Writing Advanced Applications with the C Wrapper

This chapter covers the following topics:

- Using the RPC Runtime
- Examine the RPC Runtime and Interface Object Version
- Tracing
- Programming Multithreaded RPC Clients
- Natural Logon or Changing the Library Name
- Using Variable-length Data Types AV, BV, KV and UV
- Using Unbounded Arrays
- Using Conversational RPC
- Using RPC Compression
- Using EntireX Security
- Using Natural Security
- Using SSL
- Using Compression
- Using Internationalization 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=9.0.0, Patch Level=0
```

Tracing

There are several possibilities to trace the EntireX C Wrapper. See *Tracing webMethods EntireX* under UNIX | Windows | BS2000/OSD | z/VSE.

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.

Natural Logon or Changing the Library Name

The library name sent with the RPC request to the EntireX RPC or the Natural RPC Server is specified in the IDL file (see library-definition). When the RPC is executed, this library name can be overwritten.

To overwrite the library, an EntireX C Wrapper client must do the following:

1. Set the medium ERX_TM_BROKER_LIBRARY in the ERX_SERVER_ADDRESS structure (see ERX_SERVER_ADDRESS under *API Data Descriptions*).
2. Specify the correct library in the ERX_SA_BROKER_LIBRARY structure in the szLibraryName parameter.

To force the library to be considered by Natural RPC Server

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

Warning:
Natural and EntireX RPC servers behave differently regarding the library name.

See *Natural Logon or Changing the Library Name*.

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`.

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`.

Warning:
Natural and EntireX RPC servers behave differently when ending an RPC conversation.

See *Conversational RPC*.

Using RPC Compression

EntireX and Natural RPC support a feature called RPC compression to reduce network data sizes. We recommend switching RPC compression on. See *RPC Compression*.

To switch compression on

1. Set the medium `ERX_TM_BROKER_LIBRARY` in the `ERX_SERVER_ADDRESS` structure (see `ERX_SERVER_ADDRESS` under *API Data Descriptions*).
2. Set the `cCompression` field within the `ERX_SA_BROKER_LIBRARY` structure to `"ERX_COMPRESSION_YES"`.

To switch compression off

1. When using the medium `ERX_TM_BROKER_LIBRARY` in the `ERX_SERVER_ADDRESS` structure, set the `cCompression` field within the `ERX_SA_BROKER_LIBRARY` structure to `"ERX_COMPRESSION_NO"`.
2. When using the medium `ERX_TM_BROKER` in the `ERX_SERVER_ADDRESS` structure, compression is off.

Using EntireX Security

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

To use EntireX Security

1. Specify a user ID and password in the parameters `szUserId` and `szPassword` 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 Natural Security

A Natural RPC Server may run under Natural Security to protect RPC requests. See *Natural Security*.

To authenticate an EntireX C Wrapper client against Natural Security

1. Specify a user ID and password in the parameters szUserId and szPassword of the ERX_CLIENT_IDENTIFICATION structure.
2. If different user IDs and/or passwords are used for EntireX Security and Natural Security, use the parameters szRpcUserId or szRpcPassword to provide the user IDs and/or passwords for Natural Security.

To force an EntireX C Wrapper Client to log on to a specific Natural library

1. Set the medium ERX_TM_BROKER_LIBRARY in the ERX_SERVER_ADDRESS structure.
2. Specify the correct Natural library in the ERX_SA_BROKER_LIBRARY structure in the szLibraryName parameter.
3. Set the parameter cNaturalLogon to "ERX_NATURAL LOGON_YES" in the ERX_SA_BROKER_LIBRARY structure. See also *Natural Logon or Changing the Library Name* in this document.

Using SSL

For an introduction to SSL and TLS, see *SSL or TLS and Certificates with EntireX*.

To use SSL or TLS

1. See *Running Broker with SSL or TLS Transport* under z/OS | UNIX | Windows for information on how to set up your environment.
2. Provide the SSL or TLS parameters on the pSSLParameter parameter in the ERX_CLIENT_IDENTIFICATION structure.

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`. See `COMPRESSLEVEL`.

Using Internationalization with the C Wrapper

It is assumed that you have read the document *Internationalization with EntireX* and are familiar with the various internationalization approaches described there.

The RPC runtime does not convert your application data (in RPC IDL type A, K, AV and KV fields) before it is sent to the Broker. The application's data is shipped as given by the application.

The EntireX RPC runtime running under the Windows operating system

- assumes by default that the data is given in the encoding of the Windows ANSI codepage configured for your system and
- sends the Windows ANSI codepage configured for your system as part of the locale string to tell the Broker the encoding of the data if communicating with a Broker version 7.2.*n* and above. If you want to adapt the Windows ANSI codepage, refer to the Regional Settings in the Windows Control Panel and your Windows documentation.

The EntireX RPC runtime running under the UNIX operating system

- does not send a codepage to the Broker as part of the locale string but
- assumes that the Broker's locale string defaults match. If they do not match, you will have to provide the codepage explicitly with the function `ERXSetCodepage`.

The C Wrapper programmer is responsible for providing suitable locale strings. See `ERXSetCodepage` under *API Function Descriptions*. With the function `ERXSetCodepage`:

- override or provide a codepage in the locale string sent to the broker. If a codepage is provided it must also be a codepage supported by the broker, depending on the internationalization approach, and it must follow the rules described under *Locale String Mapping*.
- force a locale string to be sent if communicating with Broker version 7.1.x and below. Under the Windows operating system, use the value `"LOCAL"` to send the default Windows ANSI codepage as the locale string to the broker.

Note:
The codepage setting is maintained inside the EntireX RPC Runtime on a per-thread basis. See *Using the RPC Runtime*. If you are using more than one codepage in a single thread, you have to provide the correct codepage before calling one of the following EntireX RPC Runtime functions:

- `ERXCall`
- `ERXConnect`
- `ERXDisconnect`
- `ERXDisconnectCommit`
- `ERXTerminateServer`
- `ERXIsServing`
- `ERXWait`

Other functions do not require a codepage.

If no locale string is provided by the C Wrapper programmer, an administrator can also force a locale string to be sent with the environment variable `ERX_CODEPAGE`.

When setting the codepage with the environment variable `ERX_CODEPAGE`:

- The `ERX_CODEPAGE` environment variable is ignored if the application programmer has already provided a codepage.
- The value of the `ERX_CODEPAGE` environment variable must be the name of the system's default codepage. Under Windows, simply apply the value `"LOCAL"` to specify the default Windows ANSI codepage.
- The codepage specified must be one that is supported by the Broker, depending on the Broker's internationalization approach. See *Locale String Mapping* for information on how the broker derives the codepage from the locale string.
- Before starting the application, set the locale string with the environment variable `ERX_CODEPAGE`.

Example:

```
ERX_CODEPAGE=LOCAL
```