Chapter 23
ARexx Support

The JForth ARexx support was written by Mike Haas, Martin Kees and Phil Burk. We at Delta Research would like to thank Martin for his generous contributions.

Note: You must have ARexx installed in your system for these tools to be useful. ARexx is a native component of AmigaDOS 2.0 or later. If you are using AmigaDOS 1.3 or earlier, you must purchase ARexx and install it.

What is ARexx?

ARexx is a programming language that can communicate with other applications. Using ARexx, for example, one could request data from a database application and send it to a spread sheet application. To support this facility, an application must be "ARexx compatible" by being able to receive commands from ARexx and execute them. A database program might have commands to search for, retrieve, and save data. MicroFiche Filer is an example of a database program with an extensive ARexx command set. A text editor might have ARexx commands corresponding to its editing command set. The Textra editor supplied with JForth has an extensive ARexx command set, which can be used with JForth to provide an 'integrated programming environment' (see the 'Integrating Textra and JForth' section in this chapter for details). AmigaVision, the multimedia presentation program, can control other programs using ARexx. ARexx can increase the power of a computer by combining the capabilities of various programs. Because of the wide acceptance of ARexx, Commodore includes it with Version 2.0 of AmigaDOS.

If you are developing an application, it is highly recommended that you provide ARexx compatibility, if appropriate. The JForth ARexx Toolbox provides the basic tools needed to call ARexx. It also provides some higher level tools for implementing a command parser, and for manipulating ARexx variables.

ARexx was developed for the Amiga by William Hawes. It is based on the REXX language described by M.F. Cowlishaw in *The REXX Language: A Practical Approach to Programming* (Prentice-Hall 1985)

Description of Files

The JForth support for ARexx is in a directory with the logical name "JRX:".

These files comprise the primary Arexx support files:

ARexxCalls.f = JForth calls to ARexx library

ARexxTools.f = tools to simplify the use of ARexx

MakeArexxMod = Make ARexx module, used when JForth is built

RXUnderKey.f = Install ARexx listener under KEY (eg. for Textra)

RexxClude.f = Support for Textra to ARexx interface

RexxVars.f = Calls for manipulating Rexx variables RexxView.f = An application that monitors ARexx messages for debugging

There are a number of test programs supplied as examples of using ARexx from JForth. When the ".f" test files are compiled, they will print instructions to the screen. The ".rexx" and ".srh" files are ARexx

scripts. The test files are in JRX:TESTS and are:

sample_rexx_host.f = sample Rexx host application. Use this as a model for your own applications.

drive.srh = rexx script to drive above sample

macro1.srh = rexx script callable as a macro from above sample

Test_RXTools.f = simple example of controlling JForth from ARexx

ToJF.rexx = sends commands to Test_RXTools.f

vartestf.f = JForth program for testing RexxVars.f

testrvi.rexx = Rexx Script for testing RexxVars.f

Low Level ARexx Support

ARexx structures and constants are defined in "include" files. These have been precompiled into a module called AREXXMOD. To access this module from your program, enter:

```
GETMODULE AREXXMOD
```

JForth compatible calls to the ARexx library are also provided. These routines are documented in the ARexx manual. ARexx is somewhat unusual in that some of the routines return multiple values. Special tools had to be added to the JForth CALL facility to support these. To load these words, enter:

```
INCLUDE JRX:AREXXCALLS.F
```

To see the stack diagram of words in the library, use FILE?. For example:

```
FILE?  STCTOKEN()
```

will show you the following definition:

```
: StcToken()  ( string -- token scan length quote )
   call>abs RexxSysLib_lib StcToken  TuckA1 TuckA0 TuckD1
;
```

ARexx Toolbox Tutorial

[Please familiarize yourself with the ARexx manual before beginning this tutorial. ARexx must be installed for this to work.]

To simplify the development of ARexx compatible programs, we have provided an optional toolbox. This toolbox has routines for initializing the ARexx system, interpreting commands and argument strings, sending messages to ARexx, and cleaning up afterwards. To load this toolbox, enter:

```
INCLUDE JRX:AREXXTOOLS.F
```

To communicate with ARexx, we need to open the RexxSysLib library, create a message port, and initialize a message. When we open a message port, we have to give it a name that will serve as an "address" for ARexx messages. If we have an application called "JAZZ", we can do all of the above with a single call. Enter:

```
: JAZZ.INIT.RX  ( -- error? , setup ARexx )
   0" JAZZ" RX.INIT
;
```

Note that we used 0" instead of just " because ARexx and AmigaDOS use NUL terminated strings. A NUL terminated string is a string of ASCII characters ending with a zero. This is different than a Forth string that begins with a count byte equal to N, followed by N characters. When looking at stack diagrams, parameters like 0NAME and 0STRING are NUL terminated strings. Parameters like $NAME and $STRING are Forth style strings. See 0COUNT >DOS DOS0 and 0" in the Forth glossary for more information.

The stack diagrams for the RX.xxx words can be found in the glossary in this chapter.

We put the call to RX.INIT in a colon definition for two reasons. One is so that the 0NAME would be permanently compiled in the dictionary instead of being on the PAD. Strings defined outside of a colon definition go on the PAD and are soon overwritten. The second reason is because when you write applications, all of the code should be in colon definitions and not just called when the file compiles. For the purposes of the rest of this tutorial, however, we will often just type things in directly because it is easier when experimenting. Now enter:

```
JAZZ.INIT.RX  .
```

The number printed should be zero. It could be equal to the constants ERR_INSUFFICIENT_MEMORY or RXERR_NO_LIBRARY if either of those errors occurred. Checking for RXERR_NO_LIBRARY is one way to determine whether ARexx is installed on a system.

Now let's try sending a message through ARexx to our program. Make sure that you have a Amiga DOS Shell or CLI window open before executing the next command! Enter:

```
RX.WAIT.MSG
```

Notice that JForth has not returned from this word. It is waiting for a message from ARexx. Now click in the Shell window and enter an ARexx command. Pay close attention to the single and double quotes. Enter in the **Shell** (NOT in JForth):

```
RX ' address "JAZZ" "play song" '
```

Notice that JForth has now returned with 1 item on the stack which is the message from ARexx. Notice also that RX in the Amiga Shell window has not returned. RX is waiting for its message back so don't lose the message in JForth.

What does the RX command mean? When you enter an RX command in the shell, you can give as a parameter either an ARexx script filename or an ARexx command delimited by quotes. We used the ADDRESS command to send the message "play song" to the address "JAZZ". Let's now look in JForth to see if we got it. Enter:

```
0 OVER RX.ARG[] \ get address of 0th argument
@ IF>REL        \ fetch it, convert to relative address
0COUNT TYPE     \ print it
```

ARexx messages have up to 16 arguments that can be referenced using RX.ARG[]. They generally contain the absolute address of an ARexxString so we must convert it to relative before using it. The message printed should have been "play song" from our RX command.

Now we must reply to the message so that RX can continue. Enter:

```
DUP .  \ see that our message is still there
RX.REPLY.MSG
```

Let's look at another way of getting messages. Enter:

```
RX.GET.MSG DUP . \ should be zero
0SP
```

Zero means that there were no messages for our program. Let's send another message from the Shell. Enter in the Shell (NOT in JForth):

```
RX ' address "JAZZ" "12345 howdy" '
```

Let's see if we can pick up that number in JForth. Enter in JForth:

```
RX.GET.MSG DUP .          \ should be non-zero
0 OVER RX.ARG[] @ IF>REL \ get 0string
RX.PARSE.NUM .S           \ extract number
```

You should now see four numbers on the stack. The first is our message address, followed by 12,345 (!), followed by the address of the remainder of the string, and a -1 = TRUE. Let's take three of them

off of the stack.

```
. \ -1
0COUNT TYPE \ howdy
.  \ 12,345
```

Now before we reply to the message, let's pretend we detected an error that we want to tell ARexx about. The error could be due, for example, to a number out of range, an illegal command, or a song name that JAZZ doesn't know how to play. The numbers that you use are decided upon by you. Let's use 23 for now. To pass an error number back to ARexx, enter:

```
23  RX-RESULT1 !
RX.REPLY.MSG
```

RX should print that the command returned '23'.

An application that supports ARexx commands is called an ARexx host. Most ARexx hosts will need to implement a command set that can be used by other programs. Each command will have a name, a particular set of arguments, and an associated Forth function. We have simplified this process for you. Suppose that we have a very small command set: one word that prints dollar signs, and another that prints a string and a number. These are trivial routines. Your application can have many powerful routines but these two will serve as examples.

First let's define our command functions.

```
: DOLLARS  ( N -- )
   0 DO ASCII $ EMIT LOOP CR
;
: TYPE.  ( ADDR COUNT N -- )
   . CR
   TYPE CR
;
```

Test them by entering:

```
5 DOLLARS
" World Peace" count 123 TYPE.
```

Now lets define a word that, sets up the command interpreter.

```
: JAZZ.SETUP  ( -- )
   2 RX.ALLOC.CTABLE  \ just two commands
   IF
      " DOLLARS" " N" 'C DOLLARS RX.ADD.COMMAND
      " TYPEDOT" " SN" 'C TYPE. RX.ADD.COMMAND
   ELSE
      ." Couldn't allocate table!" cr
   THEN
;

JAZZ.SETUP
```

The stack diagram for RX.ADD.COMMAND is:

```
RX.ADD.COMMAND ( $name $format cfa -- )
```

The format is a string of 'N's and 'S's. For each N, an number will be parsed and passed to your function. For each S, an ADDR and COUNT of a string will be parsed and passed. The stack diagram of your functions must correspond to the format that you give.

There are several ways to use the command table. See RX.EXEC.MSG below. For a simple test we can use RX.SLAVE.SAFE. This will terminate when we touch the keyboard which is handy for

testing. Enter:

```
RX.SLAVE.SAFE
```

Now enter in the Shell (NOT in JForth):

```
RX  ' address "JAZZ" "DOLLARS 10" '
RX  ' address "JAZZ" "TYPEDOT hello 123" '
RX  ' address "JAZZ" "DOLLARS 25" '
```

Notice the messages in the JForth window. Now hit <RETURN> in JForth to stop RX.SLAVE.SAFE. When we are done with our command table, we should free it by entering:

```
RX.FREE.CTABLE
```

Now let's finish by entering:

```
RX.TERM
```

Instead of entering ARexx commands directly from the Shell, we could use scripts. Study the test programs in JRX:TESTS for more examples. Pay special attention to JRX:TESTS/SAMPLE_REXX_HOST.F.

ARexx Toolbox Glossary

These words are compiled from JRX:ARexxTools.f

RX-SAVE-MSG (-- addr , saved from RX.GET.MSG)

If your program aborts before getting a chance to call RX.REPLY.MSG, you can get the last message from this variable and reply to it. The use RX.FLUSH.MSG repeatedly until the Rexx program finishes.

RX.ADD.COMMAND ($name $format cfa --)

Add a command to the command table. You must call RX.ALLOC.CTABLE first. The format is a string of 'N's and 'S's. For each N, an number will be parsed and passed to your function. For each S, an ADDR and COUNT of a string will be parsed and passed. The stack diagram of your functions must correspond to the format that you give.

: RX.ALLOC.CTABLE (n -- table | 0)

Allocate table for commands. Call RX.FREE.CTABLE when finished using table. Returns zero if not enough memory.

RX.ARG[] (n rexxmsg -- arg-addr , index into argument array)

Calculate address of an argument in a rexx message. If the argument is a string it will be stored as an absolute address so use IF>REL after you fetch it and IF>ABS before storing a string.

RX.EXEC.LINE (0arg -- error?)

Looks up command in table, parses arguments based on format, and passes as parameters to a Forth function specified by RX.ADD.COMMAND. Error? will be zero, RXERR_ILLEGAL_COMMAND, RXERR_ILLEGAL_PARAMETER or the contents of RX-ERROR as set by your code.

If the word whose CFA you pass in RX.ADD.COMMAND has the wrong stack diagram and leaves something on the stack or eats too much from the stack, you will get an error message and an abort. Please then fix the word for that command. This is considered a serious programming error thus we abort instead of returning a flag.

RX.EXEC.MSG (rexxmsg --)

Pass argument 0 to RX.EXEC.LINE then sets RX-RESULT1 to ERROR?.

RX.FIND.COMMAND (addr cnt -- index true | false)

Look up command in table.

RX.FLUSH.MESSAGES (-- #msgs)

Reply to all messages not yet processed. Sets RX-RESULT1 = 10.

RX.FREE.CTABLE (-- , free allocated table)

Free table allocated by RX.ALLOC.CTABLE.

RX.GET.MSG (-- rexxmsg | 0 , if any are ready)

Be sure to reply using RX.REPLY.MSG when done with message.

RX.INIT (0hostname -- error?)

Open RXSysLib library, create message port, and create a RexxMsg for use if needed.

RX.JFORTH.INIT (-- error? , initialize port as "JFORTH")

RX.KILL.SCRIPT (rexxmsg -- , send CTRL-C to rexx script)

RX.PARSE.NUM (0string -- num 0scan true | false)

RX.PARSE.STRING ($format 0string -- ...params... 0left true | false)

Params will contain N for each 'N' in $format, and ADDR COUNT for each 'S'. If an error is encountered, a false is returned.

RX.PUT.ACTION (action -- , set in rx-message-ptr)

Set action field in RX-MSG-PTR. See ARexx manual for an explanation.

RX.PUT.MSG (0arg 0portname -- error?)

Sends the argument to the named port. Uses NUL terminated strings. Converts the argument to an official ARexx ArgString and calls SafePutToPort(). RX.PUT.MSG then waits for and processes messages using RX.EXEC.MSG until the original packet is returned.

RX.PUT.REXX (0arg -- error?)

Send message to resident process REXX Calls RX.PUT.MSG with a portname of 0" REXX".

RX.PUT.TEXTRA (0arg -- error?)

Calls RX.PUT.MSG with a portname of 0" TEXTRA". The RXCOMM, RXFF_RESULT, and RXFF_STRING bits are first set in the action field then restored to just RXCOMM.

RX.REPLY.MSG (rexxmsg --)

Replies to message. Passes RX-RESULT1 (integer) and RX-RESULT2 (argstring) if they are set and the sender requested a reply by setting the RXFF_RESULT flag.

RX.SLAVE (-- , process ARexx commands until RX-QUIT on)

Wait for messages, execute them using RX.EXEC.MSG, then reply to them. This continues until the variable RX-QUIT is set to TRUE. You must provide a quit command to do that.

RX.SLAVE.SAFE (--)

Like RX.SLAVE but also quits if you hit a key.

RX.TERM (-- , terminate if initialized)

RX.WAIT.MSG (-- rexxmsg)

Be sure to reply using RX.REPLY.MSG when done with message. As an alternative to waiting for the ARexx message, you could AND several Signals together and use Wait(). This would allow you to get IDCMP events while waiting for ARexx events.

SafePutToPort() (msg 0portname -- ok?)

Finds a port with the given name then sends it a message. Uses Disable() and Enable() calls.

Integrating Textra and JForth

Textra is a text editor written in JForth by Mike Haas. It is optimized for program development as opposed to word processing. Textra is provided with JForth and can be found in the JTX: directory. Textra can be freely redistributed.

Mike has added an ARexx interface to Textra that allows you to manipulate text from other programs. Textra also provides the ability to compile programs directly from the Textra window without going to disk. If an error is encountered during compilation, the offending line is highlighted.

To prepare the Textra/ARexx interface, you will need to copy the scripts to the REXX: directory. Enter in the shell:

```
COPY JTX:SCRIPTS REXX: ALL CLONE
```

Now run Textra, select "AREXX..." from the Utilities menu. Click in one of the long text gadgets and enter "jcompile". We recommend the gadget associated with <F10>. Then click on the <SAVE> button. Finally, click the <CANCEL> button to close the requester.

To compile the Textra Interface into JForth, enter in JForth:

```
INCLUDE JRX:RexxClude.f
```

To activate the interface, enter:

```
RX.INSTALL
```

JForth will now be ready for any ARexx message sent to the port named "JFORTH".

Note: Do not INCLUDE directly from Textra when you are writing programs that use the RX.xxx words. They will fight over the ARexx port and both will lose. Specifically, do not call RX.INIT when RX.INSTALL is active. Instead, write the file to disk and call the normal INCLUDE.

To test this interface, open a file using textra and enter a few lines of code. Put a deliberate error in one of the lines. The hit the <F10> function key.

JForth will begin to compile the contents of the Textra window. When it reaches the bug, it will highlight it in Textra. You can now fix the bug and hit <F10> again.

You can use SAVE-FORTH to save an image with this code already compiled. When you run this image RX.INSTALL will be called automatically by AUTO.INIT.

Warning: Do NOT select other windows or edit text while JForth is compiling from the window.

I recommend that you read the documents in JTX:DOCS. The file RexxCommand.doc contains a description of the commands that Textra supports. As long as we have Textra up and RX.INSTALL active let's have some fun.

Arrange your windows so that you can see the shell window, the JForth window and a Textra window all at once. In the Textra window, you should have a file with at least 20-30 lines for this experiment.

Note: Textra commands require a '@' before the command when sent from JForth. This is to distinguish commands from invocations of a script.

Lets try repositioning the Textra cursor. Enter in JForth:

```
0" @GOTOXY 10 5" RX.PUT.TEXTRA .
```

Did you notice the cursor jump in Textra. Enter these other commands and watch the cursor move.

```
0" @LEFT 3" RX.PUT.TEXTRA .
```

Now lets try selecting a line.

```
0" @SELECTLINE 8" RX.PUT.TEXTRA .
```

Now let's try retrieving the text that is currently selected in Textra. Select a different line if it is empty in your file. Enter:

```
: GETIT  ( -- , get and type )
   0" @get select text" rx.put.textra . cr
   rx-result2 @ 0count type cr
;
GETIT
```

The text is left at HERE which can be overwritten so be sure to copy it somewhere else if you want to keep it.

Now would be a good time to try out some of the commands described in the JTX:Docs/RexxCommand.doc file. You may also want to read some of the scripts in JTX:Scripts.

ARexx Variables Interface

The REXX Variables Interface (RVI) is a set of functions to allow an ARexx host to manipulate a macro program's symbol table. Using these functions the host can retrieve values for existing variables and install new values. There is no limit (except for available memory) to the number of variables that can be created, so the variables interface is a very convenient way to pass information to a macro program.

[Martin Kees created this code by disassembling REXXVARS.O which is a linkable object file supplied with ARexx. RVI was developed by William Hawes.]

The RVI functions can be called whenever the ARexx host holds a pending command message from an ARexx program. While the command message is outstanding, the macro program is waiting for the reply and the program's symbol table is in a consistent state. All calls to the interface functions must be completed before the message is replied.

In a typical application, a macro program issues a command to the host to perform some specific action, and may pass the name of a stem variable as an argument. The host performs the command and fills in the appropriate variables before replying to the message. When the macro program resumes operation, it can check the values of the variables set by the host.

Here is an outline of that transaction:

1) JForth host calls ARexx macro (optional).

2) ARexx macro sends command to JForth host.

3) JForth host can read or write macros variables using RVI.

4) JForth host replies to command.

5) ARexx macro can use new values in variables. (optional)

Variable Names

Variable names are specified by a pointer to a null-terminated string and must follow the REXX language symbol conventions. Alphabetic characters must be in UPPERCASE. The variable name is treated as a literal string, and no substitution for compound symbols is performed.

The host application should document the variables affected by each command so that the macro programmer knows where to find the information, and to avoid accidental conflicts in variable names. If a large number of values must be passed, the host should define them as compound variables and let the macro program pass the stem name as an argument.

The RVI Glossary

To use the RVI functions, include the file "JRX:RexxVars.f". It is not necessary to define the ARexx library base (RexxSysBase), as the ARexx library is opened on the fly when required.

CheckRexxMsg (rexxmsg -- ok?)

This function verifies that the message pointer is a valid RexxMsg and that it came from an ARexx macro program. The validation test is more stringent than that performed by the ARexx library function IsRexx(). The latter verifies that the message is tagged as a RexxMsg structure, but not that it necessarily came from an ARexx macro program. Each macro program installs a pointer to its global data structure in the command message, and this pointer is necessary to gain access to the symbol table.

The return from the function will be non-zero (TRUE) if the message is valid, and 0 (FALSE) otherwise.

GetRexxVar (rexxmsg 0variablename -- 0value error?)

This function retrieves the current value for the specified variable name. It first validates the message using CheckRexxMsg() and then, if the message pointer is valid, retrieves the value string and passes it in the supplied return pointer. The return pointer is actually an argstring (an offset pointer to a RexxArg structure), but can be treated as a pointer to a null-terminated string. The value must not be disturbed by the host.

The function return will be zero if the value was successfully retrieved and non-zero otherwise. An error code of 10 indicates an invalid message.

Here is an example of getting and displaying the value of a variable named PRICE.

```
MyRexxMsg 0" PRICE" GetRexxVar 0=
IF  ." Price = " 0count type cr
ELSE drop ." AN error occured getting PRICE" cr
THEN
```

SetRexxVar (rexxmsg 0variablename addrval length -- error?)

This function installs a value in the specified variable. It validates the message pointer using CheckRexxMsg() and then installs the value, creating a symbol table entry if required. The value is supplied as a pointer to a data area along with the total length; the data may contain arbitrary values and need not be null-terminated.

The function return will be zero if the call was successful and non-zero otherwise. The possible error codes are given in the table below.

Error Code Reason for Failure

```
3     Insufficient storage
9     String too long
10 Invalid message
```

Here is an example of setting a variable named PRICE to 1234.

```
MyRexxMsg 0" PRICE" 1234 n>text SetRexxVar
IF ." Could not set PRICE!" cr
THEN
```

Note that the value returned by GetRexxVar() is an argstring (pointer), and the value passed to SetRexxVar() is just a pointer to a data area.

RVI Code Examples and Test Program

A sample JForth program and an ARexx script to exercise the RVI functions are included. They are:

```
JRX:Tests/VarTest.f
JRX:Tests/TestRVI.rexx
```

VarTest opens a public port named "VarTest" and waits for messages to arrive from ARexx. Each message is validated with a call to CheckRexMsg(), after which the value for A.1 and A.2 is retrieved using GetRexxVar(). The program then installs the value "A-OK" in the variable STATUS and replies the message. VarTest exits when it receives a "CLOSE" command. The ARexx program TestRVI.rexx will demonstrate the VarTest host.

Note how the variable values are affected by the procedure calls in ARexx. In the first call from TestMem, A.1 will be NULL. Then it will be set to "Local to TestMem" for the next call. Then it reverts to its original value when we return from the procedure TestMem. This is because PROCEDURE in ARexx creates a new symbol table. A.2 keeps its value because it is EXPOSEd.

Here is the procedure for running this test program:

In JForth:

```
INCLUDE JRX:RexxVars.f  \ takes a while to assemble
INCLUDE JRX:tests/VarTest.f
VARTEST
```

In Amiga Shell:

```
RX JRX:Tests/TestRVI
```

RexxView

RexxView is a CLI utility that monitors the REXX port. Information is listed to a file describing each message received by REXX. JForth source code is in file jrx:RexxView.f. RexxView is designed to be cloned.

Usage:

```
rexxview  outfile
```

For example:

```
rexxview con:0/0/640/100/rv
(or)
rexxview PRT:
```

Rexxview is terminated by sending the string "closerexxview" to the REXX port. To do this enter in the CLI:

```
rx closerexxview
```