ARx_Tknq.ag

COLLABORATORS			
	TITLE : ARx_Tknq.ag		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 3, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 ARx_Tknq.ag **1**

1.1 ARexxGuide | TECHNIQUES . . . 1
1.2 ARexxGuide | Techniques | ABOUT THIS SECTION . . . 2
1.3 ARexxGuide | Techniques (1 of 20) | COUNTCHAR() . . . 3
1.4 ARexxGuide | Techniques (2 of 20) | COUNTWORDS() . . . 3
1.5 ARexxGuide | Techniques (3 of 20) | FORMAT OUTPUT . . . 4
1.6 ARexxGuide | Techniques (4 of 20) | FORMAT() . . . 6
1.7 ARexxGuide | Techniques (5 of 20) | ADDCOMMA() . . . 8
1.8 ARexxGuide | Techniques | ADDCOMMA() | Note: Alternative . . . 9
1.9 ARexxGuide | Techniques (6 of 20) | WORDWRAP() . . . 10
1.10 ARexxGuide | Techniques (7 of 20) | PARSEFILENAME() . . . 11
1.11 ARexxGuide | Techniques (8 of 20) | CONSOLE WINDOWS . . . 13
1.12 ARexxGuide | Techniques (9 of 20) | PRINTER OUTPUT . . . 15
1.13 ARexxGuide | Techniques (10 of 20) | READ/WRITE FILES . . . 15
1.14 ARexxGuide | Techniques (11 of 20) | COMMAND PIPE . . . 16
1.15 ARexxGuide | Techniques (12 of 20) | USING MESSAGE PORTS . . . 17
1.16 ARexxGuide | Techniques (13 of 20) | GLOBAL VARIABLES . . . 18
1.17 ARexxGuide | Techniques (14 of 20) | ENVIRONMENTAL VARIABLES . . . 20
1.18 ARexxGuide | Techniques (15 of 20) | IN-LINE DATA . . . 22
1.19 ARexxGuide | Techniques (16 of 20) | DATA SCRATCHPAD . . . 22
1.20 ARexxGuide | Techniques (17 of 20) | SEEKTORECORD() . . . 24
1.21 ARexxGuide | Techniques (18 of 20) | INTERPRETED VARIABLES . . . 24
1.22 ARexxGuide | Techniques (19 of 20) | CHECK UNIQUE DATATYPES . . . 26
1.23 ARexxGuide | Techniques (20 of 20) | LIBVER() . . . 27

Chapter 1

ARx_Tknq.ag

1.1 ARexxGuide | TECHNIQUES

```
                  AN AMIGAGUIDE® TO ARexx                  Second edition (v2.0a)
  by Robin Evans


               About this section
                 Techniques:
    Strings

               CountChar(): count characters with COMPRESS()

               CountWords(): count words in a file

               Format data into table form

               Format(): round and format a number

               AddComma(): add commas to an integer

                  Alternative: add commas in a loop

               WordWrap(): wordwrap text to a defined length

               ParseFileName(): split name of file from path
                    Input/Output

               Open console windows for I/O

               Output text to a printer

               Read data from one file, write to another

               Retrieve result of AmigaDOS command

               Getting and sending message packets
                    Data storage and retrieval

               Store global variables on the clip list
```

Get and set environmental variables

Retrieve data from source code

Create a data scratchpad with PUSH, QUEUE, and PULL

SeekToRecord(): pull single record from data file

Use VALUE() to create interpreted variable names

Check unique datatypes with VERIFY()

Determine version number of any library
Other hints: < Press RETRACE to return to this page >
Brief examples of instructions and functions are
presented throughout ARexxGuide. The following nodes
include more extended examples.
Translate a string to lower-case
Using the elapsed time counter
Error messages: redirecting to a file
PARSE: pull field values from one-line records
Set sequential bookmarks in TurboText
Varieties of looping
Setting a default prompt for PULL instruction
Respond to asynchronous user input
Make variable declarations required
Emulate the standard-REXX WordPos() function
Store contents of a disk file in memory
Retrieve name and size of default system font
Pause a script until a program has started

1.2 ARexxGuide | Techniques | ABOUT THIS SECTION

Techniques
~~~~~~~~~~
This section presents code examples that serve two purposes:

They show ARexx instructions and functions in context of complete
program examples.

Most of the code examples presented here are complete subroutines
that can be copied and pasted into scripts to solve frequently-
encountered problems.

Users of versions of AmigaGuide that support clipboard operations can copy
the examples directly from the guide. Others can copy the examples from
the file ARx_Tknq.ag -- the file you're looking at now.

Many of the routines presented here use the PROCEDURE instruction to
isolate the variable references in the subroutine from other sections of
~~~~~~~~~~

the program. That instruction is valid, however, only when the subroutine is invoked as a subroutine. It may not be used and should be deleted if the code is stored as an external function .

1.3 ARexxGuide | Techniques (1 of 20) | COUNTCHAR()

Count characters using COMPRESS()
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Used in conjunction with LENGTH() , the COMPRESS() function provides 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 being 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. Following the syntax of built-in functions like POS() , the 'needle' (item to be found) is the first argument in this function followed by the 'haystack' -- the string to be searched. Because the ARG() function is used, the function does not make any variable assignments, so PROCEDURE is unnecessary.

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

Next: COUNTWORDS() | Prev: Tutorial | Contents: Tutorial

1.4 ARexxGuide | Techniques (2 of 20) | COUNTWORDS()

Count words in a file
~~~~~~~~~~~~~~~~~~~~

```
This is a simple word-counting program. It reads each line in a file and
counts the words. Because the contents of a line are not important,
READLN() is  nested  within the  WORDS()  function. The definition of
'word' here is that used by ARexx: any collection of characters divided by
a space or a line-end character from other collections.

   CountWords:
      arg FileName
      if FileName = '' | FileName = '?' then do
         say 'WordCount <FileName>'
         say '  Specify the name of the file to be counted.'
         exit 0
      end
      WdTotal = 0
      if open(6IFile, FileName, 'R') then do
         say 'Counting words in' FileName'.'
         do until eof(6IFile)
            WdTotal = WdTotal + words(READLN(6IFile))
         end
         call close 6IFile
         say 'There are' WdTotal 'words in' FileName'.'
         return WdTotal
      end
      else do
         say 'WordCount failed. File not found'
         exit 20
      end

   /* ------------------------ end example ------------------------ */

   Also see  ARG instruction
             SAY instruction
             OPEN() function
             DO instruction

Next: FORMAT OUTPUT | Prev: COUNTCHAR() | Contents: Tutorial
```

1.5 ARexxGuide | Techniques (3 of 20) | FORMAT OUTPUT

```
                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 twice, allowing the long
definition of a SAY expression to be spread over two line.)

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

   /* Use  compound variables  to store the values to be used in table  */
List.Prod.1 = 'Widget wacker'; List.Price.1 = 99; List.Code.1 = 'WID01-W'
```

```
List.Prod.2 = 'Foo Barian'; List.Price.2 = 182.95; List.Code.2 = 'FOO08-D'
List.0 = 2

   /* Variables determine the width of the listings */
PrdWd = 15; PrcWd = '9'; CdWd = 8

   /* Output heading */
say center('Product', PrdWd) || center('Price', PrcWd) ||,
   center('Code', CdWd-1)

   /* Output divider lines */
say  copies('-', PrdWd - 1) copies('-', PrcWd - 1) copies('-', CdWd-1)

   /* Output each product listing from within a  loop  */
do i = 1 to List.0
   say left(List.Prod.i, PrdWd) || right(trunc(List.Price.i,2),PrcWd-1),
      left(List.Code.i, CdWd)

end

/* OUTPUTS:    >>>

    Product      Price   Code
-------------- -------- -------
Widget wacker     99.00 WID01-W
Foo Barian       182.95 FOO08-D

*/
```

In constructing each listing with the DO loop, three functions are used: TRUNC() adds two decimal places to [List.Price] while RIGHT() pads the number with spaces on the left side so that numbers up to four digits will be decimal-aligned. Other versions of REXX include a function, called format(), that will perform these operations in one step. A version of that function, FORMAT(), is described in the following node and could be used in place of right() and trunc() here.

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

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

Also see

FORMAT(): User function to format numbers

ADDCOMMA(): User function to add commas

Next: FORMAT() | Prev: COUNTWORDS() | Contents: Tutorial

1.6 ARexxGuide | Techniques (4 of 20) | FORMAT()

```
FORMAT(): a user function to format numbers
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 TRL2  defines a function missing from ARexx that will round and format a
number to a given specification. Although the standard function can also be
used to control the presentation of numbers in exponential notation, its
simpler syntax is this:

      format(<number>, [<before>], [<after>])

If number alone is supplied, the result is the same as that returned by
the expression <number> + 0: leading 0's are removed from the number and
it is formatted according to the current setting of  NUMERIC DIGITS .

If <before> is supplied, it must be a number equal to or greater than the
length in the integer part of <number>. The result will be returned
right-justified to <before> spaces.

If <after> is supplied, it must be a number. The fractional part of
<number> is rounded (not just truncated) to <after> digits.

The following  user function  provides these features for an ARexx script.
If there is a problem with the received argument, the function attempts to
duplicate the kind of error reporting that would be provided by a built-in
function, but it cannot generate a true syntax error, so the error will not
be trapped by a  SIGNAL ON SYNTAX  routine.

   /* FORMAT():       format(<number>, [<before>], [<after>])  */

   Format: /* procedure */ /* Use  procedure  for  internal function   */
      arg number, before, after
         /* Record the line number from the caller to be used in case  **
         ** of a syntax error. The SIGL variable is available only if  **
         ** this is used as an internal_function.                      */
      CallLine = SIGL
         /* SIGL won't be set if called as external function           */
      if ~datatype(CallLine, 'N') then CallLine = '??'

         /* Make sure we have a number as first (required) argument    */
      if ~datatype(number, 'N') then do
         if number = '' then
            fc = 17     /* Wrong number of arguments                  */
         else
            fc = 47     /* Arithmetic conversion error                */
         signal FormatSyntaxError
      end

         /*  Arithmetic operation  reformats the number to NUMERIC     **
         **  DIGITS  setting.                                          */
      num = number + 0

         /* Return the reformatted number if other options not spec'd. */
      if before = '' & after = '' then
         return num
      else do
```

```
      /* split the number into fraction and integer. This section**
      ** mixes text operations with arithmetic operations.       */
    parse var num integer '.' fraction
      /* Set defaults for non-spec'd arguments                   */
    if before = '' then before = length(integer)
    if after = '' then after = length(fraction)
      /* Check for syntax errors.                                */
    if ~datatype(before, N) | ~datatype(after, N) then
      do fc = 18
      signal FormatSyntaxError
   end
      /* [before] argument must be at least as long as integer   */
    if before < length(integer) then do
      fc = 18
      signal FormatSyntaxError
    end
      /* add an appropriate value of .5 to number to round it    */
    if after ~= length(fraction) then do
      fraction = trunc(('.'fraction'0') +, /* cont'd on next line */
                               ('.'copies('0', after)'5'), after)
        /* Numbers created as text strings are still numbers    */
      integer = integer + (fraction % 1)
      fraction = substr(fraction, 3)
    end
    if fraction >= 0 then
      return right(integer, before)'.'fraction
    else
      return right(integer, before)
end

FormatSyntaxError:
      /* Acts like a syntax error in a built-in function would   **
      ** except that this error won't be trapped by SIGNAL ON    **
      **  SYNTAX . Output to STDERR if that file is open so msg  **
      ** will go where other error messages go.                  */
   if show('F', STDERR) then
      call writeln(STDERR, '+++ Error' fc 'in line' CallLine':',
           /* continued from line above */          errortext(fc))
   else
      say '+++ Error' fc 'in line' CallLine':' errortext(fc)
      /* Return an non-numeric value on error if this was called **
      ** as an external function. Otherwise exit script with     **
      ** error code. If called as external program, then caller  **
      ** will have to check for error return.                    */
   parse source Func .
   if Func = 'FUNCTION' then
      exit "Err"
    else
      exit 10

/*            -------- end function definition --------             */

Also see  DATATYPE() function
          SIGL special variable
          SIGNAL instruction
          IF instruction
          Arithmetic operators
```

```
             SUBSTR() function
             Standard I/O functions

Next: ADDCOMMA() | Prev: FORMAT OUTPUT | Contents: Tutorial
```

1.7 ARexxGuide | Techniques (5 of 20) | ADDCOMMA()

```
                 Add commas to a number
~~~~~~~~~~~~~~~~~~~~~~
 Numbers  in ARexx can contain only digits, an optional decimal point
‘.’, and/or an ‘e’ to indicate  exponential notation . Even though it will
no longer be considered a valid number for  arithmetic operations , a
large number will be more readable in charts and other output if it is
divided with commas. If the number needs to be used for an arithmetic
operation, the commas can be removed with the  COMPRESS()  function.

The function below will add commas in the appropriate places to a number.
It respects any fractional amount that was included with the number and
also leaves unchanged any leading spaces that were used with the number.

If the number includes leading zeros, it removes them, but adds leading
spaces in their place.

Once the number is sent to the function, it is always treated as a text
 string  and never as a number. It will not, therefore, be reformatted to
the  DIGITS()  setting. This makes it possible to format numbers that are
much larger than the current setting of  NUMERIC DIGITS .

   More information:  Numbers as character strings

The function returns the string 'ERROR' if it finds something wrong with
the received argument.

ADDCOMMA:           /*  NumWithComma = AddComma(<number>)   */
   arg integer '.' fraction
   if integer'.'fraction <= 0 then return integer'.'fraction
      /* How many leading spaces or 0's are included?       */
   LSpace = verify(integer,'123456789','m') -1
      /* Get rid of all spaces and leading 0's              */
   integer = strip(strip(integer,l,' 0'))
      /* Will format a max of 17 digits. If you need more,  **
      ** add '+3 p7' etc to parse and to 'integer = ' below */
   if length(integer) < 18 & datatype(integer, 'N') then do
         /* Where should commas start?                      */
      FPos = length(integer)//3
      parse var integer p1 +FPos p2 +3 p3 +3 p4 +3 p5 +3 p6
         /* p1 = integer when it divides equally at 3       */
      if FPos = 0 then p1 = ''
         /* Add commas, then strip off extras               */
      integer = strip(p1','p2','p3','p4','p5','p6,, ',')
   end
      /* Add fraction & leading spaces back into the string */
   if fraction > '' then
      return copies(' ', Lspace)integer'.'fraction
   else
```

```
      return copies(' ', Lspace)integer

 /* ------------------- end function   -------------------- */

            More information
                 Testing alternate coding methods
                   Also see  VERIFY() function
              STRIP() function
              LENGTH() function
              PARSE position patterns
              COPIES() function
```

Next: WORDWRAP() | Prev: FORMAT() | Contents: Tutorial

1.8 ARexxGuide | Techniques | ADDCOMMA() | Note: Alternative

```
                Testing alternate coding methods
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The
                 AddComma()
                 routine presented here uses the  PARSE  instruction to
break an integer into parts so commas can be inserted. The disadvantage of
the method is that it is limited to formatting numbers of a set maximum
length. The maximum can be easily expanded, but the routine will never be
able to handle any number that is thrown at it, and will therefore break
under unusual circumstances.

An alternative is to use the  REVERSE()  function and a  loop  to add
commas so that a number of any length can be sent to the routine:

   /**/
   arg integer
   rnum = reverse(integer)
   do cpos = 3 by 4 while cpos < length(rnum)
      rnum = insert(',', rnum, cpos)
   end
   return reverse(rnum)

This method is also more language-general: versions of the routine coded
in different programming languages will usually look similar. The parse
method, on the other hand, uses uncommon REXX conventions.

Which method should be used? It is partly an aesthetic choice. Some will
find one method more sensible and more attractive than the other, but
there are also issues of efficiency. A good way to test the efficiency of
a routine is to run it through a timed loop using the  elapsed-time
counter:

   /**/
   arg integer
   call time(r)
   do 200
      rnum = reverse(integer)
      do cpos = 3 by 4 while cpos < length(rnum)
```

```
         rnum = insert(',', rnum, cpos)
      end
      dinteger = reverse(rnum)
   end
   say '   Do:' time(r)
   do 200
      if length(integer) > 21 then say 'ERROR'
      FPos = length(integer)//3
      parse var integer p1 +FPos p2 +3 p3 +3 p4 +3 p5 +3 p6 +3 p7 +3 p8
      if FPos = 0 then p1 = ''
      pinteger = strip(p1','p2','p3','p4','p5','p6','p7','p8,,',')
   end
   say 'Parse:' time(e)
   say dinteger
   say pinteger
```

Adding another 'px' has an insignificant effect on the PARSE version which runs at about the same speed no matter what size the number is. The DO version starts out faster but will slow down significantly when the numbers get very large. (Again, though, that will depend on the machine.)

The timing results will be different on different systems, but running a test like this is often helpful in deciding which of two alternatives to use in a given situation. If most numbers are small, then the DO version will be faster in nearly all cases, but as numbers get larger, the PARSE version becomes ever more attractive.

If the code is run within a script that can assure that a number will not be larger than what can be handled by the PARSE instruction, then the length check can be deleted, making the code for that method even more efficient.

Thanks to the readers of Usenet's comp.lang.rexx for a discussion of this routine and for suggesting the REVERSE/DO alternative; and to Richard Stockton of Gramma's Software and Gramma's BBS for inspiring the original version of the routine. His approach to solving the problem is different and can be seen by calling the BBS and checking the wonderful collection of ARexx questions and solutions in the ARexx archive category.

Next: ADDCOMMA() | Prev: ADDCOMMA() | Contents: Tutorial

1.9 ARexxGuide | Techniques (6 of 20) | WORDWRAP()

Word-wrap text to a defined line length
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
When text is output to the screen or to the printer, it is often useful to have it wrap at a defined line-length. That can be done easily with this subroutine . It takes as arguments a string and a line length and creates a series of compound variables that contain the original text divided into lines that are no longer than the specified length.

```
 WordWrap: procedure expose Line.
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
   /* Arguments:                                                        **
   ** Text       := The string that is to be split into parts          **
   ** Length     := Maximum length of lines desired.                   */
   parse arg Text, Length

   Line. = ''                              /* All compounds are now null  */
   EndPos = length(Text); DivPos = 1       /* Preliminary values for loop */

   do i = 1 while EndPos <= length(Text)
      EndPos = lastpos(' ',Text' ', DivPos + Length+1)
         /* Handle a word that's bigger than the line length by         **
         ** splitting it arbitrarily at the line length                 */
      if EndPos < DivPos then do
         EndPos = DivPos + Length - 1
            /* Add a hyphen to the original string with  INSERT() .     **
            ** Since this subroutine is defined as a  procedure , this  **
            ** change to [Text] will not affect any variable with the   **
            ** same name in the calling environment.                     */
         Text = insert('- ', Text, EndPos-2)
      end
      Line.i = substr(Text, DivPos, EndPos-DivPos)
         /* Add one to DivPos because we want to get rid of the space   **
         ** at the start of each line.                                   */
      DivPos = EndPos + 1
   end
   Line.0 = i - 1
   return i - 1

   /* ------------------------ end example ------------------------ */
```

Because the subroutine creates a set of compound variables that must be accessible to the calling environment, this routine must be included as an internal function within the calling script. An external function would need to send the split lines back to the caller by a different method.

The routine will handle a word that is longer than the defined line length, but does it inelegantly. It simply splits the word at an arbitrary point. Although it is possible in English to split most words at syllable breaks by noting the positions of consonants and vowels, no attempt is made here to do that.

```
   Also see  LASTPOS() function
             SUBSTR() function
             INSERT() function
             LENGTH() function
             DO instruction
             PROCEDURE instruction
             EXPOSE instruction
```

Next: PARSEFILENAME() | Prev: ADDCOMMA() | Contents: Tutorial

1.10 ARexxGuide | Techniques (7 of 20) | PARSEFILENAME()

Split path or filename from file specification
~~~~~~
It is often necessary to separate a file name 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 routine, which can be called either as in internal or external function , 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.

Since the RETURN instruction can send back only a single value, this function can be used to retrieve either the filename only (if the second argument if 'FILE' or is omitted), or to retrieve the path specification without the filename (if the second argument is anything other than 'F' or 'FILE'). ABBREV() is used to check the second argument. Since a length was not specified, a null value (from an omitted argument) will return TRUE.

```
   /* Split filename from path                                          */
   ParseFileName: /*procedure*/  /* add  procedure  for internal func.  */
      /* Arguments:                                                     **
      ** FilePath   := Any valid AmigaDOS file specification           **
      ** Part       := [Optional] 'F', 'FILE', or omit to get filename  **
      **                          Anything else to retrieve the path    */
      parse arg FilePath, Part

      DivPos = max(lastpos(':', FilePath),lastpos('/', FilePath)) +1
      if abbrev('FILE', upper(Part))
         then return substr(FilePath, DivPos)
      else
         return strip(left(FilePath, DivPos-1),'T', '/')

   /* ------------------------ end example ------------------------ */
```

Since the function is meant to return either a filename or a path, but not both, the original string is divided using either SUBSTR() or LEFT() . If both parts were needed, however, it would be more efficient to use the PARSE instruction:

```
    parse var FilePath PathSpec =DivPos FileName
```

The '=' sign precededing [DivPos] indicates that the value of the variable is to be used as a positional marker .

The PROCEDURE keyword is used here to protect the variables declared in this subroutine from any similarly-named variables in the calling environment. In a short subroutine like this one, however, it's sometimes useful to avoid any variable assignments . The following variation of the same function uses the ARG() function and nested expressions, but returns the same information:

```
   /* Split filename from path. No assignments in subroutine            */
   ParseFileName:
```
~~~~~~

```
      /* Arguments:                                                    **
      ** arg(1)     := Any valid AmigaDOS file specification          **
      ** arg(2)     := [Optional] 'F', 'FILE', or omit to get filename **
      **                          Anything else to retrieve the path   */

      if abbrev('FILE', upper(arg(2)))
         then return substr(arg(1),,    /* Comma =  continuation token  */
                     max(lastpos(':', arg(1)),lastpos('/', arg(1))) +1)
      else
         return strip(left(arg(1),max(lastpos(':',arg(1)),,
                                      lastpos('/',arg(1)))), 'T', '/')

   /* ------------------------ end example ------------------------ */
```

Next: CONSOLE WINDOWS | Prev: WORDWRAP() | Contents: Tutorial

1.11 ARexxGuide | Techniques (8 of 20) | CONSOLE WINDOWS

```
                Custom console windows
~~~~~~~~~~~~~~~~~~~~~~
```

Several function libraries are available to add graphic requesters and menus to an ARexx script. Using those libraries can make a script more elegant since user interaction is done with the sophisticated GUI tools that Amiga users expect.

The power of console windows should not be overlooked, however. Like its parent, REXX, ARexx was designed mainly for the simple character-based input and output provided console windows. Unlike the GUI tools that require certain non-system libraries, console I/O is always available without additional programs or libraries.

Variations of the console I/O routines explained here are used throughout ARexxGuide to provide interactive examples. (See Tutorial Contents for a list of all the examples.)

```
A simple informational window:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

This subroutine can be used as an internal function in a script that needs to present simple informational messages to users. The only argument to the function, [InfoMsg], is the message that is to be presented. Multiple lines can be included in the string by indicating with the character '\' the places where the string should be broken into a new line. (The

```
                WordWrap()
```

user function could be substituted to automatically word-wrap the text.)

The window is opened as a RAW: console (see any Amiga OS reference for more information on that) rather than a CON: device because it is only in a RAW: console that READCH() is able to retrieve a single keystroke without waiting for the <Enter> key.

```
InfoCon: procedure
   /* Open a raw: window to display information                      */
```

```
   parse arg DisplayMsg
      /* Determine depth of window by multipying 11 (for interlace font **
      ** size) by the number of rows.                                     */
   depth = 27 + (11 * (countchar('\', DisplayMsg) + 3))
   if open(6Info, 'raw:10/0/346/'Depth, w) then do
      call writeln(6Info, translate(DisplayMsg, '0a'x, '\'))
      call writech(6Info, '0a'x'              <Press any key>  ')
      call readch(6Info)
      call close 6Info
      return 1
   end
   else
      return 0

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

   /* ------------- end user function ---------------- */
```

The
CountChar()
function used above is explained in another note.

Full input/output in a new window

The informational window presented in the first example uses the simplest of input and output, waiting for a single keystroke of any kind before closing the window. More complex interaction is possible in ARexx, however.

When a script is launched from a shell, it can use the same shell for its interaction. That's the approach taken with the Uncrunch.rexx utility explained in another tutorial. But since some Amiga users don't use a shell even for things like ARexx scripts, the writer of a script cannot be sure that a shell will be available.

The function below creates a new console window from within the script and redirects the standard input/output streams to that window.

```
   FullIOwindow:
      call close STDOUT
      if open(STDOUT, 'con:10/98/346/45/Window title',W) then do
         call close STDIN
         call open(STDIN, '*', R)
      end
         /*********            Do stuff here            ************/
      call close STDOUT
      call close STDIN
      call pragma('*')
```

To see this routine at work, run any of the interactive examples listed in the Tutorial Contents . All of them use a variation of this routine to open the windows in which the example is presented.

Next: PRINTER OUTPUT | Prev: PARSEFILENAME() | Contents: Tutorial

1.12 ARexxGuide | Techniques (9 of 20) | PRINTER OUTPUT

```
Output text to printer
~~~~~~~~~~~~~~~~~~~~~~
This example should output just two lines to the printer since the PrintVar
function outputs lines with the  writech()  function that does not
automatically add a line-feed to the output string.

When a line-feed is desired, as it is in the the third call to PrintVar(),
the  hex-string  '0a'x (the line-feed character) is  concatenated  to the
string.

   /* Printer test */
      call Printvar('This is a test')
      call printvar(' of adding more text')
      call printvar(' to a single line.'||'0a'x)
      call printvar('This should be the second line')
      call close 'Printer'
      exit

      /* Output the contents of a variable to the printer.             **
      ** Function can be called repeatedly without forcing a form-feed **
      ** on most printers.                                             */
      PrintVar:
         /* Argument:                                                  **
         ** ToPrint  := Text to be printed                            */
         parse arg ToPrint

            /* Use  SHOW()  to find out if channel to printer has been **
            ** opened. Open it if it's not yet available               */
         if ~show('F', 'Printer') then
               /* The  PRT:  device can be opened just like any file.  */
            if ~open('Printer', 'PRT:') then
               return 'ERROR'
            /* Return the number of characters written                 */
         return WRITECH('Printer', ToPrint)

      /* ------------------------ end example ----------------------- */

   Also see  I/O to other devices
             OPEN() function
             CALL instruction

Next: READ/WRITE FILES | Prev: CONSOLE WINDOWS | Contents: Tutorial
```

1.13 ARexxGuide | Techniques (10 of 20) | READ/WRITE FILES

```
Read and write files
~~~~~~~~~~~~~~~~~~~~
This example could be used as a script, or as an  external function  or
 internal function . It will open one file -- the first  argument  --
and output its contents to a second file -- the second argument.

The  READLN()  function is used to get input. The  WRITELN()  function puts
```

```
the line into a new file unless it begins with the character '@'. The
 ABBREV()  function is used to check for that character in column 1, but
because the function respects spaces, a line that starts with spaces is
not dropped.

   /* Output selected lines of one file to a new file                  */
         /* Arguments:                                                 **
         ** InputFile   := Name of file to read                        **
         ** OutputFile  := Name of file to create                      */
      parse arg InputFile OutPutfile .

         /* Do very simple error checking                              */
      if InputFile = '' | OutputFile = '' then return 'ERROR'
      if ~open(6Input, InputFile, 'R') then return 'ERROR'
      if ~open(6Output, OutputFile, 'W') then return 'ERROR'

         /* Read all lines in input file                               */
      do until eof(6Input)
         ThisLine = readln(6Input)
         if ~abbrev(ThisLine, '@',1) then
            call writeln(6Output, ThisLine)
      end
      call close 6Input; call close 6Output

      /* ------------------------ end example ----------------------- */

   Also see  EOF() function
             DO instruction

Next: COMMAND PIPE | Prev: PRINTER OUTPUT | Contents: Tutorial
```

1.14 ARexxGuide | Techniques (11 of 20) | COMMAND PIPE

```
Getting the output from a command
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Although most application programs allow direct interaction between ARexx
scripts that the host environment, some  hosts  will not return
information to ARexx. AmigaDOS is a prime example.

 WShell  offers an elegant way to read the output of AmigaDOS commands --
with the  ExecIO  utility. On the standard Amiga shell, however, the best
way to read the output of a  command  is to redirect the output to a file
or named pipe  whose contents can then be read by the script:

This example uses the 'TO' keyword to redirect the output of the the
AmigaDOS 'LIST' command to a file in the T: directory. (Even if a command
does not support a 'TO' option, its output can be redirected using the '>'
redirection character which is explained more fully in an OS reference.)

   /**/
   address command
   'list quick files nohead to t:ls'
   if OPEN(1List, 't:ls', 'r') then
   do i = 1 while ~eof(1List)
      File.i = readln(1List)
```

```
   end
   call CLOSE 1List
   call delete 't:ls'
```

Current versions of AmigaDOS include a facility that will send the output of a command to something that looks like a file, but uses RAM for only as long as needed to make use of the information. Called "named pipes," these virtual files are an ideal target for command output that is to be used within an ARexx script.

Instead of creating a file that will remain until deleted, a named pipe can be used to hold the information temporarily:

```
   /**/
   address command
   'run >nil: list quick files nohead to pipe:ls'
   if OPEN(1List, 'pipe:ls', 'r') then
   do i = 1 while ~eof(1List)
      File.i = readln(1List)
   end
   call CLOSE 1List
```

The PIPE: device must be mounted before it can be used. That is done automatically in the default startup procedures for 2.x and 3.x versions of the OS. Consult an AmigaDOS reference for more information about pipes.

```
   Also see  OPEN() function
             READLN() function
             EOF() function
             DELETE function
             DO instruction
```

Next: USING MESSAGE PORTS | Prev: READ/WRITE FILES | Contents: Tutorial

1.15 ARexxGuide | Techniques (12 of 20) | USING MESSAGE PORTS

```
Using message ports from an ARexx script
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This code fragment demonstrates the use of the repertoire of port
functions available in ARexx.

      More information:  Message port functions

/************ 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

  /* ------------------------- end example -------------------------- */
```

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.

```
   Also see  IF instruction
             NULL() function
```

Next: GLOBAL VARIABLES | Prev: COMMAND PIPE | Contents: Tutorial

1.16 ARexxGuide | Techniques (13 of 20) | GLOBAL VARIABLES

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  POS()  function is used to verify that there is        **
         ** at least one more clip matching format used by this app.    */
      do while pos('Rd_', Clips) > 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

   /* ------------------------ end example ------------------------ */
```

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')

   Also see  OPEN() function
             WRITELN() function
             SHOW() function
             DO instruction
             POS() function
             PARSE VAR instruction
```

Next: ENVIRONMENTAL VARIABLES | Prev: MESSAGE PORTS | Contents: Tutorial

1.17 ARexxGuide | Techniques (14 of 20) | ENVIRONMENTAL VARIABLES

```
Getting and setting environment variables
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Environmental variables have been a part of the Amiga OS since version
1.3. The values can be accessed directly in OS scripts by preceding the
variable name with the '$' character:

 'echo $kickstart' or 'copy $srcfile to $destination'.

The same syntax can be used in ARexx only when the statement is sent as a
 command  to be executed by AmigaDOS. (See  ADDRESS COMMAND .) If the
value of an enviromental variable is to be used within an ARexx script,
the  interpreter  has no simple way to access it.

The function library  rexxarplib.library  includes two functions, GetEnv()
and SetEnv(), that give scripts access to environmental variables. Other
function packages may contain similar functions, but it is difficult to
depend on function libraries if a script is distributed to other users.

An alternative is a set of user functions that can be included in any
script that needs access to environmental variables, or stored in the
REXX: directory to be called as  external functions . These function check
for rexxarplib.library and use the functions from there if the library is
available. When the library function is called, its name is quoted,
preventing the  interpreter  from calling the internal function
recursively.

   /* GetEnv()  return the value of an environmental variable          */
   GetEnv: procedure
      /* Arguments:                                                    **
      **   arg(1) := The name of the variable to retrieve             **
      ** Returns     a string                                         */
      /* Use function from rexxarplib if it's available               */
   if show('L', 'rexxarplib.library') then
      return 'GetEnv'(arg(1))

      /*  OPEN()  will fail if variable is not defined. Null will be   **
      ** returned in that case                                        */
   if open(6Env, 'env:'arg(1), 'R') then do
      EnvVar = readln(6Env)
      call close 6Env
   end
   else EnvVar = ''
```

```
   return EnvVar


   /* SetEnv()  Set the value of an environmental variable              */
   SetEnv: procedure
      /* Arguments:                                                     **
      **   arg(1) := Name of variable to set                           **
      **   arg(2) := New value for variable                            **
      ** Returns     a  Boolean value                                  */

      /* Use function from rexxarplib if it's available                */
   if show('L', 'rexxarplib.library') then
      return 'SetEnv'(arg(1),arg(2))

         /**** Add makedir() option here ****/
   if arg(2, 'E') then do  /* Open file only if second arg is supplied  */
      if open(6Env, 'env:'arg(1), 'W') then do
            /* [Success] will be TRUE or FALSE since it is assigned to  **
            ** a  logical expression .                                  */
         Success = (writech(6Env, arg(2)) > 0)
         call close 6Env
      end
      else
         Success = 0
      return Success
   end
   else                    /* Var is deleted if there's no value to set */
      return delete('env:'arg(1))

   /* ------------------------ end example ------------------------ */
```

SetEnv() uses delete() , a function from rexxsupport.library . That is one library that should be available on all Amigas using ARexx since it is part of the ARexx distribution. It does, however, need to be explicitly loaded with ADDLIB() before it is available.

Since the OPEN() function will not create a directory, SetEnv() will not be able store a variable that includes a path specification for a subdirectory in env: that does not yet exist. The OS 'SET' command has the same limitation, but it can be overcome by adding a bit of code to the user function:

```
   /* Setenv option will create a subdirectory if it doesn't exist */

   Path = ParseFileName(arg(1), 'P')
   if Path > '' then
      if ~exists('env:'Path) & arg(2, 'E') then
         call makedir('env:'Path)

   /* ------------------------ end example ------------------------ */
```

Also see OPEN() function
READLN() function
WRITECH() function
ARG() function
EXISTS() function
MAKEDIR() function

The REXX standard defines an extension to the VALUE() funtion that can be used to retrieve variables defined in an outside environment. The extension is used in many REXX implementations to get and set environment variables. It is not, unfortunately, available in ARexx.

Next: IN-LINE DATA | Prev: GLOBAL VARIABLES | Contents: Tutorial

1.18 ARexxGuide | Techniques (15 of 20) | IN-LINE DATA

Copy data from the program code
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
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 i

      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 transfers 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 ("signal SendLine") is known to be two lines above the first line of data, SendLine() returns the proper line number to the calling environment.

This technique is used in the script ARx_Reg.rexx which is distributed in the ARexxGuide archive.

```
   Also see  DO instruction
             PARSE VALUE instruction
```

Next: DATA SCRATCHPAD | Prev: ENVIRONMENTAL VARIABLES | Contents: Tutorial

1.19 ARexxGuide | Techniques (16 of 20) | DATA SCRATCHPAD
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Data scratchpad using PUSH and QUEUE
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

PUSH and QUEUE can be used for more than just stacking commands on the shell. In his ARexx manual, Bill Hawes mentions use of the instructions to create a 'private scratchpad' for a program. Strings stacked with the instructions can be retrieved later in the same script using the PARSE PULL instruction, but are also available to another script launched from the first one. (Note, however, that if the scripts terminate for some reason before data has been pulled from the scratchpad, the shell will treat whatever remains as commands, probably causing a messy series of error messages. Using SIGNAL traps to intercept error conditions and clean up the data stack is recommended in this instance.)

Although there are more efficient and elegant ways to do this, the following example suggests how PUSH and QUEUE can be used as a data scratchpad.

```
Datafile format
---------------
01-Aug-1994 1400
02-Aug-1994 1300
01-Aug-1994 1000
03-Aug-1994 1700
06-Aug-1994 1100
03-Aug-1994 1430
01-Aug-1994 0900
04-Aug-1994 1030
01-Aug-1994 0800
```

```
Program
-------------------------------------------
/* Demo of PUSH and QUEUE */
arg AptFN .
TDt = upper(translate(date(),'-',' '))
if open(1AptFile, AptFN, R) then do
   do until eof(1AptFile)
      Apt = readln(1AptFile)
      if word(upper(Apt), 1) >= TDt then
         if abbrev(upper(Apt), TDt) then
            PUSH Apt
         else
            QUEUE Apt
   end
end
do for  lines()
   parse pull Apt
   say Apt
end
```

The PUSH instruction is used to place a record with the current date at the top of the stack while QUEUE is used to put other dates at the end of the stack. (Sorting the file -- even with the AmigaDOS Sort command -- would make this step unnecessary.) Dates earlier than [TDt] are discarded. In this example, the data is simply printed to the shell. A more useful alternative might be to rewrite it to an updated file. More significantly, the PARSE PULL instruction could be left out of this script and included in another one called from here. The second script could then read the data from the stack and perform whatever actions are needed.

Also see IF instruction
OPEN() function
DO instruction
ABBREV() function
LINES() function
UPPER() function
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

1.20 ARexxGuide | Techniques (17 of 20) | SEEKTORECORD()

```
Retrieve a record from disk-based database
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
The PARSE instruction can be used to divide a one-line record into its component fields, as explained in the node Combining PARSE templates , but what about getting a specific record from a file that contains many records?

This routine uses the file I/O functions to accomplish the task:

```
   /* Retrive a record of defined length from a file                  */
   SeekToRecord:
         /* Arguments:                                                 **
         ** NamesDB  := Name of database file in disk                 **
         ** RecNum   := Sequential number of record to retrieve       **
         ** RecSize  := Total size of each record                     */
      arg NamesDB, RecNum, RecSize

      if open(DBFile, NamesDB, 'R') then do
         CurRecPos = seek(DBFile, RecNum * RecSize, 'B')
         Rec.RecNum = readch(DBFile, RecSize)
         call close DBFile
      end
```

Once the file is OPEN() , the SEEK() function is used to move to a particular spot within the file. The location is determined by multiplying the record size by the sequential record number. Finally, the entire record is retrieved by using the READCH() function to input just the number of characters used in a single record.

Next: INTERPRETED VARIABLES | Prev: DATA SCRATCHPAD | Contents: Tutorial

1.21 ARexxGuide | Techniques (18 of 20) | INTERPRETED VARIABLES

```
Interpreting strings as variables
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
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 check for a valid argument to VALUE().

Next: CHECK UNIQUE DATATYPES | Prev: SEEKTORECORD() | Contents: Tutorial

1.22 ARexxGuide | Techniques (19 of 20) | CHECK UNIQUE DATATYPES

Check unique datatypes
~~~~~~~~~~~~~~~~~~~~~~

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 something that specific. VERIFY() can, however, check for such a limited range of values:

```
/************ CheckProductNum.rexx ***********************************/
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 do
         /* The comma  continuation token  is used stretch to one    **
         ** line of code over two physical lines in the following.   */
      say 'Invalid data at position',
         VERIFY(Num, xrange(0,9)||xrange('A','G')'-')
      say 'Code must be in this format:'
      say '  nnn-AAA'
      say 'where <n> can be any number and <A> any letter between A and G'
   end

   /* ------------------------ end example ------------------------ */

   Also see  LENGTH() function
             ARG instruction
             IF instruction
             XRANGE() function
             SAY instruction
```
~~~~~~~~~~~~~~~~~~~~~~

Next: LIBVER() | Prev: INTERPRETED VARIABLES | Contents: Tutorial

1.23 ARexxGuide | Techniques (20 of 20) | LIBVER()

```
                Determine version number of any library
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Although the addlib() function accepts as an argument a mimimum version number, it acts only on the integer part of the version. It is sometimes necessary to limit a library to a version based both on the integer and the fractional part of a version number.

The version number of any library and most programs -- even something that is not yet loaded -- can be retrieved using a DOS command:

```
address command 'version' 'libs:'Libname '>env:LibVer'
LibVer = GetEnv('env:LibVer')
```

(
GetEnv()
is described as a user-function elsewhere and is also available in rexxarplib.library or in RexxDosSupport.library or rexxextend.library under different names.)

This method might be preferable to the method described below, but it is slower and perhaps more demanding of system setup because it depends on these factors:
-- The VERSION command must be available
-- the 'libs:' device must be present on a mounted volume

The function below returns the same numerical information: a version number that includes both an integer and a fraction. The name of any library available on the system can be passed to the routine. Unlike the 'VERSION' command, however, this user function is limited to libraries in memory.

To find the version of the Workbench being run, send 'version' as an argument. Any of the system libraries, including 'exec.library', can be used to get a number which indicates, by its integer, the OS version being used.

```
/* LibVer(): Retrieve the version number of a library          */
LibVer: procedure
   parse arg libname
   if right(libname,8) ~= '.library' then
      libname = libname'.library'
   if ~showlist('L', libname) then
      return -1
   else do
      call forbid    /* probibit multitasking during read      */
      libver = import(offset(showlist('L',libname,,'a'),20),4)
      call permit
   end
   return c2d(left(libver,2))'.'c2d(libver,2)

/* ------------------------ end example ------------------------ */
```

The first call to SHOWLIST() verifies that the requested library is in memory since this function will not read the version number of an unloaded library.

The function is used a second time with its address argument to determine the base address of the library. A known offset number, 20, is applied to that address with OFFSET() to calculate the actual address from which data will be copied by IMPORT() . During the process of reading the system list, multitasking is temporarily disabled with FORBID() , as it should be whenever information is retrieved in this way.

The version information is stored in four bytes. The first two bytes store the integer part of the version; the rightmost two bytes store the fractional part. LEFT() is used to split out the first two bytes from the four bytes copied. Because the information is stored as character data, the C2D() function translates it to more familiar form. The second argument to C2D() can be used to truncate input data from the right, so the RIGHT() function is not needed for the data that represents the fractional part of the version number.

The return value is built by concatenation of three expressions. The numbers returned by the two C2D() functions are concatenated with a period using one of the 'virtual' concatenation operators: Because the values are abutted against one another, ARexx will combine them without a space. The same string can be built using the explicit concatenation operator:

```
return c2d(left(libver,2)) || '.' || c2d(libver,2)
```

Because ARexx treats numbers as strings , the numbers returned by c2d() can be directly concatenated to the string '.' without the kind of translation required in most programming languages. The value returned will be a valid number that could be used directly in an arithmetic operation.

Also see PERMIT()

Next: Tutorial | Prev: CHECK UNIQUE DATATYPES | Contents: Tutorial