ARx_Tutor.ag

COLLABORATORS			
	TITLE : ARx_Tutor.ag		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 17, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ARx_Tutor.ag

1.1 ARexxGuide | TUTORIALS

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

Using ARexx with macros

              Extending keyboard macros
                ARexxGuide online help system

              Macros bring one-key ARexx reference to editors
                A complete program example:
    UnCrunch.rexx               A shell-based program to undo
   archives made with different archiving utilities.

Interactive examples:   *
    Break-key demonstration             More information
    Comparison demonstration            More information
    NUMERIC demonstration               More information
    Standard I/O demonstration          More information
    Test for valid symbols              More information
    TRACE demonstration                 More information
```


1.2 ARexxGuide | Tutorials (1 of 2) | Using ARexx with macros

```
                Using ARexx with macros
~~~~~~~~~~~~~~~~~~~~~~
Many users are introduced to ARexx through macros -- the scripts that can
be launched from most applications to automate tasks. Programs ranging
from public domain music players to the Video Toaster support sets of
commands that can be sent from an ARexx script back to the application.
```

This tutorial steps through the process of making macros for two sample programs -- the text editors TurboText and Edge. Even if you don't use those programs, the tutorial will be helpful in explaining the process.

Next: ONLINE HELP SYSTEM | Prev: Tutorials | Contents: Tutorials

1.3 ARexxGuide | Tutorials | Macros | EXAMPLE (1 of 2)

[Press < RETRACE > to return to previous position]

```
/* Macro.ttx */
MOVENEXTWORD
MARKBLK
MOVERIGHT
CONV2UPPER
MARKBLK
```

```
/* Edge default macro */

options results
signal on error
putenvvar _we_errorlevel 6

'next word'
'uppercase char'

error:
exit rc
```

1.4 ARexxGuide | Tutorials | Macros | EXAMPLE (2 of 2)

[Press < RETRACE > to return to previous position]

```
/* Macro.TTX */
[ 1]  options results
[ 2]  Rtcl = 'a an the to by from for',
[ 3]         'and or that'
[ 4]  'MoveNextWord'
[ 5]  'GetWord'
[ 6]  if find(Rtcl, result) = 0 then do
[ 7]     'MarkBlk'
```

```
/* Macro.edge */

options results
signal on error
'putenvvar _we_errorlevel 6'

Rtcl = 'a an the to by from for',
       'and or that'
'next word'
```

```
[ 8]     'MoveRight'                    'copy word resultbuff'
[ 9]     'Conv2Upper'                   CurWord = result
[10]     'MarkBlk'                      'previous word'
[11]  end                               if find(Rtcl, CurWord) = 0 then
[12]                                     'uppercase char'
[13]
[14]                                    error:
[15]                                    exit rc
```

1.5 ARexxGuide | Tutorials | Macros (1 of 7) | KEYSTROKE MACROS

Simple macro: recording keystrokes

The task we'll tackle is a simple one: Capitalize each word in a line of text. This would be useful for reports that use book-title capitalization on section headings.

Many programs use a two-tier approach to macros: Simple macros can be written without ARexx and are made up of a series of the commands understood by that program. They can often be created just by recording keystrokes.

We'll use that approach in the text editors TurboText and Edge to begin our macro. These simple macros will capitalize the first letter of the next word.

View example macros #1

The macros were made by recording keystrokes and saving the result ↩ . The two macros perform the same task with significantly different commands. That will be common. Macro ·commands· have become more consistent because of Commodore's style guidelines for the Amiga, but there are still a variety of ways to approach the same task.

Commands like 'next word' and 'MoveNextWord' are not part of the ARexx language. They are features of the program for which the macro is written and should be explained in the documentation for that program.

Different programs take different approaches to macros. In TurboText, for instance, the keystroke macro is saved in a form that can be run by TTX without using ARexx. In Edge, on the other hand, ARexx is used for even simple macros like this one. Edge, therefore, uses more ARexx features even in keystroke macros.

Both programs add comment markers to scripts. The characters '/*' and '*/' indicate the beginning and end of a comment and must be used at the start of each ARexx script. Without the comment at the beginning, ARexx won't recognize a file as a script.

Edge adds something else that we'll have to type into the TTX macro before we can use it as an ARexx script:

```
OPTIONS RESULTS
```

In almost all ARexx macros you write, that should be the first line. It tells ARexx that it should retrieve information from the ·host· program when a command is issued. The ·instruction· sets up two-way communication between the host program and the macro.

Edge also adds the quotation marks to most of the command words. (Line_5 , starting with 'putenvvar' should also be quoted.) This version of the macro uses a second of the special variables maintained by the interpreter, RC . That variable is set even if OPTIONS RESULTS in not used. An RC value of 0 usually indicates that the command was successfully executed. A value that is less than 10 is usually an informational warning, used for instance, to indicate that a search was successfully executed but did not find the specified word. A return value of 10 or greater indicates a more severe failure of the issued command.

Also see: SIGNAL ON ERROR

Next: ADDING AREXX | Prev: Macros tutorial | Contents: Macros tutorial

1.6 ARexxGuide | Tutorials | Macros (2 of 7) | ADDING AREXX

Adding ARexx control to the macro
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The keystroke macros in the first example are useful for making chapter titles but they might be more useful if they were a bit smarter. In English, some words in a title should not be capitalized. With some help from ARexx, the macro can be made to recognize those words.

View example macros #2
There are some cosmetic changes here and some significant ↩ additions. The
line numbers on the left are added for clarification only and should not be included as part of the macro. The TTX commands from the previous example were shifted to mixed case to make them more readable. Quotation marks were added because all ·commands· used in an ARexx macro should be quoted.

OPTIONS RESULTS has now been added to the TTX macro. Without the ·instruction·, the commands in line 5 of the TTX version and line 8 of the Edge version would be meaningless. With the instruction included, however, the 'GetWord' and 'Copy Word' commands send back to ARexx the word under the cursor. (There are differences in the way the macros would work if the cursor was not on a word, but we'll ignore those for now.)

The word retrieved from the document is assigned automatically to the variable RESULT . That is one of three special variables that are maintained by the ·interpreter·. The variable is handled differently in the two versions. In TurboText, the [result] variable can be used directly in the following ·function·. In Edge, on the other hand, the value of the [result] variable is assigned to a new variable which is then used in the function. That is necessary because Edge moves the cursor on the 'copy word' command. A second command 'previous word' must be issued to put the cursor back where it was, but every command issued to the ·host· will
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

cause the value of the variable [result] to be reassigned. We would have lost the result of the 'copy word' command if we did not store [result] before issuing another command.

The reason for adding elements from ARexx to a macro is to allow more processing than can be done using the program's macro commands alone. The advantage of ARexx is that, although the commands used by each program's macros will be different, the ·control structures· and logic of script will be consistent since that part of the macro is determined by ARexx.

In the next node, we look more closely at the elements of ARexx added to the example macros.

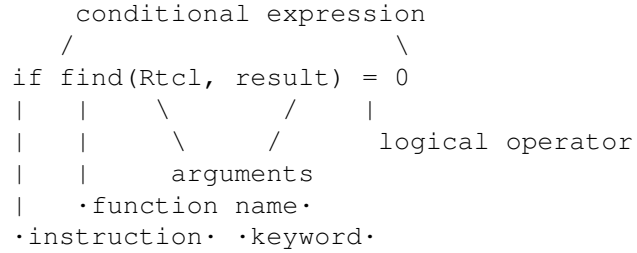
Next: CONDITIONALS | Prev: Keystroke macros | Contents: Macros tutorial

1.7 ARexxGuide | Tutorials | Macros (3 of 7) | CONDITIONALS

A closer look at the ARexx IF instruction
~~~~~~
This version of the macro still works on only one word at a time, but it leaves unchanged any of the words included in the ·variable· [Rtcl]. The ·assignment· for the variable is the same in both versions of the macro: The variable name is the first ·token· on the line; an '=' sign is the second token. Everything that follows -- an ·expression· -- becomes the value of the variable.

In both versions, the list of words is extended over two lines since we're using a short line-length in these examples. The comma at the end of line 2 in the TTX version and line 5 in the Edge version is a ·continuation token·. It tells ARexx to combine the current line with the next line.

Line 6 in the TurboText version and line 11 of the Edge version are conditional statements that will execute parts of the code only under certain conditions. The following chart identifies elements of the ·clause·:



An IF instruction always includes a conditional expression. The rules for instructions and expressions like this one are the rules of ARexx so they will remain the same no matter how different a host program's treatment of macros may be.

As it evaluates an ·expression·, ARexx goes through a process of substitution, usually in left-to-right order. The first substitution in this expression would be for variables. If the current word is 'and' then the clause would become:
~~~~~~

```
      if find('a an the to by from for and or that', 'and') = 0
```

The ARexx find() function locates a word in a longer phrase and returns a ·number· that indicates the position at which the match was found, or '0' if there was no match. The word being searched for here is the value of the variable [result] or [CurWord] -- the value that was retrieved from the document with the 'GetWord' or 'Copy Word' command.

If the word 'and' is assigned to [result], then the value returned by the function will be '8' since 'and' is the eighth word in the list assigned to [Rtcl]. Once the function has returned a value, that value effectively replaces the function call in the larger expression. The result for the word 'and' would be this:

```
      if 8 = 10
```

The conditional expression (8 = 10) is false, or 0 which means false in ARexx. (Notice that, unlike some languages, ARexx does not require that a conditional expression be surrounded with parentheses .) The final evaluation of the clause is, therefore:

```
      if 0
```

Because the conditional is false, the clause following THEN (a ·keyword· that must always accompany an IF) is not executed.

Only one clause is associated with THEN. That's fine for the Edge version since the character under the cursor can be capitalized with a single command. It would be a problem, though, for the TTX version because four commands are used which should be executed together, but only if the condition is true. A new instruction is used to group the TTX commands together: The keywords DO and END indicate that the clauses in between are to be executed as a part of one large block. The single clause tied to THEN in the TTX version is a DO/END instruction, but that instruction includes within it the four TurboText commands.

Next: LOOPS | Prev: Adding ARexx | Contents: Macros tutorial

1.8 ARexxGuide | Tutorials | Macros (4 of 7) | LOOPS

```
                Repeating macro with an ARexx loop
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The macro would be even more useful if it would automatically move from word to word, capitalizing the appropriate words in an entire line. One way to accomplish that is through a ·loop·, a set of statements that will be executed repeatedly until some condition is met. All looping in ARexx is controlled by the DO ·instruction· which has a wealth of options.

One of the simpler options is a count:

```
   DO 5
      <clause>
   END
```

```
This will repeat <clause> five times. It would work for our macro if we
knew how many words were in the line. That's easily accomplished:

   /* TTX */                              /* Edge */
   'GetLine 1'                            'copy line resultbuff'
   WordsInLine = words(result)            WordsInLine = words(result)

The ARexx  WORDS()  ·function· returns the number of words in an
·expression·. We can now use the variable [WordsInLine] to control a loop
that will move to each word, and capitalize them when appropriate. Here's
what the Edge macro would look like with the loop added:

   /* Macro.edge */

   options results
   signal on error
   'putenvvar _we_errorlevel 6'

   Rtcl = 'a an the to by from for and or that'

   'copy line resultbuff'
   WordsInLine = words(result)

   do WordsInLine
      'next word'
      'copy word resultbuff'
      CurWord = result
      'previous word'   /* Edge moves the cursor on a copy */
      if find(Rtcl, CurWord) = 0 then
       'uppercase char'
   end

error:
exit rc

When a macro becomes more complex than a simple list of keyboard commands,
it's good practice to run it first in  TRACE  mode. The process is
explained more fully in a
                 later node
                .

Next: GROWING A MACRO | Prev: Conditionals | Contents: Macros tutorial
```

1.9 ARexxGuide | Tutorials | Macros (5 of 7) | NO SURPRISES

```
Growing a macro
~~~~~~~~~~~~~~~
A macro that is used repeatedly should be made to act as transparently as
possible -- without surprises. The macros we've created here would be
sufficient for one-time use, but they don't yet satisfy the 'surprise'
test.

One problem is this: The macros work only if the cursor is located at the
start of the line. You might remember that when you've just created a
```

macro, but will probably forget about it once you've stored the macro. It's best to move the cursor to the start of the line within the macro. In TurboText, that is done with the 'MoveSOL' command and in Edge with the 'gotocolumn 1' command. The command should be issued before the first word is read in the macro.

Other improvements could also be made:

   Special handling is needed for the first word in a line since it should be capitalized even if it is the exclude list.

   The 'next word' and 'MoveNext' commands won't capture the first word in a line if it's flush with the left margin. Again, special handling should be added to treat this situation.

Macros can easily grow in complexity to meet the needs of users. A simple keystroke macro might be adequate as a start. Parts of different keystroke macros can then be pasted together to make a more complex macro. ARexx can be used to add intelligence to any macro.

The process used here is one of slow migration. A simple keystroke macro was extended first with a conditional statement and then with a loop to make it more effective. Each step could be tested as that part of the macro was developed. To make the macros grow into fully functional tools, more tests and steps should be added.

Next: MACROS & ADDRESS | Prev: LOOPS | Contents: Macros tutorial

1.10 ARexxGuide | Tutorials | Macros (6 of 7) | MACRO ADDRESS

ADDRESS and the macro
~~~~~~~~~~~~~~~~~~~~
A ·keyword· that is not used in this tutorial's macros is ADDRESS , an instruction used to control the target of interprocess communication in ARexx. When ·commands· in a script are to be sent to a new ·host· program, the ADDRESS instruction can establish the new target of commands.

Macros rarely need the instruction, however, because an ARexx script inherits a default address -- a target to which commands will be sent if no change is made. Although there may be some programs that don't properly pass off their address, the default address for macros launched from most programs will be the process that launched the macro.

For example, if a macro is launched from a TurboText window with an address of TURBOTEXT8, then the default address of the macro will be 'TURBOTEXT8'. The window that launched the macro will receive any commands issued in the macro. The ARexx address() function will return the name of the currently active host.

The ADDRESS instruction is needed, however, if a command is to be sent to a new environment. To issue an AmigaDOS command from an Edge script, for instance, the instruction 'ADDRESS "COMMAND"' must be issued to redirect the command to that environment.
~~~~~~~~~~~~~~~~~~~~

Next: DEBUGGING | Prev: Growing a macro | Contents: Macros Tutorial

1.11 ARexxGuide | Tutorials | Macros (7 of 7) | DEBUGGING

Debugging a macro
~~~~~~~~~~~~~~~~~
A number of things can go wrong with a new program in any language. Finding the problems and getting rid of them -- ·debugging· -- is a necessary part of writing a program. Since ARexx is an interpreted language, its source code can be executed immediately, making it easy to try out sections of code by typing them on the shell. If you're not sure how the word() function works, for instance, you could type this:

```
rx "say word('This is a test', 4)
```

The command will print the word 'test' to the shell. It's also easy to run small sections of code that are cut out of the main program and saved as a temporary file. (Just remember to add the opening comment .)

The most useful and powerful way to debug a program, however, is by using the tracing options. The TRACE instruction or TRACE() function can be entered as part of the source code to trace selected sections of code; or the TS command can be issued from a shell for an interactive trace of all executing scripts.

If a script is run from the shell, its tracing output will be issued to the shell, but when a script is run as a macro from another application, there may be no destination for trace output. The solution is provided by the TCO command which opens a trace console -- a window to which all tracing output and error messages are sent no matter how the script was launched. It should be opened when running a new macro since it will often give more information about error codes than the application itself will provide and because it establishes a target for any tracing output.

Not even the trace console will help, however, if the program fails with one of the problems like Error 5 , "Unmatched quote", that are detected by the ·interpreter· before execution of a script. If a macro fails before anything is output to the trace console, it's useful to run it directly from the shell:

```
rx macro.TTX
```

If there's an early problem with the macro, such a test will often provide an error message that makes it easy to find the problem and fix it:

```
rx macro.TTX
+++ Error 5 in line 2: Unmatched quote
```

The logic and syntax of the ARexx portions of a script can be tested by running a program with command inhibition turned on. Commands are not issued to the ·host· in this mode, but all ·variable· substitutions and other evaluations of ARexx ·clauses· are done. Because commands are not issued, the RESULT variable will never be initialized, but its value can be manipulated by ·assigning· to it dummy values at any pause point during an interactive trace .
~~~~~~~~~~~~~~~~~

Next: Macros Tutorial | Prev: Macro address | Contents: Macros Tutorial

1.12 ARexxGuide | Tutorials | Online help system

ARexxGuide online help system
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The best way to learn any programming language is to use it. The best way to learn how to use it is to study working examples.

ARexxGuide can help in that process when called by the online help system macros included in the 'Editors' directory of the distribution archive. Macros are provided for several editors. If a macro for your editor of choice is not provided, this section explains how to build one.

Once the macro is installed, a single keypress will invoke the appropriate ARexxGuide node when the cursor is located on an ARexx ·function name·, ·instruction· ·keyword·, or one of the ·operator tokens· or special characters. The macros turn ARexxGuide into an online help system that can be used to decipher example scripts and to answer questions about the syntax of instructions, functions, and operators while you write a script.

Next: UNCRUNCH.REXX | Prev: Keystroke-macros | Contents: Tutorials

1.13 ARexxGuide | Tutorials | Online help system (1 of 4) | SETUP

Setting up a help key macro
~~~~~~~~~~~~~~~~~~~~~~~~~~~
Most editors allow at least some keys to be defined for special purposes and most will execute an ARexx script when a properly-defined key is pressed. The appropriate ARx_Help.#? macro should be attached to one of those keys.

In the Ed editor, for instance, function keys and Control-key combinations can be user-defined. The 'sf' command is used to create a new definition. It is followed by a number between 1 and 55 to indicate which key is being defined. Key 1 is the function key F1. Key 11 is the shifted F1 key. Keys 27 through 55 define various Control-key combinations.

This command will hook the ARexxGuide help-key macro into the F1 key:

```
sf 1 "rx \arx_help\"
```

The command can be issued at a '*' prompt after pressing the <Esc> key, but will be effective only during the current editing session. To make the definition more permanent, it can be placed in the file 's:Ed-Startup'. That file is read by Ed before a window is opened. Any menu or key definitions made in the startup file become effective for each Ed editing session.

TurboText uses a similar approach. The default setup for an editing session is included in the file 'TTX_Startup.dfn' which is normally located in the 'TurboText:Support' directory. The file includes different sections for different kinds of definitions. A keyboard assignment is made in the section headed 'KEYBOARD:'. To call up ARexxGuide when the <Ctrl> key and <Help> keys are pressed at the same time, the following would be entered:

```
CTRL-HELP              ExecARexxMacro "ARx_Help"
```

In Edge, key definitions are included in the file 'edge.keyboard' in the 'EdgePrg:' directory. User variations could be made there, but should be made, instead, in a file called 'EdgePrg:user.keyboard'. The following definition might be used for ARx_Help.edge:

```
KEYCOM   KEY="Help"      Q="Control"      COMMAND ARx_Help
```

The introductory comment in each of the supplied help-key macros includes an example definition entry for the startup file of that editor. For more information, read in your editor's documentation about 'definitions' or 'startup'.

Next: ENV VARIABLES | Prev: Help system | Contents: Help system

1.14 ARexxGuide | Tutorials | Online help system (2 of 4) | ENV VARIABLES

Environmental variables for online help system
~~~~~~
So that they will run under different system configurations, the help system uses environmental variables to find some settings. A setup file is provided. It is called by the macro when it is first run and will make a valiant attempt on an unpredictable array of systems to provide a user-friendly way to establish the initial settings for these variables.

The best way to use the setup file is to make the publicly-distributed function library rexxreqtools.library available to it. It will be used if it's available. If it isn't, the setup macro tries a number of other approaches before settling on console-window interaction.

These are the environmental variables used by the help-key macros:

```
   env:AmigaGuide/path
```

This one is used by AmigaGuide itself and should be set even if you don't use this guide. The AmigaGuide viewer searches the paths listed here to find a node in external files.
~~~~~~

When a space-separated list of directories is assigned to this variable, the .guide files that are stored in any of those directories will be visible to the viewer. The variable should, therefore, include the names of all directories where AmigaGuide files are located. The ARexxGuide Help System requires that at least one directory be included here: the place where ARx_Guide.xref and the other ARexxGuide files are located.

   env:ARexxGuide/AGCmd

Must include the name of the command (like 'sys:utilities/Multiview') used to show amigaguide files. The Help System uses this command to launch the viewer.

   env:ARexxGuide/ShowFullHelp

Must be set to 1 if the
                clause information window
                is to be shown. It can
be deleted or set to any other value if you want a simple editor requester when a match isn't found.

   env:ARexxGuide/XRef

The Help System will always load the file ARx_Guide.xref (and will exit if it can't be loaded), but it will also attempt to load any additional files named here. This makes it possible to use the same macro to obtain information from other guides. If a cross-reference file to the Edge help system is added, for instance, then pressing the help key when the cursor is on the name of an Edge command will cause the node explaining that command to be loaded -- not from ARexxGuide, but from the Edge help system.

Versions of the AmigaGuide viewer that include the 'Find Document' item in a 'Navigation' menu, use .xref files to locate the named document. This makes the menu item far more powerful.

Because many guides are distributed without cross-reference files, a utility called ARx_MakeNodeList.rexx is included in the distribution archive's Extras directory. Given the name (or a valid AmigaDOS pattern) of a file, the script will create a sorted cross-reference listing for the file or for all files matching the supplied pattern.

Although one could be created with the ARx_MakeNodeList utility, the .xref file included in the archive -- ARx_Guide.xref -- does not include a complete list of nodes in ARexxGuide. Hyperlinks within this guide include a full path to the file and node, so ARx_Guide.xref is limited to a list of the nodes that might be called by the Help System. The name of the current node, for instance, is 'HelpEnvVar'. That isn't a meaningful word in ARexx, so by leaving it out of the .xref file, a false match is avoided on a word that might be used as a variable name in a script.

A few false matches of that kind are still possible, however. Try the word 'string', for instance. Even if it is used as a variable, the Help System will call the ARexxGuide section on string tokens if the word is selected.

If you create a node list for another .guide, you may delete the names of any nodes that should not be shown by the Help System, but don't do that if the .xref file was provided by the .guide's creator. You might want to

create a second file for the guide that can by loaded by the Help System
and includes only the nodes you want the macro to invoke.

The command 'expungexref' is included with many versions of the AmigaGuide
utility. It will unload cross-reference files.

Next: FULL HELP | Prev: Help system | Contents: Help system

1.15 ARexxGuide | Tutorials | Online help system (3 of 4) | FULL HELP

Getting information about a non-matching word
~~~~~~
If the the ARexxGuide help system cannot find a match in the loaded
cross-reference files for the selected word, it will do one of two things:
either display a simple requester asking if you'd like to view the
ARexxGuide index, or display a window with several buttons that include
information about the current clause.

The clause information requester is implemented with rexxarplib.library ,
a freely-distributable function library that must be available in your
libs: directory if this option is chosen.

Because the macro goes through a complex process to build the information
presented here, the window will take a few seconds to appear. Seasoned
users who simply want help with the syntax of functions or instructions
will probably prefer the simple requester. It's enough to indicate that
the chosen word is not a name recognized in ARexx.

New users should find the clause information requester helpful. Although
it can be fooled by some complex lines of code, it is usually accurate in
identifying the different kinds of clauses, and in identifying the purpose
of the current token. It recognizes function arguments, function names,
strings, variables, and more. By looking at example code and studying the
nodes available from this requester, a new user can more quickly come to
understand what is happening in an ARexx script.

Because it will usually identify strings and variables in a clause, the
window can be helpful in tracking down missing quotation marks. Note,
however, that a string will not be correctly identified if a lone
quotation mark is included within a string such as { "can't go on" }.
Because of the uneven number of marks, tokens elsewhere in the line will
be mis-identified. That can be avoided by using something like this
alternative: { 'can''t go on' }.

If the clause information window is to be shown, the environmental
variable ARexxGuide/ShowFullHelp must be set to 1. That can be done with
the following command issued from the shell:

```
   setenv ARexxGuide/ShowFullHelp 1
```

Next: BUILDING A MACRO | Prev: ENV VARIABLES | Contents: Help system
~~~~~~

1.16 ARexxGuide | Tutorials | Online help system (4 of 4) | BUILDING A MACRO

```
                Building a new help-key macro
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If a macro for the editor you use is not included in the ARexxGuide
archive, you can quickly create one. The file ARx_Help.ed is the most
generic of the macros, but any of the supplied versions can be used as a
basis for your new script.

A macro for a different editor can be created by changing the commands in
just two of the subroutines used for ARx_Help.ed and explained in the
nodes listed below: GetCLine() and GetCPos(). The editor must be able to
pass to ARexx the text of the current line and the column number where the
cursor is located.

The editor commands in the macros are located in just one section and are
called as  internal functions . The commands are entered as subroutines at
the end of the script under the heading "Editor-specific commands". Each
of the subroutines is explained in more detail in these nodes:


                  GetCLine
                    Get current line from editor

                  GetCPos
                    Get cursor position from editor

                  GetCWord
                    Calculate (or get) current word

                  DisplayStatus
                    Display a message in status bar

                  BoolReq
                    Post a Boolean requester

                  GetWinInfo
                    Get information about editor's window

                  EditorExit
                    Send special editor commands on exit

                  SetExecStr
                    Set command issued by info window

                  SetAddress
                    Set the address used by info window

                  DisplayAG
                    Display the node in AmigaGuide viewer

Next: Help system | Prev: FULL HELP | Contents: Help system
```

1.17 ...Online help system | Building a macro (1 of 10) | GETCLINE() function

```
Get current line from editor
~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

This is one of two subroutines that depends on a minimal communication between ARexx and the editor. It must be changed for each editor. The subroutine retrieves the full text of the line on which the cursor is located.

This is the Ed version:

```
GetCLine: procedure expose EdInfo.
   'rv' '/EdInfo/'
   if rc ~= 0 then signal error
   return EdInfo.Current
```

Ed returns information with just one command, 'rv'. Information is packed into a compound variable using the supplied stem name. Because the information will be used by another subroutine, the full compound variable [EdInfo] is made global to the script with the EXPOSE keyword.

The text of the current line is assigned by Ed to the compound variable EdInfo.Current, so that value is returned.

In TurboText, the same task is accomplished with this subroutine:

```
GetCLine: procedure
   'GetPrefs tabwidth'
   TabSize = result
   'GetLine 1'
   return Tab2Space(TabSize,result)
```

Unlike Ed, which expands tabs to spaces, TurboText treats tabs as the character '09'x. The character would confuse other sections of the macro, so it is removed and replaced with spaces by the internal function Tab2Space, which is defined elsewhere in the script. The conversion function needs to know the tab width defined in the editor, so that information is retrieved in this subroutine. It is safe to use the tab-conversion ·subroutine· even if there are no tab characters in a line, so it should be called if there is any chance that the character will be included.

The command 'GetLine' retrieves the current line in TurboText and assigns its value to the RESULT variable. That's the value sent to the tab function and then returned by this function. (The '1' is used with 'GetLine' to prevent a line-feed character from being appended to the retrieved line.)

Next: GETCPOS() | Prev: BUILDING A MACRO | Contents: BUILDING A MACRO

1.18 ...Online help system | Building a macro (2 of 10) | GETCPOS()

```
Get cursor position from editor
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
This is the second of two subroutines that require minimal information
from the editor. It must be changed for each editor. The subroutine
retrieves the number of the text column on which the cursor is located.

This is how the task is accomplished in Ed:

      GetCPos: procedure expose EdInfo.
         if symbol('EdInfo.X') ~= 'VAR' then do
            'rv' '/EdInfo/'
            if rc ~= 0 then signal error
         end
         return EdInfo.Base + EdInfo.X

The  conditional clause  is used to verify that the compound variable
[EdInfo] has been assigned values by a previous call to Ed's `rv' command.
If it has not, then the command is issued here.

Ed returns a number in [<stem>.X] that indicates the window column where
the cursor is positioned, but if the cursor is on a line longer than the
current window's width, then that number alone is not enough to determine
the position. The value in [<stem>.BASE] indicates how many columns are
invisible on the left side of the window and must be added to the X
position.

Watch out for that in making a macro for other editors. The macro requires
the absolute column position and will not work properly without it.

In Edge, the task is accomplished with this version of the subroutine:

      GetCPos: procedure
         'GetEnvVar _WE_Column'
         return result + 1

Edge uses 0-based counting for column numbers, so the subroutine adds 1 to
give the macro the 1-based count that it expects.

Next: GETCWORD() | Prev: GETCLINE() | Contents: BUILDING A MACRO
```

1.19 ...Online help system | Building a macro (3 of 10) | GETCWORD() function

```
Calculate (or get) current word
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Unless the cursor is located on a  special character  like `+' or `(', the
macro displays a node based on the current word. In this script, a `word'
is defined as any collection of digits or alphabetic characters separated
from other collections by non-digit or non-alpha characters. For example,
the `word' should be interpreted as `left' if cursor were placed on the
`f' in this line:

   InitCap = upper(left(TWord, 1))

TurboText uses this definition internally, so the subroutine used with
that editor is simple:

      GetCWord:
```

```
         'GetWord'
         return result
```

The definition of 'word' varies considerably from editor to editor, as does the way an editor retrieves a word when the cursor is located between words. A default routine is provided with the Ed macro that can be used with any editor and will return a consistent result to the help system. The default routine should be used in most cases.

The subroutine does not depend on editor-specific command. It determines the current word based on the two arguments that are sent to the function: The full text of the current line and the column position. The version of the subroutine included in the distribution uses ·nested· functions to avoid variable ·assignments·. This version is more verbose, but easier to follow:

```
      GetCWord:  procedure
         parse arg Line, CPos
            /* These are the characters (since tabs are gone) that are  **
            ** considered separators between words: A space, any of the **
            **  operator characters , or  special characters            */
         SepChar = ' +-*/%|&~=><^,;:().""''\'

            /* Loop will lower the value of CPos if cursor is not on a  **
            ** valid word character. The line-end character is added    **
            ** in case an editor passes it along with line text.        */
         do while (verify(substr(Line,CPos,1),SepChar || '0a'x)=0) &,
                                  /* ·continuation· */           (CPos>1)
            CPos = CPos - 1
         end
            /* Trunctate word at any of the separator characters.       **
            **  VERIFY()  will return the position of the first sep     **
            ** character. That's used by  LEFT()  as an argument        */
         Word = left(Line,verify(Line' ', SepChar,,CPos) - 1)

            /* The line is now cut off at the _end_ of the word. We'll  **
            ** use the same process to find the start of the word by    **
            ** flipping it inside-out with  REVERSE()                   */
         Word = reverse(Word)
            /* We need to reverse [CPos] to match the reversed word.    **
            ** Since this is used as an argument to verify, we use      **
            **  MAX()  to guarantee that we'll never have a value less  **
            ** than 1.                                                  **
         RPos =  max(1,length(Word) + 1 - CPos)

            /* This is the same process used above, but acting on the   **
            ** reverse value.                                           */
         Word = left(Word,verify(Word' ', SepChar, 'M', RPos)-1)

            /* The second call to REVERSE() flips word to start-first.  */
         return reverse(Word)
```

The short version of this subroutine that is included in 'ARx_Help.ed' can be used with any editor since it does not depend on editor-specific commands.

Next: DISPLAYSTATUS() | Prev: GETCPOS() | Contents: BUILDING A MACRO

1.20 ...Online help system | Building a macro (4 of 10) | DISPLAYSTATUS() function

```
Display a message in status bar
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
This is an option that is used to display a message in the status-area of the editor. If it isn't needed, the subroutine should not be deleted, but should include 'return 0' as its only instruction.

This is the method used in Edge to display the message:

```
      DisplayStatus:
         'WindowTitle "'arg(1)'"'
         return 0
```

Although it doesn't work on some systems, this should display a message in Ed:

```
      DisplayStatus:
        'sm' '/'arg(1)'/'
         return 0
```

Next: BOOLREQ() | Prev: GETCWORD() | Contents: BUILDING A MACRO

1.21 ...Online help system | Building a macro (5 of 10) | BOOLREQ() function

```
Post a Boolean requester
~~~~~~~~~~~~~~~~~~~~~~~~
```
This subroutine displays a message and retrieves a 'Yes' or 'No' response from the user. In TurboText, it can be done this way:

```
      BoolReq:
         'RequestBool "'center(arg(1),37)'" "'center(arg(2),37)'"'
            /* return a Boolean value                                  */
         return ( result = 'YES' )
```

The first argument to the function is the titlebar text of the requester. The second argument is the text of the message body. The CENTER() function is used here to make the text more readable.

The routines that invoke this function expect a ·Boolean· as a return value, so a logical expression is used in the RETURN instruction to convert the "YES"/"NO" values returned by TTX into ARexx-recognized Booleans.

I have added to all of the included macros the following conditional:

```
      if show('L', 'rexxreqtools.library') then
         return rtezrequest(arg(2), '_Yes|_No', arg(1))
```

This causes the requester from reqtools.library to be used, but only if

the function library rexxreqtools.library was previously loaded. The macro will not cause that library to be loaded.

Ed presents a problem in this subroutine since it does not include a command to open a requester. The AmigaDOS 'Request' command could be used except that it is not included in versions of the OS prior to 3.0. The most generic solution, then, is to open a console window as a requester. This routine performs the task:

```
      BoolReq: procedure
            /* Format the text with some color                          */
         csi='9b'x;bold=csi'1m';norm=csi'0m';black=csi'31m';white=csi'32m'
         blue=csi'33m'
            /* Open the window as RAW: so that a single keystroke can  **
            ** be retrieved.                                           */
         if open(6InfoWin, 'raw:50/50/300/78/'arg(1)'/') then do
            call writeln(6InfoWin, csi'302070'x || '0a'x ||,
                        /* ·continuation· */          center(arg(2)),58)
            RText = '0a'x blue '      Respond' white||bold'y'norm ||,
                                    blue 'or' white||bold'n'blue||norm'.'
            call writeln(6InfoWin, RText)
            RText = black '  [<'white'Enter'black'> =' ||,
                    white'Y'black'. <'white'Esc'black'> =' white'N'black']'
               /* Use  WRITECH()  so there won't be a final line-feed   */
            call writech(6InfoWin, RText)
               /* Read just one keystroke                               */
            resp = readch(6InfoWin)
            call close 6InfoWin
               /* The <Enter> key is returned as '0d'x and not as '0a'x **
               ** Use  logical expression  to translate response into   **
               ** a ·Boolean value·.                                    */
            return (upper(resp) = 'Y' | resp = '0d'x)
         end
```

Next: GETWININFO() | Prev: DISPLAYSTATUS() | Contents: BUILDING A MACRO

1.22 ...Online help system | Building a macro (6 of 10) | GETWININFO() function

```
                Get information about editor's window
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This function is called when the
                 clause information window
                 is opened for
an unrecognized word. It returns a three-word value. The first word is the
current X position of the editor's window. The second word is the Y
position. The third word is the name of the public screen on which the
editor is displayed. The info-window will be opened at the position and on
the screen specified by this function.

Dummy values may be returned, as they are for Ed:

      GetWinInfo:
         return 0 0 ""
```

```
In TurboText, more genuine information is retrieved:

      GetWinInfo: procedure
        'GetScreenInfo'
         PubScreen = word(result, words(result))
         'GetWindowInfo'
         return word(result,1) word(result,2) PubScreen

Next: EDITOREXIT() | Prev: BOOLREQ() | Contents: BUILDING A MACRO
```

1.23 ...Online help system | Building a macro (7 of 10) | EDITOREXIT() function

```
Send special editor commands on exit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Some editors may need to issue special commands before the macro exits.
Those commands can be placed in this subroutine which is called both by
the error-handling routines and before standard exit from the macro.

If special handling is not required, the subroutine should not be removed,
but should simply return immediately:

   EditorExit:
      return 0

Next: SETEXECSTR() | Prev: GETWININFO() | Contents: BUILDING A MACRO
```

1.24 ...Online help system | Building a macro (8 of 10) | SETEXECSTR() function

```
                Set command issued by info window
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This subroutine can remain unchanged from the default for most editors. It
is called only when the
                 clause information window
                 is used, and sets the
command string used by the window's buttons to communicate with
ARexxGuide. A node name is appended to this string by the information
window routines. When a button is pressed, the entire string is sent to
the environment specified by the  SetAddress  function.

The default setting below can be used with any editor:

      SetExecStr:
         AGCmd = GetEnv('arexxguide/agcmd')
         if AGCmd = '' then signal NoCmd
         if ~abbrev(AGCmd, 'Multi') then
            return 'address ARX_ARP quit;if ~show(P,ARX_GUIDE) then do;',
               'address command; ''run >nil:' AGCmd 'ARexxGuide.guide',
               'portname ARX_GUIDE'';waitforport ARX_GUIDE; end;',
               'address ARX_GUIDE; windowtofront; link'
         else
            return 'address ARX_ARP quit; address command; ''run' AGCmd,
```

```
               'document'

(The ·comma continuation· token is used above to spread the definition of
a long string over several lines.')

Some editors can use custom commands, however. In Edge, for instance, the
built-in 'Help' command can be used to present the ARexxGuide information
in the same window as other Edge help pages:

      SetExecStr:
         return 'QUOTE address ARX_ARP quit; Help section'

Normally, the clause information window will simply append a node name to
the ExecStr string without other changes. If the first word in the string
is 'QUOTE', however, an additional change will be made: the entire string
will be enclosed in quotation marks. The string above might result in the
following command:

   "address ARX_ARP QUIT; Help section VARIABLE"


Next: SETADDRESS() | Prev: EDITOREXIT() | Contents: BUILDING A MACRO
```

1.25 ...Online help system | Building a macro (9 of 10) | SETADDRESS() function

```
                  Set the address used by info window
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This subroutine can remain unchanged from the default for most editors. It
is called only when the
                  clause information window
                  is used, and returns
the address with which the window's command buttons will communicate.

   SetAddress:
      return 'ARX_HELP'

This address will work with virtually any editor. It causes the infomation
window routine to set up a port that waits for commands from the window.
In some cases, however, the internal port may be unnecessary. Both the
Edge and TurboText macros, for instance, send the editor's  address() :

   SetAddress:
      return address()

Commands from the information window are sent back to the editor which
then executes them. If the address returned by this function is something
other than 'ARX_HELP', then the string returned by the
                  SetExecStr
                 function should also be changed from the default to something  ↩
                    appropriate
for the environment that will receive the command.

Next: DISPLAYAG() | Prev: SETEXECSTR() | Contents: BUILDING A MACRO
```

1.26 ...Online help system | Building a macro (10 of 10) | DISPLAYAG() function

```
Display an AmigaGuide document
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This subroutine can remain unchanged from the default for most editors. It
executes commands that will launch an AmigaGuide viewer or (for users of
version 34 of AmigaGuide) display a new node if the viewer is running.

The default version is included in ARx_Help.ed and in most of the other
example macros.

It is included in the editor routines because some newer editors may make
it a trivial matter to display a help page. In Edge, for instance, a
document can be displayed with this simple command:

   DisplayAG:
      '"Help Section' arg(1)'"'
      return 0

The built-in AmigaGuide support of the editor is used in this case.

The default routine is more complex because it must launch the viewer and
cause it to display a document.

Next: BUILDING A MACRO | Prev: SETADDRESS() | Contents: BUILDING A MACRO
```