

**ARx\_Elements3.ag**

**COLLABORATORS**

	<i>TITLE :</i> ARx_Elements3.ag		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		October 17, 2022	

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>ARx_Elements3.ag</b>	<b>1</b>
1.1	"	1
1.2	ARexxGuide   Basic Elements   Expressions (4 of 5)   FUNCTIONS	1
1.3	... Expressions   Functions (1 of 5)   INTERNAL FUNCTIONS	2
1.4	... Expressions   Functions (2 of 5)   BUILT-IN FUNCTIONS	3
1.5	... Expressions   Functions (3 of 5)   LIBRARY FUNCTIONS	4
1.6	... Expressions   Functions (4 of 5)   EXTERNAL FUNCTIONS	5
1.7	... Expressions   Functions (5 of 5)   FUNCTION ARGUMENTS	6
1.8	ARexxGuide   Basic Elements   Expressions (5 of 5)   OPERATIONS	7
1.9	ARexxGuide   Expressions   NOTE (1 of 1)   CONDITIONALS	7
1.10	ARexxGuide   Expressions   NOTE (1 of 1)   ACCIDENTAL COMMANDS	8
1.11	ARexxGuide   Basic Elements (3 of 4)   CLAUSES	10
1.12	ARexxGuide   Basic Elements   Clauses (1 of 5)   ASSIGNMENT	10
1.13	ARexxGuide   Basic Elements   Clauses (2 of 5)   INSTRUCTIONS	11
1.14	ARexxGuide   Basic Elements   Clauses (3 of 5)   COMMANDS	12
1.15	ARexxGuide   Basic Elements   Clauses   Commands (1 of 5)   HOST	13
1.16	ARexxGuide   Basic Elements   Clauses   Commands (2 of 5)   DEFAULT HOST	14
1.17	ARexxGuide   Basic Elements   Clauses   Commands (3 of 5)   INITIAL HOST	15
1.18	ARexxGuide   Basic Elements   Clauses   Commands (4 of 5)   COMMAND STRINGS	15
1.19	ARexxGuide   Basic Elements   Clauses   Commands (5 of 5)   EXAMPLE	16
1.20	ARexxGuide   Basic Elements   Clauses (4 of 5)   LABELS	17
1.21	ARexxGuide   Basic Elements   Clauses (5 of 5)   NULL clauses	18

# Chapter 1

## ARx\_Elements3.ag

### 1.1 "

AN AMIGAGUIDE® TO ARexx  
by Robin Evans

Second edition (v2.0)

Note: This is a subsidiary file to ARexxGuide.guide. We recommend using that file as the entry point to this and other parts of the full guide.

Copyright © 1993,1994 Robin Evans. All rights reserved.

### 1.2 ARexxGuide | Basic Elements | Expressions (4 of 5) | FUNCTIONS

Functions

~~~~~

Functions are sub-programs that perform a task and then return a value to the calling environment. Functions can be used in either of two forms: The symbol or word that identifies the function is either followed immediately by an opening parenthesis or it is preceded by the keyword `CALL`. (Parentheses may be used even with the `CALL` keyword but are not necessary.)

The definition of a function may come from any of several sources. Two of those sources -- internal and external functions -- are easily created by the ARexx user. The characteristics of each function source is explained in the following sections:

Internal functions

Built-in functions

Library/Host functions

External functions

Whatever the source of a function, its calling sequence is the same. ←

Depending on the function's definition, it might accept a series of values, called arguments, that are passed to the function. The arguments are included after the function name and between the parentheses, if used.

#### Function arguments

When a function call is encountered within a program, ARexx ← searches the

possible sources in the order in which they are listed above unless the function name is enclosed in quotation marks. If the name quoted, ARexx will ignore any internal functions with that name and begin its search for the name at the built-in function list.

This allows a script to change the action of a function from one of the three external sources. In the following example, an internal function uses the same name as a function that might be available as a library function. If the library is available, that function is called (by using the name without the quotation marks). Otherwise, the internal function uses a different method of retrieving the information.

Technique note: Get or set environment variables

Next: OPERATIONS | Prev: Variables | Contents: Expressions

## 1.3 ... Expressions | Functions (1 of 5) | INTERNAL FUNCTIONS

#### Functions defined within a script

~~~~~

As varied and useful as ARexx

built-in functions

are, it is still often

necessary to build functions for specific tasks. ARexx allows programmers to define code within a script as an 'internal function'. Such functions work the same way as the functions described in the Function Reference . They can be passed values through an

argument list

and can return a

value to the calling environment (must return a value, in fact, unless the CALL instruction is used).

Internal functions are sections of code (called 'subroutines') that begin with a

label

clause. They are not otherwise restricted, but do exhibit some unique characteristics.

One of those characteristics concerns the scope of variables -- the way they are treated within the subroutine. By default, internal functions share all variables with the main section of code: any variable defined or changed in one part of the script is defined or altered in the other. That, however, is often not desirable. Functions are used partly because they are isolated code that can be used again in different scripts. Bugs that are difficult to find can occur when a subroutine unexpectedly changes the value of a variable in the calling environment.

The `PROCEDURE` instruction alters the default handling of variables: When used at the beginning of a function definition, it causes the subroutine to get a new set of variables that are not shared with the calling environment. The new variable table prevents accidental changes to a variable in the main program, even if a variable with the same name is used in the subroutine.

Although variables have global scope within a script, several other values inherited by subroutines have local scope by default. These settings can be manipulated within the subroutine without affecting the settings of the calling environment.

The settings established in the calling environment by `ADDRESS`, `SIGNAL`, `OPTION`, `TRACE`, `NUMERIC`, and the `elapsed-timer` are saved when a function is called and restored when control returns to the calling environment.

```
/**/
say address()      >>> REXX
say digits()      >>> 9
say trace()       >>> N
call ChangeSettings
say address()      >>> REXX
say digits()      >>> 9
say trace()       >>> N
exit
```

```
ChangeSettings:
  address FOO
  numeric digits 12
  trace r
  say address()    >>> FOO
  say digits()    >>> 12
  say trace()     >>> R
return
```

Even though several settings are changed in the subroutine `ChangeSettings`, the original values of the main section remain unchanged after the control returns from the subroutine. Tracing output will be generated only for four lines of the subroutine. After the `RETURN`, tracing will be halted.

More information: [ARexx techniques](#)

Next: [BUILT-IN FUNCTIONS](#) | Prev: [Functions](#) | Contents: [Functions](#)

## 1.4 ... Expressions | Functions (2 of 5) | BUILT-IN FUNCTIONS

Built-in functions

~~~~~

Dozens of functions are available to an ARexx program as soon as it runs. These are called 'built-in functions'. The `Function Reference` section of this guide lists and explains each of the built-in functions.

With some significant exceptions (see Compatibility issues ) the built-in functions of ARexx are also available in other implementations of the language such the REXX available with the OS/2 operating system. As long as only instructions and built-in functions are used, it is possible, with some tweaking, to use a REXX program written for another system on the Amiga.

Next: LIBRARY FUNCTIONS | Prev: Internal functions | Contents: Functions

## 1.5 ... Expressions | Functions (3 of 5) | LIBRARY FUNCTIONS

### Function libraries

~~~~~

Another source of ready-made functions is not as easily portable as built-in functions. These are library or host functions that are supplied either in libraries stored in the `libs:` directory or by host programs. To access the functions they provide, host programs must be running or libraries must be added to the system using the `ADDLIB()` function or the `RXLIB` command utility.

One such set of library functions is contained in `rexxsupport.library` which is included as part of ARexx distributions. Functions from that library are also listed and explained in this guide's Function Reference .

Dozens of function libraries have been written for ARexx -- many of them available as public-domain or shareware files. Among the available libraries are some that allow an ARexx program to harness the power of the Amiga's graphic user interface by using windows with buttons, gadgets, and menus. `'rexxarplib.library'` by Willy Langeveld is the classic of the GUI function add-ons. Another is `'rx_intui.library'` by Jeff Glatt. Both allow an ARexx programmer to define screens and windows with buttons, text gadgets, and other input facilities. Functions from `rexxarplib` library are used in the file `ARx_RarpInfoWin.rexx` (part of the ARexxGuide distribution archive) to define the optional clause information window in the ARexxGuide help system

A full suite of requesters is available in any ARexx program using `'rexxreqtools.library'` by Rafael D'Halleweyn, a file that is distributed as part of, and uses, Nico François's `ReqTools.library`. Functions from that library are used for the requesters that show glossary entries in the RQ version of ARexxGuide. The requesters are defined in the file `ARx_GlossaryPort.rexx`, which is included in the `rexx` directory of the RQ archive and in the `Extras` directory of the standard archive.

A new library first released in January, 1994 can simplify parsing of command options. `'RexxDosSupport.library'` by Hartmut Goebel is a small library with powerful functions -- functions that access the operating system's parsing routines. Command line arguments can be read using the same kind of slash-coded templates used to describe OS commands. (Type `'LIST ?'` on the shell to see the template format.) Functions to set and get environmental variables are included in this package, as they are in `rexxarplib.library`. Unlike their REXX counterparts, the pattern-matching functions in `RexxDosSupport` can ignore upper/lowercase distinctions, and can use AmigaDOS wildcard characters to match a pattern in a string.

Those add-on functions are not explained in this edition of ARexxGuide, but are highly recommended for the programmer who hopes to take full advantage of ARexx. They should be available on most networks or BBSes that support the Amiga.

Also see: ARexx Applications List

Some applications -- notably products from Gold Disk -- use a function library rather than the more common command-host to add ARexx support to the programs. There are some disadvantages and some significant advantages to this method. Unlike

commands

, functions return values directly to the

ARexx script's environment. That alone makes use of functions more elegant than using commands. Unfortunately, the standard methods of addressing multiple

hosts

are not available with this type of interface.

Also see ADDLIB()

Next: EXTERNAL FUNCTIONS | Prev: Built-in functions | Contents: Functions

## 1.6 ... Expressions | Functions (4 of 5) | EXTERNAL FUNCTIONS

Functions stored as ARexx files

~~~~~

An 'external function' is an ARexx script or a program in another language that returns a value of some sort and can therefore be called from any script as a function. These programs are completely isolated from the calling script and can communicate only through the argument string and the return value.

To locate an external function, ARexx follows the same search path used to find scripts launched with the RX command. The file extension is therefore significant. A file named 'rexx:Afunc.rexx' or a file named 'Afunc.rexx' in the current directory can be invoked from a shell-launched script with an expression similar to:

```
NewValue = Afunc('An argument')
```

If the function were used in a script launched as a macro, however, it would often not be found with this name. The default extension for ARexx files is changed in most non-shell environments. Because '.rexx' is no longer the default extension, it will not be added to the function name. Instead, the application-specific extension would be used. In TurboText, for instance, the default extension is '.ttx' so a file named 'rexx:Afunc.ttx' would be invoked if the assignment above were used in a macro started from that text editor. To assure that it will be accessible from any environment, a full file name -- including the extension -- can be used as a function name. If a file path is used in the function name, it must be quoted to avoid misinterpretation of the characters ':' and '/'.

The default extension in the current environment can be retrieved with the



instruction `PARSE SOURCE` which also indicates whether the executing script was called as a function or as a command. That can be useful information because the arguments passed to a command are treated differently than those passed to a function: Commands receive a single argument string in which commas have no special meaning. Functions, on the other hand, recognize commas within an argument string and will split the string at the position of commas into multiple strings assigned to sequential

argument slots

.

Next: ARGUMENTS | Prev: Library functions | Contents: Functions

## 1.7 ... Expressions | Functions (5 of 5) | FUNCTION ARGUMENTS

Function arguments

~~~~~

Although they can be used in a number of different ways, functions often act on data supplied by the calling environment and then return a modified version of that data. Information is passed to the function through "arguments" that must follow a syntax determined by the function being called.

Multiple arguments can be sent to a function. A comma is used to separate each argument. It is sometimes necessary to include an argument while leaving a prior argument slot blank. That can be done by using two commas:

```
Ports = show('L',, '0a'x)
```

The second argument is omitted in the example above, but its place is indicated by use of a comma.

The syntax of each built-in function is explained in this guide's Function reference . Although functions often include optional arguments that may be omitted, an error will result if an argument required by a

```
built-in
or
library
function is missing:
```

```
+++ Error 17 in line 1: Wrong number of arguments
```

Error 17 would, for instance, be generated by `'Foo = left(5)'` since the function requires two arguments.

Although variables in ARexx use a natural typing that does not distinguish between types of data, functions often require a certain datatype. An error results if a built-in or library function is sent the wrong datatype as an argument:

```
+++ Error 18 in line 1: Invalid argument to function
```

The second argument to `LEFT()` , for instance, must be an expression that results in a number. If it isn't, error 18 will be generated.

Internal functions might also require arguments of a certain type, determined by the subroutine that defines the function. ARexx will not generate either of the above errors for an internal function because it has no way to know what is required. The program should include code to check arguments to assure that they are the proper type of data.

Multiple templates can be used with the `PARSE ARG` instruction to retrieve each of the arguments sent to an internal function or the `ARG()` function can be used.

Next: [Functions](#) | Prev: [External functions](#) | Contents: [Functions](#)

## 1.8 ARexxGuide | Basic Elements | Expressions (5 of 5) | OPERATIONS

Operations

~~~~~

An expression is often built up from at least two other elements of the language or from the results of other subexpressions, with the terms tied together by an operator. `'3 + 5'`, for example, is a simple expression comprising two constant symbols tied together by the addition operator. It gives a result of `'8'`.

An operator is the glue that binds the terms of an expression together and operates on them in some way to produce a (usually different) result.

The operators can be divided into four basic groups:

- Concatenation
- Arithmetic
- Comparative
- Logical

Most of them are dyadic operators, which means they are placed between a pair of terms. A few prefix operators are also supported. They are placed immediately to the left of a term that is thereby modified.

Next: [Expressions](#) | Prev: [Functions](#) | Contents: [Expressions](#)

## 1.9 ARexxGuide | Expressions | NOTE (1 of 1) | CONDITIONALS

Conditional (True/False) expressions

~~~~~

A conditional is an expression that results in a value of either 1 (true) or 0 (false). Conditional expressions (which might also be called 'logical expressions') and the instructions that use them account for much of the apparent intelligence (or lack of it) in a computer program. When used with the instructions `IF` or `SELECT` , conditionals allow different tasks

to be performed under different conditions. Conditional expressions are often used in variations of the DO instruction to control the number of times a section of code is repeated.

The two simplest conditional expressions in ARexx are:

```
1
0
```

An instruction like 'if 1 then <action>' is valid in ARexx since '1' is a valid conditional. Such a statement would be meaningless, though, because it would always do the same thing.

When variables are used as simple conditionals, however, even a simple conditional like 'DoSomething = 1; if DoSomething the <action>' is useful. [DoSomething] can be set once and can then control actions in several places in the script.

Comparative operations are a common form of conditional since they allow two values (or two conditions) to be compared and return a value of either TRUE or FALSE. The logical operators allow for a type of arithmetic with TRUE/FALSE values.

ARexx also includes a rich set of functions that can be used directly as conditionals:

```
/**/
pull Input
if abbrev(Input, 'Y') then
    say 'You said "Yes"'
```

Because it returns a Boolean value, ABBREV() can be used directly in any instruction calling for a conditional. The OPEN() function, which is the basis of all file I/O in ARexx, also returns a TRUE/FALSE value indicating the success of the requested operation, making it useful as a conditional.

Several other functions, like SHOW() and DATATYPE() include options that make it possible to use them as conditional expressions.

Next, Prev, & Contents: EXPRESSIONS

## 1.10 ARexxGuide | Expressions | NOTE (1 of 1) | ACCIDENTAL COMMANDS

Avoiding accidental commands

~~~~~

Expressions are usually linked with other elements of the language to form an

```
instruction
or
assignment clause
-- a statement that can be
```

executed by the ARexx interpreter. If an expression is used without other language elements, it is treated as a command

clause. The result will be sent to the current host to be run in that environment.

A common error in ARexx is to use an expression (especially a function call) without the other elements that would make it a valid instruction or assignment, thus turning its result into an unintentional command for the host.

The following program fragment would generate an error in AmigaDOS:

```
/* Error-generating expression */
ADDRESS COMMAND
OPEN(TFILE, 't:tempfile', 'W')
```

AmigaDOS would generate this error:

```
Unknown command 1
```

ARexx itself would probably chime in with this:

```
1 *-* OPEN(TFile,'t:tempfile','W');
+++ Command returned 20
```

ARexx opened the file, just as it was expected to do, and the function OPEN() sent back a return value of '1' to indicate the success of the operation, but since the expression -- the function call -- was entered by itself on a line, ARexx considered it to be a command clause and therefore submitted the return value, '1', to AmigaDOS. It is fortunate that AmigaDOS does not have a command named '1' because if it did, it would have run that command. Instead, AmigaDOS reported the error and sent a return code of '20' back to ARexx.

To avoid such errors, expressions must be a part of an instruction or an assignment clause. Either of the following alternatives would be acceptable:

```
/* the expression is used in an assignment clause */
foo = OPEN(TFile,'t:tempfile','W');
/* the expression is used in an instruction */
if OPEN(TFile,'t:tempfile','W') then;
/* ... */
```

The CALL instruction can be used when the value returned by a function is unimportant (although the return value can still be retrieved in the RESULT variable):

```
CALL CLOSE TFile
```

Also see Error codes

Next, Prev, and Contents: EXPRESSIONS.

## 1.11 ARexxGuide | Basic Elements (3 of 4) | CLAUSES

Clauses -- the working statements of the language

~~~~~  
 If tokens are the atoms of ARexx, then clauses form its ever-changing table of elements. A clause is the smallest language unit that can be executed as a statement.

ARexx recognizes five types of clauses:

Assignment

Instruction

Command

Label

Null

Each clause in ARexx ends with the semicolon token , but since ↵  
 ARexx adds

one automatically at the end of each line and in several other situations, it is rarely necessary for the programmer to type the semicolon.

Next: Basic elements | Prev: Expressions | Contents: Basic elements

## 1.12 ARexxGuide | Basic Elements | Clauses (1 of 5) | ASSIGNMENT

Assignment clause

~~~~~  
 The '=' token does double-duty in ARexx, serving both as a comparative operator and as the indicator of an assignment clause in which a value is assigned to a variable .

An assignment clause takes this form:

```
<symbol> = <expression>;
```

<symbol> can be any of the three types of variable symbols . It becomes a placeholder for the value of <expression>, which may be any type of value -- numeric or text, or any combination of the two.

<symbol> can be written in uppercase or lowercase letters or a mixture of the two. The mixture can even change within the same program. ARexx always translates the symbol into uppercase for its internal symbol table.

Example:

```
/* When a symbol is uninitialized, its value is its name */
/* translated to uppercase                               */
say foo                                     >>> FOO

/* Assign a literal string to the variable [foo]       */
```

```

foo = 'Molloy, your region is vast'
say foo                                >>> Molloy, your region is vast

/* There's no distinction in ARexx between a variable */
/* that represents a numeric value and one that */
/* represents a string */
foo = 'Flooeey' /* holds a string value */
say Foo          >>> Flooeey
foo = 5          /* now it holds a numeric value */
say FOO + 8      >>> 13

```

Also see: Basic elements: Variables

Next: INSTRUCTIONS | Prev: Clauses | Contents: Clauses

## 1.13 ARexxGuide | Basic Elements | Clauses (2 of 5) | INSTRUCTIONS

### Instruction clause

~~~~~

Instructions are the basic control mechanism for ARexx scripts. An instruction may include several clauses, but always begins with a REXX keyword which must be the first token in the initial clause. Each instruction has a unique form, its syntax, which is explained in a major section of this guide: Instruction reference .

Some instructions -- notably DO , IF , and SELECT -- are composed of several distinct clauses and work in concert with other matched keywords (like END , THEN , and WHEN ).

The instruction keyword must be the first symbol in a clause and must be entered as a literal symbol since ARexx will not make a substitution for a variable that might have the value of a keyword.

The symbols used as keywords in ARexx are not reserved except within the narrow range of the instruction within which the keyword is used. Although the practice will usually lead to unnecessary confusion, it is valid in most situations to use keyword names as variables. The following fragment will not produce an error:

```

/* no error */
Now = date()
Then = '01 Apr 69'
SAY Now '&' Then          >>> 09 May 1993 & 01 Apr 69

```

The following fragment, however, will produce an error since the keyword 'then' is reserved within the range of the entire IF instruction:

```

/* invalid use of sub-keyword */
Now = date()
Then = '01 Apr 69'
IF Now > Then          +++ Error 41 in line 4: Invalid expression
THEN
  SAY Now '&' Then

```

In some of the examples included in this guide, and in the syntax diagrams

in the reference section, the keyword of an instruction is entered in uppercase letters. That is done only for the sake of clarity: A keyword may be entered in any case, or in a mixture of upper- and lowercase.

More information:

Avoiding accidental commands from expressions  
 Next: COMMANDS | Prev: Assignments | Contents: Clauses

## 1.14 ARexxGuide | Basic Elements | Clauses (3 of 5) | COMMANDS

Command clause

~~~~~

Because the language can be extended with command clauses, ARexx is able to serve as the macro language for different programs or issue AmigaDOS commands. Because commands within a single ARexx script can be directed to different hosts, several programs can interact with one another through ARexx.

A command is defined in ARexx not by what it is, but by what it is not: A command is an expression that does not meet any of the other rules for clauses. It is, in other words, a statement that isn't an assignment

```
isn't an
    ,
    instruction
    , isn't a
    label
    , and isn't a
    null
    clause.
```

The negative definition is necessary because ARexx makes no attempt to determine whether the expression is a valid command for the environment that will receive it. ARexx has no way to know which commands will be understood and which ones will generate an error, so it will submit to the current host anything it can't otherwise make sense of.

More information:

Avoiding accidental commands  
 The outside environment to which commands are submitted is called ←  
 the

'host'. It can be changed with the ADDRESS instruction. The following nodes explain hosts and commands in more detail:

Command host: what is it?

The default host

Determining the initial host

Entering commands in a script

Example script

When a command is issued in a script, ARexx will wait until the ←  
command  
sets its return code before execution continues.

Also see: Keyboard macros a tutorial

Next: LABELS | Prev: Instructions | Contents: Clauses

## 1.15 ARexxGuide | Basic Elements | Clauses | Commands (1 of 5) | HOST

Command host

~~~~~

When the interpreter encounters a command, it is sent as a text string to an external environment, which may then do something with the string. The environment that receives the command is called the host.

The host might be AmigaDOS, a text editor, a drawing program, an image processing program, a page-layout program, a database manager, a terminal program. It can be nearly any application that runs on the Amiga.

ARexx itself cannot force a program to accept commands and cannot determine the nature of the commands that will be acceptable to the host. The developer of an application must open the public message port to which ARexx will send commands and must make some provisions in the application to deal with the commands sent by ARexx.

The host is referred to within ARexx by the name of the public message port to which commands will be sent. A command `'rx "say show('P',,'0a'x)'` can be used from a CLI to display a list of available message ports.

Message ports are components of the operating system that are widely used for other purposes, so many of the listed ports will not be valid hosts for ARexx commands. (Sending a command to a port that isn't meant to receive it will sometimes cause a system crash.)

The documentation for each program that serves as an ARexx command host should include information about naming conventions for the port or ports opened by the program as well as information about the commands that can be sent to the port.

The ADDRESS instruction changes the host to which commands will be sent. The ADDRESS() function returns the name of the host that is the current target of commands. Commands sent to hosts should return a value of some sort to ARexx. The interpreter will assign values to either or both of two special\_variables:

RC is assigned a numeric code indicating success or failure of a command.

RESULT is assigned a string or numeric value returned by the external environment. This value is available, however, only when the instruction OPTIONS RESULTS is issued prior to the command.

Technique note: Read result of AmigaDOS command



### Compatibility\_issues:

The ANSI committee that is working on a formal definition of the REXX language, avoids making extensions to the language, but has made an exception in the case of ADDRESS.

The committee will probably introduce new features to the ADDRESS instruction that will make it easier to retrieve the output of commands.

The extensions will allow the output of a command to be redirected either to a file stream or to a stem variable. The format would be something like this:

```
ADDRESS <environment> ['<command>'] WITH OUTPUT STREAM PF.OUTPUT
```

```
ADDRESS <environment> ['<command>'] WITH OUTPUT STEM PF
```

If <command> is not included, the redirection would be persistent for all commands issued to the new environment.

Next: DEFAULT HOST | Prev: COMMAND | Contents: COMMAND

## 1.16 ARexxGuide | Basic Elements | Clauses | Commands (2 of 5) | DEFAULT HOST

### A script's default host

~~~~~

Each ARexx script inherits a default host to which commands will be sent until the host is changed with the ADDRESS instruction. When a script is launched with the RX command utility, the initial address for commands is a port called REXX. Most application programs that accept ARexx macros will instruct ARexx to set the initial host to the address of the program that launched the macro. A macro started from TurboText, for instance, might have an initial host address of 'TURBOTEXT1'.

### Toggling between hosts

~~~~~

ARexx maintains two addresses: the current host and the previous host. When a script begins, the current host is the same as the default host. The previous host when a program begins is usually COMMAND -- the operating system host that will execute any AmigaDOS command. When the host is changed with the ADDRESS instruction, the environment that received commands before the instruction was issued becomes the previous host. The toggle form of the ADDRESS instruction will shift between the current and previous hosts.

The REXX host understands one type of command: it will launch other ARexx programs. When a command is sent to the REXX port, the ARexx resident process handles it by looking for a program that matches the name of the first word in the command. ARexx looks for the program in the current directory and then in the REXX: directory. In each directory, it looks first for the exact name as entered and then for a the specified name with the extension '.rexx' added to the name. The script that sent the command will pause until the second script is finished. Other text included with

the command is treated as a command-line argument and can be retrieved with the `ARG` instruction or `ARG()` function in the called script.

AREXX port

~~~~~

A second port, called 'AREXX', is available to ARexx scripts. It acts like the REXX port except that it will launch ARexx scripts without causing the calling script to pause. This is called 'asynchronous' operation and is particularly useful when a script will interact with the calling script through message ports established with the `OPENPORT()` function.

Next: INITIAL HOST | Prev: HOST | Contents: COMMAND

## 1.17 ARexxGuide | Basic Elements | Clauses | Commands (3 of 5) | INITIAL HOST

Determining the initial host

~~~~~

The initial host for a program can be determined in either of two ways. Since the `ADDRESS()` function returns the name of the current host, it can be used at the beginning of a script to record the initial environment:

```
PortOfCall = address()
```

The instruction `PARSE SOURCE SrcStr` will record in the variable `[SrcStr]` information about the way the current script was called. The last word in the string is the name of the initial host for the program. It can be retrieved with these instructions:

```
PARSE SOURCE SrcStr
PortOfCall = word(SrcStr, words(SrcStr))
```

The filename used to invoke the current script is included in the string provided by `PARSE SOURCE`. Since the name might include embedded spaces, the `WORD()` function, which won't be fooled by extra words in the string, will reliably grab the host name from the end of the source string.

Some applications programs include a command that will return the names of ports opened by that program. In TurboText, for instance, the command 'GetPort <Document Name>' will send back as a `RESULT` the name of the port for the window that contains <Document Name>.

Next: COMMAND STRINGS | Prev: Default host | Contents: Command

## 1.18 ARexxGuide | Basic Elements | Clauses | Commands (4 of 5) | COMMAND STRINGS

Entering a command in a script

~~~~~

Commands are entered as strings that are sent to the host. The string may be made up of any expressions, which are evaluated by the interpreter before the command is sent to the host. Any part of the command that is not meant to be evaluated should be enclosed in quotation

marks. The punctuation is not strictly necessary (most commands will be sent to the host even if they aren't quoted), but is highly recommended for these reasons:

- To avoid conflict between an ARexx keyword and the same word used as a command to the external environment.
- To prevent ARexx from misinterpreting characters like the ':' and '/' used in Amiga file names.
- To speed up execution of the script since ARexx will not attempt to evaluate the quoted expression.

An example of a keyword conflict is provided by versions of the Amiga OS prior to 2.1 which included a command 'SAY' that caused the computer to speak a phrase. In ARexx, 'SAY' is also a keyword that causes a string to be output on the shell. How can a program distinguish between the two distinct meanings? Quoting the command makes it apparent that it is a command:

```

/**/
ADDRESS COMMAND

/* ARexx recognizes SAY as a keyword so treats this as      **
** an instruction                                           */
SAY 'Molloy, your region is vast.'

/* Because SAY is quoted, it the whole line becomes an     **
** expression (since a string token is considered an       **
** expression). ARexx won't interpret such an expression as **
** a possible instruction and will therefore send the      **
** entire string to the
   current host
   , AmigaDOS, which      **
** will speak the phrase.                                  */
'say "Molloy, your region is vast"'

```

Next: EXAMPLE | Prev: Initial host | Contents: Command

## 1.19 ARexxGuide | Basic Elements | Clauses | Commands (5 of 5) | EXAMPLE

A macro: Using commands

~~~~~

The following macro program for the TurboText text editor provides an example of how commands work. Notice that the assignment clauses and IF/THEN control structures are pure ARexx. A few ARexx functions -- ADDRESS() , GETCLIP() , and SETCLIP() -- are used as well. Statements like 'RequestBool' and 'SetBookmark' are unfamiliar, though, because they are TurboText commands rather than ARexx instructions.

```

/* $VER: 1.1 SetBkMrk.TTX (12.12.91; 05.18.93) */
/** by Robin Evans
** Sets the next available Bookmark to the current location.
** Indicates bookmark number in title bar

```

```

    **/
options results /* necessary to get info back from TTX */
PoC = address()
LastMark = getclip(Poc'CurMark')
if LastMark == '' then
    LastMark = 0
else
do
    if LastMark = 10 then
    do
        'RequestBool "All bookmarks used." "          Replace Bookmark 1?"'
        if result == 'NO' then exit
        else LastMark = 0
    end
end
CurMark = LastMark + 1
call setclip(Poc'CurMark', CurMark)
'SetBookmark' CurMark
'SetStatusBar Bookmark' CurMark 'set.'

```

This little program was made to be run from within TurboText as a macro. An ADDRESS instruction is not used because the default host will be the TurboText task that called the macro. All of the commands will be sent to that task.

Next: Command | Prev: Command strings | Contents: Command

## 1.20 ARexxGuide | Basic Elements | Clauses (4 of 5) | LABELS

Label clause

~~~~~

A simple symbol followed by a colon ':' creates a label that begins the definition of a subroutine. The code following the label can then become the target of CALL or SIGNAL instructions. If they return a value they can be used as

```

    internal functions
    .

```

Label clauses can be stacked to give the same section of code multiple names. The following construction is valid:

```

Syntax:
Error:
Novalue:
Emergency:
    <program code>

```

Because of the stacked labels, the same section of code would be invoked by each of the following instructions:

```

SIGNAL on syntax
SIGNAL on error
SIGNAL on novalue
CALL Emergency

```

ARexx will supply an implied semicolon after a label, so the symbol and its colon will be considered a clause even if something else follows on the same line, as often happens when `PROCEDURE` and `EXPOSE` are used with a subroutine. Although those keywords could be used on a separate line, they are often included on the same line as the label in this manner:

```
MakeWidget: PROCEDURE EXPOSE Foo
```

The implied semicolon makes that a valid construction.

If a label clause is encountered in the normal flow of a program (that is, outside of the shift caused by a `SIGNAL` or `CALL` instruction or by a function call), then the label clause will be ignored and the program will continue with the first clause following the label. (`PROCEDURE` cannot be used after such a label, however.) If an `EXIT` or `RETURN` instruction is not placed in the flow prior to the area where subroutines are defined, then the program will continue through and try to execute the subroutine statements.

Although it would be considered bad form in many circumstances, it is possible to use `CALL` or -- more often -- `SIGNAL` to direct program flow back to an earlier point in a program by placing a label just prior to that point. This technique provides one of the only ways to recover from an error condition that has been trapped by a `SIGNAL` instruction, but should be unnecessary under normal conditions since the various control structures available in ARexx usually offer a better way to control the flow of a program.

Next: [NULL clauses](#) | Prev: [Commands](#) | Contents: [Clauses](#)

## 1.21 ARexxGuide | Basic Elements | Clauses (5 of 5) | NULL clauses

Null clause  
~~~~~

Null clauses are program elements that don't do anything. The ARexx programmer usually doesn't need to know that they are specifically defined in the language, but the interpreter must pay attention to them so that they can be removed from the program before it is executed.

Null clauses are blank lines, comment tokens, or lines on which the only character used is the semicolon. ARexx ignores these clauses except to trace the beginning of a comment to its closing `'*/'` characters.

Because it recognizes them as an element of the language, ARexx allows and encourages the use of blank lines to enhance the readability of program code. Similarly, the language ignores the blank characters (tabs or spaces) that precede a clause so that code may be indented to make it easier to follow the flow of a program.

A null clause cannot, however, be used in place of an expression. That means that it is not enough to use the following construction that is common in C:

```
If <condition>;
```

---

```
    then <instruction>;  
else;  
    ;
```

The final semicolon is ignored and will not be interpreted (as it would be in C) to mean that no operation should be performed when the ELSE condition is met. Instead, ARexx would bind the following clause to the ELSE keyword. The NOP keyword is provided for instances when it is necessary to state explicitly that no operation is to be performed.

Next: Clauses | Prev: Labels | Contents: Clauses

---