9 software AG

webMethods EntireX

EntireX C Wrapper

Version 10.8

October 2022

WEBMETHODS

Table of Contents

1 About this Documentation

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
`Monospace font`	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at **https://documenta-tion.softwareag.com**.

In addition, you can also access the cloud product documentation via **https://www.software-ag.cloud**. Navigate to the desired product and then, depending on your solution, go to "Developer Center", "User Center" or "Documentation".

Product Training

You can find helpful product training material on our Learning Portal at **https://knowledge.soft-wareag.com**.

Tech Community

You can collaborate with Software AG experts on our Tech Community website at **https://tech-community.softwareag.com**. From here you can, for example:

- Browse through our vast knowledge base.

- Ask questions and find answers in our discussion forums.

- Get the latest Software AG news and announcements.

- Explore our communities.

- Go to our public GitHub and Docker repositories at **https://github.com/softwareag** and **https://hub.docker.com/publishers/softwareag** and discover additional Software AG resources.

Product Support

Support for Software AG products is provided to licensed customers via our Empower Portal at **https://empower.softwareag.com**. Many services on this portal require that you have an account. If you do not yet have one, you can request it at **https://empower.softwareag.com/register**. Once you have an account, you can, for example:

- Download products, updates and fixes.

- Search the Knowledge Center for technical information and tips.

- Subscribe to early warnings and critical alerts.

- Open and update support incidents.

- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

I   Introduction to C Wrapper

EntireX C Wrapper provides access to RPC-based components from C applications. It enables you to develop both client and server applications.

Description

The C Wrapper enables access to RPC servers for C client applications and access to C servers for any RPC client. The C Wrapper generation tools of the Designer take as input a Software AG IDL file which describes the interface of the RPC and generates C sources that implement the functions and data types of the interface. If there is a related server mapping file (Natural | COBOL), this is also used (internally).
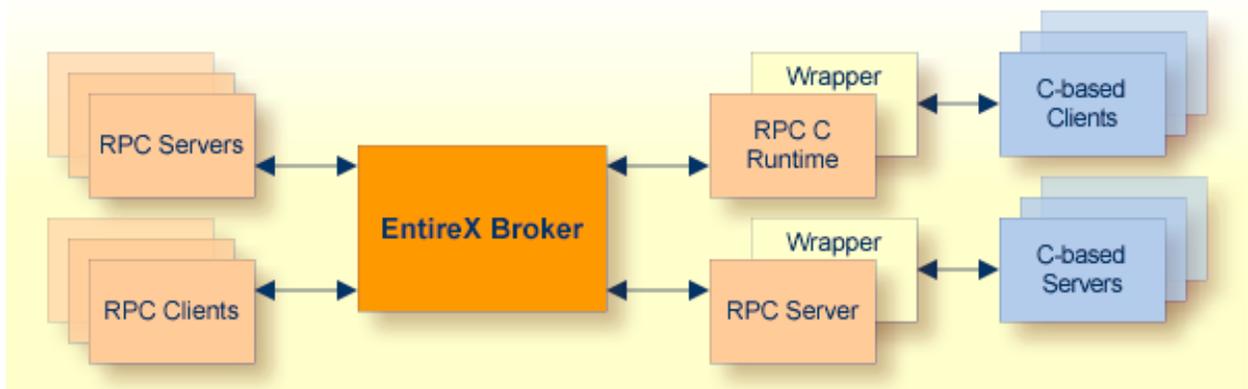
The generated functions can be compiled with the C compiler of your target platform.

The C Wrapper works as follows:

- C code is generated from the IDL file. If there is a related server mapping file (Natural | COBOL), this is also used (internally).

- The generic RPC C runtime implements functionality that is not specific to a given IDL file (e.g. broker logon and logoff, marshalling and unmarshalling of data). The generated C code makes use of the RPC C runtime functionality.
- The Software AG IDL Compiler and an appropriate template are used for the C code generation.

Generic RPC C Runtime

In order to minimize the amount of code generated for a specific IDL, all service-type functionality required by the client interface object or the server interface is implemented in a generic RPC C runtime library. The generic RPC C runtime implements functions, for example:

- marshalling C data types to Software AG IDL data types
- unmarshalling Software AG IDL data types to C data types
- connecting RPC clients to RPC servers via the broker
- etc.

C Client Applications

For a given IDL file, the Software AG IDL Compiler and a C code generation template for clients are used to generate client interface objects and header files. If there is a related server mapping file (Natural | COBOL), this is also used (internally). The source code generated by the C Wrapper can be compiled with your target C compiler. Application developers use the generated interface object(s) and the header file to write C applications that access RPC servers.
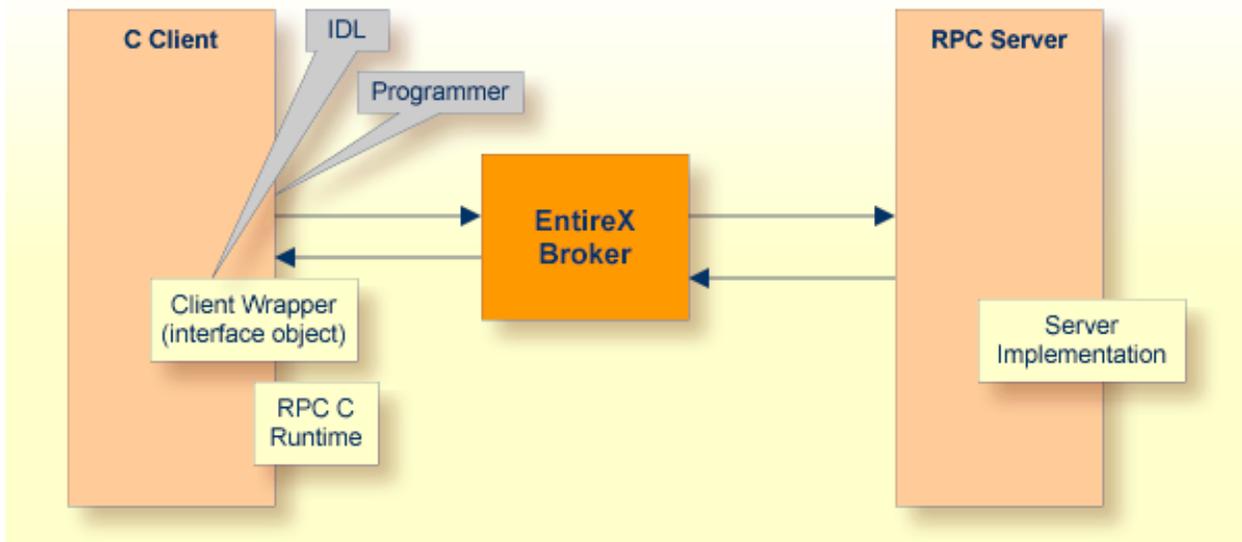
For more information, see *Using the C Wrapper*.

C Server Application

The Software AG IDL Compiler and a C code generation template for servers are used to generate interface object(s) and a server (skeleton) for a specific IDL.

Application developers use the generated server (skeleton) to write their own server code for each program in the IDL. The source code is compiled and linked with your target C compiler and linker into interface object and server libraries. Your server library name needs to match the library name as specified in the IDL file. The interface object library has the same name as your server library prefixed with the letter "D".
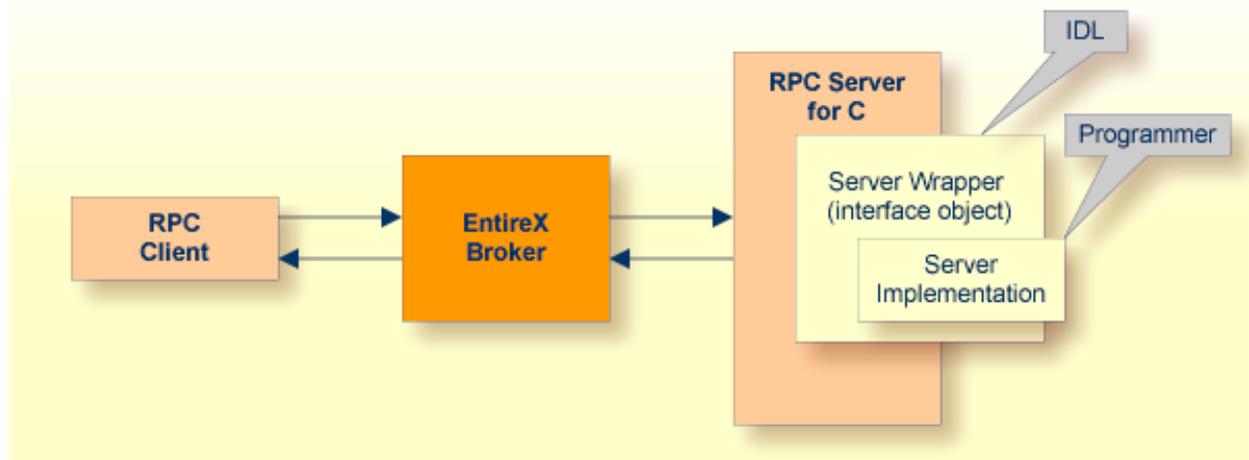
For more information, see *Using the C Wrapper*.

2 Using the C Wrapper

Using the C Wrapper for the Client Side

The C Wrapper provides access to RPC-based components from C applications and enables you to develop both clients and servers. This section introduces the various possibilities for RPC-based client applications written in C.

- Using the C Wrapper in Single-threaded Environments (UNIX, Windows)
- Using the C Wrapper in Multithreaded Environments (UNIX, Windows)

Using the C Wrapper in Single-threaded Environments (UNIX, Windows)

This mode applies to UNIX and Windows.
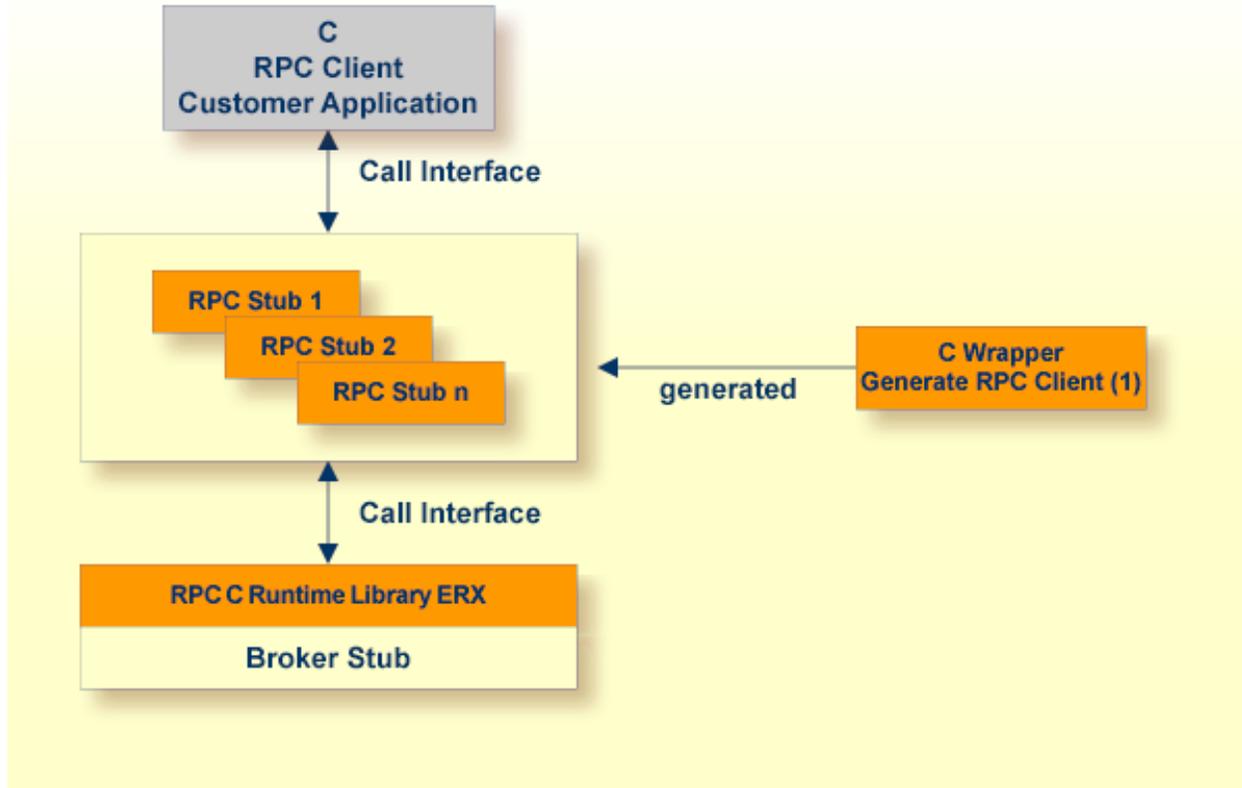
[1] For generation, see *Generate C Source Files from Software AG IDL Files*.

In this scenario, the C RPC client customer application, every generated interface object and the RPC C runtime library (erx) are linked (bound) together to an executable application.

> **To use the C Wrapper in single-threaded environments**

1 Generate the RPC client, see *Generate C Source Files from Software AG IDL Files*

   ▪ and select the *Mapping Options* according to your needs.

   ▪ Do *not* switch on the check box **Multithreaded Client**, see *Generate RPC Client*.

2 If necessary, use FTP to transfer your application and the interface object(s) to the target platform where you write your application.

3 Write your application. See *Writing a Single-threaded C RPC Client Application*.

4 If necessary, transfer your application and the client interface object(s) to the target platform where you compile your application, using FTP.

5 Using a C compiler supported by the C Wrapper and compile

   ▪ the generated client interface object(s)

   ▪ your C RPC client customer application.

   Use the standard C/C++ compiler of your target platform. Note the platform-specific settings.

6 Using the linker (binder), link (bind)

   ▪ the compiled client interface object(s)

   ▪ your C RPC client customer application

   ▪ the broker stub

   ▪ under **Windows**: the RPC C runtime library delivered as a library and DLL named *erx.lib* and *erx.dll*

   ▪ under **UNIX**: the RPC C runtime library delivered as a shared object or shared library named *liberx.so* or *liberx.sl*

   Use the standard C/C++ linker of your target platform. Note the platform-specific settings.

Using the C Wrapper in Multithreaded Environments (UNIX, Windows)

This mode applies to UNIX and Windows.

(1) For generation, see *Generate C Source Files from Software AG IDL Files*.

In this scenario, the C RPC client customer application, every generated client interface object and the RPC C runtime library (erx) are linked (bound) together to an executable application.

≫ **To use the C Wrapper in multithreaded environments**

1   Generate the RPC Client, see *Generate C Source Files from Software AG IDL Files*

■ and select the *Mapping Options* according to your needs

■ and switch on the check box **Multithreaded Client**, see *Generate RPC Client*.

2   If necessary, transfer your application and the client interface object(s) to the target platform where you write your application, using FTP.

3   Write your multithreaded C RPC Client application, see *Programming Multithreaded RPC Clients*.

4   If necessary, transfer your application and the client interface object(s) to the target platform where you compile your application, using FTP.

5   Using a C compiler supported by the C Wrapper, compile:

■ the generated client interface object(s)

- your C RPC client customer application

Use the standard C/C++ compiler of your target platform. Note the platform-specific settings.

6   Using the linker (binder), link (bind)

- the compiled client interface object(s)

- your C RPC client customer application

- the broker stub

- under **Windows**: the RPC C runtime library delivered as a library and DLL named *erx.lib* and *erx.dll*

- under **UNIX**: the RPC C runtime library delivered as a shared object or shared library named *liberx.so* or *liberx.sl*.

Use the standard C/C++ linker of your target platform. Note the platform-specific settings.

Using the C Wrapper for the Server Side (z/OS, UNIX, Windows, BS2000)

The C Wrapper provides access to RPC-based components from C applications and enables you to develop both clients and servers. This section introduces the various possibilities for RPC-based server applications written in C.

This section applies to the operating systems z/OS, UNIX, Windows and BS2000.
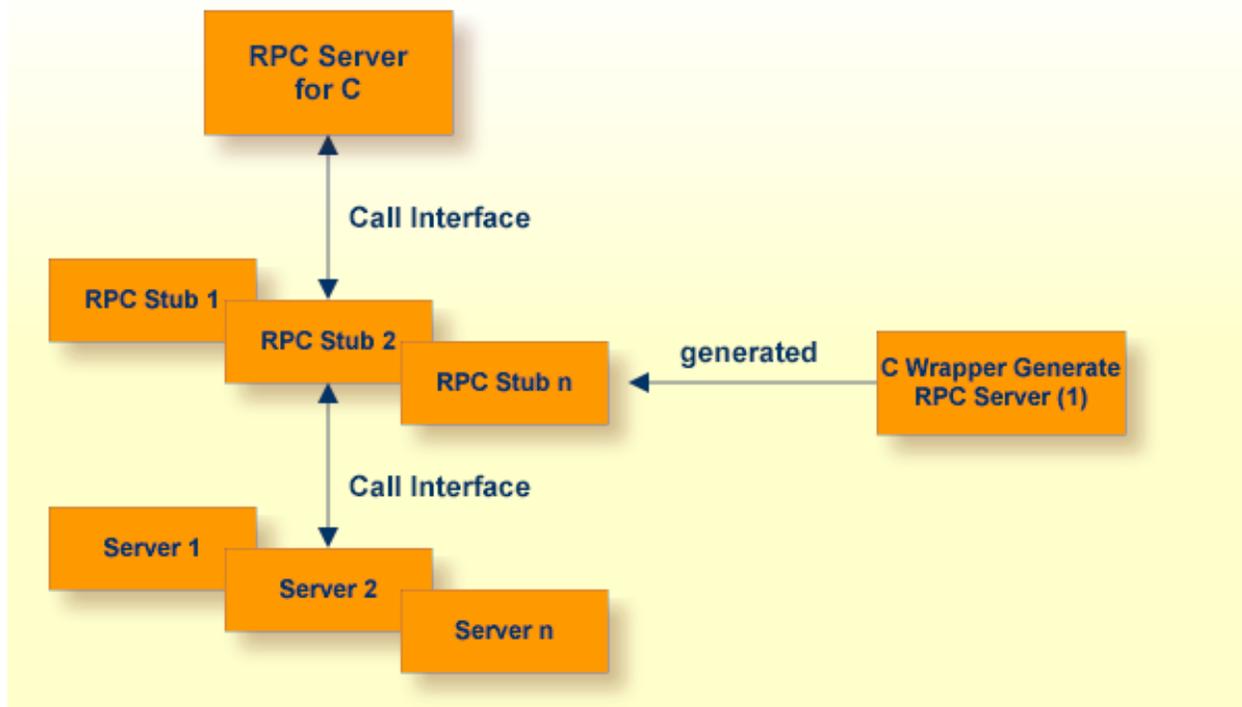
[1] For generation, see *Generate C Source Files from Software AG IDL Files*.

For C, the RPC server works with server interface objects. Your server is called dynamically using standard call interfaces.

≫ To use the C Wrapper

1   Generate the RPC Server, see *Generate C Source Files from Software AG IDL Files*, and

   ■ select the *Mapping Options* according to your needs.

   ■ The interface objects and the server (skeleton) must be generated with the same mapping options, otherwise results will be unpredictable.

> **Note:** For z/OS, the limitation of 8 characters per (physical) member name must be considered when defining the IDL library (see `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation in the Software AG IDL file. The client interface object will be generated with a prefix letter "D". Therefore an IDL library name "EXAMPLE" within the IDL file results in a physical member name "DEXAMPLE". We suggest using an IDL library name of up to 7 characters in length; if the name is longer, you will not be able to transfer (using FTP) the generated objects to the mainframe.

2   If necessary, use FTP to transfer your server (skeleton) and the server interface object(s) to the target platform where you write your application.

3   Use the generated server (skeleton) and complete it by applying your application logic. To prevent loss of implementation code when re-generating, we suggest the following before you add any implementation code to the server (skeleton):

- rename the server from *F\<library\>.c* to *\<library\>.c* (or to any other suitable name)
- or move the server *F\<library\>.c* to a different directory (folder)

4   If necessary, transfer your server and the server interface object(s) to the target platform where you compile your application, using FTP. The objects to be transferred depend on the platform:

- **z/OS**
  - your server and the interface objects to a PDS, CA Librarian etc.
- **BS2000**
  - your server and the interface objects to your application library. Note that the header files delivered in the LMS library LMS.EXP103.CSRV are required.
- **Other Platforms**
  - your server and the interface objects to a suitable directory (folder).

5   With a C compiler supported by the C Wrapper, compile the following objects, depending on the platform:

- **z/OS**
  - the generated interface object
  - your server (including your application logic)
- **BS2000**
  - the generated interface object
  - your server (including your application logic).

  Use any C/C+ ILCS-enabled compiler on BS2000.

- **Other Platforms**

  - the generated interface object

  - your server (including your application logic).

  Use the standard C/C++ compiler of your target platform. Note the platform-specific settings.

6   Using the linker (binder), link (bind) the following objects, depending on the platform:

- **z/OS**

  - Create dynamic-link libraries (DLLs) for the client interface objects and RPC server. See *Building and Using Dynamic-link Libraries* (DLLs) in the *z/OS C/C++ Programming Guide*, Order No. SC09-2362-03 or later, available through IBM and *Architecture and Software Support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point Arithmetic - References* under *http://www.research.ibm.com/journal/rd/435/abbotref.html* (subscription required).

  - There are various possibilities to combine the client interface objects and RPC server together and create dynamic-link libraries (DLLs). We suggest you keep the generated client interface object DLLs separate from RPC server DLLs:

    - Create two larger DLLs, one containing all your client interface objects and one containing all your RPC servers, and use the `FIX(dd1name)` configuration of the parameter `library` of the *RPC Server for Batch*.

    - Create separate DLLs, one for each client interface object and each RPC server and use the `PREFIX(prefix)` configuration with prefix "D", that is, `PREFIX(D)-PREFIX()` of the parameter `library` of the *RPC Server for Batch*.

- **UNIX**

  - the interface object as a shared object or shared library. If, for example, the library name within the Software AG IDL file is HUGO, the standard name of the dynamically callable interface object is *DHUGO.so* or *DHUGO.sl*. The standard name can be changed (see the Parameter Libraries of the RPC Server).

  - your server as a shared object or shared library. If, for example, the library name within the Software IDL file is HUGO, the standard name of the dynamically callable server is *HUGO.so* or *HUGO.sl*.

  Use the standard C/C++ linker of your target platform. Note the platform-specific settings.

- **Windows**

  - the interface object as a DLL. If, for example, the library name within the Software IDL file is HUGO, the name standard name of the dynamically callable interface object is *DHUGO.dll*. The standard name can be changed (see the Parameter Libraries of the RPC Server).

  - your server as a DLL. If, for example, the library name within the Software IDL file is HUGO, the standard name of the dynamically callable server is *HUGO.dll*.

Use the standard C/C++ linker of your target platform. Note the platform-specific settings.

- **BS2000**
  - the interface object as an LLM, using `BINDER`
  - your server as an LLM, using `BINDER`

  There is no need to link the object modules with the BS2000 Common Runtime Environment (CRTE) library. The CRTE is loaded once dynamically in the corresponding worker task of the RPC server where the server program is executed.

- **Other Platforms**
  - Use the standard linker of your target platform.

7 Provide the interface object library and the server library accessible to the RPC server according to the rules of your operating system.

Generate C Source Files from Software AG IDL Files

This section describes how to generate C source files from Software AG IDL files. It covers the following topics:

- Select an IDL File and Generate RPC Client or RPC Server
- Settings
- Mapping Options
- Generate RPC Client
- Generate RPC Server

Select an IDL File and Generate RPC Client or RPC Server

From the context menu, choose **Other > Generate C > RPC Client** and **> RPC Server** to generate the C source files.
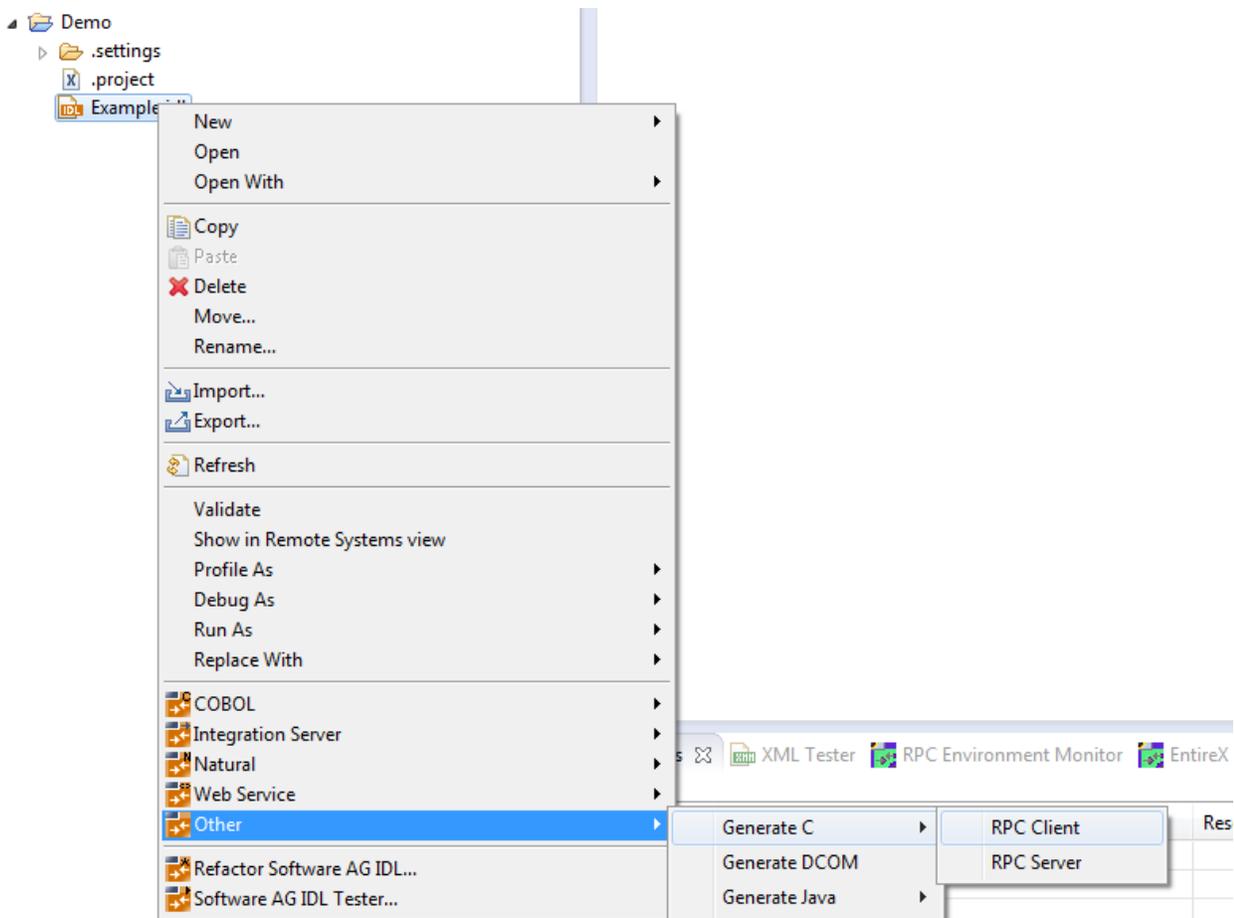
For the **RPC client**

- this creates for each library defined in the IDL file the client interface object and its associated header file. All files will be stored in parallel to the IDL file.

- In command-line mode, use the command `-c:client`. See *Using the C Wrapper in Command-line Mode*.

For the **RPC server**

- this creates for each library defined in the IDL file, the server interface object, its associated header file and the server skeleton file for your server implementation. All files will be stored in parallel to the IDL file.

- In command-line mode, use the command `-c:server`. See *Using the C Wrapper in Command-line Mode*.

> **Warning:**
> **Take care not to overwrite an existing server implementation with a server skeleton.** We recommend you move your server implementation to a different folder, or rename the server implementation.

For both **RPC client** and **RPC server**

- If you generate using the GUI and generated files exist from a previous generation, you are prompted to overwrite them.

- If you generate using command-line mode, existing files are always overwritten.

- The header file created is the same for the RPC client as for the RPC server side and contains, for example, C structure definitions for groups in the IDL file and the prototypes for your server. Use these generated C structures in your RPC client application and server implementation as required.

Settings

Use the properties of the IDL file - initialized from the C Wrapper preference page when used for the first time - to manipulate the mapping between Software AG IDL and the C source. A multithreaded client can be enforced.

Client settings
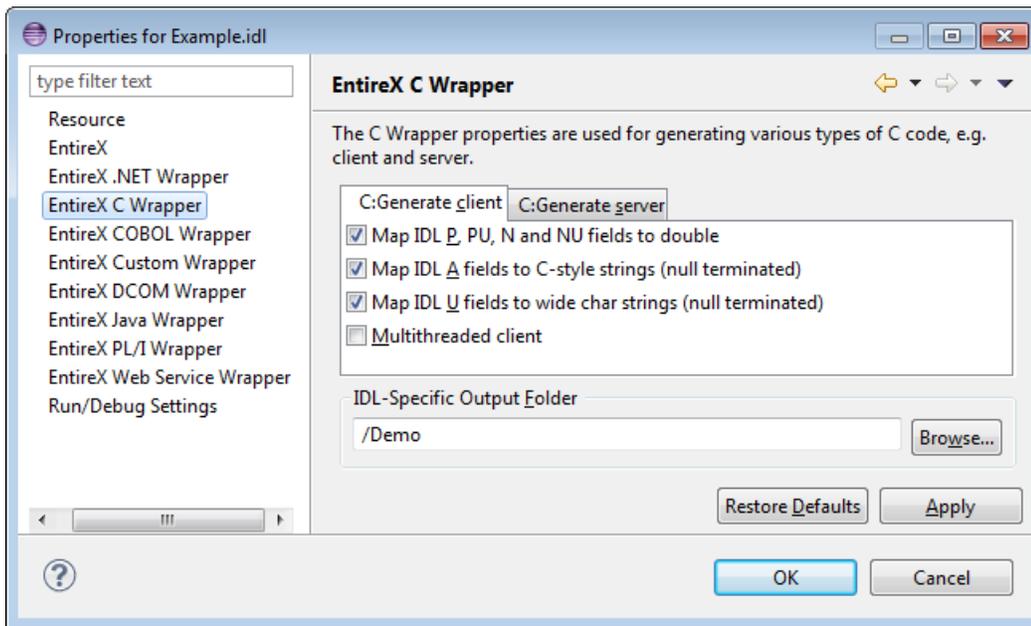
Server settings
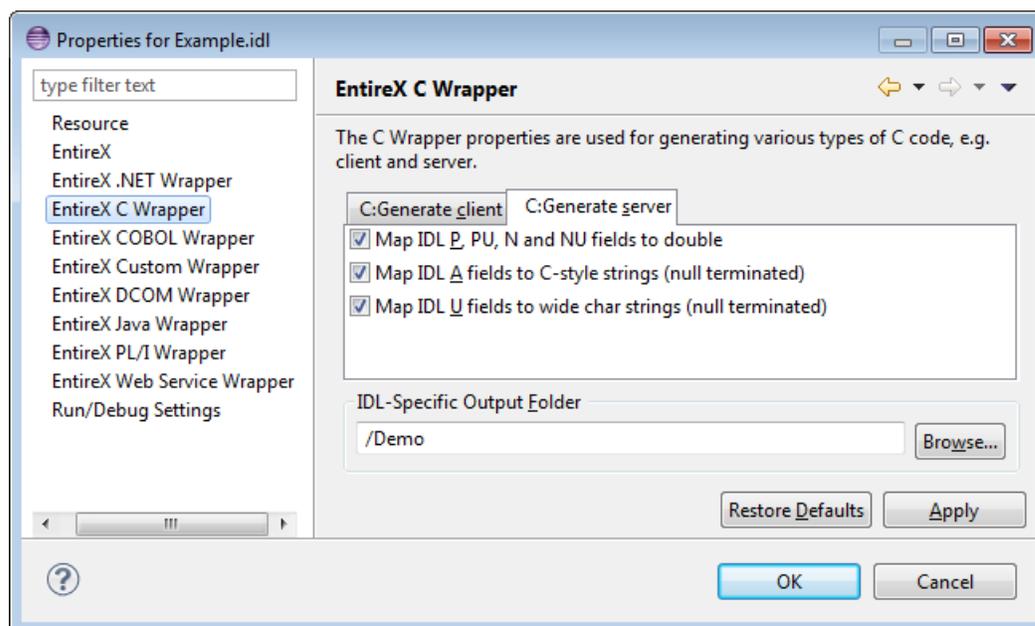
Mapping Options

Select the mapping options according to your needs. All mapping options are available for RPC clients and RPC server. See also *Mapping IDL Data Types to C Data Types*.

Option	Description
Map IDL N, P, NU and PU fields to double	If the check box is *checked*, the IDL data types N, NU, P and PU are mapped to the C data type `double`. If the check box is *not checked*, the IDL data type: ■ N and NU are mapped to the C data type `unsigned char[..]` with unpacked (mainframe Natural, COBOL, PL/I style) contents. ■ P and PU are mapped to the C data type `unsigned char[..]` with packed (mainframe Natural, COBOL, PL/I style) contents.
Map IDL A fields to C style strings (null terminated)	If the check box is *marked*, the IDL data type A is mapped to a C style string (C data type char[.. + 1] with null termination). This is recommended and comfortable for C programmers and is intended to be used with the C `str...` functions. This mapping does not allow use of trailing blanks and null values to send/receive. If the check box is *not marked*, the IDL data type A is mapped to the C data type unsigned char[..] without null termination (mainframe Natural, COBOL, PL/I style). This allows the use of trailing blanks, null values and is intended to be used with the C `mem...` functions.

Option	Description
Map IDL U fields to wide char strings (null terminated)	If the check box is *marked*, the IDL data type U is mapped to a C style wide character string (C data type `wchar_t[.. + 1]` with null termination). This is recommended and comfortable for C programmers and is intended to be used with the C `wcs...` functions. This mapping does not allow use of trailing wide character blanks (Unicode character x3000) and null values (x0000) to send/receive. If the check box is *not marked*, the IDL data type U is mapped to the C data type `unsigned wchar_t[..]` without null termination (mainframe Natural, COBOL, PL/I style). This allows the use of trailing blanks, null values and is intended to be used with the C `mem...` functions.

If the settings for the client side need to be different from the settings for the server side, generate the RPC client in a directory other than the RPC server directory.

Generate RPC Client

Select the **Multithreaded Client** option according to your needs.

Option	Description
Multithreaded Client	If the check box is *not marked*, the generated client interface object(s) can be used in single-threaded client environments. Use this option if you want to build an RPC client application as described under *Using the C Wrapper in Single-threaded Environments (UNIX, Windows)*. If the check box is *marked*, the generated client interface object(s) are thread-safe and can be used in multithreaded client environments. Use this option if you want to build an RPC client application as described under *Using the C Wrapper in Multithreaded Environments (UNIX, Windows)*.

Generate RPC Server

If you want to build an RPC server application, follow the instructions given under *Using the C Wrapper for the Server Side (z/OS, UNIX, Windows, BS2000)*.

3 Using the C Wrapper in Command-line Mode

Commands are available to generate an RPC client or RPC server from a specified IDL file.

Command-line Options

See *Using EntireX in the Designer Command-line Mode* for the general command-line syntax. The table below shows the command-line options for the C Wrapper if the Designer is used. Default values are underlined.

Task	Command	Option	Value	Description
Generate RPC client from the specified IDL file.	`-c:client`	`-DATA_CONV_NP`	0 | 1	Mapping of IDL type P, PU, N and NU fields. 0 The IDL data types are mapped to C data type unsigned char[..] with packed or unpacked (mainframe Natural, COBOL, PL/I style) contents. 1 The IDL data types are mapped to C data type double. See *Mapping Options* for more information.
		`-DATA_CONV_A`	0 | 1	Mapping of IDL type A fields. 0 Map IDL data type A to the C data type unsigned char[..] without null termination (mainframe Natural, COBOL, PL/I style). 1 Map IDL data type A to C style strings. See *Mapping Options* for more information.
		`-DATA_CONV_U`	0 | 1	Mapping of IDL type U fields. 0 Map IDL data type U to the C data type unsigned wchar_t[..] without null termination (mainframe Natural, COBOL, PL/I style). 1 Map IDL data type U to wide char strings. See *Mapping Options* for more information.
		`-DATA_CONTEXT`	0 | 1	Multithread client. 0 Off. The generated client interface objects can be used in single-threaded client environments. 1 On. The generated client interface objects are thread-safe and can be used in multithreaded client environments. See *Generate RPC Client* for more information.
		`-help`		Display this usage message.
Generate RPC server	`-c:server`	`-DATA_CONV_NP`	0 | 1	Mapping of IDL type N, P, NU and PU fields, for more information see above.

Task	Command	Option	Value	Description
from the specified IDL file.		`-DATA_CONV_A`	0 | 1	Mapping of IDL type A fields, for more information see above.
		`-DATA_CONV_U`	0 | 1	Mapping of IDL type U fields, for more information see above.
		`-help`		Display this usage message.

> **Note:** The commands "`-C:Generate client`" and "`-C:Generate server`" are deprecated. Use `-c:client` and `-c:server` instead.

Example Generating an RPC Client

```
<workbench> -c:client /Demo/example.idl -DATA_CONV_A 1
```

where *<workbench>* is a placeholder for the actual EntireX design-time starter as described under *Using EntireX in the Designer Command-line Mode*.

The generated C source files (client interface object and its associated header file)

- will be stored in parallel to the IDL file, for example in project *Demo*.
- will overwrite existing files from a previous command-line mode generation.

Example Generating an RPC Server

```
<workbench> -c:server /Demo/example.idl -DATA_CONV_A 1
```

where *<workbench>* is a placeholder for the actual EntireX design-time starter as described under *Using EntireX in the Designer Command-line Mode*.

The generated C source files (server interface object and its associated header file)

- will be stored in parallel to the IDL file, for example in project *Demo*
- will overwrite existing files from a previous command-line mode generation.

> **Warning:**
> **Take care not to overwrite an existing server implementation with a server skeleton.**
> We recommend you move your server implementation to a different folder, or rename the server implementation.

Further Examples

Windows

Example 1

```
<workbench> -c:client C:\Temp\example.idl
```

Uses the IDL file *C:\Temp\example.idl* and generates the C source files (*CEXAMPLE.c* and *CEXAMPLE.h*) in parallel to the IDL file. Slashes and backslashes are permitted in the file name. Output to standard output:

```
Using workspace file:/C:/myWorkspace/.
Exit value: 0
```

Example 2

```
<workbench> -c:client -help
```

or

```
<workbench> -help -c:client
```

Both show a short help for the C Wrapper.

Linux

Example 1

```
<workbench> -c:client /Demo/example.idl
```

If the project Demo exists in the workspace and *example.idl* exists in this project, this file is used. Otherwise, */Demo/example.idl* is used from file system. The generated output (*CEXAMPLE.c* and *CEXAMPLE.h*) will be stored in */Demo*, parallel to the IDL file.

Example 2

```
<workbench> -c:client -help
```

or

```
<workbench> -help -c:client
```

Both show a short help for the C Wrapper.

4 Using the C Wrapper in IDL Compiler Command-line Mode

The table below shows the command-line options for the C Wrapper if the IDL Compiler is used. Default values are underlined. Options can be valid for client and server side, see column Client and Server.

Option	Client	Server	Value	Description
-DCONTEXT	Yes	No	<u>0</u> | 1	Multithreaded Client. 0 Off. The generated client interface objects can be used in single-threaded client environments. 1 On. The generated client interface objects are thread-safe and can be used in multithreaded client environments. See *Generate RPC Client* for more information.
-DDATA_CONV_A	Yes	Yes	<u>0</u> | 1	Mapping of IDL type A fields. 0 Map IDL data type A to the C data type unsigned char[..] without null termination (mainframe Natural, COBOL, PL/I style). 1 Map IDL data type A to C style strings See *Mapping Options* for more information.
-DDATA_CONV_NP	Yes	Yes	0 | <u>1</u>	Mapping of IDL type P, PU, N and NU fields. 0 The IDL data types are mapped to C data type unsigned char[..] with packed or unpacked (mainframe Natural, COBOL, PL/I style) contents. 1 The IDL data types are mapped to C data type double. See *Mapping Options* for more information.
-DDATA_CONV_U	Yes	Yes	<u>0</u> | 1	Mapping of IDL type U fields. 0 Map IDL Data type U to the C data type unsigned wchar_t[..] without null termination (mainframe Natural, COBOL, PL/I style).

Option	Client	Server	Value	Description
				1 Map IDL Data type U to wide char strings. See *Mapping Options* for more information.

See also *Starting the IDL Compiler* and *IDL Compiler Usage Examples*.

Example Generating an RPC Client

```
erxidl -t client.tpl -DDATA_CONV_A=1 -DDATA_CONV_U=1 example.idl
```

The generated C source files (client interface object and its associated header file) will be stored in parallel to the IDL file.

Example Generating an RPC Server

```
erxidl -t server.tpl -DDATA_CONV_A=1 -DDATA_CONV_U=1 example.idl
```

The generated C source files (server interface object and its associated header file) will be stored in parallel to the IDL file.

5 Software AG IDL to C Mapping

This chapter describes the specific mapping of Software AG IDL data types, groups, arrays and structures to the C programming language. See also hints and restrictions on the Software AG IDL data types valid for all programming language bindings under *IDL Data Types* in the IDL Editor documentation.

Mapping IDL Data Types to C Data Types

In the table below, the following metasymbols and informal terms are used for the IDL.

- The metasymbols "[" and "]" enclose optional lexical entities.

- The informal term *number* (or in some cases *number1.number2*) is a sequence of numeric characters, for example 123.

Software AG IDL	Description	C Data Type	Note
A*number*	Alphanumeric	`unsigned char [number]`	1,3
		`char [number + 1]`	1,3
AV	Alphanumeric variable length	`ERX_HVDATA`	4
AV*number*	Alphanumeric variable length with maximum length	`ERX_HVDATA`	3,4
B*number*	Binary	`unsigned char [number]`	3
BV	Binary variable length	`ERX_HVDATA`	4
BV*number*	Binary variable length with maximum length	`ERX_HVDATA`	4
D	Date	`unsigned char [ERX_GET_PACKED_LEN(7)]`	3,5,8,12,14,15
F4	Floating point (small)	`float`	11
F8	Floating point (large)	`double`	11
I1	Integer (small)	`signed char`	
I2	Integer (medium)	`signed short`	
I4	Integer (large)	`signed long`	
K*number*	Kanji	`unsigned char [number]`	3

Software AG IDL	Description	C Data Type	Note
KV	Kanji variable length	ERX_HVDATA	4
KV*number*	Kanji variable length with maximum length	ERX_HVDATA	3,4
L	Logical	unsigned char	6
N*number1*[.*number2*]	Unpacked decimal	double	9
		unsigned char[*number1* + *number2*]	7,9
NU*number1*[.*number2*]	Unpacked decimal unsigned	double	9
		unsigned char[*number1* + *number2*]	7,9
P*number1*[.*number2*]	Packed decimal	double	10
		unsigned char[ERX_GET_PACKED_LEN(*number1* + *number2*)]	8,10
PU*number1*[.*number2*]	Packed decimal unsigned	double	10
		unsigned char[ERX_GET_PACKED_LEN(*number1* + *number2*)]	8,10
T	Time	unsigned char[ERX_GET_PACKED_LEN(13)]	5,8,13,14,16
U*number*	Unicode	wchar_t[number]	2,17,18
		wchar [number + 1]	2,17,18
UV	Unicode variable length	ERX_HVDATA	4,17
UV*number*	Unicode variable length with maximum length	ERX_HVDATA	4,17,19

See also hints and restrictions on the Software AG IDL data types valid for all programming language bindings under *IDL Data Types* in the IDL Editor documentation.

📄 **Notes:**

1. It is possible to map the data type to the C data type unsigned char[..] without null termination (mainframe Natural, COBOL, PL/I style) or C style string (C data type char[.. + 1] with null termination). The mapping is controlled with the *Mapping Options* when you generate source files from IDL. See *Generate C Source Files from Software AG IDL Files*.

2. It is possible to map the data type to the C data type wchar_t[..] without null termination (mainframe Natural, COBOL, PL/I style) or to a C style wide character string (C data type wchar_t[.. + 1] with null termination). The mapping is controlled with the *Mapping Options*

when you generate source files from IDL. See *Generate C Source Files from Software AG IDL Files*.

3. The field length is given in bytes.

4. The data type `ERX_HVDATA` is an RPC-specific data type in the header file *erxVData.h* for handling variable-length data types. See *API Function Descriptions for Variable-length Data Types AV, BV, KV and UV*.

5. Date and Time values are mapped in mainframe-style packed format.

6. The binary value zero is treated as `FALSE`. Contents other than a binary zero are treated as `TRUE`.

7. Unpacked format is an internal representation of numbers with a specified number of digits used in mainframe environments (Natural, COBOL, PL/I).

8. Packed format is an internal representation of numbers with a specified number of digits used in mainframe environments (Natural, COBOL, PL/I). `ERX_GET_PACKED_LENGTH` is a macro within the header file `erx.h` used to evaluate the length of a field in bytes with packed contents.

9. It is possible to map the data type to double or unpacked. The mapping is controlled with the *Mapping Options* when you generate source files from IDL. See *Generate C Source Files from Software AG IDL Files*.

    ▪ For unpacked, the total number of digits (*number1*+*number2*) is 99.

    ▪ For double, it depends on your target environment how many significant digits are supported. In most environments 15 - 16 digits, which means for exact results the total number of digits (*number1*+*number2*) should be less than 16.

    If you connect two endpoints, the total number of digits used must be lower or equal than the maxima of both endpoints. For the supported total number of digits for endpoints, see the notes under data types N, NU, P and PU in section *Mapping IDL Data Types* in the respective Wrapper or language-specific documentation.

10. It is possible to map the data type to `double` or `packed`. The mapping is controlled with the *Mapping Options* when you generate source files from IDL. See *Generate C Source Files from Software AG IDL Files*.

    ▪ For unpacked the total number of digits (*number1*+*number2*) is 99.

    ▪ For double, it depends on your target environment how many significant digits are supported. In most environments 15 - 16 digits, which means for exact results the total number of digits (*number1*+*number2*) should be less than 16.

    If you connect two endpoints, the total number of digits used must be lower or equal than the maxima of both endpoints. For the supported total number of digits for endpoints, see the notes under data types N, NU, P and PU in section *Mapping IDL Data Types* in the respective Wrapper or language-specific documentation.

11. For `float` and `double`, rounding errors can occur, so that the values of senders and receivers might differ.

12. Count of days AD (anno domini, after the birth of Christ). The valid range is from 1.1.0001 up to 28.11.2737. Mapping of the number to the date in the complete range from 1.1.0001 on, follows the Julian and Gregorian calendar, taking into consideration the following rules:

   1. Years that are evenly divisible by 4 are leap years.

   2. Years that are evenly divisible by 100 are not leap years unless rule 3, below, is true.

   3. Years that are evenly divisible by 400 are leap years.

   4. Before the year 1582 AD, rule 1 from the Julian calendar is used. After the year 1582 AD, rules 1, 2 and 3 of the Gregorian calendar are used.

   See the following table for the relation of the packed number to a real date:

Date / Range of Dates	Value / Range of Values
1.1.0000	0 (special value - no date)
undefined dates	1 - 364 (do not use)
1.1.0001	365
1.1.1970	719527 (start of C-time functions)
28.11.2737	999999 (maximum date)

13. Count of tenth of seconds AD (anno domini, after the birth of Christ). The valid range is from 1.1.0001 00:00:00.0 up to 16.11.3168 9:46:39 plus 0.9 seconds. See the following table for the relation of the packed number to a real time:

Time / Range of Times	Value / Range of Values
1.1.0000 00:00:00.0	0 (special value - no time)
undefined times	1 - 315359999
1.1.0001 00:00:00.0	315360000
1.1.1970 00:00:00.0	621671328000 (start of C-time functions)

14. The relation between the packed number of a Date and Time data type is as follows:

tenths of a second per day	=24*60*60*10 = 864000	
number of time	= number of date	* 864000
315360000	= 365	* 864000 1.1.0001 00:00:00.0
621671328000	= 719527	* 864000 1.1.1970 00:00:00.0
number of date	= number of time	/ 864000
365	= 315360000	/ 864000 1.1.0001
719527	= 621671328000	/ 864000 1.1.1970

15. The `no date` value is the internal state of a `#DATE` (Natural type D) variable after a `RESET #DATE` is executed within Natural programs. This internal state is not a valid date. The `no date` value

can be transferred as the invalid date 1.1.0 from RPC clients to servers and vice versa. C Wrapper supports Natural `no date` value. C Wrapper passes 0 to the C application when `no date` is received. With the same value of 0, the C application can send `no date` to its partner (client or server).

16. The `no time` value is the internal state of a `#TIME` (Natural type T) variable after a `RESET #TIME` is executed within Natural programs. This internal state is not a valid time. The `no time` value can be transferred as the invalid time 1.1.0 0:00:00.0 from RPC clients to servers and vice versa. C Wrapper supports Natural `no time` value. C Wrapper passes 0 to the C application when `no time` is received. With the same value of 0, the C application can send `no time` to its partner (client or server).

17. The Unicode encoding provided (on receive) or expected (on send) on the API depends on the width of the `wchar_t` data type in your environment:

Size of `wchar_t`	UTF Version Used
2 bytes (Windows and some UNIX environments)	UTF-16
4 bytes (some other UNIX environments)	UTF-32

The endianness (big endian or little endian) of the Unicode encoding that is used (UTF-16-LE, UTF-16-BE, UTF32-LE or UTF32-BE) is the same as the hardware architecture of the machine.

18. In environments where `sizeof(wchar_t)` is 4 bytes, only Unicode characters from the Basic Multilingual Plane are supported (code points U+0000 to U+FFFF except U+D800 to U+DFFF reserved for leading and trailing surrogates). Unicode characters from Supplementary Planes (code points U+10000 to U+10FFFF) are not supported.

19. In environments where `sizeof(wchar_t)` is 4 bytes, the number of UTF-16 Unicode code points (after character conversion from UTF-32 to UTF-16) must be less than or equal to *number*. Note the following:

  ▪ Unicode characters from the Basic Multilingual Plane (code points U+0000 to U+FFFF except U+D800 to U+DFFF, reserved for leading and trailing surrogates) require one code point in UTF-16

  ▪ Unicode characters from Supplementary Planes (code points U+10000 to U+10FFFF) require two code points in UTF-16

Mapping Library Name and Alias

The library name as specified in the IDL file is sent from a client to the server. Special characters are not replaced. The library alias is not sent to the server.

In the RPC server, the IDL library name sent may be used to locate the target server. See *Locating and Calling the Target Server* under z/OS (CICS, Batch, IMS) | C | .NET | BS2000 in the respective Administration or RPC Server documentation.

The library name as given in the IDL file is used to compose the names of the generated output files. See `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation. Therefore the allowed characters are restricted by the underlying file system.

For the server interface object, the name is composed with a prefix D as `Dlibrary-name.c` and for the server as `library-name.c`. For the client interface object the name is composed with a prefix C as `Clibrary name.c`. For both interface objects the same header file `Clibrary-name.h` is also used. When the name of the generated sources is built, lower and uppercase characters are considered and the special characters '#', '$', '&', '+', '-', '.', '/' and '@' used in the name for libraries are replaced by the character underscore '_'. Other special characters used in the library name are not changed and may lead to problems with your underlying file system.

Aliases for the library name in the IDL file are not supported in C Wrapper language binding. See `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation.

Examples:

- A library name of `#HU$GO.` results in `C_HU_GO_.c` and `C_HU_GO_.h` as the client interface object file name for the generated source.
- A library name of `#HU$GO.` results in `D_HU_GO_.c` and `C_HU_GO_.h` as the server interface object file name for the generated source.

Mapping Program Name and Alias

The program name is sent from a client to the server. Special characters are not replaced. The program alias is not sent to the server.

In the RPC server, the IDL program name sent is used to locate the target server. See *Locating and Calling the Target Server* under z/OS (CICS, Batch, IMS) | C | .NET | BS2000 in the respective Administration or RPC Server documentation.

The program names as given in the IDL file are mapped to functions within the generated C sources. See `program-definition` under *Software AG IDL Grammar* in the IDL Editor documentation. When building function names, lower and uppercase characters are considered and the special

characters '#', '$', '&', '+', '-', '.', '/' and '@' are replaced by the character underscore '_' valid for C names. Other special characters used in the program name are not changed and may lead to compilation errors when compiling the generated sources.

Aliases for the program name in the `program-definition` under *Software AG IDL Grammar* in the IDL Editor documentation are not supported in C Wrapper language binding.

Example

■ A parameter name of `#HU$GO.` results in `_HU_GO_` as the function name for the C programming language.

Mapping Parameter Names

The parameter names as given in the IDL file are mapped to parameters of the generated C functions. See `parameter-data-definition` under *Software AG IDL Grammar* in the IDL Editor documentation. When building parameters, lower and uppercase characters are considered and the special characters '#', '$', '&', '+', '-', '.', '/' and '@' are replaced by the character underscore '_' valid for C names.

■ **Examples:**
A parameter name of `#HU$GO.` results in `_HU_GO_` as the parameter name for the C programming language.

Mapping Fixed and Unbounded Arrays

■ Fixed arrays within the IDL file are mapped to fixed C arrays. The upper bound given in the IDL file is decremented by 1 because C arrays always start with the lower bound 0. For example the number `(I2/5)` in the IDL file will be mapped to `signed short number [4]`.

See the `array-definition` under *Software AG IDL Grammar* in the IDL Editor documentation for the syntax on how to describe fixed arrays within the IDL file and refer to `fixed-bound-array-index`.

■ Unbounded arrays within the IDL file are mapped to the `ERX_HARRAY` data type found in the header file `erxArray.h`. See *Using Unbounded Arrays* for more information.

See the `array-definition` under *Software AG IDL Grammar* in the IDL Editor documentation for the syntax of unbounded arrays within the IDL file and refer to `unbounded-array-index`.

Mapping Groups and Periodic Groups

Groups within the IDL file are mapped to the C data type `struct`. See the `group-parameter-definition` under *Software AG IDL Grammar* in the IDL Editor documentation for the syntax on how to describe groups within the IDL file.

Mapping Structures

Structures within the IDL file are mapped to the C data type `struct` like groups. See `structure-definition` under *Software AG IDL Grammar* in the IDL Editor documentation for the syntax on how to describe structures within the IDL file.

Mapping the Direction Attributes In, Out, InOut

The IDL syntax allows you to define parameters as `IN` parameters, `OUT` parameters, or `IN OUT` parameters (which is the default if nothing is specified). This direction specification is reflected in the generated C interface object as follows:

- Parameters with the `IN` attribute are sent from the RPC client to the RPC server. When the parameter is a simple parameter (that is, no fixed or unbounded array, no group and no structure) the parameter is provided with the method call by value. Complex parameters such as fixed and unbounded arrays, groups and structures are provided with the call by reference method.

- Parameters with the `OUT` attribute are sent from the RPC server to the RPC client. They are always provided with the call by reference method.

- Parameters with the `IN` and `OUT` attribute are sent from the RPC client to the RPC server and then back to the RPC client. They are always provided with the call by reference method.

Note that only the direction information of the top-level fields (level 1) is relevant. Group fields always inherit the specification from their parent. A different specification is ignored.

See the `attribute-list` under *Software AG IDL Grammar* in the IDL Editor documentation for the syntax on how to describe attributes within the IDL file and refer to `direction` attribute.

Mapping the ALIGNED Attribute

The ALIGNED Attribute is not relevant for the programming language C. However, a C client can send the `ALIGNED` attribute to an RPC server where it might be needed.

See the `attribute-list` under *Software AG IDL Grammar* in the IDL Editor documentation for the syntax of attributes in the IDL file and refer to the `aligned` attribute.

Calling Servers as Procedures or Functions

The IDL syntax allows definitions of procedures only. It does not have the concept of a function. A function is a procedure which, in addition to the parameters, returns a value. Procedures and functions are transparent between clients and servers. This means a client using a function can call a server implemented as a procedure and vice versa.

It is possible to call the remote procedure as a function and not as a procedure, if you prefer it and if it suits your interface.

Example

The function `float sin(float x)` will be called as a function - and not as a procedure - when defined in the IDL file as follows:

```
Library ... is
  Program 'sin' is
    Define Data Parameter
      1 x (F4) In
      1 Function_Result (F4) Out
    End-Define
```

it can be invoked as :

```
y = sin(x);
```

When you generate source files from IDL (see *Generate C Source Files from Software AG IDL Files*), a C function instead of a C procedure is generated if the following is true for the interface description in the IDL file:

- The last parameter's name is `function_result`. The name `function_result` is case-insensitive.

- The last parameter's direction is `Out`. See `attribute-list` under *Software AG IDL Grammar* in the IDL Editor documentation.

- The last parameter is a scalar variable, that is, not an array.

■ The last parameter is of type and length:

Software AG IDL	Description	Note
I1	Integer (small)	2
I2	Integer (medium)	2
I4	Integer (large)	2
A1	Alphanumeric with length 1	2
B1	Binary with length 1	2
L	Logical	2
N*number* [*.number*]	Unpacked decimal	1, 2
P*number* [*.number*]	Packed decimal	1, 2
NU*number* [*.number*]	Unpacked decimal unsigned	1, 2
PU*number* [*.number*]	Packed decimal unsigned	1, 2

Notes:

1. The data types must be mapped to double. See *Mapping Options* when you generate C source files from Software AG IDL files.

2. The type of the function returned is defined by *Mapping IDL Data Types to C Data Types*.

II

6   Writing a Single-threaded C RPC Client Application

This chapter describes in six steps how to write your first C client program.

The example given here demonstrates how to write a single-threaded C RPC client application. It demonstrates an implicit broker logon (because no broker logon/logoff calls are implemented), where it is required to switch on the AUTOLOGON feature in the broker attribute file.

The following steps describe how to write a single-threaded C client program. We recommend reading them first before writing your first RPC client program and following them where appropriate.

Step 1: Base Declarations Required by the C Wrapper

Step 1a: Include the Generated Header File

Define the generated client header file. This header file includes the RPC runtime header file erx.h and defines structures and prototypes for your RPC requests.

```
/* include generated header file */
#include "example.h"
```

Step 1b: Define Global Variables to Communicate with the Client Interface Objects

For single-threaded clients you have to declare in your main program the following global variables, used for communication with the interface objects:

```
/* Needed global variables for the CLIENT interface object */
ERXCallId                ERXCallId;
ERXeReturnCode           ERXrc;
ERX_ERROR_INFO           ERXErrorInfo;
ERX_Server_ADRESS        ERXServer;
ERX_CLIENT_IDENTIFICATION ERXClient;
```

Step 2: Required Settings for the C Wrapper

Step 2a: Identify the User with a Broker User ID

For *Implicit Logon*, if required in your environment, the client password can be given here. It is provided then through the interface object call.

```
/* set client identification */
memset( &ERXClient, 0, sizeof(ERXClient) );
ERXClient.pUserId   = "ERX-USER";
ERXClient.pPassword = "ERX_PASS";
```

Step 2b: Set the Broker and Service to be Called

Your application will wait a maximum of 55 seconds for a server response. If the server does not answer within this period, the broker gives your program control again with an error code 00740074.

```
ERXServer.Medium = ERX_TM_BROKER;
ERXServer.ulTimeOut = 55;

/* set Broker-Id, server-name, class-name and service-name */
strcpy( (char*) ERXServer.Address.BROKER.szEtbidName,   "ETB001" );
strcpy( (char*) ERXServer.Address.BROKER.szServerName,  "SRV1" );
strcpy( (char*) ERXServer.Address.BROKER.szClassName,   "RPC" );
strcpy( (char*) ERXServer.Address.BROKER.szServiceName, "CALLNAT" );
```

Step 3: Register with the RPC Runtime

As a general rule, before using the RPC runtime you have to register it. After registration, the RPC runtime holds information on a per-thread basis. See *Using the RPC Runtime* for more information.

```
/* register to the RPC runtime */
ERXrc = ERXRegister( ERX_V81 );
If ( ERX_FAILED( ERXrc ) )
{
/* code for  error handling */
}
```

Step 4: Issue the RPC Request

The RPC interface object `CALC` is called as C function (see *Calling Servers as Procedures or Functions*).

```
/* do the remote procedure call */
result = CALC( '+', 123456, 78910 );
```

Step 5: Examine the Error Code

When a return from the RPC request has been received, check whether the call was successful with the macro ERX_FAILED.

```
if( ERX_FAILED( ERXrc ) )
{
/* code for error handling */
}
```

Detailed information about an error can be retrieved with the function `ERXGetLastError`. For the error messages returned, see *Error Messages and Codes*.

Step 6: Deregister with the RPC Runtime

As a general rule, after using the RPC runtime you should unregister from it. This will free all resources held by the RPC runtime for the caller. See *Using the RPC Runtime* for more information.

```
/* unregister to the RPC runtime */
ERXUnregister();
```

7 Writing Advanced Applications with the C Wrapper

Using the RPC Runtime

As a general rule, before using the EntireX RPC runtime, a program/thread must be registered with it using the `ERXRegister` function. Hence `ERXRegister` must be the first call to the EntireX RPC runtime and `ERXUnregister` the last. The number of registrations and unregistrations should be symmetric for every thread, otherwise the thread's resources that are held by the EntireX RPC runtime will not be freed. However, successful unregistration of the last thread within a process will free all EntireX RPC runtime resources.

Each thread of a process has to register separately with the EntireX RPC runtime. After registration the EntireX RPC runtime maintains on a per-thread basis

- codepage settings, see `ERXSetCodepage` and *Using Internationalization with the C Wrapper*

- broker security settings, see `ERXSetBrokerSecurity`, `ERXSetSecurityToken` and *Using EntireX Security*

- the RPC conversation context, if any RPC conversation is ongoing, see `ERXConnect` and *Using Conversational RPC*

- the last error, which can be retrieved using `ERXGetLastError`

The following limitations and restrictions also apply:

- Up to 256 threads can be registered in parallel within a process.

- Multiple registration up to 32,767 per thread before unregistration is possible but not recommended.

All functions provided by the EntireX RPC runtime to handle the variable-length data types and unbounded arrays can be used without registration.

- Functions to handle variable-length data types are defined in the header file *erxvdata.h* and are prefixed with "erxVData".

- Functions to handle unbounded arrays are defined in the header file *erxarray.h* and are prefixed with "erxArray".

Examine the RPC Runtime and Interface Object Version

The EntireX C Wrapper API provides an interface to examine the version of the RPC Runtime, see `ERXGetVersion`.

Examine the Interface Object Version

If you generate interface objects according to the instructions given in *Using the C Wrapper for the Client Side*, a function to examine the interface object version on a per-library basis is also generated:

```
int ERX_CALL_DECLARATION GetVersionEXAMPLEStub(
    char    *pMessage,
    size_t   uMessageLength
);
```

Calling this function will provide you with the version and patch level under which the interface object was generated.

Example

```
EntireX C Wrapper Version=10.8.0, Patch Level=0
```

Tracing

There are several possibilities to trace the EntireX C Wrapper. See *Tracing webMethods EntireX under UNIX | Windows | BS2000 | z/VSE* in the platform-specific Administration documentation.

Programming Multithreaded RPC Clients

The EntireX C Wrapper runtime supports RPC clients in multithreaded environments. Every thread can establish its own RPC and broker context for communication, which is separate from every other thread's context, see also *Using the RPC Runtime*.

The functions `ERXSetContext` and `ERXGetContext` together with client interface objects generated using the instructions given in *Using the C Wrapper in Multithreaded Environments (UNIX, Windows)* assist in programming multithreaded RPC clients.

The `ERXSetContext` function can be executed prior to any business logic to provide the RPC and broker context individually on a per-thread basis. `ERXSetContext` saves the context information in a structure `ERX_CONTEXT_BLOCK`. The client interface object picks up the context from the calling

thread using the reverse function `ERXGetContext`. Hence legacy applications may not be changed to transport this information.

Additional Notes:

- To use the `ERXSetContext` and `ERXGetContext` functions, client interface objects must be generated with the check box **Multithreaded Client** switch. See *Generate C Source Files from Software AG IDL Files*.

- A maximum of 256 threads are supported in parallel.

- The `ERXSetContext` function can be called multiple times (within the same thread). This also makes it possible to change RPC and broker context with each RPC request.

- Nothing needs to be considered for servers. EntireX RPC servers support multithreading without any further activities.

Logon to Natural Library

By default the library name sent to the RPC server is retrieved from the IDL file (see `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation). The library name can be overwritten. This is useful if communicating with a Natural RPC server.

> **To overwrite the library name**

1 Set the medium `ERX_TM_BROKER_LIBRARY` in the `ERX_SERVER_ADDRESS` structure.

2 Specify the library in the `ERX_SA_BROKER_LIBRARY` structure in the `szLibraryName` parameter you want to log on to.

3 Set the parameter `cNaturalLogon` to `"ERX_NATURAL LOGON_YES"` in the `ERX_SA_BROKER_LIBRARY` structure.

If you need to set security credentials too, refer to *Using the Broker and RPC User ID/Password*.

Using Variable-length Data Types AV, BV, KV and UV

The following functions are used to send and receive the variable-length data types (IDL data types AV, BV, KV and UV). All variable-length data is controlled by so-called `VData` instances. A `VData` instance is a handle (pointer) to a memory location encapsulated in the `erxVData...` functions in the EntireX RPC runtime.

A `VData` instance has the following type definition:

```
typedef void * ERX_HVDATA; /* Handle of VData instance */
```

See the following overview of `VData` functions.

Task	Function

Allocate a new `VData` instance	`erxVDataAllocBytes`
	`erxVDataAllocString`
	`erxVDataAllocWideString`

Remove a `VData` instance	`erxVDataFree`

Get the contents held by a `VData` instance	`erxVDataGetByteAddress`
	`erxVDataGetLength`
	`erxVDataGetString`
	`erxVDataGetWideString`

Assign new contents to the `VData` instance	`erxVDataCopy`
	`erxVDataReAllocBytes`
	`erxVDataReAllocString`
	`erxVDataReAllocWideString`
	`erxVDataReset`

Usage with EntireX RPC Client

Before issuing the RPC request, allocate all `VData` instances including the instances for direction out (which is returned by the RPC server only), see `attribute-list` under *Software AG IDL Grammar* in the IDL Editor documentation.

≫ **To allocate and create `VData` instances**

1   For the directions in and inout, use `erxVDataAllocBytes` (IDL data type BV and KV), `erxVDataAllocString` (IDL data type AV) or `erxVDataAllocWideString` (IDL data type UV) with the appropriate parameters to allocate the `VData` instances.

2   For the direction out use `erxVDataAllocBytes(NULL,0)` (IDL data type BV and KV), `erxVDataAllocString(NULL)` (IDL data type AV) or `erxVDataAllocWideString(L"")` (IDL data type UV)to create an empty `VData` instance, which will contain the data returned by the server.

Following the RPC request, you can examine the server reply.

≫ **To examine the server reply**

■   Use the functions `erxVDataGetString` (IDL data type AV), `erxVDataGetWideString` (IDL data type UV) and `erxVDataGetLength, erxVDataGetByteAddress` (IDL data type BV and KV).

⟩ To remove `VData` instances

■ Use the function `erxVDataFree` if they are no longer needed.

Usage with EntireX RPC Server

When your implemented server is called, all `VData` instances are allocated by the RPC C runtime and RPC Server. The data sent by the client can be examined in the server program (in the same way the client does upon server reply). The RPC Server and the RPC C runtime will remove the `VData` instances if they are no longer needed. Do not remove any `VData` instances in server programs yourself!

⟩ To examine the client data

■ Use the functions `erxVDataGetString` (IDL data type AV), `erxVDataGetWideString` (IDL data type UV) and `erxVDataGetLength`, `erxVDataGetByteAddress` (IDL data type BV and KV).

⟩ To assign data to be returned

■ Use the functions `erxVDataReAllocBytes` (IDL data type BV and KV), `erxVDataReAllocWideString` (IDL data type UV) and `erxVDataReAllocString` (IDL data type AV).

Using Unbounded Arrays

The following functions are used to send and receive unbounded array data types of EntireX RPC (data types defined with V in the indices). All unbounded arrays are controlled by so-called arrays instances. An arrays instance is a handle (pointer) to a memory location encapsulated by the EntireX RPC Runtime.

An array instance has the following type definition:

```
typedef void * ERX_HARRAY; /* Handle of Array instance */
```

See the following overview of functions for use with unbounded arrays.

Task	Function
Allocate a new array instance	`erxArrayAlloc`
Remove an array instance	`erxArrayFree`
Get the contents of an array instance	`erxArrayGetElement`

Assign new contents to an array instance	`erxArrayCopy`
	`erxArrayReset`
	`erxArraySetElement`

Get the characteristics of an array instance	`erxArrayGetAttributes`
	`erxArrayGetBounds`
	`erxArrayGetDimension`
	`erxArrayGetElementLength`
	`erxArrayGetTypeCode`

Change upper bounds of an array instance	`erxArrayRedimAll` `erxArrayRedimVector`

Usage with EntireX RPC Client

Before calling the client interface object (that is, before issuing the remote procedure call) allocate all array instances and create instances for out data. You cannot change any type, attribute, length or dimension. When the instances are created, only the upper bounds can be changed.

For the directions in and inout, use `erxArrayAlloc` with appropriate parameters to allocate an array instance. For the direction out, create an array instance of correct type, attributes, length and dimension with all upper bounds set to 0. This is an empty array instance with no elements. Upon return it will contain the elements assigned by the server.

EntireX RPC supports unbounded arrays which must not necessarily be a square (when 2-dimensional) or a cube (when 3-dimensional). Any vector within any dimension could have different upper bound settings. Such an array could be created in two ways:

- Start with an empty array and set the upper bounds of the first dimension with `erxArrayRedimVector`. Subseqeuently loop through this dimension and set any vector of the second dimension using also `erxArrayRedimVector`. If it is a 3-dimensional array, do the same with the third dimension.

- Create a square (2-dimensional) or cube (3-dimensional) with `erxArrayAlloc` or `erxArrayRedimAll` and subsequently deform the array with `erxArrayRedimVector`.

Data to be sent can be assigned using the function `erxArraySetElement` as long as the index is within the current upper bounds. Otherwise an error will occur. Because any vector of any dimension could have different upper bound settings, the upper bounds must be examined separately for every vector. See the following code fragment:

```
int             i,j,k;
ERX_HARRAY      hArray;
ERX_ARRAY_INDEX uArrayIndex[3];
.....
for (i=0;i<erxArrayGetBounds(hArray,(unsigned int)1,NULL);i++)
{
        uArrayIndex[0] = i;
        for (j=0;j<erxArrayGetBounds(hArray,(unsigned int)2,uArrayIndex);j++)
        {
                uArrayIndex[1] = j;
            for (k=0;k<erxArrayGetBounds(hArray,(unsigned int)3,uArrayIndex);k++)
              {
                      Data = ...;

                   uArrayIndex[2] = k;
                      rc = erxArraySetElement(
                              hArray,
                              uArrayIndex,
                              &Data
                      );
              }
        }
}
```

To examine the data received from the server the same scheme can be used:

```
int             i,j,k;
ERX_HARRAY      hArray;
ERX_ARRAY_INDEX uArrayIndex[3];
.....
for (i=0;i<erxArrayGetBounds(hArray,(unsigned int)1,NULL);i++)
{
        uArrayIndex[0] = i;
        for (j=0;j<erxArrayGetBounds(hArray,(unsigned int)2,uArrayIndex);j++)
        {
                uArrayIndex[1] = j;
            for (k=0;k<erxArrayGetBounds(hArray,(unsigned int)3,uArrayIndex);k++)
              {
                      uArrayIndex[2] = k;
                      rc = erxArrayGetElement(
                              hArray,
                              uArrayIndex,
                              &Data
                      );

                  ... = Data;
              }
        }
}
```

Remove previously created array instances (if they are no longer needed) with the function `erxArrayFree`.

Usage with EntireX RPC Server

When the server is called, all array instances are allocated by the EntireX RPC Runtime and EntireX RPC Server.

The data sent by the client can be examined in the server program the same way the client examines data upon server reply.

The upper bounds of the array instance can be changed with the `erxArrayRedimAll` or `erxArrayRedimVector` function before setting any return data:

- When you use the `erxArrayRedimAll` function, the result will be an array in the form of a vector (1-dimensional), a square (2-dimensional) or a cube (3-dimensional). Thus all vectors of a dimension have the same upper bounds. Subsequently with the `erxArrayRedimVector` the square or cube can be deformed.

- You can also remove all elements of the unbounded array. This results in an unbounded array with no elements when setting the bounds parameter of `erxArrayRedimAll` to "0". Afterwards you can set the upper bounds of the first dimension with `erxArrayRedimVector`. Subseqeuently loop through this dimension and set any vector of the second dimension also using `erxArrayRedimVector`. If it is a 3-dimensional array, do the same with the third dimension. You cannot change any type, attribute, length or dimension. Only upper bounds can be changed.

Data to be returned can be assigned using the function `erxArraySetElement` the same way the client does before send.

⚠️ **Important:** Do not remove any array instances in server programs.

Using Conversational RPC

It is assumed that you are familiar with the concepts of conversational and non-conversational RPC. See *Common Features of Wrappers and RPC-based Components*.

≫ **To use conversational RPC**

1   Open a conversation with the `ERXConnect` function call (see `ERXConnect`). Save the server address `ERX_SERVER_ADDRESS` and reuse it for the complete RPC conversation.

2   Issue your RPC requests as within non-conversational mode using the generated interface objects. Different interface objects can participate in the same RPC conversation.

≫ **To abort a conversational RPC communication**

■ Abort an RPC conversation unsuccessfully with the function call `ERXDisconnect`.

≫ **To close and commit a conversational RPC communication**

■ Close the RPC conversation successfully with the function call `ERXDisconnectCommit`.

🛑 **Caution:** Natural and EntireX RPC servers behave differently when ending an RPC conversation.

See *Conversational RPC*.

Using EntireX Security

EntireX C Wrapper Applications which require security can use the security services offered by EntireX Security. See *EntireX Security* for a general overview.

≫ **To use EntireX Security**

1   Specify a user ID and password in the parameters `*pUserId` and `*pPassword` of the `ERX_CLIENT_IDENTIFICATION` structure.

2   Set security with the function `ERXSetBrokerSecurity` to force a secure call to a broker running with EntireX Security. You can use the same values as for broker ACI field `KERNELSECURITY`. See `KERNELSECURITY` under *Broker ACI Fields*. The function works together with any broker kernel version that supports EntireX Security, regardless of the ACI version used.

📄 **Note:** The broker's security token is maintained inside the EntireX RPC Runtime on a per-thread basis, see *Using the RPC Runtime*. If you are communicating with more than one broker in a single thread:

■ you have to save the broker's security token provided in the `ERX_CLIENT_IDENTIFICATION` structure after an `ERXLogon` function call

■ you have to provide the correct previously saved Broker's security token with the `ERXSetSecurityToken` function to the RPC Runtime before calling one of the following functions:

   ■ `ERXCall`
   ■ `ERXConnect`
   ■ `ERXDisconnect`
   ■ `ERXDisconnectCommit`
   ■ `ERXLogon`

- `ERXLogoff`
- `ERXTerminateServer`
- `ERXIsServing`
- `ERXWait`

Other functions are executed locally and do not communicate with the Broker, the Broker's security token is not required.

Using the Broker and RPC User ID/Password

EntireX supports two user ID/password pairs: a broker user ID/password pair and an (optional) RPC user ID/password pair sent from RPC clients to RPC servers. With EntireX Security, the broker user ID/password pair can be checked for authentication and authorization.

The RPC user ID/password pair is designed to be used by the receiving RPC server. This component's configuration determines whether the pair is considered or not. Useful scenarios are:

- Credentials for Natural Security
- Impersonation in the respective RPC Server documentation
- Web Service Transport Security with the RPC Server for XML/SOAP, see *XML Mapping Files*
- Service execution with client credentials for EntireX Adapter Listeners, see *Configuring Listeners*
- etc.

Sending the RPC user ID/password pair needs to be explicitly enabled by the RPC client. If it is enabled but no RPC user ID/password pair is provided, the broker user ID/password pair is inherited to the RPC user ID/password pair.

With `cNaturalLogon` (see below) sending the RPC user ID/password pair is enabled for C RPC clients. If you do so, we strongly recommend *Using SSL/TLS*.

> **To use the broker and RPC user ID/password**

1   Specify a broker user ID and optional broker password in the parameters `pUserId` and `pPassword` of the `ERX_CLIENT_IDENTIFICATION` structure.

2   Set the medium `ERX_TM_BROKER_LIBRARY` in the `ERX_SERVER_ADDRESS` structure.

3   Set the parameter `cNaturalLogon` to "ERX_NATURAL LOGON_YES" in the `ERX_SA_BROKER_LIBRARY` structure to enable sending the RPC user ID/password pair.

4   If different user IDs and/or passwords are used for broker and RPC, use the parameters `pRpcUserId` and `pRpcPassword` to provide a different RPC user ID/password pair.

5   By default the library name sent to the RPC server is retrieved from the IDL file (see `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation). The library name can be overwritten. This is useful if communicating with a Natural RPC server. Specify a library in the `ERX_SA_BROKER_LIBRARY` structure in the `szLibraryName` parameter.

If you need to log on to a Natural library without setting security credentials, refer to *Logon to Natural Library*.

Using SSL/TLS

RPC client applications can use Secure Sockets Layer/Transport Layer Security (SSL/TLS) as the transport medium. The term "SSL" in this section refers to both SSL and TLS. RPC-based clients are always SSL clients. The SSL server can be either the EntireX Broker or Direct RPC in webMethods Integration Server (IS inbound). For an introduction see *SSL/TLS, HTTP(S), and Certificates with EntireX* in the platform-independent Administration documentation.

With the C Wrapper, the SSL parameters (e.g. certificates) are provided in the `ERX_CLIENT_IDENTIFICATION` structure.

≫ **To use SSL**

1   To operate with SSL, certificates need to be provided and maintained. Depending on the platform, Software AG provides default certificates, but we strongly recommend that you create your own. See *SSL/TLS Sample Certificates Delivered with EntireX* in the EntireX Security documentation.

2   Specify the Broker ID, using one of the following styles:

   ▪ *URL Style*, for example:

   ```
   ssl://localhost:2010
   ```

   ▪ *Transport-method Style*, for example:

   ```
   ETB024:1609:SSL
   ```

   If no port number is specified, port 1958 is used as default.

3   Specify SSL parameters on the `pSSLParameter` parameter in the `ERX_CLIENT_IDENTIFICATION` structure, for example:

```
"VERIFY_SERVER=N&TRUST_STORE=c:\\certs\\CaCert.pem"
```

If the SSL client checks the validity of the SSL server only, this is known as *one-way SSL*. The mandatory `trust_store` parameter specifies the file name of a keystore that must contain the list of trusted certificate authorities for the certificate of the SSL server. By default a check is made that the certificate of the SSL server is issued for the hostname specified in the Broker ID. The common name of the subject entry in the server's certificate is checked against the hostname. If they do not match, the connection will be refused. You can disable this check with SSL parameter `verify_server=no`.

If the SSL server additionally checks the identity of the SSL client, this is known as *two-way SSL*. In this case the SSL server requests a client certificate (the parameter `verify_client=yes` is defined in the configuration of the SSL server). Two additional SSL parameters must be specified on the SSL client side: `key_store` and `key_passwd`. This keystore must contain the private key of the SSL client. The password that protects the private key is specified with `key_passwd`.

The ampersand (&) character cannot appear in the password.

SSL parameters are separated by ampersand (&). See also *SSL/TLS Parameters for SSL Clients*.

4   Make sure the SSL server to which the RPC component connects is prepared for SSL connections as well. The SSL server can be EntireX Broker, Broker SSL Agent, or Direct RPC in webMethods Integration Server (IS inbound). See:

  ■ *Running Broker with SSL/TLS Transport* in the platform-specific Administration documentation

  ■ Broker SSL Agent in the platform-specific Administration documentation

  ■ *Support for SSL/TLS* in the EntireX Adapter documentation (for Direct RPC)

Using Compression

EntireX C Wrapper Applications may compress the messages sent to and received from the broker.

≫ **To use compression**

■   Specify a compression level in the `ERX_CLIENT_IDENTIFICATION` structure. Possible compression levels are identical to the broker ACI field `COMPRESSION`.

Using Internationalization with the C Wrapper

For RPC clients, the RPC runtime itself does not convert your application data. The application's data is shipped and received as given. The application must ensure the encoding of the data matches the codepage sent to the broker.

Under the Windows operating system:

1. The RPC runtime assumes by default the data is given in the encoding of the Windows ANSI codepage configured for your system. A codepage identifier of this Windows ANSI codepage is automatically transferred to tell the broker how the data is encoded, even if no codepage name is provided with function `ERXSetCodepage`, see *API Function Descriptions*.

2. If you want to adapt the Windows ANSI codepage, see the Regional Settings in the Windows Control Panel and your Windows documentation.

3. If you want to encode the data different to your Windows ANSI codepage, provide the data in the desired encoding and provide the codepage name with function `ERXSetCodepage`. During receive, decode the data if necessary.

Under all other operating systems:

- The RPC runtime does not automatically send a codepage identifier to the broker.

- It is assumed the broker's locale string defaults match. See *Locale String Mapping*. If they do not match, provide the correct codepage name with function `ERXSetCodepage`.

Enable character conversion in the broker by setting the service-specific attribute `CONVERSION` to `"SAGTRPC"`. See also *Configuring ICU Conversion* under *Configuring Broker for Internationalization* in the platform-specific Administration documentation. More information can be found under *Internationalization with EntireX*.

8 Writing RPC Clients for the RPC-ACI Bridge with the C Wrapper

The EntireX RPC-ACI Bridge allows standard RPC clients to communicate with an ACI server. The RPC-ACI Bridge transforms RPC requests from clients into ACI messages.

⟩ To write a C client

■ Follow the instructions under *Using the C Wrapper for the Client Side*.

The RPC-ACI Bridge reports errors from the RPC server side and the ACI side to the RPC clients. Errors from the ACI side include errors by the Broker for ACI.

The RPC-ACI Bridge reports the same error classes and error codes for the RPC server side as the RPC Server for Java. The RPC-ACI Bridge reports errors of the ACI side in a client-specific way as error 10010007 (internal error of the RPC protocol). The detailed message of the error has the form `RPCACIBridge: < text >`, where *text* indicates the cause of the error. See *Message Class 1018 - EntireX RPC-ACI Bridge* for additional information.

9   Writing Callable RPC Servers with the C Wrapper

The callable RPC server interface enables you to write your own RPC Server. This offers the possibility to integrate RPC servers into third-party systems, as well as to call target servers in programming languages other than C by wrapping them. The programming language for writing a callable RPC Server is C.

Introduction to Callable RPC Servers

The callable RPC server consists of a function containing a loop for recurring RPC execution using a callback technique for user interaction. The function performs all of the necessary communication with the broker such as logon and logoff, service registration, receive and send. The behavior of the callable RPC server can be configured with a server configuration file.

Writing a Callable RPC Server

The main part of the callable RPC server is the C Wrapper runtime function `ERXServingCallback`. This function manages Broker communication as well as the marshalling and unmarshalling of RPC requests using callbacks. The function consists of a loop for RPC execution using callback technique for user interaction. The behavior of `ERXServingCallback` is driven by a configuration file where you set the necessary broker parameters etc. See *Configuring the RPC Server* under *Administering the RPC Server for C*. In the example below, the name of the configuration file is passed as a parameter to the callable RPC server and given to the `ERXServingCallback` function.

The callback function `ERX_Callback_SERVER_CALL` is called when an RPC request has to be executed. Implement a call to your target server within this callback event. See *Writing the Callback*. The interface of the unmarshalled data given to the callback is compatible with the generated server interface objects for the programming language C. See *Software AG IDL to C Mapping*. Upon return from the callback function, the same applies to parameters replied to the client.

`ERXServingCallback` requires registration of the events before getting control of them. Use `ERXRegisterEvent` with the Event ID `ERX_EVENT_SERVER_CALL` to register the callback function `ERX_Callback_SERVER_CALL` for this purpose.

The criteria to break/stop the RPC execution loop and give control back to the caller can be configured with the configuration file parameter `endworkers`. See *Configuring the RPC Server* under *Administering the RPC Server for C*. The example below implements a single RPC worker thread within the main function which is ended by a shutdown from outside. Hence use `endworkers=never` as the setting for the configuration file parameter `endworkers` for the example below. This ensures the RPC execution loop is not stopped by broker timeouts or after an RPC request is executed, etc. Use one of the usual ways to stop RPC servers on your platform to stop the callable RPC server. See *Break/Stop the RPC Execution Loop* for more information.

As a general rule before using the RPC C runtime at all, every worker thread must be registered with it using the ERXRegister function. ERXRegister is therefore the first call to the RPC C runtime and ERXUnregister the last. See *Using the RPC Runtime*.

Example

```
void main( int argc, char *argv[ ])
{
   int            bRuntimeRegistered = 0;
   char           myConfigurationFile[512] = "..\\server\\server.cfg";
   void *         myParms = NULL;
   ERXeReturnCode rc = ERX_S_SUCCESS;
   ERX_ERROR_INFO ErrorInfo;
   memset(&ErrorInfo,'\0',sizeof(ErrorInfo));

   /* Treat the input Parameter */
   if( argc == 2)
   {
      strncpy( myConfigurationFile, argv[ 1 ], 512 );
   }
   printf("\nEntireX callable RPC server is running:\n"
           "   Configuration File: %s\n",
             myConfigurationFile);

   /* Register to EntireX RPC Runtime */
   rc = ERXRegister( ERX_CURRENT_VERSION );
   if ( ERX_FAILED(rc ))
   {
      PrintReturnCode(rc,&ErrorInfo);
      goto done; /* ===> */
   }
   bRuntimeRegistered = 1;

   /* Register the Callback Event */
   rc = ERXRegisterEvent(ERX_EVENT_SERVER_CALL,
                          myERX_Callback_SERVER_CALL);
   if ( ERX_FAILED(rc))
   {
       PrintReturnCode(rc,&ErrorInfo);
       goto done; /* ===> */
   }

   /* Execute the Callable RPC Server */
   rc = ERXServingCallback( myConfigurationFile,
                             myParms,
                             (ERX_CF_NOTHING)
   );
   PrintReturnCode(rc,&ErrorInfo);

done:
   if (bRuntimeRegistered == 1)
```

```
    {
        ERXUnregister();
    }
    return;
}
```

Writing the Callback

This very simple example of a callback implementation uses the `szLibraryName` and `szProgramName` from the `ERX_CALL_INFORMATION_BLOCK` to select requests for the library named `EXAMPLE` and the programs named `CALC` and `SQUARE` in a hard-wired fashion. Other RPC requests are rejected with appropriate error messages. The library and program names correspond to the names given in the IDL file of the calling client.

The focus here is not to show how functions can be called dynamically. Dynamic calling depends on the possibilities of your implementation platform and support by the programming languages in use.

The interface of the unmarshalled data given to the callback is compatible with the generated server interface objects for the programming language C. See *Software AG IDL to C Mapping*. Upon return from the callback function, the same applies to parameters expected by and replied to the client.

- For the IDL data type A, null terminated strings are supplied and expected (corresponding to `DATA_CONV_A=1`).

- For the IDL data types N and P, double is supplied and expected (corresponding to `DATA_CONV_NP=1`).

Returning Errors

With the structure `ERX_ERROR_INFORMATION` either success or failure must always be returned.

User-specific Data

The callable RPC server supports a concept of user-specific data. With this feature it is possible to pass a pointer through the `ERXServingCallback` function directly into the callback. The pointer `myParms`, second parameter of the `ERXServingCallback` in the example above, is available "as is" in the callback here in the first parameter as pointer `pUserInfo`. It can be used, for example, to provide a pointer to a memory location with user-specific data.

Example

```
void myERX_Callback_SERVER_CALL
(
   void                        * pUserInfo,
   ERX_CLIENT_IDENTIFICATION   * pClientInformation,
   ERX_CALL_INFORMATION_BLOCK  * pCallInformation,
   void                        * pParameterArea,
   ERX_ERROR_INFO              * pReturnInfo
)
{
   ERXeReturnCode              rc = ERX_S_SUCCESS;
   printf("\nThis is Callback_SERVER_CALL, serving for %s,%s \n",
          pCallInformation->Callee.szLibraryName,
          pCallInformation->Callee.szProgramName);

   if (strcmp(pCallInformation->Callee.szLibraryName,"EXAMPLE") == 0)
   {
      if (strcmp(pCallInformation->Callee.szProgramName,"CALC") == 0)
      {
         S_CALC  *pParm = (S_CALC *) pParameterArea;
         /* Execute Function */
         pParm->function_result = CALC( pParm->operation,
                                        pParm->operand_1,
                                        pParm->operand_2 );
      }
      else if (strcmp(pCallInformation->Callee.szProgramName,"SQUARE")== 0)
      {
         S_SQUARE  *pParm = (S_SQUARE *) pParameterArea;

         /* Execute Function */
         SQUARE( pParm->operand, &(pParm->result) );

      }
      else
      {
         rc = ERX_E_RPC_CALLEE_NOT_FOUND;
      }
   }
   else
   {
     rc = ERX_E_RPC_LIBRARY_NOT_FOUND;
   }
   pReturnInfo->rc = rc;
   return;
}
```

Break/Stop the RPC Execution Loop

The RPC execution loop should normally run continuously until the RPC server is shut down from outside. With the setting of the configuration file parameter `endworkers` you can configure when the RPC execution loop is stopped and control is given back to the caller. See *Configuring the RPC Server* under *Administering the RPC Server for C*.

The following table explains the `endworkers` parameter.

Value	Explanation
N	**Never** The callable RPC server's function `ERXServingCallback` breaks/stops the RPC execution loop only if a normal shut down of the RPC server takes place. This setting makes sense with a simple callable RPC server (see ***Writing a Callable RPC Server***).
T	**Timeout** The callable RPC server's function `ERXServingCallback` breaks/stops the RPC execution loop if ■ a normal shutdown of the RPC server takes place. ■ the time specified by the timeout server parameter has elapsed and no further new RPC request or RPC conversation was active. See *Configuring the RPC Server* under *Administering the RPC Server for C*. This setting makes sense when working with a scalable number of worker threads. See ***Scalable Number of Worker Threads***.
I	**Immediately** The callable RPC server's function `ERXServingCallback` breaks/stops the RPC execution loop if ■ a normal shut down of the RPC server takes place. ■ immediately after execution of an RPC request or complete RPC conversation. With this setting you receive control in your callable RPC server after every RPC request or RPC conversation. See ***Writing a Callable RPC Server***. You can, for example, use this setting for logging purposes and put a repeat loop around the `ERXServingCallback` function.

Scalable Number of Worker Threads

This section provides some hints on how to implement a callable RPC server with a scalable number of worker threads. This is a more complex server with the ability to clone worker threads to satisfy a high load of client requests.

■ Implement a main function registering as an attach server by the broker using `REGISTER`, `OPTION=ATTACH`. When this server receives attach service requests for clients waiting to be served, start a suitable number of worker threads. See *Implementing an Attach Server*.

- Implement a callable RPC server and its callback to be attached in a thread as described under *Writing a Callable RPC Server* and *Writing the Callback*.

- Use `endworkers=timeout` for the configuration file parameter `endworkers`, if you wish to

  - implement a server that does not exit after the first conversation

  - reduce the number of servers when they are no longer needed

  Use `endworkers=immed` if you wish to

  - implement a server that handles only one client for one conversation

See *Implementing Servers Started by an Attach Server* under *Writing Applications: Attach Server* in the ACI Programming documentation for more details.

III   Reliable RPC for C Wrapper

10 Reliable RPC for C Wrapper

Introduction to Reliable RPC

In the architecture of modern e-business applications (such as SOA), loosely coupled systems are becoming more and more important. Reliable messaging is one important technology for this type of system.

Reliable RPC is the EntireX implementation of a reliable messaging system. It combines EntireX RPC technology and persistence, which is implemented with units of work (UOWs).

■ Reliable RPC allows asynchronous calls ("fire and forget")

■ Reliable RPC is supported by most EntireX wrappers

■ Reliable RPC messages are stored in the Broker's persistent store until a server is available

■ Reliable RPC clients are able to request the status of the messages they have sent
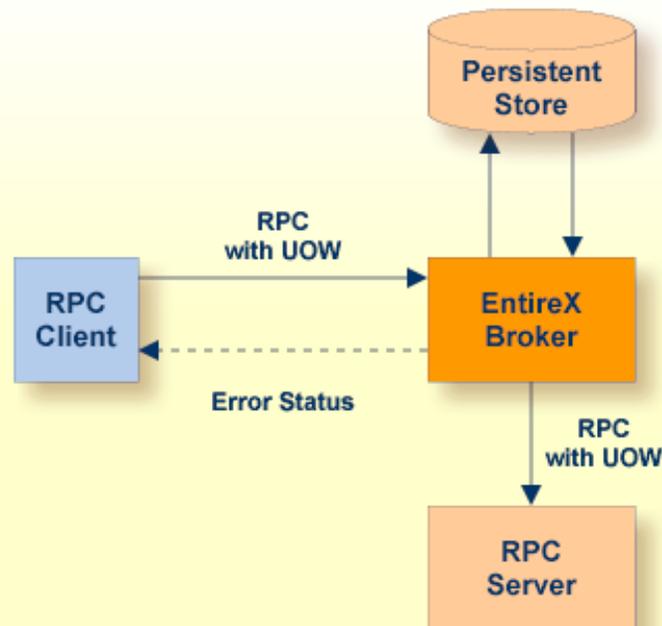
Reliable RPC is used to send messages to a persisted Broker service. The messages are described by an IDL program that contains only `IN` parameters. The client interface object and the server interface object are generated from this IDL file, using the EntireX C Wrapper.

Reliable RPC is enabled at runtime. The client has to set one of two different modes before issuing a reliable RPC request:

■ `AUTO_COMMIT`

■ `CLIENT_COMMIT`

While `AUTO_COMMIT` commits each RPC message implicitly after sending it, a series of RPC messages sent in a unit of work (UOW) can be committed or rolled back explicitly using `CLIENT_COMMIT` mode.

The server is implemented and configured in the same way as for normal RPC.

Writing a Client

This section shows a reliable RPC client for `CLIENT_COMMIT` mode. All methods for reliable RPC are defined in `erx.h`. The methods applicable to reliable RPC as described under *API Function Descriptions for Reliable RPC* are:

- `ERXGetReliableState`

- `ERXSetReliableState`

- `ERXReliableCommit`

- `ERXReliableRollback`

- `ERXGetReliableID`

- `ERXGetReliableStatus`

Step 1: Base Declarations Required by the C Wrapper

Step 1a: Include the Generated Header File

Define the generated client header file. This header file includes the RPC runtime header file erx.h and defines structures and prototypes for your RPC messages.

```
/* include generated header file */
#include "cmail.h"
```

Step 1b: Define Global Variables to Communicate with the Client Interface Objects

```
/* Required global variables for the CLIENT interface */
ERXeReturnCode           ERXrc;
ERX_CLIENT_IDENTIFICATION  ERXClient;
ERX_SERVER_ADDRESS         ERXServer;
ERX_SERVER_ADDRESS         ERXServerDefault;
ERXCallId                  ERXCallID;
ERX_ERROR_INFO             ERXErrorInfo;
```

Step 2: Required Settings for the C Wrapper

Step 2a: Identify the User with a Broker User ID

For *Implicit Logon*, if required in your environment, the client password can be given here. It is provided then through the RPC interface object call.

```
/* set client identification */
memset( &ERXClient, 0, sizeof(ERXClient) );
ERXClient.pUserId   = "ERX-USER";
ERXClient.pPassword = "ERX_PASS";
```

Step 2b: Set the Broker and Service to be Called

Your application will wait a maximum of 55 seconds for a server response. If the server does not answer within this period, the broker gives your program control again with an error code 00740074.

```
ERXServer.Medium = ERX_TM_BROKER_LIBRARY;
ERXServer.ulTimeOut = 55;

/* set Broker-Id, server-name, class-name and service-name */
strcpy( (char*) ERXServer.Address.BROKER.szEtbidName,   "ETB001" );
strcpy( (char*) ERXServer.Address.BROKER.szServerName,  "SRV1" );
strcpy( (char*) ERXServer.Address.BROKER.szClassName,   "RPC" );
strcpy( (char*) ERXServer.Address.BROKER.szServiceName, "CALLNAT" );
```

Step 3: Register with the RPC Runtime

As a general rule, you have to register the RPC runtime before you use it. After registration, the RPC runtime holds information on a per-thread basis. See also *Using the RPC Runtime*.

```
/* register to the RPC runtime */
ERXrc = ERXRegister(ERX_CURRENT_VERSION );
If ( ERX_FAILED( ERXrc ) )
{
/* code for  error handling */
}
```

Step 4: Broker Logon

We logon by EntireX Broker.

```
/* Logon to EntireX Broker Middleware */
ERXrc = ERXLogon( &ERXClient,
                  ERXServer.Address.BROKER_Library.szEtbidName );
if(ERX_FAILED(ERXrc))
{
/* code for  error handling */
}
```

Step 5: Set Reliable-State

Before reliable RPC can be used, the reliable state must be set to either ERX_RELIABLE_CLIENT_COMMIT
or ERX_RELIABLE_AUTO_COMMIT.

```
/* Set reliable RPC state to client commit */
ERXrc = ERXSetReliableState(ERX_RELIABLE_CLIENT_COMMIT);
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 6: Send the RPC Message

The RPC interface object SENDMAIL is called as a C procedure. See *Calling Servers as Procedures
or Functions*.

```
/* do the remote procedure call */
SENDMAIL( gTo, gSubject, gText);
```

Step 7: Get the Reliable RPC Message ID

Get the reliable RPC message ID before you commit any reliable RPC messages, otherwise the
reliable ID will be lost and checking for the RPC message status will not be possible.

```
/* Get the reliable ID */
ERXrc = ERXGetReliableID( &ERXServer, pReliableID );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 8: Check the Reliable RPC Message Status

After the reliable RPC message ID has been got, you can query the status of the reliable RPC message. This is a separate call independent of any reliable RPC messages, so we use the default server connection (`ERXServerDefault`). Valid reliable RPC message states can be found in header file *etbcdef.h*. See *Broker ACI Control Block Definition*.

See *Using Persistence and Units of Work*, *Understanding UOW Status* and *Broker UOW Status Transition* in the platform-independent Administration documentation for more information.

```
/* Check the reliable RPC message status */
ERXrc = ERXGetReliableStatus( &ERXClient,
                              &ERXServerDefault,
                              pReliableID,
                              pReliableStatus );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 9: Send a Second RPC message

Send a second reliable RPC message.

```
/* do the remote procedure call */
SENDMAIL( gTo, gSubject, gText);
```

Step 10: Commit Both Reliable RPC Messages

Now we commit both reliable RPC messages. This will deliver all reliable RPC messages to the server if it is available.

```
/* Commit all made reliable RPC messages */
ERXrc = ERXReliableCommit( &ERXServer );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 11: Reset ERX_SERVER_ADDRESS

For reliable RPC, the ERX_SERVER_ADDRESS will be overwritten by the RPC runtime, so it is necessary to reset the ERX_SERVER_ADDRESS structure with the required values.

```
/*
 * After an ERXReliableCommit we have to use a new server connection
 * so we restore our default server connection for further calls.
 */
memcpy(&ERXServer, &ERXServerDefault, sizeof(ERX_SERVER_ADDRESS));
```

Step 12: Check the Reliable RPC Message Status

To determine that reliable RPC messages are delivered, we query the reliable RPC message status again. See also *Step 8* above.

Step 13: Send a Third RPC message

Send a third reliable RPC message.

```
/* do the remote procedure call */
SENDMAIL( gTo, gSubject, gText);
```

Step 14: Get the Reliable RPC Message ID

Get the reliable RPC message ID. See also *Step 7*.

```
/* Get the reliable ID */
ERXrc = ERXGetReliableID( &ERXServer, pReliableID );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 15: Check the Reliable RPC Message Status

After the reliable RPC message ID has been got, query the status of the reliable RPC message again.

```
/* Check the reliable RPC message status */
ERXrc = ERXGetReliableStatus( &ERXClient,
                              &ERXServerDefault,
                              pReliableID,
                              pReliableStatus );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 16: Roll back the Third Message

Roll back the current reliable RPC message.

```
/* Roll back Message 3 */
ERXrc = ERXReliableRollback( &ERXServer );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 17: Check the Reliable RPC Message Status

After rolling back the reliable RPC message, query the status of the reliable RPC message.

```
/* Get the reliable RPC message status */
ERXrc = ERXGetReliableStatus( &ERXClient,
                              &ERXServerDefault,
                              pReliableID,
                              pReliableStatus );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 18: Broker Logoff

Log off from EntireX Broker.

```
/* Logoff from EntireX Broker Middleware */
ERXrc = ERXLogoff( &ERXClient,
                   ERXServerDefault.Address.BROKER_Library.szEtbidName );
if ( ERX_FAILED( ERXrc ) )
{
/* code for  error handling */
}
```

Step 19: Deregister with the RPC Runtime

As a general rule, after using the RPC runtime you should unregister from it. This will free all resources held by the RPC runtime for the caller. See *Using the RPC Runtime* for more information.

```
/* unregister to the RPC runtime */
ERXUnregister();
```

Writing a Client using AUTO COMMIT

This section gives some hints for reliable RPC `AUTO_COMMIT` mode. It is not a complete example and shows only the correct order of reliable RPC method calls. The reliable ID to check the message status must be retrieved immediately after the reliable RPC message is sent and before any other RPC runtime calls - otherwise the reliable ID is lost and retrieving the message status is not possible.

```
/* Initialize pERXServer */
...

/*
 * After initializing pERXServer  with your connection settings (broker ID,
 * server-name, class-name, service-name) create a copy of it
 * (pERXDefaultServer). Use this copy to resolve the reliable status after
 * a reliable RPC message.
 */
memcpy(pERXServer, pERXDefaultServer, sizeof(ERX_SERVER_ADDRESS));

...

/* Set reliable state to AUTO_COMMIT */
ERXSetReliableState( ERX_RELIABLE_AUTO_COMMIT );

...



/* reliable RPC message 1 */
SENDMAIL( gTo, gSubject, gText );
/*
 * The reliable ID must be resolved directly
 * after a reliable RPC message
 */
ERXGetReliableID( pERXServer, pReliableID );




...

/* Resolve the reliable status */
ERXGetReliableStatus( pERXClient, pERXDefaultServer, pReliableID,
                      pReliableStatus );

...
```

```
/* For a second AUTO_COMMIT RPC message, use a new server connection */
memcpy(pERXServer, pERXDefaultServer, sizeof(ERX_SERVER_ADDRESS));


...

/* reliable RPC message 2 */
SENDMAIL( gTo, gSubject, gText );
/*
 * The reliable ID must be resolved directly
 * after a reliable RPC message
 */
ERXGetReliableID( pERXServer, pReliableID );


...

/* Resolve the reliable status */
ERXGetReliableStatus( pERXClient, pERXDefaultServer, pReliableID,
                      pReliableStatus );

...
```

Writing a Server

There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it returns an error code greater than zero. This causes the transaction (unit of work inside the Broker) to be cancelled, and the error code is written to the user status field of the unit of work.

For writing reliable RPC servers, see *Using the C Wrapper for the Server Side (z/OS, UNIX, Windows, BS2000)*.

To execute a reliable RPC service with an RPC server, the parameter `logon` must be set to `YES`, see *Configuring the RPC Server* in the BS2000 Administration documentation or *Configuring the RPC Server* under *Administering the RPC Server for C*

Broker Configuration

A Broker configuration with `PSTORE` is recommended. This enables the Broker to store the messages for more than one Broker session. These messages are still available after Broker restart. The attributes `STORE`, `PSTORE`, and `PSTORE-TYPE` in the Broker attribute file can be used to configure this feature. The lifetime of the messages and the status information can be configured with the attributes `UOW-DATA-LIFETIME` and `UOW-STATUS-LIFETIME`. Other attributes such as `MAX-MESSAGES-IN-UOW`, `MAX-UOWS` and `MAX-UOW-MESSAGE-LENGTH` may be used in addition to configure the units of work. See *Broker Attributes*.

The result of the procedure `ERXGetReliableStatus` depends on the configuration of the unit of work status lifetime in the EntireX Broker configuration. If the status is not stored longer than the message, the procedure returns the error code 00780305 (`no matching UOW found`).

IV   API Function Descriptions for the C Wrapper

This chapter describes the API Functions available for the C Wrapper.

API Function Descriptions

The API for the RPC C runtime is defined in the following header file:

```
#include <erx.h>
```

ERXCall

Remote Procedure Call - Conversationless or Conversational.

Syntax

```
extern ERXeReturnCode ERXAPI ERXCall(
      ERX_CLIENT_IDENTIFICATION  ERXPTR *pClient,
      const ERX_SERVER_ADDRESS   ERXPTR *pServer,
      ERXCallId                  ERXPTR *pCallId,
      const ERX_CALL_INFORMATION_BLOCK ERXPTR *pCallInfoBlock,
      void                       ERXPTR *(ERXPTR pParameterBlock)[],
      const ERXeControlFlags          fFlags);
```

Description

This performs a remote procedure call. It is used for both connectionless and connection-oriented calls:

■ **Connectionless Calls**
The server is identified by the `&Server` parameter

```
ERXCall(&Client,&Server,&CallId,&CIB,&Parameter,ERX_CF_NOTHING|ERX_CF_STRUCTURED);
```

■ **Connection-oriented Calls**
The server is identified by referring to an established connection: `&Server_Connection` parameter returned by an `ERXConnect` call, as follows:

```
ERX_SERVER_ADDRESS Server_Connection
...
ERXConnect(&Client,&Server,&Server_Connection);
ERXCall(NULL,&Address,&CallId,&CIB,&Parameter,ERX_CF_NOTHING|ERX_CF_STRUCTURED);
...
ERXDisconnectCommit(&Address);
```

We suggest using `ERX_TM_BROKER_LIBRARY` as the medium of the server address (see `ERX_SERVER_ADDRESS`). Appropriate values must be provided for all fields. See also `ERXConnect`.

The called procedure is identified by the call information block (see `ERX_CALL_INFORMATION_BLOCK`), which contains its name and location (library). The call information block also points to an array of parameter definitions (`ERX_PARAMETER_DEFINITION_V3`). The parameter definition contains the type, size, count of indices, occurrences in the respective dimensions, the addresses of the parameters, etc. The `pParameterBlock` array contains the pointers to each parameter's data.

The Software AG IDL Compiler (with the template files *client.tpl* and *server.tpl*) generates interface objects in which the parameter data is grouped in consecutive storage. This is referred to as

structured mode and is indicated by specifying the `ERX_CF_STRUCTURED` flag as part of the `fFlags` `ERXCall`. In structured mode, only one address, the address of the structure, is passed in the `pParameterBlock` array. That is, the `pParameterBlock` array only has one entry.

For information on the messages, see *Error Messages and Codes*.

Parameters

pClient
    in out: The client's identification, see `ERX_CLIENT_IDENTIFICATION`.

pServer
    in: The address of the server, see `ERX_SERVER_ADDRESS`.

pCallId
    out: The `CallId` returned to the caller, used in asynchronous communication to receive the reply with the `ERXWait` API call.

pCallInfoBlock
    in: The description of the program to call and its parameter definition, see
    `ERX_CALL_INFORMATION_BLOCK`

pParameterBlock
    in out: The array of pointers to the actual parameter data.

fFlags
    in: `ERX_CF_STRUCTURED`, i.e. the parameters are collected in one data structure.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
00020002	ERX_ETB_USER_DOES_NOT_EXIST
0003nnnn	ERX_ETB_CONVERSATION_ENDED
00070007	ERX_ETB_SERVICE_NOT_AVAILABLE
00740074	ERX_ETB_WAIT_TIMEOUT
02150148	ERX_ETB_BROKER_NOT_AVAILABLE

Related Functions

ERXConnect
ERXDisconnect
ERXDisconnectCommit

ERX_Callback_SERVER_CALL

Syntax

```
void    ERX_Callback_SERVER_CALL (
        void                         *pUserInfo,
        ERX_CLIENT_IDENTIFICATION    *pClientInformation,
        ERX_CALL_INFORMATION_BLOCK   *pCallInformation,
        void                         *pParameterArea,
        ERX_ERROR_INFO               *pReturnInfo
);
```

Description

This callback function (see *Writing the Callback*) is called by the Callable RPC Server (see *Writing Callable RPC Servers with the C Wrapper*). Success or failure is returned with the structure `ERX_ERROR_INFO`.

Parameters

*pUserInfo

in: User specific data. The data is provided "as is" in the function `ERXServingCallback`. It can be used to provide a pointer to a memory location with user specific data in the callback.

*pClientInformation

in: The client's identification such as user ID, etc., see `ERX_CLIENT_IDENTIFICATION`.

*pCallInformation

in: The description of the library and program to call and its parameter definition, see `ERX_CALL_INFORMATION_BLOCK`

*pParameterArea

in: IDL in and inout Parameters from the client. Upon return IDL out and inout parameters are replied back to the client. Parameters are provided and expected in a contiguous memory location.

*pReturnInfo

in: Returning success or errors from the callback function. Possible Return Codes to give back are: `ERX_E_RPC_LIBRARY_NOT_FOUND`
`ERX_E_RPC_CALLEE_NOT_FOUND`
`ERX_E_RPC_OUT_OF_MEMORY`
`ERX_E_RPC_ABNORMAL_TERMINATION`
`ERX_S_SUCCESS`

Related Functions

`ERXRegisterEvent`
`ERXServingCallback`

ERXConnect

Establish a conversation to the named server.

Syntax

```
extern ERXeReturnCode ERXAPI ERXConnect(
       ERX_CLIENT_IDENTIFICATION  ERXPTR *pClient,
       const ERX_SERVER_ADDRESS   ERXPTR *pServer,
       ERX_SERVER_ADDRESS         ERXPTR *pAddress
);
```

Description

Establishes an RPC conversation (connection) to the named server.

The information supplied covers the identification of the client, for example user ID and password, and the server address.

We suggest using `ERX_TM_BROKER_LIBRARY` as the medium of the server address. Appropriate values must be provided for all fields.

See *Using Conversational RPC* for more information.

For information on the messages, see *Error Messages and Codes*.

Parameters

pClient
  in out: The client's identification, see `ERX_CLIENT_IDENTIFICATION`.

pServer
  in: The address of the server, see `ERX_SERVER_ADDRESS`.

pAddress
  out: The connection ID returned to the caller. The pointers `pServer` and `pAddress` must not be the same. Otherwise an error will occur.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
00020002	ERX_ETB_USER_DOES_NOT_EXIST
00070007	ERX_ETB_SERVICE_NOT_AVAILABLE
00740074	ERX_ETB_WAIT_TIMEOUT
02150148	ERX_ETB_BROKER_NOT_AVAILABLE

Related Functions

ERXDisconnect
ERXDisconnectCommit

ERXDisconnect

Give up the conversation with Backout.

Syntax

```
extern ERXeReturnCode ERXAPI ERXDisconnect(
      ERX_SERVER_ADDRESS        ERXPTR *pAddress
);
```

Description

Aborts the specified RPC conversation (connection). In contrast to `ERXDisconnectCommit`, calling this function leads to an abnormal, unsuccessful end of the RPC Conversation. See *Using Conversational RPC* for more information.

For information on the messages, see *Error Messages and Codes*.

Parameters

pAddress
> in: The pointer to the connection to the RPC Server that is to be aborted. The connection ID contained in the `ERX_SERVER_ADDRESS` data structure was retrieved by a previous `ERXConnect` call.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
00020002	ERX_ETB_USER_DOES_NOT_EXIST
00070007	ERX_ETB_SERVICE_NOT_AVAILABLE
00740074	ERX_ETB_WAIT_TIMEOUT
02150148	ERX_ETB_BROKER_NOT_AVAILABLE

Related Functions

`ERXConnect`
`ERXDisconnectCommit`

ERXDisconnectCommit

Give up the conversation with Commit.

Syntax

```
extern ERXeReturnCode ERXAPI ERXDisconnectCommit(
      ERX_SERVER_ADDRESS        ERXPTR *pAddress
);
```

Description

Close the specified RPC conversation (connection). In contrast to `ERXDisconnect`, calling this function leads to a normal, successful end of the RPC Conversation. See *Using Conversational RPC* for more information.

For information on the messages, see *Error Messages and Codes*.

Parameters

pAddress

 in: The pointer to the connection to the RPC Server to close. The connection ID contained in the `ERX_SERVER_ADDRESS` data structure was retrieved by a previous `ERXConnect` call.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
00020002	ERX_ETB_USER_DOES_NOT_EXIST
00070007	ERX_ETB_SERVICE_NOT_AVAILABLE
00740074	ERX_ETB_WAIT_TIMEOUT
02150148	ERX_ETB_BROKER_NOT_AVAILABLE

Related Functions

ERXConnect
ERXDisconnect

ERXGetBrokerSecurity

Get the current setting of broker kernel security value.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetBrokerSecurity(
      char                  * pKernelSecurity
);
```

Description

With this function you retrieve the current settings for security set by a previously issued
ERXSetBrokerSecurity function call maintained internally by the RPC C runtime on a per-thread
basis. See *Using EntireX Security* for more information. For information on the messages, see *Error
Messages and Codes*.

Parameters

pKernelSecurity
    in

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXSetBrokerSecurity

ERXGetCodepage

Get the current setting of the codepage.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetCodepage(
      char szCodepage [ERX_MAX_CODEPAGE_LENGTH + 1]
);
```

Description

With this function you retrieve the current settings for the locale string set by a previously issued ERXSetCodepage function call maintained internally by the RPC C runtime on a per-thread basis. See *Using Internationalization with the C Wrapper* for more information. For information on the messages, see *Error Messages and Codes*.

Parameters

szCodepage
    in

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXSetCodepage

ERXGetContext

Get the current assigned context.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetContext(
      ERX_CONTEXT_BLOCK          ERXPTR ** ppContextBlock
);
```

Description

This function supports RPC clients in multithreaded environments. It is used to retrieve (thread-safe) RPC and broker context information, which was supplied by a preceding `ERXSetContext` call. See *Programming Multithreaded RPC Clients*.

For information on the messages, see *Error Messages and Codes*.

Parameters

ppContextBlock
   in out: Pointer to context block, see `ERX_CONTEXT_BLOCK`

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

`ERXSetContext`

ERXGetLastError

Get Information on Return Codes.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetLastError(
      ERX_ERROR_INFO              ERXPTR *pErrorInfo
);
```

Description

Retrieve information about the error that occurred last (the status of the last executed RPC C runtime function). When an API function is called, the error information is reset and, in case of error, the applicable information is placed in the error information structure. Exceptions to this rule are the functions ERXRegister and ERXUnregister, which only return the ERXeReturnCode. If ERXGetLastError itself is erroneous, the error information structure will be empty. For information on the messages, see *Error Messages and Codes*.

Parameters

pErrorInfo
out: A pointer to the data structure receiving the error information, see ERX_ERROR_INFORMATION.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXGetMessage

ERXGetMessage

Get Message Text.

Syntax

```
extern int ERXGetMessage(
      ERXeReturnCode  rc,
      char            *szMsg,
      unsigned int    ulMsg
);
```

Description

The function `ERXGetMessage` delivers the message text of the error codes in the following error classes. See also *Error Messages and Codes*:

- *Message Class 0001 - RPC C Runtime*
- *Message Class 1000 - RPC C Runtime System*
- *Message Class 1001 - RPC Protocol*
- *Message Class 1003 - Conversion*
- *Message Class 1004 - IDL Compiler*
- *Message Class 1005 - RPC Server*
- *Message Class 1006 - DCOM Wrapper*
- *Message Class 1008 - EntireX License*

Message texts from other error classes cannot be retrieved with this function. Use `ERXGetMessage` only to access errors from the error classes listed above. To always retrieve the correct error message after a C Wrapper API function call (`ERX` call), use the function `ERXGetLastMessage`.

Parameters

rc
   in: ID of the message text to retrieve.

szMsg
   out: Pointer to message text buffer.

ulMsg
   in: Length of message text buffer.

Return Codes

Value   Meaning

`int==0`   OK

Value   **Meaning**

`int!=0`   something has failed.

Related Functions

ERXGetLastError

ERXGetSecurityToken

Get the current setting of the Security Token.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetSecurityToken(
      char                        szSecurityToken
                                  [ERX_MAX_securityToken_LENGTH + 1]
);
```

Description

Returns the current value of the Broker's Security Token maintained internally by the RPC C runtime on a per-thread basis. See *Using EntireX Security* for more information.

For information on the messages, *Error Messages and Codes*.

Parameters

szSecurityToken
    out: The security token returned

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXSetSecurityToken

ERXGetTraceLevel

Get the current trace level setting of the RPC C runtime and the broker stub.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetTraceLevel(
      long                    *puTraceLevel
);
```

Description

With this function you can retrieve the current trace level setting of the RPC C runtime and the broker stub.

Parameters

puTraceLevel
    out

Return Codes

Value   **Meaning**

00000000 ERX_S_SUCCESS

00010008 ERX_E_NOT_REGISTERED

Related Functions

ERXSetTraceLevel

ERXGetVersion

Determine Version of RPC C runtime.

Syntax

```
extern int ERXGetVersion(
      char   *pMessage,
      size_t  uMessageLength
);
```

Description

Determine version of RPC C runtime. See *Examine the RPC Runtime and Interface Object Version* for more information.

Parameters

pMessage
  in out: Pointer to buffer for version string.

uMessageLength
  in: Length of buffer

Return Codes

Value   Meaning

`int==0` OK

`int!=0` something has failed.

ERXIsServing

Ping the Server.

Syntax

```
extern ERXeReturnCode ERXAPI ERXIsServing(
      ERX_CLIENT_IDENTIFICATION  ERXPTR *pClient,
      ERX_SERVER_ADDRESS         ERXPTR *pAddress,
      ERX_IS_SERVING             ERXPTR *pIsServing
);
```

Description

Check whether the server is available. Before issuing `ERXIsServing`, you must provide the following:

- the client identification
- the server address
- a pointer to the message area
- the length of the message area

For information on the messages, see *Error Messages and Codes*.

Parameters

pClient
in out: Pointer to the client identification, see `ERX_CLIENT_IDENTIFICATION`.

pAddress
in: Pointer to the server address, see `ERX_SERVER_ADDRESS`.

pIsServing
in out: Pointer to `ERX_IS_SERVING` structure.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
00020002	ERX_ETB_USER_DOES_NOT_EXIST
0003nnnn	ERX_ETB_CONVERSATION_ENDED
00070007	ERX_ETB_SERVICE_NOT_AVAILABLE
00740074	ERX_ETB_WAIT_TIMEOUT
02150148	ERX_ETB_BROKER_NOT_AVAILABLE

ERXLogoff

Logoff by Broker and EntireX Security, i.e. free Broker resources.

Syntax

```
extern ERXeReturnCode ERXAPI ERXLogoff(
      ERX_CLIENT_IDENTIFICATION  ERXPTR *pClient,
      char szEtbidName [ERX_BROKER_ETBID_NAME_LENGTH + 1]
);
```

Description

Logs off from the Broker, frees the resources within the Broker and makes them available to other users. For information on the messages, see *Error Messages and Codes*.

Parameters

pClient

in out: The client's identification, see `ERX_CLIENT_IDENTIFICATION`

szEtbidName

in: Identification of the Broker. Correponds to the `BROKER-ID` field of the Broker ACI control block.

Return Codes

Value	Meaning
`00000000`	`ERX_S_SUCCESS`
`00010008`	`ERX_E_NOT_REGISTERED`
`00020002`	`ERX_ETB_USER_DOES_NOT_EXIST`
`02150148`	`ERX_ETB_BROKER_NOT_AVAILABLE`

Related Functions

`ERXLogon`

ERXLogon

Logon by EntireX Broker and EntireX Security.

Syntax

```
extern ERXeReturnCode ERXAPI ERXLogon(
      ERX_CLIENT_IDENTIFICATION  ERXPTR *pClient,
      char                              szEtbidName
                                        [ERX_BROKER_ETBID_NAME_LENGTH + 1]
);
```

Description

Logon to Broker.

This function allows the client or server application to logon to Broker, which allocates the necessary structures to handle the new participant. If the Broker is running in a secure environment, `ERXLogon` performs the authentication process. Whether `ERXLogon` is required depends on the customization of the Broker. See `AUTOLOGON` in the Broker attribute file and `cForceLogon` in the `ERX_CLIENT_IDENTIFICATION` structure.

We suggest using `ERXLogon` and `ERXLogoff` to logon/logoff to/from Broker.

For information on the messages, see *Error Messages and Codes*.

Parameters

pClient
in out: The client's identification, see `ERX_CLIENT_IDENTIFICATION`

szEtbidName
in: Identification of the Broker. Correponds to the `BROKER-ID` field of the Broker ACI control block.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
02150148	ERX_ETB_BROKER_NOT_AVAILABLE

Related Functions

ERXLogoff

ERXRegister

Prepare use of RPC C runtime.

Syntax

```
extern ERXeReturnCode ERXAPI ERXRegister(
      const unsigned long                    ulVersionRequested
);
```

Description

Register with RPC C runtime. Any thread within a process requiring RPC C runtime must register with it. When the RPC C runtime is no longer needed, any registered thread should unregister itself (see `ERXUnregister`). See *Using the RPC Runtime* for more information. For information on the messages see *Error Messages and Codes*.

Parameters

ulVersionRequested
   in: The RPC C runtime version to be used. (See `ERX_CURRENT_VERSION` in `erx.h` for the most recent version). If the version is not supported, an error will occur.

Return Codes

Value	Meaning
`00000000`	`ERX_S_SUCCESS`
`00010007`	`ERX_E_ALREADY_REGISTERED`

Related Functions

`ERXUnregister`

ERXRegisterEvent

Syntax

```
extern ERXeReturnCode ERXAPI ERXRegisterEvent(
      const long                        EventId,
      void                         (* Callback)()
);
```

Description

The function registers events to the RPC C runtime used by the callable RPC Server during execution of the `ERXServingCallback` function. All events must be registered prior to the execution of the `ERXServingCallback` function. The following events are supported:

Event ID	Callback Prototype	Description
ERX_EVENT_SERVER_CALL	ERX_Callback_SERVER_CALL	Event for calling the server.

See *Writing the Callback* for more information. For information on the messages, see *Error Messages and Codes*.

Parameters

EventID
in: Event to register

(* Callback)()
in: Callback function belonging to the event

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERX_Callback_SERVER_CALL
ERXServingCallback

ERXReset

Reset a parameter block.

Syntax

```
extern ERXeReturnCode ERXAPI ERXReset(
 const ERX_CALL_INFORMATION_BLOCK ERXPTR *pCallInfoBlock,
      void                        ERXPTR *(ERXPTR pParameterBlock)[ ],
      ERXeParameterDirection             eDirection,
      const ERXeControlFlags             fFlags
);
```

Description

Reset the specified program parameters:

Software AG IDL	Value
A, AV, K, KV	blank
B, BV, I, F	0
D	1.1.1582
T	0:00, the date portion is reset as type D
N	+0
P	+0

For information on the messages, see *Error Messages and Codes*.

Parameters

pCallInfoBlock

in: The pointer to the description of the program and its parameter definition, see
ERX_CALL_INFORMATION_BLOCK.

pParameterBlock

in out: The array of pointers to the actual parameter data.

eDirection

in: Type of parameters to be reset, input and/or output parameters. For ERX_IN_PARM, parameters
with the IN attribute; for ERX_OUT_PARM parameters with the OUT attribute; for ERX_INOUT_PARM,
all parameters are reset.

fFlags

in: ERX_CF_STRUCTURED, i.e. the parameters are collected in one data structure.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXCall

ERXServingCallback

Syntax

```
extern ERXeReturnCode ERXAPI ERXServingCallback(
      char                              *pConfigurationFile,
      void                              *pUserInfo,
      const ERXeControlFlags             fFlags
);
```

Description

This function implements the main function of the Callable RPC Server, see *Writing the Callback*. For information on the messages, see *Error Messages and Codes*.

Parameters

*pConfigurationFile
   in: Location consisting of path and file name of the configuration file in relative and absolute notation.

*pUserInfo
   in: User-specific data. The data is provided "as is" and can be used to provide a pointer to a memory location with user-specific data in callback functions.

fFlags
   in: For future use.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
00100050	ERX_ETB_SHUTDOWN_IMMED
02150148	ERX_ETB_BROKER_NOT_AVAILABLE

Related Functions

ERX_Callback_SERVER_CALL

ERXSetBrokerSecurity

Set the broker kernel security value.

Syntax

```
extern ERXeReturnCode ERXAPI ERXSetBrokerSecurity(
      char                         cKernelSecurity
);
```

Description

This function exposes the Broker ACI field `KERNELSECURITY` as a method to users of C Wrapper. The security settings are maintained internally by the RPC C runtime on a per-thread basis. See *Using EntireX Security* for more information.

For information on the messages, see *Error Messages and Codes*.

Parameters

cKernelSecurity
    in

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXGetBrokerSecurity

ERXSetCodepage

Set the codepage.

Syntax

```
extern ERXeReturnCode ERXAPI ERXSetCodepage(
      char                    szCodepage [ERX_MAX_CODEPAGE_LENGTH + 1]
);
```

Description

With this function you assign a codepage to the locale string maintained internally by the RPC Runtime (see *Using the RPC Runtime*) on a per-thread basis. If you are using more than one codepage in a single thread, provide the correct codepage before calling one of the following RPC Runtime functions:

- ERXCall

- ERXConnect

- ERXDisconnect

- ERXDisconnectCommit

- ERXTerminateServer

- ERXIsServing

- ERXWait

Other functions do not require a codepage. For more information see *Using Internationalization with the C Wrapper* and *Error Messages and Codes*.

Parameters

szCodepage
    in

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXGetCodepage

ERXSetContext

Set a context (thread-safe RPC information).

Syntax

```
extern ERXeReturnCode ERXAPI ERXSetContext(
      ERX_CONTEXT_BLOCK     ERXPTR * pContextBlock
);
```

Description

This function supports RPC clients in multithreaded environments. It is used to set (thread-safe) RPC and broker context information, see *Programming Multithreaded RPC Clients*. For information on the messages, see *Error Messages and Codes*.

Parameters

pContextBlock
    in: Pointer to context block, see ERX_CONTEXT_BLOCK

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXGetContext

ERXSetSecurityToken

Get the current setting of the Security Token.

Syntax

```
extern ERXeReturnCode ERXAPI ERXSetSecurityToken(
      char                    szSecurityToken[ERX_MAX_securityToken_LENGTH + 1]);
```

Description

Sets the Broker Security Token. The security settings are maintained internally by the RPC C runtime on a per-thread basis. See *Using EntireX Security* for more information.

For information on the messages, see *Error Messages and Codes*.

Parameters

szSecurityToken
  in: The security token to set.

Return Codes

Value   Meaning

00000000 ERX_S_SUCCESS

00010008 ERX_E_NOT_REGISTERED

Related Functions

ERXGetSecurityToken

ERXSetTraceLevel

Set the trace level of the RPC C runtime and the broker stub's trace.

Syntax

```
extern ERXeReturnCode ERXAPI ERXSetTraceLevel(
      long                 uTraceLevel
);
```

Description

With this function you can programmatically set the trace level of the RPC C runtime and the broker stub's trace. Use the provided defines in the erx.h header file for assigning trace levels:

```
#define ERX_TRACE_NONE    (0L)
#define ERX_TRACE_LEVEL1 (1L)
#define ERX_TRACE_LEVEL2 (2L)
#define ERX_TRACE_LEVEL3 (3L)
#define ERX_TRACE_LEVEL4 (4L)
```

Parameters

uTraceLevel
    in

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXGetTraceLevel

ERXTerminateServer

Terminate running Server.

Syntax

```
extern ERXeReturnCode ERXAPI ERXTerminateServer(
      ERX_CLIENT_IDENTIFICATION  ERXPTR *pClient,
      ERX_SERVER_ADDRESS         ERXPTR *pAddress,
      ERX_TERMINATE_SERVER       ERXPTR *pTerminateServer
);
```

Description

Shut down the running server. Before issuing `ERXTerminateServer` you must provide the following:

- the client identification
- the server address
- the shutdown command
- a pointer to a message area
- the length of the message area

See description of the `ERX_SERVER_ADDRESS` and control block.

For information on the messages, see *Error Messages and Codes*.

Parameters

pClient
  in out: Pointer to the client identification, see `ERX_CLIENT_IDENTIFICATION`.

pAddress
  in: Pointer to the server address, see `ERX_SERVER_ADDRESS`.

pTerminateServer
  in out: Pointer to terminate structure, see `ERX_TERMINATE_SERVER`.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
00020002	ERX_ETB_USER_DOES_NOT_EXIST
0003nnnn	ERX_ETB_CONVERSATION_ENDED
00070007	ERX_ETB_SERVICE_NOT_AVAILABLE

Value   Meaning

00740074 ERX_ETB_WAIT_TIMEOUT

02150148 ERX_ETB_BROKER_NOT_AVAILABLE

Related Functions

ERXCall

ERXUnregister

RPC C runtime is not needed anymore, i.e. free local resources.

Syntax

```
extern ERXeReturnCode ERXAPI ERXUnregister(void);
```

Description

Unregister from RPC C runtime. When a thread no longer needs the RPC C runtime, it must unregister itself from the runtime. See *Using the RPC Runtime*. For information on the messages, see *Error Messages and Codes*.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXRegister

ERXWait

Wait for the completion of asynchronous non-conversational call.

Syntax

```
extern ERXeReturnCode ERXAPI ERXWait(
      ERXCallId                        CallId,
 const ERX_CALL_INFORMATION_BLOCK ERXPTR *pCallInfoBlock,
      void                         ERXPTR *(ERXPTR pParameterBlock)[]
);
```

Description

Wait for an incoming request. Only applicable to connection-oriented processing.

For information on the messages, see *Error Messages and Codes*.

Parameters

CallId
    in: The `CallId` returned by a previous `ERXCall`.

pCallInfoBlock
    in: Pointer to the description of the program and its parameter definition, see
    `ERX_CALL_IDENTIFICATION`.

pParameterBlock[]
    in out: The array of pointers to the actual parameter data.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010008	ERX_E_NOT_REGISTERED
00020002	ERX_ETB_USER_DOES_NOT_EXIST
0003nnnn	ERX_ETB_CONVERSATION_ENDED
00070007	ERX_ETB_SERVICE_NOT_AVAILABLE
00740074	ERX_ETB_WAIT_TIMEOUT
02150148	ERX_ETB_BROKER_NOT_AVAILABLE

API Function Descriptions for Variable-length Data Types AV, BV, KV and UV

The API of the RPC C runtime for variable-length data is defined in the following header file:

```
#include <erxvdata.h>
```

erxVDataAllocBytes

Allocates a new `VData` instance and copies `uLen` bytes from `pSource` into it. Intended for use with the IDL data types AV, BV, KV and UV together with the C programming language `mem...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*. Any allocated `VData` instance must be freed with `erxVDataFree` if no longer used.

Syntax

```
extern ERX_HVDATA erxVDataAllocBytes(void *pSource, size_t uLen);
```

Parameters

pSource
in: A pointer to `uLen` bytes to copy.

uLen
in: Number of bytes to copy from `pSource` location.

Return Values

Points to a copy of the `VData` instance. No `VData` instance is allocated and null is returned in the following case:

■ Insufficient memory

An empty `VData` instance (which holds an empty data area) is allocated in the following cases:

■ A null pointer is passed for `pSource`.

■ A value of zero is passed for `uLen`.

Related Functions

erxVDataFree
erxVDataGetByteAddress
erxVDataGetLength
erxVDataReAllocBytes

erxVDataAllocString

Allocates a new `VData` instance, and copies the string from `pSource` into it. Intended for use with the IDL data types AV and KV together with the C programming language `str...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*. Any allocated `VData` instance must be freed with `erxVDataFree` if no longer used.

Syntax

```
extern ERX_HVDATA erxVDataAllocString(char *pSource);
```

Parameters

pSource
  in: A pointer to a string to copy

Return Values

Points to a copy of the `VData` instance. No `VData` instance is allocated and null is returned in the following case:

- Insufficient memory

An empty `VData` instance, which holds an empty (null-terminated) string, is allocated in the following case:

- A null pointer is passed for `pSource`

Related Functions

`erxVDataFree`
`erxVDataReAllocString`

erxVDataAllocWideString

Allocates a new `VData` instance, copies the wide character string from the passed `pSource` location into it. Intended for use with the IDL data type UV together with the C programming language `wcs...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*. Any allocated `VData` instance must be freed with `erxVDataFree` if no longer used.

Syntax

```
extern ERX_HVDATA erxVDataAllocWideString(wchar_t *pSource);
```

Parameters

pSource
    in: A pointer to a wide character string to copy.

Return Values

Points to a copy of the `VData` instance. No `VData` instance is allocated and null is returned in the following case:

- Insufficient memory exists

An empty `VData` instance, which holds an empty (null-terminated) string is allocated in the following case:

- A null pointer is passed for `pSource`

Related Functions

erxVDataFree
erxVDataReAllocWideString

erxVDataCopy

Copies an existing source `VData` instance to an existing target `VData` instance. Can be used for all IDL data types AV, BV and KV and UV. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern ERX_HVDATA erxVDataCopy(ERX_HVDATA hVDataTo, ERX_HVDATA hVDataFrom);
```

Parameters

hVDataTo
   Handle of existing `VData` target instance.

hVDataFrom
   Handle of existing `VData` source instance.

Return Values

Points to the target `VData` instance. The `VData` instance is not copied and null is returned in the following cases:

■ An invalid handle is passed for `hVDataTo`

■ An invalid handle is passed for `hVDataFrom`

Related Functions

erxVDataAllocBytes
erxVDataAllocString
erxVDataAllocWideString
erxVDataReAllocBytes
erxVDataReAllocString
erxVDataReAllocWideString

erxVDataFree

Frees all the memory used by a `VData` instance of IDL data types AV, BV, KV and UV. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern void erxVDataFree(ERX_HVDATA hVData);
```

Parameters

hVData
> in: Handle of existing `VData` instance to free the resources.

Return Values

None.

Related Functions

```
erxVDataAllocBytes
erxVDataAllocString
erxVDataAllocWideString
```

erxVDataGetByteAddress

Get the address of binary data held by a `VData` instance of IDL data types AV, BV, KV and UV. Intended for use with the function `erxVDataGetLength` together with the C programming language `mem...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern void * erxVDataGetByteAddress(ERX_HVDATA hVData);
```

Parameters

hVData
    in: Handle of `VData` instance from which to retrieve the address of the data

Return Values

Returns the address of the data held by the `VData` instance. A null pointer is returned in the following case:

- An invalid handle is passed

A pointer to an undefined area is returned in the following case:

- The `VData` instance is empty.

Related Functions

erxVDataAllocBytes
erxVDataGetLength
erxVDataReAllocBytes

erxVDataGetLength

Get the length in bytes of the data held by a `VData` instance of IDL data types AV, BV, KV and UV. Intended for use with the function `erxVDataGetByteAddress` together with the C programming language `mem...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern size_t erxVDataGetLength(ERX_HVDATA hVData);
```

Parameters

hVData
> in: Handle of `VData` instance from which to retrieve the length.

Return Values

Number of bytes or length of string (excluding the terminating null or terminating wide-character null) held by the `VData` instance. Zero is returned in the following cases:

- An invalid handle is passed.

- The `VData` instance is empty.

Related Functions

```
erxVDataAllocBytes
erxVDataGetByteAddress
erxVDataReAllocBytes
```

erxVDataGetString

Get the address of the string held by a `VData` instance of IDL data types AV and KV. It will always have the address of a valid null-terminated string. The returned string can be used in conjunction with C language `str...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern char * erxVDataGetString(ERX_HVDATA hVData);
```

Parameters

hVData
   in: Handle of `VData` instance from which to retrieve the string address.

Return Values

Returns a pointer to the address of the string held by the `VData` instance. A null pointer is returned in the following case:

■ An invalid handle is passed.

A pointer to an empty string is returned in the following case:

■ The `VData` instance is empty.

Related Functions

erxVDataAllocString
erxVDataReAllocString

erxVDataGetWideString

Get the address of the wide character string held by a `VData` instance of IDL data type UV. It will be guaranteed always to have the address of a valid null-terminated wide character string. The returned wide character string can be used in conjunction with C programming language `wcs...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern wchar_t * erxVDataGetWideString(ERX_HVDATA hVData);
```

Parameters

hVData
> Handle to `VData` instance to get the wide character string address from.

Return Values

Returns a pointer to the address of the wide character string held by the `VData` instance. A null pointer is returned in the following case:

■ An invalid handle is passed

A pointer to an empty wide character string is returned in the following case:

■ The `VData` instance is empty

Related Functions

erxVDataAllocWideString
erxVDataReAllocWideString

erxVDataReAllocBytes

Assign new binary data to an existing `VData` instance. The function copies `uLen` bytes from `pSource` into the `VData` instance. Note that the address of the data held by the `VData` instance may have changed upon return. The location of the `VData` instance itself will always remain fixed. Intended for use with IDL data types AV, BV, KV and UV together with the C programming language `mem...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern void * erxVDataReAllocBytes(
              ERX_HVDATA hVData,
              void *pSource,
              size_t uLen);
```

Parameters

hVData
   in: Handle of existing `VData` instance.

pSource
   in: A pointer to `uLen` bytes to copy, or null to set the `VData` instance empty.

uLen
   in: Number of bytes to copy from `pSource` location. Zero will set the `VData` instance empty.

Return Values

Returns a pointer to the data held by the `VData` instance. A null pointer is returned in the following case:

- an invalid handle is passed

A pointer to empty data is returned in the following cases:

- Insufficient memory exists to hold the new value
- A null pointer is passed for `pSource`
- Zero is passed for `uLen`.

Related Functions

erxVDataAllocBytes
erxVDataGetByteAddress

erxVDataReAllocString

Assign a new string to an existing `VData` instance of IDL data types AV and KV. The function copies the string from `pSource` into the `VData` instance. Note that the address of the data held by the `VData` instance may have changed upon return. The location of the `VData` instance itself will always remain fixed. Intended for use with IDL data type AV and KV together with the C programming language `str...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern char * erxVDataReAllocString(
            ERX_HVDATA hVData,
            char *pSource);
```

Parameters

hVData
  in: Handle of existing `VData` instance.

pSource
  in: A pointer to the string to copy, or null to set the `VData` instance to an empty string.

Return Values

Returns a pointer to the data held by the `VData` instance. A null pointer is returned in the following case:

▪ An invalid handle is passed

A pointer to a null string is returned in following cases:

▪ Insufficient memory exists to hold the new value

▪ A null pointer is passed for `pSource`

Related Functions

erxVDataAllocString
erxVDataGetString

erxVDataReAllocWideString

Copies the wide character string from the passed `pSource` location into the `VData` instance of IDL data type UV. Be aware that the address to the data held by the `VData` instance can be changed upon return. The location of the `VData` instance itself will always stay fixed. Intended for use with the IDL data type UV together with the C programming language `wcs...` functions. See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern wchar_t * erxVDataReAllocWideString(ERX_HVDATA hVData, wchar_t *pSource);
```

Parameters

hVData
> Handle to `VData` instance to put the wide character string into.

pSource
> A pointer to the string to put, or null to set the `VData` instance to an empty wide character string.

Return Values

Returns a pointer to the data held by the `VData` instance. A null pointer is returned in the following case:

▪ An invalid handle is passed

A pointer to a null wide character string is returned in the following cases:

▪ Insufficient memory exists to hold the new value

▪ A null pointer is passed for `pSource`

Related Functions

erxVDataAllocWideString
erxVDataGetWideString

erxVDataReset

Sets an existing `VData` instance to empty (a null string for IDL data types AV and KV; a null wide-character string for IDL data type UV; and zero length for IDL data type BV). See *Using Variable-length Data Types AV, BV, KV and UV*.

Syntax

```
extern void erxVDataReset(ERX_HVDATA hVData);
```

Parameters

in hVData

Handle of existing `VData` instance.

Return Values

None.

Related Functions

```
erxVDataGetWideString
erxVDataGetString
erxVDataGetLength
```

API Function Descriptions for Unbounded Arrays

The API of the RPC C runtime for unbounded arrays is defined in the following header file:

```
#include <erxarray.h>
```

erxArrayAlloc

Allocates a new array instance of the given dimensions to be used as a so-called unbounded array. Array elements are initialized with their correct null value or zero corresponding to their IDL data type provided by the ERXeTypeCode. See *Using Unbounded Arrays*. Any allocated array instance must be freed with `erxArrayFree` if no longer used.

Syntax

```
extern ERX_HARRAY erxArrayAlloc(ERXeTypeCode      usType,
                                ERXeAttributes    usAttributes,
                                size_t            uLength,
                                unsigned int      uDimension,
                                ERX_ARRAY_INDEX   uArrayBound[]);
```

Description

usType	uLength	usAttributes	Note
ERX_TYPE_A	1 - 1GB	ERX_ATTR_STRING, ERX_ATTR_MF_ALPHA	1
ERX_TYPE_AV	0		2
ERX_TYPE_B	1 - 1GB		1
ERX_TYPE_BV	0		2
ERX_TYPE_D	0		2
ERX_TYPE_F	4,8		
ERX_TYPE_G	1 - 1GB		3
ERX_TYPE_I	1,2,4		
ERX_TYPE_K	1 - 1GB		1
ERX_TYPE_KV	0		2
ERX_TYPE_L	0		2
ERX_TYPE_N	1 - 29	ERX_ATTR_DOUBLE, ERX_ATTR_UNPACKED	4
ERX_TYPE_NU	1 - 29	ERX_ATTR_DOUBLE, ERX_ATTR_UNPACKED	4
ERX_TYPE_P	1 - 29	ERX_ATTR_DOUBLE, ERX_ATTR_PACKED	4
ERX_TYPE_PU	1 - 29	ERX_ATTR_DOUBLE, ERX_ATTR_PACKED	4
ERX_TYPE_S	1 - 1GB		3
ERX_TYPE_T	0		2

usType	uLength	usAttributes	Note
ERX_TYPE_U		ERX_ATTR_STRING, ERX_ATTR_MF_ALPHA	1
ERX_TYPE_UV	0		2

Note:

1. When mapped to ERX_ATTR_MF_ALPHA the length is exactly the length given in the IDL file. When mapped to ERX_ATTR_STRING the length is the length + 1 given in the Software AG IDL file for the terminating null character or terminating wide-character null.

2. The length is implicitly defined by the IDL data type.

3. A Group or structure is normally associated with a struct typedef. The length to specify is the value of the sizeof() operator applied to the struct.

4. When mapped to ERX_ATTR_UNPACKED or ERX_ATTR_PACKED the length to specify relates to the IDL data type. The number of digits before and after the decimal point must be added. Example: For 5.2 specify 7. When mapped to ERX_ATTR_DOUBLE the length is implicit and thus obsolete.

Parameters

usType

The IDL data type stored in the array instance. See the description above.

usAttributes

The description above lists valid values for IDL data types. The values must exactly match the mapping options used when the RPC client is generated. See *Generate C Source Files from Software AG IDL Files*.

uLength

Depending on the data type (see table above) the length is required.

uDimension

Number of dimensions. The dimension must be at least 1. Up to 3 dimensions are allowed. The lower bound is always 0.

uArrayBound

Pointer to a vector containing the number of elements for each dimension. The number of vector elements must correspond to the number of array dimensions. The left (most significant) dimension is uArrayBound[0].

Return Values

Points to the created copy of the array instance. No array instance is allocated and null is returned in the following cases:

◼ Insufficient memory exists

◼ The IDL data type provided by the ERXeTypeCode is invalid

- missing `uLength` depending on the data type
- `uDimension` is zero
- `uArrayBound` is invalid

Related Functions

`erxArrayFree`

erxArrayCopy

Copies an existing source array instance to an existing target array instance. Source and target array instance must exist, otherwise an error is returned. The contents of the target array instance are overwritten by the contents of the source instance. See *Using Unbounded Arrays*.

Syntax

```
extern ERXeReturnCode erxArrayCopy(ERX_HARRAY hArrayTo, ERX_HARRAY hArrayFrom);
```

Parameters

phArrayTo
Points to the target array instance created previously by `erxARrayAlloc`.

hArrayFrom
Points to the source array instance created previously by `erxArrayAlloc`.

Return Codes

Value	Meaning
`00000000`	`ERX_S_SUCCESS`
`00010005`	Out of Memory
`00010028`	Illegal Type
`00010079`	Invalid Unbounded Array

Related Functions

`erxArrayAlloc`

erxArrayFree

Frees all the memory used by the array instance. See *Using Unbounded Arrays*.

Syntax

```
extern ERXeReturnCode erxArrayFree(ERX_HARRAY hArray);
```

Parameters

hArray
    Points to an array instance created by `erxArrayAlloc`.

Return Codes

Value   Meaning

`00000000` `ERX_S_SUCCESS`

`00010079` Invalid Unbounded Array

Related Functions

`erxArrayAlloc`

erxArrayGetAttributes

Returns all attributes defined for the array instance during allocation with `erxArrayAlloc`. See *Using Unbounded Arrays*.

Syntax

```
extern ERXeAttributes erxArrayGetAttributes(ERX_HARRAY hArray);
```

Parameters

hArray
Points to an array instance created by `erxArrayAlloc`.

Return Values

The attributes defined for the array instance.

Related Functions

`erxArrayAlloc`
`erxArrayGetElementLength`
`erxArrayGetTypeCode`

erxArrayGetBounds

Returns the array bound for a vector (the number of elements which can be stored in and retrieved from a given vector of an array instance). See *Using Unbounded Arrays*.

Syntax

```
extern ERX_ARRAY_INDEX erxArrayGetBounds(ERX_HARRAY      hArray,
                                         unsigned int    uDimension,
                                         ERX_ARRAY_INDEX uArrayIndex[]);
```

Parameters

hArray
> Points to an array instance created by `erxArrayAlloc`.

uDimension
> The dimension for which to set the array bound.

uArrayIndex
> Pointer to a vector of indices defining an array position. The left-most (most significant) dimension is `uArrayIndex[0]`.

Return Values

The array bound for a given dimension. Zero is returned in the following cases:

- `hArray` is invalid
- `uDimension` is invalid or outside the current dimensions
- the unbounded array or specified vector has no elements
- `uArrayIndex` is invalid

Related Functions

erxArrayGetDimension
erxArrayRedimAll
erxArrayRedimVector

erxArrayGetDimension

Returns the number of dimensions of the array instance. See *Using Unbounded Arrays*.

Syntax

```
extern unsigned int erxArrayGetDimension(ERX_HARRAY hArray);
```

Parameters

hArray
 Points to an array instance created by `erxArrayAlloc`.

Return Values

The number of dimensions of the unbounded array instance. Zero is returned in the following case:

- `hArray` is invalid

Related Functions

```
erxArrayGetBounds
erxArrayRedimAll
erxArrayRedimVector
```

erxArrayGetElement

Retrieves a single element of the array instance. The caller must provide a storage area of the correct size to receive the data. See *Using Unbounded Arrays*.

Syntax

```
extern ERXeReturnCode erxArrayGetElement(ERX_HARRAY        hArray,
                                         ERX_ARRAY_INDEX   uArrayIndex[],
                                         void            * pData);
```

Parameters

hArray
> Points to an array instance created by `erxArrayAlloc`.

uArrayIndex
> Pointer to a vector of indices defining an array position. The number of vector elements must correspond to the number of array dimensions. The left-most (most significant) dimension is `uArrayIndex[0]`.

pData
> Pointer where to put the data stored in the given array position.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010079	Invalid Unbounded Array
00010084	Unbounded Array indices out of bounds
00010085	Invalid Data for Unbounded Array

Related Functions

erxArrayReset
erxArraySetElement

erxArrayGetElementLength

Retrieves the explicit logical length of the IDL data type of the array instance defined during allocation with `erxArrayAlloc`. See ***Using Unbounded Arrays***.

Syntax

```
extern size_t erxArrayGetElementLength(ERX_HARRAY hArray);
```

Parameters

hArray
Points to an array instance created by `erxArrayAlloc`.

Return Values

The explicit logical length of the IDL data type of the array instance. Zero is returned in the following cases:

- the IDL data type has no explicit logical length, for example for the types L,D and T.

- `hArray` is invalid

Related Functions

```
erxArrayAlloc
erxArrayGetAttributes
erxArrayGetTypeCode
```

erxArrayGetTypeCode

Returns the IDL data type of the array instance defined during allocation with `erxArrayAlloc`. See *Using Unbounded Arrays*.

Syntax

```
extern ERXeTypeCode erxArrayGetTypeCode(ERX_HARRAY hArray);
```

Parameters

hArray
    Points to an array instance created by `erxArrayAlloc`.

Return Values

The IDL data type of the array instance. `ERX_TYPE_UNKNOWN` is returned in the following case:

- `hArray` is invalid

Related Functions

erxArrayAlloc
erxArrayGetAttributes
erxArrayGetElementLength

erxArrayRedimAll

Changes all bounds of the array instance. The cardinality (number of dimensions) cannot be changed. For a 2 or 3-dimensional array, the result will be a square or a cube. See *Using Unbounded Arrays*.

Syntax

```
extern ERXeReturnCode erxArrayRedimAll(ERX_HARRAY       hArray,
                                       ERXeArrayPreserve  ePreserveData,
                                       ERX_ARRAY_INDEX    uNewArrayBound[]);
```

Parameters

hArray
> Points to an array instance created by `erxArrayAlloc`.

ePreserveData
> Determines whether the redimensioned array is to be initialized with null or whether the contents are to be kept (when elements exist in the old and new array). Valid values are: `ERX_PRESERVE_NO`, `ERX_PRESERVE_YES`

uNewArrayBound
> Pointer to a vector containing the number of elements for each dimension. The number of vector elements must correspond to the number of array dimensions. The left-most (most significant) dimension is `uNewArrayBound[0]`.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010005	Out of Memory
00010079	Invalid Unbounded Array
00010084	Unbounded Array indices out of bounds
00010086	Invalid Preserve flag for Unbounded Array

Related Functions

erxArrayGetBounds
erxArrayGetDimension
erxArrayRedimVector

erxArrayRedimVector

Changes the specified bounds of the given vector of the array instance. For 2 and 3-dimensional arrays, the result can be a deformed array (that is, not a square or cube). See *Using Unbounded Arrays*.

Syntax

```
extern ERXeReturnCode erxArrayRedimVector(ERX_HARRAY          hArray,
                                          ERXeArrayPreserve   ePreserveData,
                                          unsigned int        uDimension,
                                          ERX_ARRAY_INDEX     uArrayIndex[],
                                          ERX_ARRAY_INDEX     uNewBound);
```

Parameters

hArray

Points to an array instance created by `erxArrayAlloc`.

ePreserveData

Determines whether the redimensioned vector (when it is the last dimension) is to be initialized with null or whether the contents are to be kept (when elements exist in the old and new array). New elements are always initialized with null. Valid values are:ERX_PRESERVE_NO, ERX_PRESERVE_YES

uDimension

The dimension for which to set new vector bound.

uArrayIndex

Pointer to a vector of indices defining an array position. The left-most (most significant) dimension is `uArrayIndex[0]`.

uNewBound

The number of elements to redimension the vector with.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010005	Out of Memory
00010079	Invalid Unbounded Array
00010084	Unbounded Array indices out of bounds
00010086	Invalid Preserve flag for Unbounded Array
00010087	Invalid Dimension

Related Functions

erxArrayGetBounds
erxArrayGetDimension
erxArrayRedimAll

erxArrayReset

Sets all elements of the array instance to null value or zero corresponding to the IDL data type given when the array was created. See *Using Unbounded Arrays*.

Syntax

```
extern ERXeReturnCode erxArrayReset(ERX_HARRAY hArray);
```

Parameters

hArray
    Points to an array instance created by `erxArrayAlloc`.

Return Codes

Value	Meaning
`00000000`	`ERX_S_SUCCESS`
`00010079`	Invalid Unbounded Array

Related Functions

erxArrayGetElement
erxArraySetElement

erxArraySetElement

Stores the data element at a given location in the array instance. See *Using Unbounded Arrays*.

Syntax

```
extern ERXeReturnCode erxArraySetElement(ERX_HARRAY       hArray,
                                         ERX_ARRAY_INDEX  uArrayIndex[],
                                         void           * pData);
```

Parameters

hArray
Points to an array instance created by `erxArrayAlloc`.

uArrayIndex
Pointer to a vector of indices defining an array position. The number of vector elements must correspond to the number of array dimensions. The left-most (most significant) dimension is `uArrayIndex[0]`.

pData
Pointer to the data set into the given array position.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010079	Invalid Unbounded Array
00010084	Unbounded Array indices out of bounds
00010085	Invalid Data for Unbounded Array

Related Functions

`erxArrayGetElement`
`erxArrayReset`

API Function Descriptions for Reliable RPC

ERXGetReliableState

Get the current reliable RPC state.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetReliableState(
      unsigned long             *pulReliableState
);                                                                    ↵
```

Description

Get the current reliable RPC state. For a list of possible states with description, see ERXSetReliableState.

Parameters

pulReliableState
out: The current reliable RPC state

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010009	ERX_E_PARAMETER_ERROR
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXSetReliableState

ERXSetReliableState

Set the reliable RPC state.

Syntax

```
extern ERXeReturnCode ERXAPI ERXSetReliableState(
      unsigned long             ulReliableState
);                                                                        ↩
```

Description

Set the current reliable RPC state to enable/disable reliable RPC.

State	Description
ERX_RELIABLE_OFF	The ERX_RELIABLE_OFF state represents the "normal" RPC.
ERX_RELIABLE_AUTO_COMMIT	The ERX_RELIABLE_AUTO_COMMIT puts each RPC request in a single reliable RPC message and commits each message automatically. To query the status of the sent reliable RPC message, you first have to resolve the reliable ID with ERXGetReliableID. With the retrieved reliable ID you can query the status of the reliable RPC message with ERXGetReliableStatus at any time. See also *Writing a Client using AUTO COMMIT*.
ERX_RELIABLE_CLIENT_COMMIT	On ERX_RELIABLE_CLIENT_COMMIT the client application can send a sequence of reliable RPC messages and can commit them whenever it is required. For this purpose ERXReliableCommit is offered. The client application also has the option to roll back the sequence of reliable RPC messages by using ERXReliableRollback. See also *Writing a Client*.

Parameters

ulReliableState
    in: The reliable RPC state to set the values to

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010009	ERX_E_PARAMETER_ERROR
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXGetReliableState

ERXReliableCommit

Commits a sequence of reliable RPC messages.

Syntax

```
extern ERXeReturnCode ERXAPI ERXReliableCommit(
      ERX_SERVER_ADDRESS        ERXPTR *pAddress
);                                                                        ↵
```

Description

Commits a sequence of reliable RPC messages in mode `ERX_RELIABLE_CLIENT_COMMIT`. See `ERXSetReliableState`.

Parameters

pAddress
   in: The server address to send the commit to.

Return Codes

Value   Meaning

`00000000 ERX_S_SUCCESS`

`00010009 ERX_E_PARAMETER_ERROR`

`00010010 ERX_E_CONTROL_BLOCK_NOT_FOUND`

`00010008 ERX_E_NOT_REGISTERED`

Related Functions

`ERXReliableRollback`

ERXReliableRollback

Rolls back a sequence of reliable RPC messages.

Syntax

```
extern ERXeReturnCode ERXAPI ERXReliableRollback(
       ERX_SERVER_ADDRESS        ERXPTR *pAddress
);                                                                            ↵
```

Description

Rolls back a sequence of reliable RPC messages in mode `ERX_RELIABLE_CLIENT_COMMIT`. See `ERXSetReliableState`.

Parameters

pAddress
    in: The server address to which to send the rollback.

Return Codes

Value   **Meaning**

`00000000 ERX_S_SUCCESS`

`00010009 ERX_E_PARAMETER_ERROR`

`00010010 ERX_E_CONTROL_BLOCK_NOT_FOUND`

`00010008 ERX_E_NOT_REGISTERED`

Related Functions

`ERXReliableCommit`

ERXGetReliableID

Get the reliable ID of the current reliable RPC message or message sequence.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetReliableID(
 ERX_SERVER_ADDRESS          ERXPTR *pAddress,
 ETB_CHAR                    ERXPTR *pReliableID
);                                                                    ↵
```

Description

Get the current reliable ID. The reliable ID is required to get the status of reliable RPC messages, see `ERXGetReliableStatus`.

In the case of `ERX_RELIABLE_CLIENT_COMMIT`, this method must be called before `ERXReliableCommit` or `ERXReliableRollback` is invoked, otherwise you might get the error 00010010.

In the case of `ERX_RELIABLE_AUTO_COMMIT`, this method must be called directly after the RPC message is sent and before any other RPC runtime calls, otherwise the reliable ID is lost and you cannot retrieve the message status.

Parameters

pAddress
in: The server address which was used for the interface object call.

pReliableID
out: The reliable ID of the current reliable RPC message.

> ⚠ **Important:** The pointer `pReliableID` must point to a field defined like `ETB_CHAR szReliableID[16+1]`, otherwise unpredictable results occur.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010003	ERX_E_UNKNOWN_MEDIUM
00010009	ERX_E_PARAMETER_ERROR
00010010	ERX_E_CONTROL_BLOCK_NOT_FOUND
00010008	ERX_E_NOT_REGISTERED

Related Functions

ERXGetReliableStatus

ERXGetReliableStatus

Get the status of the reliable RPC messages.

Syntax

```
extern ERXeReturnCode ERXAPI ERXGetReliableStatus(
 ERX_CLIENT_IDENTIFICATION  ERXPTR *pClient,
 ERX_SERVER_ADDRESS         ERXPTR *pAddress,
 ETB_CHAR                   ERXPTR *pReliableID,
 ETB_BYTE                          *pReliableStatus
);                                                        ↵
```

Description

Get the status of the reliable RPC messages given by the reliable ID. by given Reliable ID.

Status can be one of the values listed under *ACI Fields used for Units of Work*.

Parameters

pClient
    in: Client information.

pAddress
    in: Server information.

pReliableID

   ⚠️   **Important:**   in: The reliable ID of the reliable RPC messages. The pointer `pReliableID`
        must point to a field defined like `ETB_CHAR szReliableID[16+1]`, otherwise unpredict-
        able results occur.

pReliableStatus

   ⚠️   **Important:**   out: the status of the requested reliable RPC message (identified by the given
        reliable ID). pReliableStatus points to a field of one byte.

Status can be one of the values listed under *ACI Fields used for Units of Work*.

Return Codes

Value	Meaning
00000000	`ERX_S_SUCCESS`
00010003	`ERX_E_UNKNOWN_MEDIUM`
00010009	`ERX_E_PARAMETER_ERROR`

Value   Meaning

00010008 ERX_E_NOT_REGISTERED

Related Functions

ERXGetReliableID

ERXControl

Control of RPC C runtime.

Syntax

```
extern ERXeReturnCode ERXAPI ERXControl
(
    const ERXCallId         CallId,
          ERXeControlCommand eCmd
);
```

Description

For future use.

Parameters

CallId
   in: `ERXCallId`.

eCmd
   in: command to use.

Return Codes

Value	Meaning
00000000	ERX_S_SUCCESS
00010009	ERX_E_PARAMETER_ERROR
00010010	ERX_E_CONTROL_BLOCK_NOT_FOUND
00010008	ERX_E_NOT_REGISTERED

Related Functions

None.

11   API Data Descriptions for the C Wrapper

This chapter describes the client API data structures available for the C Wrapper and covers the following topics:

Conventions Used for API Data Descriptions

The following naming conventions are used to describe the EntireX RPC API data structures:

Naming Convention	Data Type
ul*XXXXX*	unsigned long
us*XXXXX*	unsigned short
u*XXXXX*	unsigned ... (predefined types such as `ptrdiff_t`)
sz*XXXXX*	zero terminated string
p*XXXXX*	pointer to ...

API Data Descriptions

ERX_CLIENT_IDENTIFICATION

```
typedef   struct                    tagERX_CLIENT_IDENTIFICATION
{
   const char    * pUserId;              /* string, required max [32 + 1] bytes */
   const char    * pPassword;            /* string, any length */
   const char    * pToken;               /* string, max [32 + 1] bytes */
   const char    * pNewPassword;         /* string, any length */
   const char    * pRpcUserId;           /* string, any length */
   const char    * pRpcPassword;         /* string, any length */
   const char    * pSSLParameter;        /* string, any length */
   char            szSecurityToken    [ ERX_MAX_SECURITY_TOKEN_LENGTH + 1 ];
   char            cForceLogon;
   unsigned char   uEncryptionLevel;  /* Deprecated. Use AT-TLS on z/OS
                                         ATLS on z/VSE or SSL/TLS on
                                         other platforms. */
   unsigned char   uCompressionLevel;
}  ERX_CLIENT_IDENTIFICATION;
```

Field	Description	More Information
*pUserId	Mandatory. The user ID for access to the broker.	*Using the Broker and RPC User ID/Password*
*pPassword	Optional. The password for secured access to the broker.	
*pToken	Optional. Token used by the EntireX Broker to identify the caller.	`USER-ID` and `TOKEN` under *Writing Client and Server Applications* in the ACI Programming documentation
*pNewPassword	Optional. For changing the password to a new password.	*Changing your Password* under *Writing Applications using EntireX Security*
*pRpcUserId	Optional. The RPC user ID sent to the RPC server.	*Using the Broker and RPC User ID/Password*
*pRpcPassword	Optional. The RPC password sent to the RPC server.	
*pSSLParameter	Secure Sockets Layer settings are provided here as a null-terminated string.	*Using SSL/TLS*
szSecurityToken	Security token generated by EntireX Security and EntireX Broker after successful security validation.	*Role of Security Token (STOKEN) during Authentication* under *Writing Applications using EntireX Security*
cForceLogon	Mandatory. Determines whether explicit logon or autologon is used by the caller.	`FORCE-LOGON` under *Writing Applications using EntireX Security*
uEncryptionLevel	Deprecated. For encrypted transport we strongly recommend using the Secure Sockets Layer/Transport Layer Security protocol. See *SSL/TLS, HTTP(S), and Certificates with EntireX* in the platform-independent Administration documentation.	
uCompressionLevel	Optional.	*Data Compression* under *Writing Client and Server Applications* in the ACI Programming documentation

ERX_SERVER_ADDRESS

```
typedef  struct              tagERX_SERVER_ADDRESS
{
   ERXeMedium Medium;
   unsigned long             ulTimeOut;
   union
   {

     ERX_SA_BROKER            BROKER;
     ERX_SA_BROKER_LIBRARY    BROKER_Library;
     ERX_SA_CONNECTION        Connection;
```

```
    }  Address;
}  ERX_SERVER_ADDRESS;
```

Field	Description

`Medium`	Mandatory.
	Selects an RPC server.
	The following types are supported:
	`ERX_TM_BROKER` (for backward compatibility)
	`ERX_TM_BROKER_LIBRARY`
	`ERX_TM_CONNECTION`
	This type of medium is used for the connection-oriented (conversational) RPC. After successful `ERXConnect` (that is, after opening the conversation), the RPCs are invoked using `ERX_TM_CONNECTION`. Any open conversation must be closed with `ERXDisconnectCommit` or aborted with `ERXDisconnect`.

`ulTimeOut`	Mandatory. Gives the timeout value for the transport system in seconds. Corresponds to the `WAIT` field of the ACI control block. **Note:** Zero is not a valid value.

Address

Depending on the Medium field, the Address union holds the necessary information to address a server:

BROKER	
`szEtbidName`	Mandatory. Broker ID used. Corresponds to the `BROKER-ID` field of the ACI control block.
`szClassName`	Mandatory. Class Name of the EntireX/Natural RPC server. Use RPC for Natural RPC Server. Corresponds to the `SERVER-CLASS` field of the ACI control block.
`szServerName`	Mandatory. Server Name of the EntireX/Natural RPC server. Corresponds to the `SERVER-NAME` field of the ACI control block.
`szServiceName`	Mandatory. Service Name of the EntireX/Natural RPC server. Use `CALLNAT` for Natural RPC Server. Corresponds to the `SERVICE` field of the ACI control block.
BROKER_LIBRARY	
`szEtbidName`	Mandatory. Broker ID used. Corresponds to the `BROKER-ID` field of the ACI control block.
`szClassName`	Mandatory. Class Name of the EntireX/Natural RPC server. Use RPC for Natural RPC Server. Corresponds to the `SERVER-CLASS` field of the ACI control block.

BROKER	
szServerName	Mandatory. Server Name of the EntireX/Natural RPC server. Corresponds to the SERVER-NAME field of the ACI control block.
szServiceName	Mandatory. Service Name of the EntireX/Natural RPC server. Use CALLNAT for Natural RPC Server. Corresponds to the SERVICE field of the ACI control block.
szLibraryName	Mandatory. Library sent to the target RPC server by the client. The library specified here overrides any library information specified in the IDL file, see library-definition under *Software AG IDL Grammar* in the IDL Editor documentation. Usage depends on the target RPC server type: ■ **Natural RPC server** Set cNaturalLogon (see below) to Y for library usage, that is, log on to the library. ■ **EntireX RPC server** The library given here is used.
cNaturalLogon	Enable to send the RPC user ID/password pair. For more information see *Using the Broker and RPC User ID/Password*. Valid values: Y,N. Use the macro definition ERX_NATURAL LOGON_YES and ERX_NATURAL LOGON_NO from erx.h to set the values.
cCompression	Mandatory. Specify ERX_COMPRESSION_YES always. **Note:** *Data Compression in EntireX Broker* is something different and is controlled by field uCompressionLevel in API structure ERX_CLIENT_IDENTIFICATION.
Connection	
Contains internal information for EntireX runtime.	
Caution: Do not modify	

ERX_CALL_IDENTIFICATION

```
typedef      struct          tagERX_CALL_IDENTIFICATION
{
   char      szLibraryName  [ ERX_MAX_LIBRARY_NAME_LENGTH + 1 ];
   char      szProgramName  [ ERX_MAX_PROGRAM_NAME_LENGTH + 1 ];
   unsigned long            ulVersion;
}  ERX_CALL_IDENTIFICATION;
```

Field	Description
szLibraryName	The name of the library where the program to be called resides. The format depends on the environment. For a mainframe Natural server, for example, the name of the library must be uppercase.
szProgramName	The name of the program to be called. The format depends on the environment. For a mainframe Natural server, for example, the name of the program must be uppercase.
ulVersion	Reserved for future use, should be set to zero.

ERX_PARAMETER_DEFINITION_V3

```
typedef                    struct                tagERX_PARAMETER_DEFINITION
{
    char                   szParameterName    [ ERX_MAX_PARAMETER_NAME_LENGTH + 1 ];
    ERXeTypeCode           usType;
    ERXeAttributes         usAttributes;
    size_t                 uElementLength;
    ERX_OBJECT_SIZE        uSize
    unsigned long          uParent;
    unsigned long          uOccurrence        [ ERX_MAX_INDICES ];
    ERX_POINTER_DIFFERENCE uBase;
    ERX_POINTER_DIFFERENCE uDelta             [ ERX_MAX_INDICES ];
    void                   *pCallInfoBlock;
}   ERX_PARAMETER_DEFINITION_V3;
```

Field	Description
szParameterName	The name of the parameter.

usType	The data type including parameter type direction and index count. See *IDL Data Types*. Bits:
	
	```FEDCBA9876543210```
	```UUUUUUTTTTTTDDNN```
	
	U: Unused
	T: Type Code
	ERX_TYPE_A, STRING FIXED LENGTH
	ERX_TYPE_AV, STRING VARIABLE LENGTH
	ERX_TYPE_K, KANJI FIXED LENGTH
	ERX_TYPE_KV, KANJI VARIABLE LENGTH
	ERX_TYPE_L, LOGICAL
	ERX_TYPE_I, INTEGER
	ERX_TYPE_N, UNPACKED DECIMAL
	ERX_TYPE_NU, UNPACKED DECIMAL UNSIGNED
	ERX_TYPE_P, PACKED DECIMAL
	ERX_TYPE_PU, PACKED DECIMAL UNSIGNED
	ERX_TYPE_F, FLOAT
	ERX_TYPE_T, DATE & TIME
	ERX_TYPE_D, DATE

Field	Description

	`ERX_TYPE_B,`BINARY FIXED LENGTH
	`ERX_TYPE_BV,`BINARY VARIABLE LENGTH
	`ERX_TYPE_G,`GROUP
	`ERX_TYPE_S,`STRUCTURES
	`ERX_TYPE_U,`UNICODE FIXED LENGTH
	`ERX_TYPE_UV,`UNICODE VARIABLE LENGTH
	D: Parameter Direction
	`ERX_IN_PARM`
	`ERX_OUT_PARM`
	`ERX_INOUT_PARM`
	N: Index Count (0 .. 3) All other bits must be zero.

`usAttributes`	Attributes can be combined by OR when they are not exclusive, such as `ERX_ATTR_STRING and ERX_ATTR_MF_ALPHA`. For a list of `Attributes` see the following table.
`uElementLength`	Information on the logical length of the parameter. For `ERX_TYPE_A`, for example, A10 in the IDL file has a `uElementLength` of 11 when mapped to string with `ERX_ATTR_STRING`. It has a `uElementLength` of 10 when mapped with `ERX_ATTR_MF_ALPHA`.
`uSize`	The physical size of the parameter in bytes.
`uOccurrence`	The count of elements in each dimension in ascending order. For unbounded arrays: when set presents a possible maximum. Unbounded arrays with value zero have no maximum.
`uBase`	The address of the base of the parameter.
`uDelta`	The difference for each dimension between following elements in ascending order.
`uCallInfoBlock`	Pointer to `CallInfoBlock` for structures `ERX_TYPE_S`.

Attributes

Attribute	Description
`ERX_ATTR_ARRAY_V1`	First dimension of array is unbounded. Attribute evaluated by Software AG IDL Compiler.
`ERX_ATTR_ARRAY_V2`	Second dimension of array is unbounded. Attribute evaluated by Software AG IDL Compiler.
`ERX_ATTR_ARRAY_V3`	Third dimension of array is unbounded. Attribute evaluated by Software AG IDL Compiler.
`ERX_ATTR_ALIGNED`	The parameter is aligned on the server side. The attribute is evaluated by the Software AG IDL Compiler. Used by EntireX RPC server on CICS.
`ERX_ATTR_DOUBLE`	The parameter is mapped to C data type double. Valid for: ■ `ERX_TYPE_P` ■ `ERX_TYPE_PU`

Attribute	Description
	■ ERX_TYPE_N ■ ERX_TYPE_NU The mapping can be forced by a template by adding 32768 to the %TypeAttribute macro of the Software AG IDL Compiler.
ERX_ATTR_PACKED	The parameter is mapped to C data type char[...] contained in IBM mainframe packed format. Valid for: ■ ERX_TYPE_P ■ ERX_TYPE_PU Default mapping for P and PU data types; used when nothing is added to the %TypeAttribute.
ERX_ATTR_UNPACKED	The parameter is mapped to C data type char[...] contained in IBM mainframe packed format. Valid for: ■ ERX_TYPE_N ■ ERX_TYPE_NU Default mapping for N and NU data types; used when nothing is added to the %TypeAttribute.
ERX_ATTR_STRING	The parameter is mapped to a null terminated string. Valid for ■ ERX_TYPE_A The mapping can be forced by a template by adding 16384 to the %TypeAttribute macro of the Software AG IDL Compiler.
ERX_ATTR_MF_ALPHA	The parameter is mapped to C data type char[...] - but without a NULL terminator. Valid for: ■ ERX_TYPE_A Default attribute for A data types; is used when nothing is added to the %TypeAttribute.
ERX_ATTR_NOTHING	Use when none of the mappings described above apply.

ERX_CALL_INFORMATION_BLOCK

```
typedef  struct                        tagERX_CALL_INFORMATION_BLOCK
{
   ERX_CALL_IDENTIFICATION            Callee;
   unsigned short                     uParameterCount;
   ERX_PARAMETER_DEFINITION_V3        *pParmDef;
}  ERX_CALL_INFORMATION_BLOCK;
```

Field	Description
Callee	The identification of the program to be called, see ERX_CALL_IDENTIFICATION
uParameterCount	The total count of the parameters (the number of entries in the parameter definition array).
pParmDef	A pointer to the parameter definition array. See ERX_PARAMETER_DEFINITION_V3_V3

ERX_ERROR_INFORMATION

```
typedef   struct          tagERX_ERROR_INFO
{
   ERXeReturnCode          rc;
   char                    szMessage[ 256 ];
}  ERX_ERROR_INFO;
```

Field	Description
rc	The last return code returned. See *Error Messages and Codes*.
szMessage	Error message. Text associated with the last return code issued.

ERX_IS_SERVING

```
typedef struct          tagERX_IS_SERVING
{
   char                    *pMessage;
   int                     uMessageLength;
}  ERX_IS_SERVING;
```

Field	Description
pMessage	Pointer to the provided buffer for server's "alive" message.
uMessageLength	Length of the provided buffer.

ERX_TERMINATE_SERVER

```
typedef   enum
{
   ERX_SHUTDOWN_IMMED_ALL = 1,
   ERX_SHUTDOWN_ANYONE = 2
}  ERXeShutdownCommand;

typedef struct tagERX_TERMINATE_SERVER
{
   ERXeShutdownCommand       eShutdownCommand;
   char                      *pMessage;
   int                       uMessageLength;
}  ERX_TERMINATE_SERVER;
```

Field	Description
eShutdownCommand	The shutdown method to be used. ■ ERX_SHUTDOWN_IMMED_ALL Using EntireX Broker Command Service. Using this method forces all instances of RPC servers (Java, Natural, UNIX, Windows, CICS, etc.) to be shut down. ■ ERX_SHUTDOWN_ANYONE Directly to EntireX RPC Server. If multiple instances of RPC servers (Java, Natural, UNIX, Windows, CICS, etc.) are registered under the same class/server/service name at the broker, only one single RPC server instance is shut down. There is no control over which instance is shut down. To shut down all RPC server instances, you have to repeat the function until no more RPC server instances are registered. This method is compatible with the method from versions of EntireX RPC prior to EntireX 5.3.1.
pMessage	Pointer to the provided buffer for server's "completion" message.
uMessageLength	Length of the provided buffer.

ERX_CONTEXT_BLOCK

```
typedef struct tagERX_CONTEXT_BLOCK
{
    ERXCallId                 ERXCallId;
    ERXeReturnCode            ERXrc;
    ERX_ERROR_INFO            ERXErrorInfo;
    ERX_SERVER_ADDRESS        ERXServer;
    ERX_CLIENT_IDENTIFICATION ERXClient;
}  ERX_CONTEXT_BLOCK;
```

Field	Description
ERXCallId	The CallId returned by a caller.
ERXrc	EntireX RPC Error Code. See *Error Messages and Codes*.
ERXErrorInfo	EntireX RPC Error information, see ERX_ERROR_INFORMATION.
ERXServer	The server address, see ERX_SERVER_ADDRESS.
ERXClient	The client identification, see ERX_CLIENT_IDENTIFICATION.

ERX_SVM_VERSION_1

```
#define ERX_SVM_VERSION_1 (unsigned long) 1
typedef struct tagERX_SVM_V1
{
   unsigned long  version;
   char          *pProtocol;
   char          *pSM;
   char          *pFA;
   char          *pVA;
   char          * pSA;
}  ERX_SVM_V1;
```

Field	Description
version	Version of this control block. Initialitze with `ERX_SVM_VERSION_1`.
pProtocol	RPC protocol version from the related server mapping file (Designer file with extension .cvm) evaluated by the Software AG IDL Compiler and provided in the `output_substitution_sequence %SVMRpcProtocol`.
pSM	Pointer to the meta data part of the related server mapping file evaluated by the IDL Compiler and provided in the `output_substitution_sequence %SVMMetaData`.
pFA	Pointer to the format area of the related server mapping file evaluated by the IDL Compiler and provided in the `output_substitution_sequence %SVMFormatArea`.
pSA	Pointer to the string area of the related server mapping file evaluated by the IDL Compiler and provided in the `output_substitution_sequence %SVMStringArea`.