

ARexxBox_E

COLLABORATORS

	TITLE : ARexxBox_E		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		July 23, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	ARexxBox_E	1
1.1	ARexxBox_E.guide	1
1.2	ARexxBox_E.guide/Copyright	2
1.3	ARexxBox_E.guide/Important	2
1.4	ARexxBox_E.guide/Addresses	3
1.5	ARexxBox_E.guide/Introduction	4
1.6	ARexxBox_E.guide/Requirements	5
1.7	ARexxBox_E.guide/Installation	6
1.8	ARexxBox_E.guide/Usage	6
1.9	ARexxBox_E.guide/Input	7
1.10	ARexxBox_E.guide/MsgPort Basename	7
1.11	ARexxBox_E.guide/CommandShell	8
1.12	ARexxBox_E.guide/Merge in	8
1.13	ARexxBox_E.guide/Print	9
1.14	ARexxBox_E.guide/Generate Source	9
1.15	ARexxBox_E.guide/Clipboard	10
1.16	ARexxBox_E.guide/Concept	10
1.17	ARexxBox_E.guide/Initializing	10
1.18	ARexxBox_E.guide/Commands from ARexx	11
1.19	ARexxBox_E.guide/Expanding commands	11
1.20	ARexxBox_E.guide/Commands to ARexx	12
1.21	ARexxBox_E.guide/Closedown	12
1.22	ARexxBox_E.guide/Files	13
1.23	ARexxBox_E.guide/InterfaceFunctions	14
1.24	ARexxBox_E.guide/Arguments&Results	17
1.25	ARexxBox_E.guide/Errors	20
1.26	ARexxBox_E.guide/How2use	21
1.27	ARexxBox_E.guide/Oberon-2	23
1.28	ARexxBox_E.guide/ReadArgs	24
1.29	ARexxBox_E.guide/Library	27

1.30	ARexxBox_E.guide/ARexxBox-ARexxDispatch	27
1.31	ARexxBox_E.guide/ARexxBox-CloseDownARexxHost	28
1.32	ARexxBox_E.guide/ARexxBox-CommandShell	29
1.33	ARexxBox_E.guide/ARexxBox-CommandToRexx	30
1.34	ARexxBox_E.guide/ARexxBox-CreateRexxCommand	30
1.35	ARexxBox_E.guide/ARexxBox-DoShellCommand	31
1.36	ARexxBox_E.guide/ARexxBox-ExpandRXCommand	32
1.37	ARexxBox_E.guide/ARexxBox-FindRXCommand	33
1.38	ARexxBox_E.guide/ARexxBox-FreeRexxCommand	34
1.39	ARexxBox_E.guide/ARexxBox-ReplyRexxCommand	34
1.40	ARexxBox_E.guide/ARexxBox-SendRexxCommand	35
1.41	ARexxBox_E.guide/ARexxBox-SetupARexxHost	36
1.42	ARexxBox_E.guide/ARexxBox-StrDup	37
1.43	ARexxBox_E.guide/FileFormat	38
1.44	ARexxBox_E.guide/History	39
1.45	ARexxBox_E.guide/Thanks	45
1.46	ARexxBox_E.guide/Index	45

Chapter 1

ARexxBox_E

1.1 ARexxBox_E.guide

ARexxBox 1.11 Documentation

Common stuff:

Copyright	Copyright and other legal stuff
Important	Read this!
Addresses	Where to send bug reports, comments and gifts

The ARexxBox (ARB) :

Introduction	Why ARexxBox?
Requirements	68040, 16 MB RAM, ... ;-)
Installation	What to put where
Usage	What happens if I push this button?

The generated Code:

Concept	How it is supposed to work
Files	What is where
InterfaceFunctions	Interface between ARexx and your program
Arguments&Results	Parameters for and result values from commands
Errors	ARB's extended error code system
How2use	Step by step guidelines
Oberon-2	Remarks to the Oberon-2 source

Lookup chapters:

ReadArgs	ReadArgs switches and their types
Library	The library functions of ARB

Appendices:

FileFormat	Syntax of the .arb file format
History	The development history of ARexxBox
Thanks	The author would like to thank...
Index	Index for this document

1.2 ARexxBox_E.guide/Copyright

Copyright and other legal stuff

Copyright (C) 1992,1993 Michael Balzer

Oberon-2 source copyright (C) 1993 hartmut Goebel

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

No guarantee of any kind is given that the programs described in this document are 100% reliable. You are using this material at your own risk. The authors *can not* be made responsible for any damage which is caused by using these programs.

This package is freely distributable, but still copyright by Michael Balzer. This means that you can copy it freely as long as you don't ask for a more than nominal copying fee. This fee *must not* be more than US \$5 or 5 DM.

This limit applies to German Public-Domain dealers too!!

Permission is granted to include this package in Public-Domain collections, especially in Fred Fish's Amiga Disk Library (including CD ROM versions of it). The distribution file may be uploaded to Bulletin Board Systems or FTP servers. If you want to distribute this program you *must* use the original distribution archive 'ARexxBox111.lha'.

The authors would like to see this packet included in any kind of developer's tool chest (e.g. the Native Developer's Upgrade Kit by Commodore), but please tell us if you do so.

None of the programs *nor* the source code (nor parts of it) may be used on any machine which is used for the research, development, construction, testing or production of weapons or other military applications. This also includes any machine which is used for training persons for *any* of the above mentioned purposes.

1.3 ARexxBox_E.guide/Important

Important remarks

Before you do anything else: ARexxBox is a tool by programmers for programmers. If you're a user, this program is of no use for you, please pass it to a friend who develops software.

One of the primary design goals of the ARexxBox and especially the generated code was to come as close as possible to the ideas of the "User Interface Style Guide" by Commodore.

If you haven't read this book yet, I'd recommend you do so before reading on and using the ARexxBox. Many of the potentially confusing things will be clearer to you.

In my opinion, we have to establish a standard for ARexx interfaces; I think it's a very annoying thing for the user having to learn a new syntax and usage of ARexx for each new program she buys.

Of course, the Style Guide can't give final solutions to all problems -- many of the ideas mentioned in the Guide need to be extended a bit. Especially some standard command's parameters leave some questions. I have tried to complete these things when converting them to ARexxBox, but there are certainly many more places than I recognized. If you do such an extension, please let me know, so we can share it amongst all ARB users.

ARexxBox is FreeWare, meaning you may copy and distribute it freely as long as you follow the conditions mentioned in the copyright chapter, but the complete ARexxBox is still copyrighted by me, the author. However, I grant you the right to use the generated code freely as long as you do not use it for military purposes as mentioned in the copyright chapter. You don't have to pay any licenses for any part of the software.

All I'm asking for is that you put a remark into both the About window and the documentation for any software using ARB generated code stating that the ARexx interface was designed using the ARexxBox. Also, I'd like to get (email preferred) a list of the supported ARexx commands, but that's not a must.

1.4 ARexxBox_E.guide/Addresses

Addresses of the Authors

You can reach the author under one of the following addresses:

Mail:

Michael Balzer
Wildermuthstraße 18
W-5828 Ennepetal (58256 from July 1993 on!)
GERMANY

InterNet Electronic Mail:

At work:
balzer@heike.informatik.uni-dortmund.de

At home:
bilbo@bagsend.aworld.de

or in the german Z-Net
m.balzer@aworld.zer
or from Fido
Michael Balzer of 2:241/5604.19

The Fido address hasn't been tested up to now, if you don't get a
reply please try another way.

If you have questions concerning the Oberon-2 source, please contact:

Postadresse:

hartmut Goebel
Aufseßplatz 5
W-8500 Nürnberg 40 (90459 from july 1993 on!)
GERMANY

InterNet Electronic Mail:

At home:
hartmut@oberon.nbg.sub.org
or in the german Z-Net (not very frequently)
hartmut@asn.zer
or from Fido
Harmut Goebel of 2:246/81.1

1.5 ARexxBox_E.guide/Introduction

Introduction

You probably already know how important ARexx is to the Amiga. So
I'll concentrate on convincing you to use ARexxBox for your ARexx
interfaces ;-).

Why ARexxBox?

=====

ARexxBox (inspired by Jan van den Baard's GadToolsBox) is a tool
that eases the design, construction and implementation of an ARexx
interface for a program. An interface generated by ARexxBox
automatically conforms to the ARexx standards mentioned in Commodore's
"User Interface Style Guide".

Here are the features:

- * Syntax and result method conform to the "Style Guide", e.g. the
code uses ReadArgs to give the commands a syntax similar to DOS
commands, and there is automatic support for the keywords VAR and
STEM.
 - * Each command of an interface may have as many arguments and results
as you like.
 - * All ReadArgs() templates are allowed for arguments and will be
recognized by ARexxBox, for results you may use the options /N and
-

/M -- also coming with full support by the Box.

- * Graphical user interface (made with GadToolsBox :-).
- * Generates C or Oberon-2 source(1)
- * There's an option to generate code for a CommandShell where the user may enter ARexx commands directly and view the results.
- * The CommandShell can execute batch programs as well.
- * A program may open an arbitrary number of ARexx ports and CommandShells.
- * The standard commands mentioned by the "Style Guide" come in separate files, ready--to--use. There's example interface code for some of these.

----- Footnotes -----

(1) The Oberon-2 source was designed to be object oriented and "type-save".

1.6 ARexxBox_E.guide/Requirements

Requirements

Hardware

=====

The hardware requirements are neglectible, ARB should run on any Amiga.

Okay, there's **one** thing: As the ARB window is taller than 200 lines, you have to use something larger than that. Or you could set the system font to some 6 or 7 point font.

As programmers normally don't work on a 640x200 screen, this shouldn't cast any problems.

Software

=====

Both ARB and the generated code will only run on OS2.04 or higher.

The Box itself also makes use of the regtools library by Nico Francois. This is only for the Box, the generated code does not use it!

C source

=====

You will need an ANSI compatible C compiler which offers an option to include (link) functions from 'amiga.lib' by Commodore.

The 'amiga.lib' must be version 37.32 (12.11.91) or later. This version is available as part of the Native Developer's Upgrade Kit for OS2.04.

You can do without 'amiga.lib'. I reassembled the needed functions ('GetRexxVar' and 'SetRexxVar') and included them as separate source. Aztec spends very much time scanning 'amiga.lib', so if you link with this object you will get shorter turn-around times.

I, on the other hand, cannot guarantee that this code is completely compatible to those from 'amiga.lib', and of course you won't make use of any future enhancements in the lib.

Oberon-2 source
=====

You will need AmigaOberon V3.0 or later.

The source is based upon a non standard class library and some other modules (see Oberon subdirectory).

The module RVI is an interface to some part of the amiga.lib, you have to link with this module if using amiga.lib functions:

OLink RxTest OBJ oberon:obj/amiga.lib

If you have got the object file "rexxvars.o" from one of the ARexx developer kits, you can use the \$JOIN option to link it onto the object file of the RVI module. In this case, you don't need to specify amiga.lib for linking. Doing the same with amiga.lib itself is not recommended, your files would become very big.

1.7 ARexxBox_E.guide/Installation

Installation

Basically you don't have to install anything. Just copy all parts to a place where you (or your compiler) can find them. Some Oberon modules are not only for ARB but also of general use.

1.8 ARexxBox_E.guide/Usage

Usage

There's nothing unusual in using ARexxBox. If you've ever used an amiga application, you should have no problems at all.

Before defining some complex ARexx interface, you should probably

first read the chapter explaining the concept of the Box. To get a first impression, you could now load the file 'arb/Misc.arb'. Click on 'GETATTR' in the leftmost list.

The remaining two listviews should now fill with the parameters and result fields of the command 'GETATTR'. Each element of any list can be changed using the gadgets below of the list.

Input	How to enter a new command
MsgPort Basename	
CommandShell	
Merge in	
Print	
Generate Source	To C or not to C, that is the question
Clipboard	Cut, Copy and Paste

1.9 ARexxBox_E.guide/Input

Input
=====

To enter a command definition, do the following steps: Push key N, enter the name of the ARexx command and confirm with RETURN. ARexxBox will now convert all characters to upper case and those not allowed to underscores ('_') and then check if the name is unique. If there is already a command with that name, you can change the name of the new command and try again.

Then enter the arguments. For each argument, push E and enter the name (optionally with ReadArgs style type tags). This name will also be checked for collisions with existing arguments.

With results, it's just the same, only the shortcut for a new result field is key W.

You can move args and results up and down, either just after entering them or later after selecting them via clicking on them, by pushing the gadgets labeled 'up' and 'do'. You can remove any list element by selecting it and then clicking on 'Remove'.

The types of the arguments and results must be given in ReadArgs() style (see AutoDocs), for example ARG1/K/N or LIST/M. Only the (optional and combinable) switches /N and /M are allowed for results.

1.10 ARexxBox_E.guide/MsgPort Basename

MsgPort Basename
=====

The 'MsgPort Basename' is the default base name for all ARexx message ports opened by the application. If necessary, a proper number will be appended to the name to make it unique (e.g. "FOOWRITE.13").

Any application should recognize the shell or workbench argument "PORTNAME" to change this base name. Such a user supplied base name will also be made unique via an appended number if necessary.

In addition, the MsgPort Basename will be used as the standard extension for ARexx script files for the application, so you should choose a short text for it. The PORTNAME tooltype has no effect on this.

1.11 ARexxBox_E.guide/CommandShell

CommandShell
=====

For C, ARB will generate additional code for a command shell if you turn this gadget on. You should normally leave it on, as the command shell code can also be used to execute external macro programs.

The switch makes no difference for Oberon, where the command shell code is always available.

The command shell is a kind of shell, which gives the user a direct interface to all ARexx commands of an application. The user can execute the ARexx commands from this shell as if it were a normal shell and the commands were DOS commands. The results of commands get printed in a readable form in the shell window.

For example, you could use the commandshell to quickly execute some macro without having to use ARexx directly. Or you can use this as a way to make some commands only available to the advanced user.

If you give a command shell a file as its input channel, it will execute the commands in the file line by line. So you can also use it to execute external macro programs.

1.12 ARexxBox_E.guide/Merge in

Merge in
=====

Merge is like Open, only it does not clear the command list before loading the data file. Merge will only add commands which are not already defined in the current project.

So with merge, you can build your own ARexx command packages for different purposes and later combine them as you need them. I already entered all standard commands mentioned by the Style Guide, try to

combine some of these packages into your test file.

1.13 ARexxBox_E.guide/Print

Print
=====

This shall become a documentation aid. It currently only prints name, syntax and results for all commands. If you have a good suggestion on a better output format for this function, please tell me.

1.14 ARexxBox_E.guide/Generate Source

Generate Source
=====

Starts the source generator. You may select any part of an already existing source output in the file requester or enter a complete new base name. Any ARB extension (`.c`, `.h` or `_rxif.c` or for Oberon `.mod`) will be cut off. Some safety checks will be done (e.g. if you are going to destroy another file accidentally) and then the source will be generated.

The new source files first go into `T:`, to prevent possible damages caused by crashes or other bad things. The new files will be copied to the actual destination directory as soon as they have been completed in `T:`.

The new files in `T:` won't be deleted if some error happens while copying them to their real location. So if some bad things happen, you still have a chance to save them yourself.

When generating the source, the Box will set the internal status of all commands to `old`. A new command has status `new` and if you edit a command, the Box sets the status to `changed`. The source generator relies on this status flag to tell which commands have been altered since the last source generation. If a command has been altered, the generator will put a remark ("ATT: Interface changed!") into the source right before the interface function correlating to that command.

You should look for this comment and check the consistency of the interface and the variable naming of the RXD structure against the real (perhaps on some older version basing) code before compiling again. Some cases the compiler won't find...

An important thing to have this remarks working is to save an ARB project after generating the source to it. If you do not save it, the changes of the status flags won't be saved as well. That's not tragic, but the Box will not know about the changes next time.

There are currently two source generators available, one for C (the original thing) and one for Oberon (by hartmut Goebel). Please ask hartmut if you have questions concerning the Oberon part.

1.15 ARexxBox_E.guide/Clipboard

Clipboard
=====

Cut, Copy and Erase work on either the currently selected command or on a range of commands.

To mark a range for these operations, choose "Mark Range", click on the first and then on the last command of the wanted range.

Selecting a single command clears the range selection.

Paste, like Merge, will only include commands not already there.

1.16 ARexxBox_E.guide/Concept

Concept

Besides conforming to the Style Guide, another primary design goal of ARexxBox was to ease the usage of ARexx to a programmer as much as possible.

Initializing	Open port
Commands from ARexx	Dispatcher and interface functions
Expanding commands	ExpandRXCommand() and external commands
Commands to ARexx	SendRexxCommand()
Closedown	Close port

1.17 ARexxBox_E.guide/Initializing

Initializing
=====

The function SetupARexxHost() will open the ARexx port and return a pointer to an object of the type 'struct RexxHost'. This pointer is a kind of file handle for the ARexx port, all other ARexx functions generated by the Box need this pointer.

In Oberon-2 SetupARexxHost() also returns a pointer, which serves as the 'receiver' of the methods.

So all operations relate to a specific port (an application can have any number of ARexx ports, e.g. a text editor could open one port for each open file).

1.18 ARexxBox_E.guide/Commands from ARexx

Commands from ARexx
=====

To handle its ARexx port, all the application has to do is listen to the port (signal) and call the function `ARexxDispatch()`, which will fetch and handle all incoming messages until the port is empty and then return.

Each command has its corresponding interface function (the interface between ARexx and your code), the address of this interface function is in the command list entry to the command. Normally such an interface function just checks the parameters and then calls the right core functions of your application. All you have to do is implementing these interface functions -- and the Box will help you by generating the skeleton code for each new command.

The actual (user supplied) parameters for a command are passed to the interface function via a specially created data structure. The interface function in turn returns its results by storing them in the same data structure before returning. The ARB routines take care to pass the results back to ARexx according to their types and conforming to the Style Guide.

1.19 ARexxBox_E.guide/Expanding commands

Expanding commands
=====

To support both the ALIAS standard command and other things like macros, there is a callback function which will be called by the parser before discarding an unknown command.

The parser analyzes incoming command lines and determines which interface function to call. If the command line does not match any command of the list, the parser will call `ExpandRXCommand()`, which could then do an ALIAS expansion and return the substituted command to the parser.

The parser will then try once again to recognize the command. If this second try fails again, the original command will be sent to the ARexx server process. So if there is an external script program of this name, it will be executed as if it were an internal part of your application, external commands are totally transparent to the user. The

result of an external command will be sent back to the original caller.

Demonstration of this feature: Change to directory 'test' and start 'test'. Close the CommandShell by typing EOF (C-\) and then enter the following command in another shell:

```
'rx "options results; address 'arbtest'; test 'FooBar'; say result'"'
```

As the test program does not have an internal command called "test", the external script 'test.arbtest' will be executed, which in turn will output a message into the output window of the test program and then return the string "Testtext!" to the caller (yeah, I know, very imaginative).

If no external command of the given name exists, the command will finally be classified as "unknown" and the parser will return an error to the caller.

Once again the complete process for a command:

1. Command in internal command list?
2. If not: ExpandRXCommand() -> new Command known?
3. If not: Is it an external script?
4. If not: Error - not implemented

1.20 ARexxBox_E.guide/Commands to ARexx

Commands to ARexx
=====

Of course the application may also send commands to ARexx, there are some special support functions for this direction available (see SendRexxCommand).

As such commands generally (being just normal Exec messages) run asynchronously, the ARexxBox has a special mechanism to react on the return of such a call. So you can start an ARexx command and then later get the result from it. Of course you have to make sure you can identify which result belongs to which command (see chapter on how to use the code).

To prevent late replies to break, the Box routines will count all sent commands and reject to close the port until all replies have been received (see Closedown).

1.21 ARexxBox_E.guide/Closedown

Closedown
=====

SetupARexxHost's counterpart is obviously CloseDownARexxHost. This one closes the port and releases all allocated resources.

The final closedown may be delayed automatically, if there are some commands which were sent to ARexx but have not returned yet. The Box will then wait for all these commands to be finished before closing the port. Any new commands coming up during this phase will be returned immediately with an error "Host closing down".

1.22 ARexxBox_E.guide/Files

Files

The source generated by ARB should be compilable without any changes. All needed interface functions (the only part you have to work on) should at least be there as empty template code.

C
=

The C source consists of five files:

- 'name.c' contains the basic functions like SendRexxCommand(), SetupARexxHost() and ARexxDispatch(). Normally, you only have to compile this module once, as there are no changes from editing the command list to this module. The only change in the Box which also changes this module is the switch for 'CommandShell', as the command shell functions reside in this module.
- 'name.h' contains the basic constants and structures for the ARexx port. Also found here are the prototypes and structure definitions for all the needed interface functions.
- 'name_rxcl.c' carries the global variables, first of all the list of commands for the interface.
- 'name_rxif.c' contains the interface functions. This is the only module you have to change. The Box will automatically keep any existing code and save functions no longer needed (e.g. if you have deleted some commands) to the file
- 'name_rxifstore', from where they can be retrieved later if you decide to put them in again or want to use them in another application.

Oberon
=====

The Oberon source consists of four files:

- 'RxName.mod' carries the command list and some basic functions like SetupARexxHost().
- 'NameARB.mod' defines the basic constants.
- 'NameRXIF.mod' contains all the interface functions and their data structures. This is the only module you have to edit. Existing functions will be preserved, functions no longer needed will be moved to the file
- 'Name_rxifstore', from where you can get them back later.

ARB special comments

=====

In the interface module, the Box will generate some comments of the form '/* \$ARB: ... */' (or on Oberon '(* !ARB: ... *)', which must not be changed in any way.

Somewhere at the top of the interface module there will be a comment telling to which project file this source module belongs. The Box will check this ID and reject to extend an existing code which has been generated from another ARB project.

Each interface function is enclosed in two comments ('B'egin and 'E'nd with ID of function). All source code between these two comments will be considered as belonging to this function and will always be preserved (or moved to the storage file) completely. So if you need some variables or support functions for an interface function, be sure to define them between the proper comment lines.

If you changed the definition of a command, the Box will generate a warning comment right ahead of the corresponding interface function. This comment of the form "ATT: Interface changed!" may be removed, but you really should first check if the code needs to be changed according to the new arguments or results.

1.23 ARexxBox_E.guide/InterfaceFunctions

Interface Functions

Basics

=====

An interface function consists of three phases,

- * initialization,
- * the working phase and
- * closedown,

which will be called by the ARexxBox routines in this order.

The first phase serves for allocating and initializing the memory for the transfer data structure that will transfer the arguments and result values between the basic functions and the interface function.

You can also use this phase for other initializations, e.g. allocating local buffer memory or the like.

The second phase should do the actual work and use the parameters from the previously (phase 1) allocated transfer structure. Any results should be stored into this structure as well before returning.

There are two special elements in every transfer structure, rc and rc2. These two relate to the normal return codes, any ARexx command may produce. In addition, the ARB offers an extended error code system (see chapter on Errors).

After execution of phase 2, the Box will extract all results and send them back to the ARexx server process. Then phase 3 will be called for freeing all locally allocated resources.

You will find some working examples on the form and function of interface functions in the directory 'rxif'. Especially the -- completely working -- HELP function should serve well as a complex example.

Anyway, here is the concrete structure of an interface function:

```
void rx_help( struct REXXHost *host, struct rxd_help **rxd,
              long action, struct REXXMsg *rexxmsg )
{
    struct rxd_help *rd = *rxd;    /* for simplification and local
                                    extensions */

    struct rxs_command *rxc;        /* local variables */
    int cnt = 1;

    switch( action )
    {
        case RXIF_INIT:
            /* Phase 1! */
            /* allocate the transfer structure */
            *rxd = calloc( sizeof *rd, 1 );
            /* no defaults to set, so just return */
            break;

        case RXIF_ACTION:
            /* Phase 2! */
            /* do it! */

            if( rd->arg.prompt )
            {
                rd->rc = -10;
                rd->rc2 = (long) "Prompt option not yet implemented";
                return;
            }
    }
```

```

        and so on, making also some allocations:
        rd->res.commanddesc = malloc( ... )
        and
        rd->res.commandlist = [calloc];
        which have to be freed in phase 3...

        break;

case RXIF_FREE:
    /* Phase 3! */
    /* free any allocated ressources */

    if( rd->res.commanddesc )
        free( rd->res.commanddesc );
    if( rd->res.commandlist )
        free( rd->res.commandlist );

    /* free the transfer object */
    free( rd );
    break;
}

/* back to the dispatcher */
return;
}

```

Local Memory =====

If you need local memory in an interface function, you must not use static or global variables in any case! Always remember: Your interface code has to be reentrant, any interface function may be called many times at once -- not only if more than one port is open! Also local static variables do not fit into object oriented design (and would cause big troubles in Oberon for example).

There's a simple trick for local memory: Just extend the transfer structure, add elements for your local variables locally at the end. Example:

```

void rx_rx( struct RexxHost *host, struct rxd_rx **rxd,
            long action, struct RexxMsg *rexxmsg )
{
    struct {
        /* first the original structure */
        struct rxd_rx rd;
        /* followed by our local extension(s) */
        long tempval;
        char *tempbuffer;
        ...
    } *rd = (void *) *rxd;
    ...
}

```

Or in Oberon:

```

PROCEDURE Rx * (host: rxh.RexxHost; VAR rxd: rxh.RXDPtr; action: INTEGER);

```

```

VAR
  rd: POINTER TO RECORD (rxdRx) (* Extension of the original structure *)
  tempval: LONGINT;
  tempbuffer: BT.DynString;
  ...
END;
...

```

As you can see, we made a local extension of the rxd_rx structure, giving us 'local' variables (tempval and tempbuffer). As the transfer structure including these extensions will be allocated during the first phase and freed in the last phase, these variables can be used like local memory without any conflict.

Examples

=====

There are some example commands (their interface functions) in the directory named 'rxif'. Some of them are ready-to-use, e.g. the HELP command only needs an application specific graphical user interface.

If you have written a commonly usable (i.e. application independant) command and would like to give it to the whole community of ARexxBox users, please send it to me (command definition, functionality and source), I will include it in the next release.

I would like to see a library of standard commands arise, which can easily be included into any project. Such a library of ARexx commands would make ARexx interface development even easier.

1.24 ARexxBox_E.guide/Arguments&Results

Arguments & Results

Arguments

=====

The ARexxBox concept has been developed from the ideas and suggestions of the Style Guide.

One of the essential basics of this concept is to have the arguments of ARexx commands looking and feeling just like normal shell command arguments. So the Box uses the DOS function ReadArgs() to parse the parameters of a concrete command.

First of all, you should now read the chapter on ReadArgs.

Okay. Now to the Box's specialities:

If a command has at least one result field, the Box will automatically generate the two (invisible) arguments VAR and STEM.

These give the user of the final application the option to choose the

result method. There are two methods, the first (RESULT/VAR) returns all assigned result fields concatenated and separated by spaces in one variable, the second (STEM) uses an ARexx stem variable to separate the result fields.

If neither VAR nor STEM were specified by the user, method one (concatenated) will be used to return all results in the standard ARexx variable 'RESULT'.

If the user wants to have structured results, she should choose the STEM method.

VAR and STEM are not mutually exclusive, the user may use both methods at the same time. There will be no 'RESULT' variable assignment if at least one of these options is in use.

Results
=====

In order to offer a consistent interface in both directions, the ARexxBox code implements *exactly the same* syntax and method which is used for arguments also for the results. There's one restriction on the type tags allowed for results: Only the optional type switches /N and /M are supported up to now. Switches like /K do not make any sense for results, and instead of boolean results (/S or /T) you better use integer fields, which are more flexible, as an integer only appears in the output if set.

Here are the types for results:

- * 'FOOBAR' is a string
- * 'FOOBAR/M' is an array of strings
- * 'FOOBAR/N' is an integer (or bool)
- * 'FOOBAR/N/M' is an array of integers

As you can see, the results (which *your* interface function will produce!) may be of the same types as the arguments. Of course, all result fields are optional as well, if you don't assign any value to the foobar result above, it will not show up in the result of the command.

An example: Start the Box, open the file 'arb/Other.arb' and select the 'HELP' command. As you can see, the HELP command has two result fields, one being a simple string ('COMMANDDESC') and one being an array of strings ('COMMANDLIST').

This means, the ARexxBox will generate the following result fields in the transfer structure for this command, 'struct rxd_help':

In C:

```
char *commanddesc;  
char **commandlist;
```

In Oberon (the types have been resolved for this example):

```
commanddesc: BasicTypes.DynString;
commandlist: POINTER TO ARRAY OF BasicTypes.DynString;
```

Your HELP command may set any of these two fields by assigning values to them, e.g.

```
static char *myarray[4] = { "foo", "bar", "dubidu", NULL };
rxd->res.commanddesc = "Blafasel";
rxd->res.commandlist = myarray;
```

In Oberon you also have to take care of the string length, but the lists also don't need a closing NIL:

```
rxd.res.commanddesc := MoreStrings.CopyString("Blafasel");
NEW(rxd.res.commandlist,3); (* three entries *)
rxd.res.commandlist[1] := MoreStrings.CopyString("foo");
rxd.res.commandlist[2] := MoreStrings.CopyString("bar");
rxd.res.commandlist[3] := MoreStrings.CopyString("dubidu");
```

(Remark: The normal HELP command will set only one of the two results (depending on the parameters), but it could assign both if necessary.)

The ARB does all the conversions and assignments which are necessary to have the result method beeing Style Guide conform: Integers will be converted to strings, array entries will be counted and their number will be put in front of them.

You can check it out easily: Start the test program ('test/test') and enter 'help' in the command shell. Without any parameter, the command will produce the list of commands understood by the test program's ARexx interface, doing so by only assigning the field 'rxd->res.commandlist'.

The output contains the number of commands (list entries that is) followed by the commands (entries). This is the result method for single, unstructured variables, what you see is exactly what would have been assigned to the RESULT variable if the command would have been triggered from ARexx instead of the command shell. This output may be redirected into any variable using the keyword 'VAR'.

Now enter 'help stem kl.'. Examine the output, this is exactly what would have been used as an assignment list for ARexx. For this structured result method, arrays get a new first element called 'resultname.COUNT' followed by the entries of the array numbered from 0 upto COUNT-1.

The example above would produce the following result:

```
Using VAR method:
"BlaFasel 3 foo bar dubidu"
```

Using STEM method with root name hr (recommended method because of the multiple, structured result values):

```
hr.COMMANDDESC = "BlaFasel"
hr.COMMANDLIST.COUNT = 3
```

```
hr.COMMANDLIST.0 = "foo"
hr.COMMANDLIST.1 = "bar"
hr.COMMANDLIST.2 = "dubidu"
```

An ARexx program may traverse these lists using a simple loop:

```
help stem a.
say 'There are' a.commandlist.count 'commands available, that are:'
do i=0 to a.commandlist.count-1
  say a.commandlist.i
end i
```

Do you still have that test program running? Good. Enter 'help stem single. help'. The HELP command will now only assign a value to the result field 'COMMANDDESC' (Feld 'rxd->res.commanddesc'). As you can see, the output reflects also clearly, which result field has been used for the result. This only applies to the STEM method.

1.25 ARexxBox_E.guide/Errors

Error Codes

If a command cannot execute normally, it should return an error code reflecting this condition so the caller can react on the problem.

For this purpose ARexx normally only offers the 'RC' variable, which may be set to an integer error code (normal range 0..20).

The ARexxBox offers a slightly extended method with an automatically generated variable called RC2. This variable can hold error codes as well as strings (descriptions), which may be read by the calling ARexx program after command execution.

So we have two error variables, rc and rc2, in every transfer structure. The sign of rc serves as the type tag for rc2, if rc is positive, rc2 is an integer value, if rc is negative, rc2 is a string pointer.

Here's an example for an additional integer error code (you should use DOS error codes where possible):

```
rd->rc = 10;
rd->rc2 = ERROR_NO_FREE_STORE;
```

And here's one for an additional error description:

```
rd->rc = -10;
rd->rc2 = (long) "Prompt option not yet implemented";
```

The last one again for Oberon:

```
rd.rc = -10;
rd.rc2 = SYSTEM.ADR("Prompt option not yet implemented");
```


An integer value will of course be converted to a string, strings will be passed to ARexx without change. A negative rc will be converted to positive.

This extended error return method is not a Style Guide suggestion, but I think it's a good and usable extension, as you can give the user detailed error strings in addition to the simple RC value.

1.26 ARexxBox_E.guide/How2use

How to use the generated Code

Design questions

=====

It's a good idea to build applications in an object oriented manner beginning with the core functions inside and building layers of interface functions around the core. On an ideal application core one should be able to build interface functions for any user interface - be it graphical or textual or...

With such a design, you don't have much problems implementing the interface layer for the graphical user interface. And the layer for ARexx consists of the interface functions pregenerated by the ARexxBox and later completed by you. Another interface layer could for example be made for terminals at the serial port.

A core of basic application functions greatly simplifies the implementation of the ARexx layer.

A simple example: Imagine an application opening files. This opening of a file with a given path and name could be a basic core function, errors could be returned through defined return values to the caller (that is an interface function).

The GUI layer would then contain a function which gets called by the window dispatcher if the user chooses submenu "Open" from the projects menu. This function would then open a file requester, read the name and then call the core function to actually load the file. Any error codes would be given to the user via suitable requesters.

The corresponding function of the ARexx interface layer needn't do as much, as it normally will receive the name of the file to open as a parameter. All it has to do is do a parameter check and then call the same core function. Any problems would be returned to ARexx through the error result fields.

Example

=====

This example is not an example for the ideal layer design mentioned in the chapter before, it's merely an example on how to use the basic

functions generated by the ARexxBox.

You should now load the file 'test/test.c' into your text editor and examine the implementation of the following steps.

Initializing

There's only one library you have to open for the ARexxBox generated code, the 'rexsyslib.library'. Of course you should offer an option to the user for changing the base name of the ARexx port.

If you intend to make use of the result hook mechanism (remember: to handle asynchronous results from commands you sent off to ARexx), you should assign the address of your result hook function to the global variable 'ARexxResultHook'. For Oberon you would redefine the abstract method 'HandleResult' in the proper module of your project.

Now, to open the ARexx port and do all the necessary internal inits, just call SetupARexxHost().

CommandShell

The command shell needs an input and an output channel. The example just opens a normal console window, but you could also use files for that purpose (e.g. to execute scripts).

The command shell runs synchronously, the call will not return until the input channel is empty (EOF or Ctrl-\ from the console).

Of course the caller has to close the I/O channels.

SendRexxCommand

As you can see, sending commands to ARexx is very simple.

If you want to differ between the results your result hook will get, you have to take care of the result recognition yourself. Remember: The results will come in an unpredictable order, depending on what the commands did and what else was running.

One possible method is to store the addresses of all ARexx messages you send off and later compare them to the addresses of the result messages, as ARexx will (currently) use the same message object for results. But that may change anytime, so I'd recommend this method: Use one of the free ARG fields of an ARexxMessage to store any kind of message ID there. For this method you will have to do the two steps of SendRexxCommand() yourself by first calling CreateRexxCommand(), then assigning the ID and then calling CommandToRexx().

ARexxDispatch

Now you do what any Amiga application does - wait for signals. If the ARexx port triggers a signal, you simply call ARexxDispatch(). The

ARexx dispatcher will process all messages on the port and then return.

Of course, the same structure (signal -> dispatcher) can be used for windows (the whole GUI layer).

CloseDown

The example program does an indirect closedown, the CloseDownARexxHost() call is in the function closedown() which was hooked onto the exit procedure before. The return of the CloseDown routine may be delayed if there are any outstanding replies. Take care: Also the routine pointed to by ARexxResultHook may be called in that situation!

1.27 ARexxBox_E.guide/Oberon-2

Hints concerning the Oberon-2 source

These are some remarks to the concept, implementation and some design decisions of the Oberon-2 source by hartmut Goebel.

Concept
=====

One design goal of this implementation was to give the programmer full type safety while retaining the comfortable aspects of Oberon (e.g. the garbage collector). This was a challenge to me, as the parameters of the result structure depend on the result template and cannot be checked at compile time(1).

Another goal was to build upon the PortHandle class library, that was under development right then -- some ideas from ARB have been used there as well.

As a conclusion from this, the source has been designed in an object oriented manner. This was no problem, as the original C design is also object oriented. Merely the interface functions have not been designed to be methods, as they couldn't be called in a generic way otherwise.

Results
=====

While the parameters will be parsed by Dos.ReadArgs() which will put their values into the special transfer structure (see also 'ReadArgs'), this job has to be done by ARBRexxHost for the results. This offers the chance to at least have the full Oberon type safety and comfort with the results.

The following type equivalence applies to the results (type descriptors have been resolved):

```
* 'FOOBAR' -> 'BasicTypes.DynString;'
```

```
* 'FOOBAR/M' -> 'POINTER TO ARRAY OF BasicTypes.DynString;'
* 'FOOBAR/N' -> 'POINTER TO ARRAY 1 OF LONGINT;'
* 'FOOBAR/N/M' -> 'POINTER TO ARRAY OF LONGINT;'
```

As you can see, /M results are dynamic arrays. You only need to allocate them with the needed number of elements and enter the data, for example:

```
NEW(rx.res.foobar,10);
for 10 list entries.
```

You don't need the final NIL (like with C) with Oberon. This is also the reason for /M/N results being directly an 'ARRAY OF LONGINT' instead of an 'ARRAY OF POINTER TO LONGINT'. This is a little inconsistent for the interface, but much more comfortable.

Parameters and simple numerical results have been declared as 'POINTER TO ARRAY 1 OF CHAR', as Oberon does not support pointers to unstructured types (although AmigaOberon does).

Documentation
=====

As the design and language is completely different from C, the C documentation is not very good for the Oberon side. This applies especially to the procedures defined by the PortHandle class library and its extension ARBRexxHost.

I therefore recommend using the documentation to this library. You should find all needed information there. If not, please tell me what misses and I will complete it.

hartmut Goebel

----- Footnotes -----

(1) I think the generic procedures in ARBRexxHost (especially ARBRexxHost.CreatStem) do an elegant job on this problem. I recommend you have a look at them.

1.28 ARexxBox_E.guide/ReadArgs

ReadArgs

ReadArgs() uses ascii templates to know the number and types of the arguments. The destination memory is just an array of longwords, which have to be interpreted by the program corresponding to their types.

Example: A template "DATEI,ARG1,FORCE" means there are three possible arguments. So the destination array must be able to hold at least three longwords.

Template syntax

=====

ReadArgs() allows five basic data types identified by special appended switches in the template string:

- String: Default, no switch
- Integer: '/N'
- Boolean: '/S' for switch or '/T' for toggle
- List: Additionally appended '/M' -- lists of strings and integers are supported

It's essential to remember, that every argument (besides those with '/A' for 'Always required') is optional and doesn't need to have any assignment.

Example: 'FILE/A,LINES/N,STDIN/S' means, three arguments are possible, but only the first one must be specified by the user. The first one is a string, the second an integer and the third is a boolean.

It is possible to force the user to put the keyword before an argument by adding the '/K' switch, this may be necessary for complex templates. Also you can define aliases for a keyword (e.g. abbreviations).

Example: 'FILES/M/A,AS=TO/K/A' is the template for the AmigaDOS join command. It forces the user to put either 'AS' or 'TO' right before the destination file. Valid argument lines for this template would be:

- * 'file1 file2 as ram:dest'
- * 'foo bar test to blafasel'
- * 'as=ram:test source1 source2 source3 source4'

The last example demonstrates that ReadArgs() doesn't need a fixed order of the parameters if the user specifies the keywords. And 'KEYWORD DATA' is equivalent to 'KEYWORD=DATA'.

Forcing the user to use the keywords is a must be in many situations with multiple arguments, as ReadArgs() would otherwise have problems to differ between those arguments for the array and those for the other arguments.

Results

=====

Boolean arguments can be specified or not in an actual argument line, so their state is represented by a plain longword.

All other types are completely optional, so they are represented not by objects but by pointers to objects. If a pointer is NULL, no value was specified for the related argument. Strings are pointers to

characters, integers are pointers to longwords. Lists are NULL--terminated arrays of pointers to either strings or longwords.

The following type equivalence results for C:

```
* 'String' => 'char *'
* 'Array of strings' => 'char **'
* 'Integer' => 'long *'
* 'Array of integers' => 'long **'
* 'Boolean' => 'long'
```

The module ARBRexxHost defines these types for Oberon.

Another example: The template 'FILES/M/A,AS=TO/K/A,FORCE/S,MAX/N,SOUND/T,LINES/N/M' would have the following data structure for the argument values:

```
struct readargs_result
{
    char **files;
    char *to;
    long force;
    long *max;
    long sound;
    long **lines;
} rda_result;
```

(Remark: This template is not correct, it's merely an example for the type equivalence. Actually a template may contain only **one** /M switch.)

The fields of the structure may be assigned default values before calling ReadArgs(), the system call will only alter those elements whose arguments have been specified by the user. Boolean arguments are different: /S arguments will always be set according to their keyword being specified (if not specified, the value will be zero), but /T arguments will toggle the previously set default value -- in theory, I haven't tested this behaviour yet.

So for strings and integers you can see from the primary pointer in the structure above, if the parameter has been specified at all. Otherwise the pointer is NULL.

Arrays may be of any size, so their end has to be marked by a NULL entry.

More Features

=====

There's another switch, /F, used to assign an argument the whole rest of the command line from that point, no matter what keywords or special characters may be in it. You normally don't need this one and should take care if you use it.

Parameters which are not set in between quotation marks by the user will first be checked against the possible keywords. So an input of "foo bar all qwe" to a template "Dir/M,All/S" will set the switch ALL and return the array "foo", "bar", "qwe" in DIR. If the user had specified "all" in quotation marks, it would have been taken into the array instead of being considered as being the ALL switch.

As mentioned above, you should not have more than one array (/M) per template to avoid problems.

If there are unassigned /A arguments left after parsing, ReadArgs() will first try to fill them with the last elements of any /M type argument. An example for this behaviour is the copy command: The template "From/A/M,To/A" with an input of "copy file1 file2 file3 destination" will assign the last word to the TO argument -- just like you'd expect it to do.

On ReadArgs please also see 'AutoDocs:dos.doc'.

1.29 ARexxBox_E.guide/Library

Library

TABLE OF CONTENTS

ARexxBox-ARexxDispatch
ARexxBox-CloseDownARexxHost
ARexxBox-CommandShell
ARexxBox-CommandToRexx
ARexxBox-CreateRexxCommand
ARexxBox-DoShellCommand
ARexxBox-ExpandRXCommand
ARexxBox-FindRXCommand
ARexxBox-FreeRexxCommand
ARexxBox-ReplyRexxCommand
ARexxBox-SendRexxCommand
ARexxBox-SetupARexxHost
ARexxBox-StrDup

1.30 ARexxBox_E.guide/ARexxBox-ARexxDispatch

ARexxBox/ARexxDispatch
=====

ARexxBox/ARexxDispatch

ARexxBox/ARexxDispatch

NAME

ARexxDispatch -- get ARexx command from MsgPort and execute it

SYNOPSIS

```
ARexxDispatch( rexxhost );
```

```
void ARexxDispatch( struct REXXHost * );
```

FUNCTION

ARexxDispatch fetches and executes all queued commands from the given REXXHost's message port.

If a reply for some previously (with SendREXXCommand()) sent command comes in, the counter variable for still outstanding replies will be decreased by one and the REXXMsg and it's associated memory will be freed by FreeREXXCommand().

In the main program, you should just check for the signal of the host's message port and call ARexxDispatch() without actually getting the message. All work will be done by the dispatcher.

INPUTS

rexxhost - pointer to an active REXXHost structure with a valid MsgPort

RESULTS

SEE ALSO

SendREXXCommand(), SetupARexxHost(), DoShellCommand()

1.31 ARexxBox_E.guide/ARexxBox-CloseDownARexxHost

ARexxBox/CloseDownARexxHost

=====

ARexxBox/CloseDownARexxHost

ARexxBox/CloseDownARexxHost

NAME

CloseDownARexxHost -- close & free ARexx host

SYNOPSIS

```
CloseDownARexxHost( rexxhost );
```

```
void CloseDownARexxHost( struct REXXHost * );
```

FUNCTION

CloseDownARexxHost() waits until replies for all pending ARexx commands have been received and then closes the ARexx port and frees all memory associated with that REXXHost structure.

All messages sent to a closing host will be replied immediately with an error "Host closing down".

INPUTS

rexhost - the RexxHost to close down

RESULTS

SEE ALSO

SetupARexxHost(), SendRexxCommand()

1.32 ARexxBox_E.guide/ARexxBox-CommandShell

ARexxBox/CommandShell

=====

ARexxBox/CommandShell

ARexxBox/CommandShell

NAME

CommandShell -- process Commands from a file

SYNOPSIS

```
CommandShell( rexhost, fhin, fhout, prompt );
```

```
void CommandShell( struct RexxHost *, BPTR, BPTR, char * );
```

FUNCTION

CommandShell() sets the Flag ARB_HF_CMDSHELL in the RexxHost's flag field and then processes input from fhin until EOF or the CmdShell flag in the RexxHost being cleared (e.g. by the standard Rexx command "CMDSHELL CLOSE").

The input is read line-wise, with newline as EOL. Each line will be parsed and executed just like a built-in custom ARexx command, exactly like it was called via an ARexx host messageport.

The parsing and execution of each line is done by the function DoShellCommand().

If fhout is not NULL, the output of the commands will be printed to fhout. The output of the commands will NOT be assigned to any variables, as there is no underlying ARexx script program that could hold these variables. Instead, the output will be formatted to be human-readable.

The prompt string (if not NULL) will be printed to fhout as an input request before reading an input line.

New (ARB 0.99d): The rexhost parameter has to point to a valid RexxHost structure. This is for identifying which command shell belongs to which window/instance of the main process.

New(ARB 0.99e): To support the "RX" command sending asynchronous messages to ARexx, this function now catches the replies of those messages and frees them using

FreeRexxCommand(). Messages sent to this host will be replied immediately with an error "CommandShell Port".

INPUTS

rexhost - an initialized RexxHost structure
 fhin - the input FileHandle (see dos.library/Open())
 fhout - the output FileHandle (or NULL)
 prompt - the prompt string (or NULL)

RESULTS

SEE ALSO

DoShellCommand(), ARexxDispatch(), dos.library/Open()

1.33 ARexxBox_E.guide/ARexxBox-CommandToRexx

ARexxBox/CommandToRexx

=====

ARexxBox/CommandToRexx

ARexxBox/CommandToRexx

NAME

CommandToRexx -- send a prepared RexxMsg to the ARexx server

SYNOPSIS

```
msg = CommandToRexx( rexhost, rexxmessage );
```

```
struct RexxMsg *CommandToRexx( struct RexxHost *, struct RexxMsg * );
```

FUNCTION

CommandToRexx just sends the given RexxMsg to the ARexx server process without changing any fields of the Msg. It will also increment the counter for outstanding replies in the RexxHost structure.

You can use this function together with CreateRexxCommand() to easily create customizable command messages for Rexx.

INPUTS

rexhost - an initialized RexxHost structure
 rexxmessage - an initialized ARexx message

RESULTS

msg - the same as rexxmessage, just for easy further processing

SEE ALSO

CreateRexxCommand(), SendRexxCommand()

1.34 ARexxBox_E.guide/ARexxBox-CreateRexxCommand

ARexxBox/CreateRexxCommand

=====

ARexxBox/CreateRexxCommand

ARexxBox/CreateRexxCommand

NAME

CreateRexxCommand -- allocate & initialize rexxmsg for a command

SYNOPSIS

rexxmsg = CreateRexxCommand(rexxhost, command, fh);

struct RexxMsg *CreateRexxCommand(struct RexxHost *, char *, BPTR);

FUNCTION

This function will create a RexxMsg structure for the given RexxHost, create an Argstring from the command string and use that string to initialize the message as a RXCOMM type with RXFF_RESULT requested.

The file handle will be used for both input and output.

You can use this function to create a standard ARexx command with the additional possibility to set some extra parameters before sending the command to ARexx using CommandToRexx().

INPUTS

rexxhost - an initialized RexxHost structure

command - the command string

fh - the input/output FileHandle (see dos.library/Open())

RESULTS

rexxmsg - a pointer to the new RexxMsg structure

SEE ALSO

CommandToRexx(), SendRexxCommand(), dos.library/Open()

1.35 ARexxBox_E.guide/ARexxBox-DoShellCommand

ARexxBox/DoShellCommand

=====

ARexxBox/DoShellCommand

ARexxBox/DoShellCommand

NAME

DoShellCommand -- parse & execute a command line

SYNOPSIS

DoShellCommand(rexxhost, commandline, fhout);

void DoShellCommand(struct RexxHost *, char *, BPTR);

FUNCTION

DoShellCommand parses the given string assuming it contains

an ARexx-style command line.

New (ARB 0.99e): If normal parsing fails, the external function `ExpandRXCommand()` will be called to expand any macros. If the expansion fails or the expanded command can't be recognized either, an error will be returned.

If no errors occur during parsing, it tries to execute the command with the given arguments. The results of the command's execution will be printed in a human-readable format to `fhout` if `fhout` is not `NULL`.

If errors occur, `DoShellCommand` prints a string describing the error to `fhout` (if not `NULL`).

New (ARB 0.99d): The `rexxhost` parameter has to point to a valid `RexxHost` structure. This is for identifying which command shell belongs to which window/instance of the main process.

INPUTS

`rexxhost` - an initialized `RexxHost` structure
`commandline` - the string to be parsed & executed
`fhout` - the output `FileHandle` (or `NULL`)

RESULTS

none

SEE ALSO

`CommandShell()`, `ExpandRXCommand()`, `<dos/dos.h>`

1.36 ARexxBox_E.guide/ARexxBox-ExpandRXCommand

ARexxBox/ExpandRXCommand

=====

ARexxBox/ExpandRXCommand

ARexxBox/ExpandRXCommand

NAME

`ExpandRXCommand` -- expand macros and/or aliases (V0.99e)

SYNOPSIS

`newcommand = ExpandRXCommand(rexxhost, oldcommand)`

`char *ExpandRXCommand(struct RexxHost *, char *);`

FUNCTION

This is an 'external' function you should provide if you want to have command aliases or the like. The minimal version of this function is just a `return(NULL)` as generated in the `rxif` module.

`ExpandRXCommand()` will be called by the parser if it doesn't know how to interpret a command string. Expansion could now

for example be a look up in the host's macro table.

Any strings returned by this function have to be allocated explicitly using the standard C memory functions. The calling parser will free() them.

INPUTS

rexshost - the RexxHost we are working on
oldcommand - the commandline the parser doesn't know

RESULTS

newcommand - an explicitly allocated memory area containing the expanded command (or NULL)

SEE ALSO

DoShellCommand(), ARexxDispatch()

1.37 ARexxBox_E.guide/ARexxBox-FindRXCommand

ARexxBox/FindRXCommand

=====

ARexxBox/FindRXCommand

ARexxBox/FindRXCommand

NAME

FindRXCommand -- search the ARexxBox command table (V0.99e)

SYNOPSIS

```
rxscmd = FindRXCommand( command )
```

```
struct rxs_command *FindRXCommand( char * );
```

FUNCTION

This function returns a pointer to the given command's entry in the ARexxBox-generated command table. It exists to support those functions working on/with commands, like HELP or ENABLE/DISABLE.

This function does no macro expansion. The comparisons are case independant so you don't have to convert your input to upper case beforehand. As this is exactly the routine used by the parser to find a command, it will handle abbreviations.

INPUTS

command - the command name to search for

RESULTS

rxscmd - the rxs_command structure of that command
(or NULL if command not found)

SEE ALSO

1.38 ARexxBox_E.guide/ARexxBox-FreeRexxCommand

ARexxBox/FreeRexxCommand

=====

ARexxBox/FreeRexxCommand

ARexxBox/FreeRexxCommand

NAME

FreeRexxCommand -- free the associated memory of a RexxMsg

SYNOPSIS

```
FreeRexxCommand( rexxmessage );
```

```
void FreeRexxCommand( struct RexxMsg * );
```

FUNCTION

This is basically a PD ARexx routine provided by William S. Hawes.

It frees all memory associated with a particular (previously sent) ARexx message structure. It will also close any stdin/stdout channels associated to that Rexx message.

You normally shouldn't have to bother with this one because the dispatcher will call it for you.

INPUTS

rexxmsg - the rexx message to free

RESULTS

SEE ALSO

SendRexxCommand()

1.39 ARexxBox_E.guide/ARexxBox-ReplyRexxCommand

ARexxBox/ReplyRexxCommand

=====

ARexxBox/ReplyRexxCommand

ARexxBox/ReplyRexxCommand

NAME

ReplyRexxCommand -- reply a rexx message from rexxmast

SYNOPSIS

```
ReplyRexxCommand( rexxmsg, primary, secondary, result );
```

```
void ReplyRexxCommand( struct RexxMsg *, long, long, char * );
```

FUNCTION

This is a PD ARexx routine provided by William S. Hawes.

It replies a given rexx message to the rexx master process, filling in a primary and a secondary return code plus

optionally a supplied result string.

The result string will only be converted to an ARexx string, if the primary return code equals 0, and will then destroy the contents of the secondary return code. So you provide either primary and secondary return codes or a result string.

You normally shouldn't have to call this function! It is only mentioned here, because it is not part of the ARexxBox routines, but part of the original ARexx distribution by William S. Hawes.

New (ARB V0.99d): Now creates an ARexx variable "RC2" for the secondary return code. If primary is positive, secondary is interpreted as a long, if primary is negative, secondary is interpreted as a char *. RC will become positive in any case.

RC2 will only be assigned if the ARexx RESULT flag is set.

INPUTS

rexmsg - the message structure to reply
 primary - the primary return code (rc) (>0 <0)
 secondary - the secondary return code (rc2) (long or char *)
 result - the result string

RESULTS

SEE ALSO

SendRexxCommand(), FreeRexxCommand()

1.40 ARexxBox_E.guide/ARexxBox-SendRexxCommand

ARexxBox/SendRexxCommand

=====

ARexxBox/SendRexxCommand

ARexxBox/SendRexxCommand

NAME

SendRexxCommand -- invoke rexx command script

SYNOPSIS

```
rexmsg = SendRexxCommand( rexxhost, command, filehandle )
```

```
struct RexxMsg *SendRexxCommand( struct RexxHost *, char *, BPTR );
```

FUNCTION

This is basically a PD ARexx routine provided by William S. Hawes.

This function sends the given command string to the ARexx master process for execution as an ARexx command. The command string contains the file name of the ARexx script to be started. If the filehandle is not NULL, it will be used as stdin and stdout for the Rexx script. If it is

NULL, the REXX program will use stdin/stdout of the calling process.

If necessary, the default extension (defined in the generated header file under the name REXX_EXTENSION) will be added to the file name.

Messages sent using this function will be replied to by the ARexx master process as soon as the execution of the command script stops. The application MUST NOT close it's messageport before all replies have been received! To simplify things, ARexxBox does this book-keeping for you. CloseDownARexxHost() will wait for all missing replies to arrive before closing down the messageport.

The dispatcher will automatically detect any replies, count them and do a FreeRexxCommand() for each reply, so you don't have to bother with this either.

Internally, this function is implemented using the two more atomic functions CreateRexxCommand() and CommandToRexx().

INPUTS

rexhost - the RexxHost to be used to send the command
 command - the file name of the ARexx script
 filehandle - Filehandle for stdin/stdout or NULL

RESULTS

rexmsg - the sent rexx message structure (for comparisons)

SEE ALSO

FreeRexxCommand(), CloseDownARexxHost(), ARexxDispatch(),
 CreateRexxCommand(), CommandToRexx()

1.41 ARexxBox_E.guide/ARexxBox-SetupARexxHost

ARexxBox/SetupARexxHost

=====

ARexxBox/SetupARexxHost

ARexxBox/SetupARexxHost

NAME

SetupARexxHost -- initialize and open an ARexx host

SYNOPSIS

```
rexhost = SetupARexxHost( basename );
```

```
struct RexxHost *SetupARexxHost( char * );
```

FUNCTION

This function allocates and initializes a RexxHost structure. It opens a public message port under the given basename. If no basename (NULL) was specified,

the default basename as entered in the ARexxBox window will be used instead.

Anyway, if a public port of that name already exists, SetupARexxHost() will start adding numbers to the name until a unique name is found. So if for example the basename is "myhost" and there is already a port of that name in the system, the name will be changed to "myhost.1" (then to "myhost.2" and so on).

The actual name will be copied to the portname field of the RexxHost structure. It is a good idea to tell the user about the actual port name of the new host.

INPUTS

basename - the messageport basename or NULL

RESULTS

rexhost - the initialized RexxHost structure, ready to go. Pass this pointer to CloseDownARexxHost(), ARexxDispatch() and SendRexxCommand().

SEE ALSO

CloseDownARexxHost(), ARexxDispatch(), SendRexxCommand()

1.42 ARexxBox_E.guide/ARexxBox-StrDup

ARexxBox/StrDup

=====

ARexxBox/StrDup

ARexxBox/StrDup

NAME

StrDup -- duplicate a string

SYNOPSIS

```
copy = StrDup( origin );
```

```
char *StrDup( char * );
```

FUNCTION

This routine will do exactly the same as its standard C lib counterpart strdup(), but use the exec function AllocVec() for allocating the needed memory.

(So YOU are responsible for freeing the copy!)

INPUTS

origin - the original string

RESULTS

copy - the copy of the original string

SEE ALSO

```
strdup(), exec.library/AllocVec(), exec.library/FreeVec()
```

1.43 ARexxBox_E.guide/FileFormat

File Format

The ARB file format is an extendable ASCII format (line separator is LF). Besides the obvious data (syntax of the commands, their arguments and results) an ARB file contains common and command specific tags.

The first line of an ARB file contains the ID string 'ARB'. Beginning with line two comes the first, global tag block. Each tag line begins with a dollar followed without a space by the tag ID (type: long). The rest of the line contains alphanumeric parameters, separated by spaces.

Up to now there's only one global tag ID:

* ID 1, two parameters:

1. File-ID (unsigned long): This is the ID used by ARB to tell if an existing source belongs to the project.
2. Next-Cmd-ID (unsigned long): The ID for the next new command (which is **not** the number of commands!).

The first line after this tag block contains the MsgPort base name. The next line contains the status of the CommandShell gadget (1 for checked). The following lines up to the end of the file are the command definitions. Each command entry is of the form:

- * Command name
- * Tag block (first char = '\$', followed by the tag ID as above)
- * Arguments (one per line)
- * Separator (line containing just a 'minus')
- * Results (one per line)
- * Another separator line

There are currently two tag IDs for commands:

* ID 1, two parameters:

1. Command-ID (unsigned long): For identifying the command in an existing source.
2. Status (char): 'N' for 'New', 'C' for 'Changed' or 'O' for 'Old'.

* ID 2, one parameter:

1. ExtStatus (char): Like Status, for documentation status of

cmd.

There will possibly be other tag IDs in the future, so please write an intelligent parser for the tag blocks. Unknown tags should be ignored.

1.44 ARexxBox_E.guide/History

History

History of ARexxBox Releases:

V0.99

first beta release

V0.99a

FIXED: Argument and result lists could work on an invalid command -> Enforcer-Hit/Crash.

(Report: RALF_KAISER@AWORLD)

V0.99b

ENHANCED: GadToolsBox source fixed to use the System Default Font instead of the Screenfont for Layout and Gadgets.

(Report: SYSOP@INSIDER [Garry Glendown])

V0.99c

FIXED: The equal sign '=' wasn't allowed for arguments.

FIXED: After changing some argument or result, the display now shows the changed element, rather than the last one in that list.

FIXED: The font routines still had some error.

(Report: F.J.Reichert [F_J_REICHERT@SAARAG])

V0.99d

FIXED: All pointer conversions are now clean. The code should now be compileable without warnings.

Added pragma #includes and _toupper() for SAS/C.

(Report: W_KUETTING@HSP)

V0.99e

FIXED: When _no_ result field was given, the code generated an "out of memory" error.

ENHANCED: ReplyRexxCmd() now creates the variable RC2, it contains the "Secondary Returncode", which can be set in any rxif-structure under the name rc2. This variable can be used to return detailed error codes to the calling rexx program.

This feature is an extension of the style guide conventions, IMHO a good one. If you have complaints or comments on

this idea, please tell me!

RC2 will only be generated if a) 'options results' was given and b) RC != 0. The field rc2 may contain an error code (long) as well as an error string (char *). Code is the default. The software distinguishes these two types by the sign of rc, rc positive means rc2 is a long, rc negative means rc2 is a char *. A negative rc will be converted to positive before being returned to the caller.

Example for error CODE:

```
rd->rc = 10;
rd->rc2 = ERROR_OBJECT_NOT_FOUND;
```

Example for error STRING:

```
rd->rc = -10;
rd->rc2 = (long) "You idiot! No object!";
```

The standard error strings are `_not_` localized. As soon as I get some infos and tools for this, I'll do it.

ENHANCED: `SendRexxCommand()` now has a new parameter for better supporting the standard command "RX". It is a `FileHandle (BPTR!)`, which becomes the `stdin/stdout` of the executed arexx script. `FreeRexxCommand()` closes the file (if necessary).

`SendRexxCmd()` now returns the address of the sent `RexxMsg` structure (or `NULL` for error).

ENHANCED: The `RexxHost` structure now has a new field called "userdata". It can be used to hook the `RexxHost` onto your own structures, e.g. for linking hosts and and their project instances together.

ENHANCED: Every command now has a global "enabled" flag. If the flag is 0, the dispatcher will not execute that command anymore until the flag will be set again. The user should have the ability to manipulate this flag using the (standard) arexx commands `ENABLE` and `DISABLE` (see `rxif/*.c`).

CHANGED: Any `CommandShell` now must have its own `RexxHost`, that means that host `_must_not_` be used as a normal `ARexx` port simultaneously.

ENHANCED: As a consequence, every `rxif` function now gets the `RexxHost` of the `ARexx` port resp. `CmdShell` as it's (first) parameter. Therefore the function can decide whether it was called from `ARexx` or from a `Shell` and on which project it should work.

ENHANCED: Another consequence is the new flag `ARB_HF_CMDSHELL` for supporting the standard command `CMDSHELL`. The flag is in the 'flags' field of the `RexxHost` structure. If it's set, that host is currently running a shell. You can

clear the flag from anywhere causing the shell to close (after processing the next command).

FIXED: ARB no longer accepts argument or result names consisting of only spaces or only options.

CHANGED: As some rxif functions have to access them, the structs 'rxs_command' and 'rxs_commandlist' are now defined in the headerfile. For accessing the command list, there is a new function:

```
struct rxs_command *FindRXCommand( char *name );
```

FIXED: Arguments with '=' are now translated correctly into variable names for C. The last alias component will be used as the var name.

ENHANCED: There is now a callback function for words the parser doesn't recognize:

```
char *ExpandRXCommand( struct RexxHost *host,  
                      char *commandline );
```

This function will be called before the parser returns an error. If the function returns NULL, the error will be generated, if not, the parser assumes that the function was able to analyze the input and tries to parse the returned string.

This function can be used to implement command aliases or processing of non-standard commands.

Attention! ExpandRXCmd() has to allocate memory for the returned string, using standard C functions! This memory will be freed by the parser after processing the string!

ENHANCED: ARB now does more safety checks on saving the source modules, so you should never accidentally destroy other code.

NOTE: One of the standard commands as suggested by the style guide uses the keywords "VAR" and "STEM" to describe the INPUT variables instead of the former usage as OUTPUT descriptors. I don't think this is a good choice as there can't be any results from this function that way. I suggest other names for those arguments, e.g. "FROMVAR" and "FROMSTEM".
Comments?

(The ARB doesn't look at the already existing args, if a command has some result, "VAR" and "STEM" will be added to the template.)

Some of the style guide suggestions are not very precise and need extensions to operate correctly. I added some extensions to the arexx standard commands. Comments welcome.

FIXED: Numeric results now work.

ENHANCED: ARB now uses two FileRequesters, one for the binaries and the other for sources. The patterns now remain what you set them to and have defaults with #? instead of *.

(Report: Stefan Zeiger)

FIXED: Window->TopEdge is now font sensitive.

(Report: Stefan Zeiger)

V0.99g

FIXED: The parser now returns the ReadArgs() error code in rc2.

CHANGED: No longer appends \n to the template string before parsing (not needed).

CHANGED: The ARexxBox now uses the ASL file requester.

(Wish: Garry Glendown [Sysop@Insider])

V1.00

ENHANCED: Source now compatible to GCC. Remaining warnings may be ignored.

CHANGED: FindRXCommand() now only accepts real abbreviations of commands.

--- FIRST PUBLIC RELEASE ---

V1.01

ENHANCED: FindRXCommand() now uses binary search (so it's now complexity $O(\log n)$ instead of $O(n)$)

V1.02

ADDED: Clipboard support

FIXED: CreateVAR() didn't check an allocation

(Report: Rüdiger Dreier)

CHANGED: SendRexxCommand() is now divided into two functions, CreateRexxCommand() and CommandToRexx(). So one can change a created msg if needed before sending it to ARexx.

ENHANCED: Unknown commands will now be sent to ARexx before being treated as "unknown". External ARexx programs will be called in a transparent way with this method.

(Suggestion: Rüdiger Dreier)

ENHANCED: DoRXCommand() and DoShellCommand() polished a bit.

(Report: Hartmut Goebel)

FIXED: The dispatcher now checks incoming msgs.

(Report: Hartmut Goebel)

FIXED: In free_stemlist() was a potential FreeMem bug.

CHANGED: Now using AllocVec/FreeVec instead of (m|c)alloc and free. Needed a new function as well, a replacement for strdup:

```
char *StrDup( char *string )  
(Suggestion: Rüdiger Dreier)
```

FIXED: The CloseDown could cause running scripts on that port to hang.
(Report: Rüdiger Dreier)

V1.03

ENHANCED: New parameter for the Box: FONT=<name>/<size>, specifies the font to use. Default is now again the Screen font (Style Guide).
(Suggestion: Timothy J. Aston)

CHANGED: If an empty port name was specified, the default will be used.
(Suggestion: hartmut "Essich" Goebel)

CHANGED: VAR and STEM parameters will now be converted to upper case, as SetRexxVar() doesn't do this by itself.
(REXX standard)

V1.04

FIXED: Path for project files will be kept when importing files via "Merge".
(Report: Stefan Reisner)

FIXED: ARG0 now casted to (char *) everywhere.
(Report: Marc Schröer)

ENHANCED: The header now has got it's own #define.
(Marc Schröer)

V1.05

ENHANCED: Source generation now intelligent, old interface functions will be recognized through special comments in the source and either be kept or moved to a storage file.
(Suggestion: all.all; ID idea: Christoph Teuber)

CHANGED: File format changed to reflect the new source generation. The Box will read the old format.

ENHANCED: SetupARexxHost() now needs a second parameter: An optionally previously allocated MsgPort. So one can use a special port, e.g. to use one signal for all ARexx ports.
(Suggestion: Stefan Reisner)

V1.10

ENHANCED: The Oberon source generator is there at last. The Oberon source is completely by the Oberon Wizard Extraordinary hartmut "Essich" Goebel. Questions and comments concerning Oberon please go directly to him, as I don't know Oberon.

ENHANCED: Implemented result hook method, if the variable ARexxResultHook contains the address of a function, it will be called for all replies to messages sent to ARexx.

FIXED: FreeRexxCommand() no longer closes stdin and stdout if these were entered.

V1.11

FIXED: Some last changes for Oberon.

----- Release 2 -----

V1.12

FIXED: The HELP command produced Enforcer hits (strlen(0)).
(Report: Klaas Hermanns)

ENHANCED: Introduced tag #2 for commands: External status flag,
"New" state of this flag will only be reset by external programs,
e.g. documentation aids like ARB2TeXinfo by Albert Weinert.
(Suggestion: Albert Weinert)

FIXED: The command list was generated not alphabetically but
numerically sorted (by their IDs).
(Report: Klaas Hermanns)

ENHANCED: Instead of the binary search algorithm, the Box now
generates a real finite state machine for searching of commands
(FindRXCommand). The binary search method had a big design flaw
with abbreviations on the alphabetically sorted list of commands.
(ATT: This has been implemented for C only up to now!)
(Report: Klaas Hermanns)

FIXED: For SAS C, toupper() will be #undefined. Also there is now
the real function cast in the list of commands.
(Report: Klaas Hermanns)

FIXED: The MIN and MAX results of Misc.arb/GETATTR have to be
numbers, of course.
(Report: Klaas Hermanns)

FIXED: "Merge" trashed the command list when merging in existing
commands.

CHANGED: Now declares the external library bases with their
appropriate types (C).
(Wunsch: Klaas Hermanns)

CHANGED: In arb/advanced.arb, commands REQUESTNUMBER and
REQUESTSTRING, I changed the argument called "DEFAULT" to "DEFAULTNUM"
resp. "DEFAULTSTR" to avoid conflicts with the 'default' keyword in C.
(Report: Klaas Hermanns)

ENHANCED: There's now another parameter for all interface functions
called "struct REXXMsg *rexxmsg", which will contain the address of
the message from REXX if called by ARexx. If called from a command
shell, it will contain NULL, so you can also use it to determine the
caller. This is to support GetRexxVar(), which needs this pointer.
(Report: Klaas Hermanns)

FIXED: I now define toupper() as a function for GCC. The macro caused
problems by evaluating the parameter more than once.

FIXED: The command shell parameter parsing had a bug (the ReadArgs
buffer wasn't cleared).

----- Release 3 -----

1.45 ARexxBox_E.guide/Thanks

Thanks

I'd like to thank the following persons:

- * The people behind the Amiga, for the Amiga and AmigaOS Release 2.
- * William S. Hawes, for his excellent port of Rexx.
- * Jan van den Baard, for his GadToolsBox, which inspired me and was a great help in designing the user interface.
- * Nico Francois, for his regtools.library with its really programmer and user friendly requesters.
- * hartmut Goebel, for the Oberon-2 source.
- * All my active beta testers and all who sent me suggestions (see history).
- * Christoph Teuber and Oliver Wagner, for the AWorld Mailbox System.

1.46 ARexxBox_E.guide/Index

Index

Addresses

amiga.lib

amiga.lib

ANSI C

ARB input

ARexxBox/ARexxDispatch

ARexxBox/CloseDownARexxHost

ARexxBox/CommandShell

ARexxBox/CommandToRexx

ARexxBox/CreateRexxCommand

ARexxBox/DoShellCommand

ARexxBox/ExpandRXCommand

ARexxBox/FindRXCommand

ARexxBox/FreeRexxCommand

ARexxBox/ReplyRexxCommand

ARexxBox/SendRexxCommand

ARexxBox/SetupARexxHost

Addresses

Requirements

Requirements

Requirements

Input

ARexxBox-ARexxDispatch

ARexxBox-CloseDownARexxHost

ARexxBox-CommandShell

ARexxBox-CommandToRexx

ARexxBox-CreateRexxCommand

ARexxBox-DoShellCommand

ARexxBox-ExpandRXCommand

ARexxBox-FindRXCommand

ARexxBox-FreeRexxCommand

ARexxBox-ReplyRexxCommand

ARexxBox-SendRexxCommand

ARexxBox-SetupARexxHost

ARexxBox/StrDup	ARexxBox-StrDup
Arguments	Arguments&Results
ATT: Interface changed!	Generate Source
Bug reports	Addresses
CommandShell	CommandShell
Comments	Files
Concept	Concept
Concept: Closedown	Closedown
Concept: Commands to ARexx	Commands to ARexx
Concept: Expanding commands	Expanding commands
Concept: Init	Initializing
Copy	Clipboard
Copyright	Copyright
Cut	Clipboard
Data structure	InterfaceFunctions
Design questions	How2use
Distribution	Copyright
E-Mail	Addresses
Erase	Clipboard
Error Codes	Errors
Example test	How2use
Features	Introduction
File Format	FileFormat
FreeWare	Important
Generate Source	Generate Source
Gifts	Addresses
History	History
How to use the Code	How2use
Important remarks	Important
Interface Functions	InterfaceFunctions
Interface structure	InterfaceFunctions
InterNet Address	Addresses
Konzept: Commands from ARexx	Commands from ARexx
Legal stuff	Copyright
Local Memory	InterfaceFunctions
Mail Address	Addresses
Mark Range	Clipboard
Merge in	Merge in
MsgPort Basename	MsgPort Basename
Oberon-2	Oberon-2
OLink	Requirements
Paste	Clipboard
Print	Print
RC	Errors
RC2	Errors
ReadArgs	Arguments&Results
ReadArgs	ReadArgs
ReadArgs: More Features	ReadArgs
ReadArgs: Results	ReadArgs
ReadArgs: Template syntax	ReadArgs
reqtools.library	Requirements
RESULT	Arguments&Results
Results	Arguments&Results
rexxvars.o	Requirements
RVI	Requirements
rxif examples	InterfaceFunctions
Shortcuts	Input

Software layers	How2use
Source: ARB comments	Files
Source: C	Files
Source: Oberon	Files
static	InterfaceFunctions
STEM	Arguments&Results
Suggestions	Addresses
Thanks	Thanks
Transfer structure	InterfaceFunctions
Types	Arguments&Results
up & do	Input
VAR	Arguments&Results
