY LEFT(book,3) WORD(realrange,1) insert,
    'written to RAM:bcut w/o [Hrt] except between verses.'
    EXIT 0
    END /*do if 1st verse found*/

  END /*do until EOF input file*/

IF typ='THINK' THEN 'input First verse not found. Any key exits.'
SAY 'First verse not found. Exiting...'
EXIT 12
END /*output file sucessfully opened*/

ELSE DO
  SAY 'Output file not there!'
  IF typ='THINK' THEN 'input Output file not there! Any key exits.'
  EXIT 12
  END
```

```
/* bv.thnkr Thinker Host:Input Bible Verses Output to ram:bcut temporary file */
OPTIONS RESULTS
PARSE UPPER ARG inline verses answ insrt .
RC=OPEN('bvout','RAM:bcut','WRITE')
RC=CLOSE('bvout')
answer = Y
flag=0
book=''
entry:
DO WHILE answer = Y
```

```
   'input Enter book aaaa or aa# (ex:john,judg(es),jo2 = II JOHN)(X=exit)'
   inline = RESULT
   PARSE UPPER VAR inline 1 inbook 4 inrest
   IF inbook='X' THEN LEAVE
   IF (inbook='' & inrest='' & book~='') THEN
      DO
        inbook=book
        inrest=rest
      END
   book=inbook;rest=inrest
   IF inbook='' THEN
      DO
        SIGNAL entry
      END
   'Input Input (f)irst/(l)ast (c)hapter:(v)erse <fc:fv lc:lv> X=Exit'
   verses = RESULT
   PARSE UPPER VAR verses 1 range 12
   IF range='X' THEN LEAVE
   typ='think'
   CALL bv2.rexx book, rest, range, typ
   IF result=12 THEN EXIT
   flag=1
   'input Add (append) some more verses to file? Y/N'
   answ = RESULT
   PARSE UPPER VAR answ 1 answer 2 .
END
IF flag = 1 THEN call bft.thnkr
EXIT 0
```

```
/* bft.thnkr Format Bible */
OPTIONS RESULTS
t=OPEN('bbk','RAM:bcut','READ')
'GET CURSOR' /*need this to initialize current statement*/
prevline = ''
firstline = ''
flg = 0
flag = 0
'input Add verses after cursor: S=Same, D=Down, U=Up level;X=Exit.'
letter = RESULT
PARSE UPPER VAR letter 1 level 2
IF level = 'X' THEN EXIT
   DO UNTIL EOF('bbk')
      nowline = READLN('bbk')
      /* If verse is at beginning, then... */
      IF datatype(left(word(nowline,2),1))=num THEN DO
            /* put on labels */
            pos = LASTPOS('  ',nowline,20)
            front = LEFT(nowline,pos-1)
            front ='('WORD(front,1)','WORD(front,2)','front')'
            nowline = front||nowline
            /* nowline is first line of verse with labels */
            IF flg = 1 THEN DO
               firstline = nowline
               nowline = '' /* don't want firstline in next statement! */
               END
            END /* put on labels */
      /* combine nowline and previous lines */
      prevline = prevline||nowline
      flg = 1
      /* should we write the verse to thinker? */
      IF (firstline ~= '')|EOF('bbk') THEN DO
            /* add a statement after current one in Thinker */
            IF level = 'S' THEN 'add after same' prevline
            IF level = 'U' THEN DO
               'add after up' prevline
               level = 'S'
               END
             IF level = 'D' THEN DO
               'add after down' prevline
               level = 'S'
               END
            prevline = '' /* clear line for new input */
            prevline = firstline /* the start of new statmnt */
            firstline = '' /* clear firstline */
            END /* add a statement after current one in Thinker */
      END  /*go back and read another line */
'SAVE'
EXIT 0
```

Table of Equivalent Terminology

Official ARexx Nomenclature	Similar Terms (not rigorous)
Tokens	**Words, entities**
Comment Tokens	Comments
Symbol Tokens	Variables
Fixed Symbol Tokens	Numeric Constants
Simple Symbol Tokens	Variable names
Stem Symbol Tokens	Array names
Compound Symbol Tokens	Array elements
Symbol Values	Variable values
String Tokens	String literal, String, Name
Operator Tokens	Operators
Special Character Tokens	Special Characters
Clauses	**Statements**
Null Clause	Blank line
Label Clause	Label statement
Assignment Clause	Assignment statement
Instruction Clause	Instruction statement
Command Clause	Command statement
Expressions	**Expressions**
Combinations of:	
String Tokens Symbol Tokens Operator Tokens Parentheses	Compound expressions Functions, function calls
Internal Function Call	Subroutine

Note: References to other manuals occur *first*, and are in *italics:*
C=Commodore ARexx manual
H=Hawes ARexx manual
E=Electric Thesaurus manual
P=Proper Grammar manual
T=TurboText manual

more

more

more

About the Author

Merrill Callaway is the author of the monthly ARexx Column in *Amazing Computing* magazine. He holds degrees in Applied Mathematics and Fine Art from Brown University. A former aerospace Logistics Engineer with extensive technical writing and programming experience, he has devoted his full time to the Amiga since 1990. He lives in Albuquerque, NM where he is currently developing a tutorial work on Art Department Professional and pursuing techniques for making original art on the Amiga.

★ The ARexx Cookbook

by Merrill Callaway

A Tutorial Guide to the ARexx Language on the Commodore Amiga® Personal Computer

★ Features

● ***Tutorial Approach*** *Not another reference manual, The ARexx Cookbook is a step by step approach to learning ARexx, in graduated lessons from simple stand alone programs to complex interprocess control programs. Along with presenting good programming techniques, a thorough treatment of parsing, string handling and arrays insures that the readers will have the equipment to write any ARexx programs they need. This book makes ARexx easy!*

● ***Useful Projects*** *Real programs that do useful tasks make up all the examples. You will keep the programs you learn on! Beginners, intermediates, and advanced users of ARexx will find useful programs as well as inspiration and ideas here. There are programs to sort your data, pick out a list of words without duplicates, turn your text editor into a full blown word processor with Thesaurus and Grammar checker, scale IFF images to fit your screen, search large text files, and more. Along the way, you'll learn many powerful programming techniques so that you can write any program you need for your own custom environment.*

● ***ARexx and PostScript*** *There are no other books dealing with ARexx controlling PostScript, but this extremely powerful combination is presented clearly by The ARexx Cookbook. The projects include a slick utility to print envelopes right from the Shell or CLI.*

● ***Multiple Reference Index*** *Page numbers for the Hawes and the Commodore ARexx Manuals, as well as for the manuals of the application software mentioned in the text occur both in the index and in the margins of the text. Every ARexx instruction and function and every application program command is referenced thoroughly.*

● ***Companion Disk*** *All programs in the book, as well as several more are available on disk.*

WHITESTONE
511-A Girard Blvd. SE
Albuquerque, NM 87106
(505) 268-0678

ISBN 0-9632773-0-8

52495

9 780963 277305