ARx_Notes.ag

COLLABORATORS			
	TITLE : ARx_Notes.ag		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 8, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ARx_Notes.ag

1.1 Comments and other notes

```
AN AMIGAGUIDE® TO ARexx                 Edition: 1.0c
by Robin Evans
```

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, Robin Evans. All rights reserved.

1.2 Finding VALUE()

The VALUE() function is similar to a localized INTERPRET instruction. It allows an expression to be used as a variable name that is then treated as the variable itself would be if entered directly in the clause.

VALUE() may only be used where an expression is expected. It cannot, for instance, be used as the left-value of an assignment clause since only a symbol is valid in that position.

With VALUE(), the contents of one variable can be used to name another variable:

```
      /**/
[1]   SunshineMom = 'Winnie'
[2]   Winnie = 'Foo.1'
[3]   Foo.1 = 'Minnie''s Daughter'
[4]   say SunshineMom                       >>> Winnie
[5]   say value(SunshineMom)                >>> Foo.1
[6]   say value(value(SunshineMom))         >>> Minnie's Daughter
[7]   Child = 'Sunshine'; Relat = 'Mom'
      /* a dynamically constructed variable is used below */
[9]   say value(Child||Relat)               >>> Winnie
      /* [SunshineMom]'s value is output since that's the derived var  */
```

Lines 1 through 3 are standard assignment clauses , just as line 4 is the same kind of SAY instruction used throughout this Guide.

The output of line 5, however, might seem strange. The value of [SunshineMom], is 'Winnie'. It is the name [Winnie] that becomes the object of the SAY instruction, so line 5 outputs the same result as the simpler instruction {SAY Winnie}.

VALUE() is doubled up in line 6, showing that a function can be used as the argument to VALUE(). Two substitutions have taken place here, giving the same result as {SAY Foo.1}.

Line 9 demonstrates the use of variables in a concatenation operation to build the variable name used in the instruction.

As explained in the section on compound variables , the value of the stem symbol -- unlike that of the symbols forming its branches -- is never a target of substitution when ARexx interprets the derived name of a compound variable, but the VALUE() function allows even the stem to have a derived name.

In the second example below, the value of [A] is substituted for the unquoted variable and then concatenated with the string '.1'. The concatenation results in the variable FOO.1 whose value is output by the SAY instruction.

```
foo.1 = 67; a = foo; say a.1                 >>> A.1
foo.1 = 67; a = foo; say value(a'.1')        >>> 67
```

VALUE() can also be used in some circumstances to substitute a value for the branches of a compound variable, which might be useful when the name of one branch is assigned to another compound variable:

```
a.12 = 'foo'; c.1 = 12; say a.c.1;          >>> A.C.1
a.12 = 'foo'; c.1 = 12; say value('a.'c.1)  >>> foo
```

The same result could be obtained more safely, however, by transferring the value of the second compound variable to a simple variable:

```
a.12 = 'foo'; c.1 = 12; Hold = c.1; say a.Hold   >>> foo
```

The argument to VALUE() must be a valid symbol. If it is not an error will be generated:

```
/**/
Name = 'Winnie Foo'; Foo.Name = 1; Test.1 = 'Winnie Foo'
say Foo.Name
say value('Foo.'Test.1)
```

This will output:

```
        1
        +++ Error 31 in line 4: Symbol expected
```

Line 3 will generate the expected value { 1 }, showing that a compound variable with a derived name of 'FOO.Winnie Foo' is valid. Line 4 causes an error because the space in 'Winnie Foo' makes it an invalid symbol name.

The SYMBOL() function can be used to validate the argument to VALUE().

1.3 Formatting output with RIGHT(), LEFT(), and TRUNC()

Because they will pad a string with spaces as well as truncate it, the RIGHT() and LEFT() functions are useful in preparing formatted output for tables and lists.

The following program segment demonstrates the technique:

(The comma continuation character is used in three of the assignments below, allowing the long definition of the assignment to be spread over two line.)

```
/*************  TableFormat.rexx             *************************
**************  Format variables for a table ************************/

Prod = 'Widget wacker'; Price = 99; Code = 'WID01-W'
PrdWd = 20; PrcWd = '9'; CdWd = 8

Heading = center('Product', PrdWd) || center('Price', PrcWd) ||,
   center('Code', CdWd)
Divider = copies('-', PrdWd - 1) copies('-', PrcWd - 1),
   copies('-', CdWd - 1)
Listed = left(Prod, PrdWd) || right(trunc(Price, 2),PrcWd-2)' ',
   left(Code, CdWd)

say Heading; say Divider; say Listed

/* OUTPUTS:    >>>
                        Product         Price    Code
                  ------------------- -------- -------
                  Widget wacker         99.00  WID01-W
*/
/*********************************************************************/
```

In constructing the [Listed] variable, three functions are used: TRUNC() adds two decimal places to [Price] while RIGHT() pads the number with spaces on the left side so that numbers up to four digits will be decimal-aligned.

COPIES() replicates the '-' enough times to produce a dashed divider of the appropriate size under each heading.

All three concatenation operators are used in building the [Listed] variable. The '||' operator is used to prevent an extra space from being introduced between the [Prod] and [Price]. But extra spaces are wanted between [Price] and [Code]. To get them, a one-space string is added to the right side of [Price] using the abuttal operator and another space is added by concatenating that value to [Code] using the space operator.

1.4 Checking unique datatypes with VERIFY()

The VERIFY() function can be used to expand the range of DATATYPE() checking in ARexx since it allows for validation of a specific range of characters. A product number, for instance, might be constructed of any 3-digit number, followed by a dash, followed by any of the upper case letters 'A' through 'G'. The DATATYPE() function can't validate anything that specific. VERIFY() can, however, check for such a limited range of values:

```
/************ CheckProductNum.rexx ********************************/
parse arg Num
   if LENGTH(Num) ~= 7 then do
      say 'Number must be 7 characters'
      exit
   end
      /*  xrange()  returns a string of characters between those     **
      ** specified as its arguments                                  */
   if VERIFY(Num, xrange(0,9)||xrange('A','G')'-') = 0 then
      if DATATYPE(left(Num,3), N) & DATATYPE(right(Num,3), U) then
         say 'Good Number'
      else
         say 'Number is improper format.'
   else
      say 'Invalid data at position' VERIFY(Num, xrange(0,9)||xrange('A','G')'-')
```

1.5 The elapsed time counter

The 'E' and 'R' options to TIME() control a clock that allows an ARexx script to measure time intervals. The clock is started with the first call to either TIME(E) or TIME(R). The result of the first call will always be '0.00'. The next call to TIME(E) will report the interval in the form <ss.tt> where <s> is seconds and <t> is ticks of the internal clock (1/50 second on NTSC systems).

TIME(R) will reset the interval counter to 0.00.

Changes to the interval counter made within a subroutine are local to that subroutine and do not affect the settings of the clock in the calling environment.

Next, Prev, & Contents: TIME()

1.6 Note: Persistence of DATE() and TIME() settings

The DATE() and TIME() settings are persistent within a single clause . A record is made of the initial value of both functions when either of them is first used in a clause. Thereafter, each call within the clause to one of the functions will return the initial value recorded at the first call.

The following, entered as three distinct clauses will return a different value for time() because of the delay() between the clauses:

```
   say time();call delay(100);say time()
                                          >>> 11:45:29
                                          >>> 11:45:31
```

When the function calls are combined into a single clause, however, the value of the first call is returned on both calls to time():

```
   say time() delay(100) time()
                                          >>> 11:45:43 0 11:45:43
```

A call to either date() or time() will freeze the values returned by both functions:

```
   say time();say date() delay(100) time()
                                          >>> 11:54:03
                                          >>> 02 Nov 1993 0 11:54:03
```

This persistence guarantees that calls to the functions will return a consistent value within a single clause.

Next: TIME() | Prev: DATE() | Contents: Information func.

1.7 Locating file names with LASTPOS() and MAX()

It is often necessary to separate a file name in a program from the full path specification. LASTPOS() is ideally suited to this task since it will locate the last divider character '/' even in a deeply nested file specification.

In the following program, the LASTPOS() function is used twice, once to locate the device ':' specification (which could also have been found with POS() since there should be only one colon in the name), and again to find the last directory divider. MAX() , then, returns the larger of those numbers.

```
 /**********  FileName.rexx  ****************************************/

 parse arg FilePath
 DivPos = max(lastpos(':', FilePath),lastpos('/', FilePath)) +1
 parse var FilePath PathSpec =DivPos FileName
 say 'File name is' Filename', located in' PathSpec

 /*********************** End of program segment**********************/
```

In this example, a PARSE instruction was used with [DivPos] as a positional marker . This has the advantage of setting both the [PathSpec] and [FileName] variables at the same time, but it could be replaced with these two calls to SUBSTR() and LEFT() :

```
 PathSpec = left(FilePath, DivPos-1)
 FileName = substr(FilePath, DivPos)
```

1.8 Using ports in ARexx programs

This program fragment demonstrates the use of the repertoire of port functions available in ARexx.

```
/************ Ports example ******************************************/

       /* 'MYPORT' will appear on ports list */
       /* [OPort] holds the address that will be used to close the port */
   OPort = openport('MYPORT')
      /* Loop until a Cmd changes the value of [Status] */
   do until Status = 'CLOSE'
      call waitpkt('MYPORT')
      Packet = getpkt('MYPORT')
         /* Make sure we have a real message in [Packet] */
      if Packet ~= null() then do
         Cmd = getarg(Packet)
            /*               Do something with Cmd                       **
            ** Since the command and its arguments are usually provided **
            ** as a single string, the following could be used as well: **
            **    interpret Cmd                                          **
            ** It's a good idea to check the command, however, to make  **
            ** sure it's valid for this context.                        */
         call reply(Packet, rv)
            /* [rv], above, should be an appropriate return code        */
      end
   end
   call closeport OPort
   exit

/*********************************************************************/
```

Because of the loop at DO UNTIL , this example will keep a port open until it is specifically closed with a command such as 'Status = CLOSED' received from an external process.

Commands would be sent to this process by ADDRESS MYPORT <Cmd> ' where <Cmd> is a command that will be understood by other routines in this program.

1.9 Counting characters using COMPRESS()

Used in conjunction with LENGTH() , the COMPRESS() function presents a way to count characters in a string. The following fragment demonstrates the technique:

```
   /* Count characters */
   Str = 'Molloy|Mollone|Godot|Krapp|'
   CharNum = length(Str)-length(compress(Str,'|'))
   say 'There are' CharNum '"|" characters in' Str'.'
```

If the character counted is used as a field divider, as it is in this example, then this technique will count the number of fields in the string.

Using the SHOW() function, the following fragment will return a count of public message ports even if some of them use names with spaces:

```
/* Count ports */
PLIST = show('P',,'0a'x)
NumPorts = length(PList) - length(compress(PList, '0a'x))
say Numports 'ports are open.'
```

The count can include multiple characters:

```
/* Count digits */
PrdNum = '1289-ABC'
Dig    =  length(PrdNum) - length(compress(PrdNum, xrange(0,9)))
say 'There are' Dig 'digits in "'PrdNum'".'
```

This technique could be generalized as a function. The string should be sent as the first argument to the function and the character(s) to be counted as the second argument.

```
CountChar:
   return length(arg(1)) - length(compress(arg(1), arg(2)))
```

1.10 In-line data

Combined with the special variable SIGL , the SOURCELINE() function provides a way to copy data from the program code. In the following fragment, a range of compound variables is set in this manner:

```
InLineData:
   DataL = GetLine()
   do i = 0 until Data.i.FVal = 'ENDDATA'
      parse value sourceline(i + DataL) with Data.i.FVal Data.i.SVal .
   end
   return

   SendLine:
      return SIGL + 2
   GetLine:
         /* this sets the location of the data to be copied */
      signal SendLine
   /* DATA:
   FooBar 78
   MooBar 98
   FooIsh 89
   ENDDATA
   */
```

The location of the data is determined by calling the internal function GetLine(), which tranfers control, using the SIGNAL instruction, to the subroutine SendLine(). The special variable SIGL is set to the line number of the clause that called the subroutine. Since the clause is known to be two lines above the first line of data, SendLine() returns the proper line number to the calling environment.

1.11 Global variables on the clip list

The clip list gives an ARexx script access to a pool of global variables maintained by the resident process.

Clip list variables are set or cleared in a special way by using the SETCLIP() function or the RXSET command utility. Their values are retrieved using the GETCLIP() function.

Because they retain their values even after the program that sets them exits, clip list variables can be used to maintain user settings called by different scripts.

The following fragment demonstrates how the clip list might be used to hold information for a set of ARexx scripts used as an online message reader. The file containing these instructions can be called by the script that launches the reader. Any other script needing the information can then retrieve (or change) the values set in the initial script.

```
/* Preferences clips for a message reader */
call setclip("Rd_Sig", "Robin Evans")
call setclip("Rd_RepDir", "temp:")
call setclip("Rd_DlDir", "temp:")
call setclip("Rd_MalFile", "cap:Email.snd")
call setclip("Rd_InsName", "1")
call setclip("Rd_InsMsg", "0")
```

Macros in an ARexx command host like TurboText could retrieve values from the clip list whenever needed, giving an overall consistency to a complex set of related scripts. In TurboText and several other programs an in-line script can be bound to a particular key, so that pressing that key will call the macro. The following line from a TurboText key-definitions file would cause a name from the clip list to be inserted in the document when the key combination Alt-I is pressed:

```
ALT-I    ExecARexxString Insert getclip('Rd_Sig')
```

An application using the clip list in this way will need some way to save preferences that were changed while the scripts were running, and should, ideally, clean up the clip list when the values it has set are no longer needed. The following program accomplishes both tasks and could be called by the script that closes the reader:

```
/* Save values from clip list to a file and clear the clips */

if open(PrfFile, "rexx:Rd_Prefs.rexx", 'w') then do
      /* This file will be called as a program, so add comment       */
   call writeln(PrfFile, '/* Preferences clips for a message reader */')
      /* The  SHOW('C')  function returns a list of all clips        */
   Clips = show('C')
      /* The  INDEX()  function is used to verify that there is      **
      ** at least one more clip matching format used by this app.    */
   do while index(Clips, 'Rd_') > 0
         /* An iterative  PARSE  is used to separate the name of     **
         ** each clip.                                               */
      parse var clips "Rd_" OneNam Clips
```

```
            /* The current value is saved in a format that can be      **
            ** called as a subroutine.                                   */
          call writeln(PrfFile, 'call setclip("Rd_'OneNam'", "',
                                  getclip('Rd_RTnam'||OneNam)'")')
            /* Each clip set by the application is now cleared          */
          call setclip('Rd_'OneClip)
       end
       call close(PrfFile)
    end
```

The values in the clips need not be limited to short items like those listed above. They may be used to hold sections of frequently-used code that can be entered in the form of an in-line script and executed using the INTERPRET instruction.

As an example, the string files defined in the TurboText key definitions mentioned above are limited to a length of 255 characters. The limitation isn't severe, since disk macros can be called via key definitions, but there are times when the performance penalty of calling a disk file can be problematic. The clip list provides a middle ground: A complex in-line script that is not bound by the 255 character limitation could be stored on the clip list. The following key definition could then be used to launch the script:

```
ALT-CURSOR_RIGHT  ExecARexxString interpret getclip('Rd_MoveDn')
```

1.12 ARexxGuide | Functions reference | File I/O (1 of 4) | OVERVIEW

```
Overview of file I/O functions
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The most basic of the file I/O functions is OPEN() , which gives the opened file a 'logical name' that the other functions like READLN(), WRITECH(), and SEEK() will then use when acting on that file. The logical name used with the OPEN() function can be any literal string or symbol . The name has significance only for the current script.

The input functions are READLN() , which reads characters from the specified file until an ASCII 13 end-of-line character is encountered, and READCH() , which reads one character by default but can be made to read a specified number of characters.

The complimentary output functions are WRITELN() , which adds a specified string to a file and appends an end-of-line (EOL) character to the string, and WRITECH() , which adds characters to the file without adding the EOL character.

The EOF() function returns a Boolean flag of TRUE or 1 when the end of a file has been reached. The SEEK() function moves to a specified point within the file.

As information is read from or written to a disk file, ARexx (through AmigaDOS) keeps track of the current position within the file with what is called a file pointer. When a file is first opened, the initial position of the pointer is determined by the <mode> argument in OPEN(<handle>, <file name>, <mode>).

The <mode> may be 'R' for read (the default -- used when nothing else is specified), which opens an existing file with the file pointer at the beginning of the file; 'A' for append, which also opens an existing file, but with the file pointer at the end of the file. The OPEN() function will fail and return a value of 0 if a 'R' or 'A' mode is specified for a file that does not yet exist. The third mode option is 'W' for write, which will create a new file or clear out everything in an existing file of the same name.

The mode used to open a file does not affect the other I/O functions. It is possible to read from a file opened in 'W' or 'A' mode and it is possible to write to a file opened in 'R' mode. Unless SEEK() is used to reposition the pointer, however, there will be nothing to read with the file pointer is located at the end of a file as it is in 'A' and 'W' modes. Writing to an existing file with the pointer located at its beginning will overwrite existing data.

The SEEK() function performs two tasks: it returns the current byte location within a file and may be used to move the file pointer to a new location. Because the AmigaDOS file system is byte-oriented rather than line-oriented, there is no simple way to move to the beginning of a new line unless the lines are all of the same length.

AmigaDOS allows for different levels of access protection for opened files. ARexx uses two of those levels. Files opened in write mode are given an exclusive lock: until it is closed, the file cannot be accessed except through use of its ARexx handle by the script that opened the file. Files opened in the other modes are given a non-exclusive lock: not only may other processes have access to the file, but the same file can be the subject of multiple OPEN() statements.

Next: LOGICAL FILE NAMES | Prev: File I/O | Contents: File I/O

1.13 ARexxGuide | Functions reference | File I/O (2 of 4) | FILE NAMES

Naming logical files
~~~~~~~~~~~~~~~~~~~~

When a file or other device is opened using the OPEN() function, it is given a logical name. In the original manual to ARexx, Bill Hawes uses a string for the logical name:

```
say open('outfile', 'ram:temp', 'W')
```

Using a literal string makes it apparent that no assignment takes place in the function. 'outfile' is simply a name used to refer to the file. It isn't assigned an address or anything else.

The problem with this usage is that the name becomes case sensitive. The following will generate an error:

```
call writeln('Outfile', String)
```

'Outfile' and 'outfile' are not the same name because of the difference in letter-case. Such a subtle difference might give rise to what Cowlishaw
~~~~~~~~~~~~~~~~~~~~

calls a "high astonishment factor." He notes, "If a feature, accidentally misused, gives apparently unpredictable results, then it has a high astonishment factor and is therefore undesirable."

That's a good test for each programmer of the best method to use when naming files. If a using a literal string often gives rise to errors, then it is probably better to avoid the usage.

Fortunately, REXX is a language designed to be adaptable to different styles, but most of all it is a language designed to use something as close as possible to a natural English-like style.

Any valid symbol can be used as the logical name. Entering the names without quotation marks -- as simple symbols -- means that the name will be treated as upper-case by ARexx no matter how it is written. The disadvantage of this construction is that the name could be used later in a variable assignment , which would change its value and make it no longer the same name for the purposes of the file I/O functions -- another astonishing situation.

There is an interesting third alternative to using a literal (quoted) string or a variable symbol; an alternative which, like using a literal string, prevents the accidental assignment of a new value to <name>, but which also -- like the use a simple symbol -- preserves the general case insensitivity of REXX statements. The third alternative? Use a constant symbol for the name.

Unlike the symbols used for variables, constants cannot be assigned a value. There's no danger of accidentally using the symbol for something else. Constants are usually numbers (567.43 is a constant symbol, for instance), but they don't have to be. Any token beginning with a digit or a period (such as {.InputFile} or {2File}) is considered a constant. Such a symbol can be used as <name> in the OPEN() function. The name will be case insensitive since ARexx will translate it each time to uppercase.

An assigned variable may also be used as the file <name>. In that case, the logical name of the file is the value of the variable and not the name of the variable. There are times (opening multiple files in a loop, for example) when it is far more elegant to use a variable.

This will write a line to the file 't:vartest':

```
/**/
LFname = 'TFile'
   /* the variable's name can be written in any mixture of U&lc */
if open(LFName, 't:vartest', 'W') then
      /* 'TFile' is now the logical name of the file        */
   call writeln('TFile', 'See, it works with a variable.')
call close LFName
```

Next: NON-FILE DEVICES | Prev: Overview | Contents: File I/O

1.14 ARexxGuide | Functions reference | File I/O (3 of 4) | OTHER DEVICES

```
   Using I/O functions with other devices
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The Amiga operating system makes the file I/O functions even more useful because it extends the concept of 'file' to cover a range of devices including text windows and printers. Because the OS is able to treat a printer as a file-like device, ARexx can send output to a printer using a simple variation of the file I/O functions: The device 'PRT:' may be specified as the file name in the OPEN() function:

```
   /**/
   if open(Printer, 'prt:', 'W') then do
      call writeln(Printer, 'Hello world')
   end
```

(The READLN() input function cannot be used when communicating with the PRT: printer device.)

Using the operating system's console device, a window can be opened and treated in much the same way as a disk file:

```
   /**/
   if open(OutWin, "con:8/8/272/88/Output Window", W) then do
      call writeln(OutWin, 'Hello there, you big bad world.')
      call delay 500
      call close OutWin
   end
```

Even the input functions READLN() and READCH() can be used with the console device and will act much like the instruction PARSE PULL does on the standard input window.

Next: STANDARD I/O FILES | Prev: Logical file name | Contents: File I/O

1.15 ARexxGuide | Functions reference | File I/O (4 of 4) | STANDARD I/O

```
   Standard input/output files: STDOUT, STDIN, STDERR
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The instructions SAY and PARSE PULL are closely related to the functions WRITELN() and READLN() . SAY and PULL output and retrieve items from a defined logical file, except that the file used by the two instruction need not be opened because it will always be available in some form to a script.

The function SHOW('F') will return the names of all currently open logical files. The logical name of any file added with OPEN() will appear on the list. In virtually all cases, the returned list will also contain the names of at least two files that were not explicitly opened within the script: STDIN and STDOUT are logical files that are available by default to all scripts. The names refer to the standard input and output devices.

SAY outputs a specified string to STDOUT, making it a simpler variation of the clause { call writeln(STDOUT, <string>) }. In the same way, PULL retrieves its input from the STDIN device much like { Input = readln(STDIN) }. The instruction PARSE EXTERNAL also retrieves output

from a logical file, one named STDERR , that is normally available only when the trace console is open.

The STDIN and STDOUT files can be redirected to other devices using a standard AmigaDOS facility: When a command is followed by the character { < }, STDIN -- the standard input device -- is redirected to the device specified after that character. Similarly, the { > } redirects standard output or STDOUT to a specified device.

The interactive example uses the following simple script to demonstrate the effect of redirection.

```
/**/
options prompt "0a"x||"Enter any text then press <Enter>: "
pull T$
say T$

 Run interactive example    *
```

Redirection is often used on the Amiga to suppress output by setting up a dummy device called 'nil:' as the destination and source for a command. When the output of an ARexx program is redirected to nil: with the {>NIL:} option, the instruction SAY will have no effect. Its output will disappear. Similarly, the instruction PULL will return with an empty string when input is redirected to nil: with {<NIL:}.

Next: File I/O | Prev: Non-file devices | Contents: File I/O