RFC C++ Class Library (BC-FES-AIT)

HELP.BCFESDEA

Release 4.6C

SAP™

Copyright

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Contents

RFC C++ Class Library (BC-FES-AIT)

RFC C++ Class Library (BC-FES-AIT)

This document explains the relationships of the classes to one another and instructs you on how to work with the class library.

What's New in Release 4.6B?

New Feature

An auto-creating server feature [Page 22] has been added to the RFC C++ Class Library. This feature allows you to develop RFC servers that create parameters automatically for selected RFC server functions if these functions already exist in the target R/3 System.

New Functions

CRfcConnection Class

ConnectInfoFromConnectParam [Page 54]

ConnectParamFromConnectInfo [Page 55]

CrfcFunction Class

GetConnectionRfcHandle [Page 87]

CrfcServerApp Class

GetConnectionRfcHandle [Page 113]

UnInstallFunction [Page 120]

CrfcServerFunc Class

CallReceive [Page 121]

Clone [Page 122]

MultipleThreadDataProcess [Page 124]

ReceiveData [Page 125]

ReturnCall [Page 126]

Changed Functions

CRfcConnection Class

GetR3SystemInfoCount has been renamed R3SystemInfoCount [Page 67]

Overview

The Remote Function Call (RFC) Class Library is built on the Remote Function Call API, and is designed to provide ease-of-use. The RFC Class Library is useful for integrating SAP R/3 functionality with the desktop.

Who Should Use this Help File

SAP R/3 provides a Remote Function Call API that enables external programs to communicate with R/3 either as a client- or as a server-process. The API contains a set of C functions. In the past, programming with this API required low-level parameter packing and unpacking, internal table data manipulation, memory mapping, communication signal detection, and so forth. In addition, there were many actions that had to be coded the same way for all server or all client programs.

The Remote Function Call (RFC) Class Library has been developed so that low-level knowledge of R/3 and Remote Function Calls is not so important. You can use the RFC Class Library with any R/3 system that has been set up to receive remotely callable functions. Designed to simplify the programming of external applications that communicate with R/3, the RFC Class Library enables you to focus more attention upon your application logic and less upon the low-level details of RFC.

The class library contains a set of C++ classes that encapsulate the functionality of the RFC API. In addition, the RFC Class Library provides a C++ object-oriented framework for developers to use as a base for future effort with all the benefits that come with object-oriented programming.

This version of the class library supports the features of R/3 version 2.X. In addition, it supports R/3 version 3.X and 4.X features.

Who Should Use this Help File

This document is for developers who are going to apply RFC Class Library Functions to integrate R/3 with user desktops. It is aimed at people who are already using remote function calls (RFCs) and want to switch to a more object-oriented approach. Developers need knowledge of the R/3 system, particularly those remotely callable functions they need. To use RFC Class Libraries effectively, you also need to be able to program in the C++ language.

General Class Design Principles

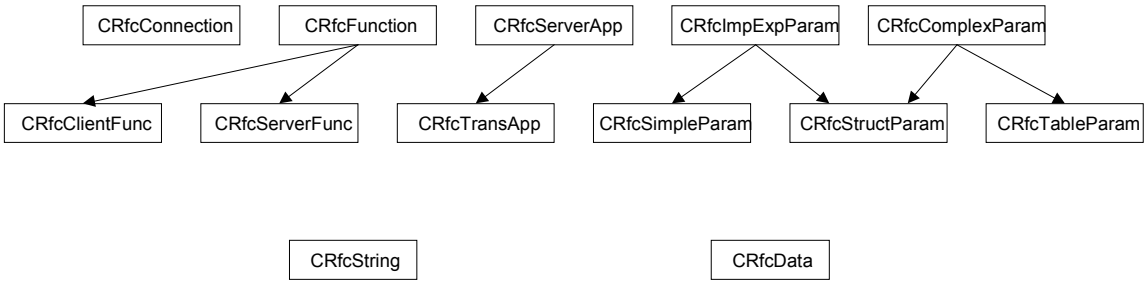
The following objectives were used in developing this implementation of RFC Class Libraries:

- Platform independent. ANSI C++ compliant.
- Major reduction in the effort to write an application for RFC.
- Execution speed comparable to that of the C-language RFC API.
- Minimum code size overhead.
- Easier conversion of existing C-language RFC applications to C++.
- Ability to leverage from the existing base of C-language RFC programming experience.
- Easier use of the RFC API with C++ than with C.
- Easier-to-use yet powerful abstractions of complicated operations such as the server application process loop, table management, and so forth.
- True RFC API for C++ that effectively uses C++ language features.
- Protect all data members of classes, only allow access through methods.

Class Hierarchy Diagram

The diagram below shows the inheritance relationships among the classes in the RFC Class Library:

Data Types and Heirarchy in RFC Function Library



Prerequisites

You must review the following documents to implement some required customizations before you can use the RFC C++ Class Library:

System Requirements [Page 13]

Building C++ RFC Client or Server Applications [Page 14]

Customizing the MS Visual C++ Compiler [Page 15]

System Requirements

Development Requirements

To create applications using the C++ RFC Class Library, you need the following:

- One of the following operating systems: Windows NT, Windows 95, Windows 98, Solaris, HP-UX or IBM OS/400. The library has not been tested with other operating systems.
- SAP R/3 Release 2.X or higher
- A C++ compiler. On systems running Win32 operating systems (Windows NT, Windows 95 and Windows 98), SAP has tested only the Microsoft Visual C++ 5.0 compiler. On systems running HP-UX, SAP has tested only the aCC and AC++ compilers. On systems running Sun Solaris OS, only Sun's WorkShop Compiler 4.2 has been checked. On the IBM AS/400 platform, SAP has used only the Visual Age C++ compiler.

SAP has not tested the library with any other compilers except those mentioned above. If you are going to use Microsoft Visual C++, please follow the steps in Customizing the MS Visual C++ Compiler [Page 15] before attempting to use this class library.

- The *librfc32.dll* and *librfc32.lib* files for Win32 operating systems, or the *librfc.a* file on Solaris, HP-UX, and IBM OS/400 systems. You receive the DLL and LIB files when you install the RFC library.
- The RFC header files *saprfc.h* and *sapitab.h*. These files are available with the RFC library release.
- All C++ RFC class library header files. These files are included in this release of the C++ Class Library.

Run-time Requirements

To use the applications developed with this C++ Class Library, you need the following:

- One of the following operating systems: Windows NT, Windows 95, Windows 98, Sun Solaris, HP-UX or IBM OS/400.
- SAP R/3 release 2.X or higher.
- The *librfc32.dll* for Win32 operating systems.

See also:

Building C++ RFC Client or Server Applications [Page 14]

Configuring the MS Visual C++ Compiler [Page 15]

Building C++ RFC Client or Server Applications

Purpose

Before you can develop and use C++ RFC applications, you must properly configure both your development and client workstations, and the R/3 System.

Prerequisites

See System Requirements [Page 13]

Process Flow

1. Set up the R/3 System so that your developed applications can call RFC functions or call functions in external programs through the R/3 gateway.
2. Set the paths in your C++ compiler to the RFC library and C++ RFC class library header files and libraries.
3. If you are developing applications for a Win32 operating system, you must also install a copy of the RFC dynamic link library (*librfc32.dll*) on all clients when you install RFC client or

server application executables developed with the C++ library. In other operating systems, you only need to distribute the executables.

4. If you develop C++ RFC applications in Microsoft Visual C++ 5.0 running on Windows NT 4.0, you must change some default settings to successfully compile and link applications. See Configuring the MS Visual C++ Compiler [Page 15] for more information about this process.
5. To develop an RFC server application, you must derive a sub-class from CRfcServerAPP because it is an abstract class. We also recommend that you derive a subclass from CRfcClientFunct if you develop RFC client applications.

Customizing the MS Visual C++ Compiler

Use

If you use Microsoft Visual C++ 5.0 running on Windows NT 4.0, you must change some default settings before you can successfully compile and link applications.

Procedure

1. Start Microsoft Visual C++
2. Choose *Project* → *Settings* → C/C++
3. From the *Category* field, select *Code Generation* from the drop down menu. The fields displayed on the tab change.
4. From the *Use run-time library* drop down, select *Debug multithreaded DLL*.
5. Save the changes.

If this is a debug version, you also need to do the following:

1. Choose Project → Settings → C/C++
2. From the *Category* field, select *General* from the drop down menu. The fields displayed on the tab change.
3. From the *Debug info* drop down, select *C7 compatible*.
4. Save the changes.

Major Class Categories

The classes of the RFC Class Library can be described in five groups, shown here.

The Connection Class [Page 16]

The Function Classes [Page 16]

The Server Application Classes [Page 16]

The Parameter Classes [Page 17]

Other Auxiliary Classes [Page 18]

The Connection Class

The CRfcConnection Class [Page 47] establishes connections to R/3 following the user-specified connection mode. You can also use it to verify user logon information. For Windows applications, all logon and connection information can be written and read from the registry.

The Function Classes

The CRfcFunction Class [Page 83] is an abstract class that encapsulates the actual remote function invocation (sending and receiving data) to and by R/3.

It packs parameter information in the appropriate RFC format before sending data over and it unpacks information after receiving data back. It contains a reference to a CRfcConnection object [Page 47], two lists of CRfcImpExpParam objects [Page 100] for importing and exporting parameters, and a list of CRfcTableParam objects [Page 160] for table parameters. (See The Parameter Classes [Page 17]) The *CRfcFunction* class is thus a container of parameter objects. Its descendant class, *CRfcClientFunc*, encapsulates RFC function module calls as a client. The other descendant, *CRfcServerFunc* encapsulates data processing when called as a server.

General descriptions of the function classes are available under the following topics:

What the CRfcFunction Class Represents [Page 18]

How the CRfcClientFunc Class Works [Page 19]

How the CRfcServerFunc Class Works [Page 19]

Complete details are provided in the class library reference topics shown here:

The CRfcFunction Class [Page 83]

The CRfcClientFunc Class [Page 32]

The CRfcServerFunc Class [Page 120]

The Server Application Classes

The CRfcServerApp Class [Page 109] contains the logic of a server application program. The Run [Page 118] method encapsulates the main process loop for listening to incoming requests and dispatching them to appropriate server functions. It contains a list of CRfcServerFunc objects [Page 120] that correspond to all server functions that the application supports. Its descendant, CRfcTransApp [Page 192] contains transactional RFC functionality for R/3 3.0.

General descriptions about the server application classes are available under the following topics:

The CRfcServerApp Class: Encapsulation of the Server Process Loop [Page 20]

Using the CRfcServerApp and Derived Classes [Page 21]

The CRfcTransApp Class and Transactional RFC [Page 23]

Getting System Information from the Calling System [Page 24]

Complete details are provided in the class library reference topics shown here:

The CRfcServerApp Class [Page 109]

The CRfcTransApp Class [Page 192]

The Parameter Classes

The parameter class group consists of two abstract classes and three derived classes.

CRfcImpExpParam [Page 100] is an abstract class that contains common features for RFC import and export parameters. One descendant, CRfcSimpleParam [Page 127], represents import or export parameters with a single data type. The other descendant, CRfcStructParam [Page 149], represents import or export parameters with a complex data type, such as a structure with multiple fields.

The CRfcComplexParam [Page 40] class is an abstract class that contains common features for RFC parameters that involve complex data types, such as structure parameters and table parameters. It keeps a list of definitions for each field in the complex structure. The CRfcStructParam [Page 149] multiply inherits from both *CRfcImpExpParam* and *CRfcComplexParam* since it needs to contain features from both classes. The CRfcTableParam [Page 160] descends directly from *CRfcComplexParam*. It encapsulates all internal table-related operations: getting cell values, setting cell values, reading and writing a whole row, appending, inserting, and deleting rows, and others.

General descriptions about the parameter classes are available under the following topics:

Setting and Getting Values of Parameter Objects [Page 24]

Example of Set and Get Value Methods [Page 26]

Row Manipulation of the CRfcTableParam Class [Page 26]

Complete details are provided in the class library reference topics shown here:

The CRfcImpExpParam Class [Page 100]

The CRfcSimpleParam Class [Page 127]

The CRfcStructParam Class [Page 149]

The CRfcComplexParam Class [Page 40]

The CRfcTableParm Class [Page 160]

Other Auxiliary Classes

The two remaining classes are *CRfcString* and *CRfcData*; these are auxiliary classes that are used by various classes mentioned above.

The CRfcString Class [Page 134] contains some basic string operations.

The CRfcData Class [Page 72] objects are used as intermediate objects when the user sets and gets values from any of the parameter objects. It automatically does appropriate data conversion before the set and get procedures act on a value, given that the related parameter object contains data type information. Normally, this class should not be the concern of the user.

Working With the Class Library

What the CRfcFunction Class Represents

The CRfcFunction Class [Page 83] directly corresponds to remotely callable function modules. For each remotely callable function, an instance of a *CRfcFunction*-derived class (whether CRfcClientFunc [Page 32] or CRfcServerFunc [Page 120]) can be created. The instance then acts as a container for all parameters required for its function. It packs and unpacks parameter values to process RFC calls.

The *CRfcFunction* class keeps as its data members three lists:

- a list of CRfcImpExpParam objects [Page 100] for importing parameters,
- a list of CRfcImpExpParam objects [Page 100] for exporting parameters

- a list of CRfcTableParam objects [Page 160] for table parameters.

It also keeps two references: one to a CRfcConnection object [Page 47] that keeps a live connection to R/3, and another to a string to store the function name.

No parameter object is contained in the function object immediately after it is created. The user of the function object is required to create all parameter objects needed by the function, and to provide their pointers to the function object. The function object then adds these to its appropriate parameter lists.

Note that a function object only keeps records of parameter object pointers in its lists. The function object is not responsible for creating and destroying the parameter objects. When the function object is destroyed, the object pointers stored in its lists are simply purged from memory. The actual destruction of these parameter objects is up to the user. The same is true for the connection object pointer stored in the function object.

How the CRfcClientFunc Class Works

The CRfcClientFunc Class [Page 32] is designed to encapsulate all calling actions that can be made to R/3 as a client. When you provide parameter values required by the function before the call, the *CRfcClientFunc* class uses methods inherited from the CRfcFunction class [Page 83] to carry out the following actions:

- Reads parameter values from parameter objects
- Packs the information into RFC format before each call
- Unpacks values after the call
- Has the values written into appropriate parameter objects

After a call, if there are returned values, they can be easily retrieved from the individual parameter objects.

How the CRfcServerFunc Class Works

The CRfcServerFunc Class [Page 120] is designed to encapsulate all calls for processing actions that can be made to R/3 as a server function. This class uses methods inherited from the CRfcFunction class [Page 83] to carry out the following actions:

- unpacks data received from R/3 and writes this data into appropriate parameter objects (the *ReceiveData* member function is not directly available to the user)
- does the appropriate processing (See the Process [Page 124] member function)
- reads data from parameter objects and packs such data for sending to R/3 (the *ReturnCall* member function is not directly available to the user)

The three member functions listed above are automatically called by CRfcServerApp [Page 20]

The *CRfcServerFunc* class is an abstract, base class for the custom server function classes needed by users. It establishes a framework by providing an overridable pure virtual function known as Process() [Page 124]for the derived classes to fill with function-specific processing logic.

Another overridable member function of the *CRfcServerFunc* class is GetDescription [Page 123] You can use it to return a text description of the particular server function represented by the class. This function is automatically called when testing in R/3 (3.0 onwards) under Transaction SM59 using the *Function List* utility. The default implementation of *GetDescription* returns NULL.

In summary, the only major work that needs to be done by the user of the *CRfcServerFunc* class is to implement the function-specific logic in the *Process()* method of his/her own custom server function class, which is derived from *CRfcServerFunc*.

The CRfcServerApp Class: Encapsulation of the Server Process Loop

The CRfcServerApp Class [Page 109] encapsulates the initialization, running and termination of an RFC server application. (Running consists of listening for requests from R/3 and dispatching them to corresponding functions). An application built with the framework provided by the class library must have exactly one object of the *CRfcServerApp* class. We refer to this as the "application object."

The application object keeps a pointer list of objects of classes derived from CRfcServerFunc [Page 83], which we refer to as "server function objects". These server function objects, in turn, are known as the user server functions. User server functions are user-defined, application-specific functions to be called from R/3. The user implements these functions as *CRfcServerFunc* classes (see How the CRfcServerFunc Class Works [Page 19]) and registers them with the *CRfcServerApp* object during runtime.

The application object's Run [Page 118] member function contains the main loop for server applications. In the loop, as a regular process for server programs, it keeps listening for incoming requests from R/3. Upon getting a request, the *Run* member function calls the C API to look for a matching function in the server function list. *Run* then locates the server function object by calling its *ReceiveData*, Process() [Page 124], and *ReturnCall* member functions one after the other and asks the C API to dispatch processing to the server function object. After each request is processed, *Run* goes back to the listening mode, ready for the next request until the other end indicates intent for termination.

When you design and implement the derived classes of *CRfcServerFunc*, you are freed from the need to concern yourself with retrieving data sent from R/3 and with sending data back after processing. Instead, you can concentrate on the function processing logic: Once inside the *Process* method, all data sent by R/3 is already stored in the appropriate parameter objects in C++ format, so functional data processing can start right away. When you exit the *Process* method, all appropriate data is automatically sent back to R/3. All this automation is done by the application object.

The application object's OnIdle [Page 115] member function can be overridden by the user who wishes to do some optional side work between requests. *OnIdle* is called in the main loop of *Run* while *Run* waits for requests. If you do not intend to override *OnIdle*, you can use the *CRfcServerApp* class without modification.

Using the CRfcServerApp and Derived Classes

In the main program of the user's server application, you can use the following steps to integrate R/3 functionality with the user desktop:

1. Establish a connection to R/3.
2. Declare an instance of the *CRfcServerApp* or its derived class.
3. Create the custom server function objects and add them to the application object.
4. Invoke the application object's *Run.*

(See the server application sample, "Server.cpp" under the topic Programming Examples [Page 196], for details.)

Writing Multithreading RFC Server Applications

You can use the class library to develop multithreading RFC clients and servers with a minimum of multithreading knowledge. The C++ class library allows you to develop multithreading RFC servers in almost the same way as you develop normal RFC servers; the library encapsulates all the details so you do not have to worry about them.

To write an RFC server with the class library, you must first derive a new class from CRfcServerFunc [Page 120] (CRfcServerFunc is an abstract class). You must also define the member function Process() [Page 124] in the derived class.

If you want to develop a multithreading RFC server using the class library, you must define a new member function, clone() [Page 111], in the derived class. Usually, it is easier to implement a copy constructor in the derived class, and then define the clone() function. Unlike Process(), the clone() member function is not a pure virtual function. This means that if you do not implement this function in the derived class, your program will still compile and run. If your application does not use multithreading, your application will run correctly without this function. However, if your application IS multithreading, and you do NOT implement a clone() function, you will receive a run-time error.

After you implement the clone() function for the CRfcServerFunc derived class, a CRfc server application performs almost like a non-multithreading RFC server program. The only exception is when an application calls the CRfcServerApp class Run() [Page 118] function. In this case, you need to define the following argument:

```
Run(int nThreads=0)
```

The previous definition of this function had no arguments. If the application does not pass an argument, or if the application passes an argument with a value of zero or less to the function, the CRfc server program does not multithread the application. In this case, you do not need to implement a clone() function.

The parameter nThreads is the number of threads you want the class library to create for this server application. Currently, you can use up to 1000 threads with the library.

Determining the Number of Threads

How many threads does your application need? There is no firm rule, so use your experience to guide you. Below are some tips to help you select the number of threads:

Case 1: The server has only one CPU and very few clients.

It is very unlikely that more than one client will call this server simultaneously. If so, then do NOT use multithreading. You can call Run without an argument, or use zero as the argument. You do not need to implement the clone() function in the derived CRfcServerFunc class.

Case 2: The server has more than one CPU and many clients.

Usually, there will be only one client at a time calling this server. Do NOT use multithreading.

Case 3: The server has more than one CPU.

There might be more than one client calling the server simultaneously. However, the number of clients is usually smaller than the number of CPUs. Set the number of child threads equal to the number of CPUs.

Case 4: The server has one or more CPUs.

Usually the number of clients calling the server is greater than the number of CPUs. In this instance, the number of child threads needs to be selected experimentally. Your client response and total processing time are the determining factors.

If you want to monitor which threads are performing which tasks at what times, you can access the global variable, **`rfc_thread[10000]`**, exposed by the class library **`(extern int rfc_thread[10000];)`**. This array holds the IDs of threads created by the class library. The array index is the RFC_HANDLE for the RFC connection.

For example, using the derived class Process() member function, (this function must be derived because it is defined as a virtual function in CRfcServerFunc), a user can display the activities of each thread on the console or a GUI. To do this, get the RFC_HANDLE, by using the following syntax:

```
(RFC_HANDLE RfcHandle; GET_VERIFIED_CONNECTION(RfcHandle);)
```

You can then use RFC_HANDLE RfcHandle to access the ID **`(rfc_thread[RfcHandle])`** of the current running thread. For details, please see the sample programs.

To conclude, if you have already developed an Rfc server with the class library, it is easy to implement multithreading. You only need to use the clone() function and a copy constructor derived from the CRfcServerFunc, and enter the number of threads as the argument when calling **`Run(int nThreads=0)`**. For more help on developing and testing a multithreaded server with the class library, please see the ThreadServer.h [Ext.] and ThreadServer.cpp [Ext.] programming examples.

Developing RFC Servers with RFC Server Functions

An auto-creating server function has been added to the RFC C++ Class Library. This feature allows you to develop RFC servers that create parameters automatically for selected RFC server functions if these functions already exist in the selected R/3 System.

Previous to Release 4.6B, the RFC C++ Class Library allowed you to install and register with a destination, one or more RFC server functions on an RFC server. This feature enabled the server to use an RFC connection to listen for the incoming RFC function calls from the RFC clients. Additionally, this feature allowed RFC clients to request services from one or more RFC server functions. However, if some of these RFC functions were already being used or could be created by your R/3 System, it would be very useful to retrieve the parameter information for these functions. To meet this need, SAP added an auto-creating server function at Release 4.6B.

To use one of these RFC functions, you need to supply connection information to the R/3 System. Once you do this, the function automatically retrieves the parameter information. However, these connections are different from the one the server uses to listen to incoming RFC function calls.

For an example on how to develop a RFC server with the auto-creating server feature, please see the sample programs *crautsrv.cpp* and *crautsrv.h*. After starting the sample server program, you can write RFC clients to test this RFC server. You can do this either by writing an ABAP client program or an external C++ client. There is a sample ABAP client program inside the *crautsrv.h* file. For help writing an external C++ client program to test the auto-creating RFC server feature, please see the *crtstasv.cpp* and *crtstasv.h* files.

The CRfcTransApp Class and Transactional RFC

The CRfcTransApp Class [Page 192] is a direct descendent of the CRfcServerApp Class [Page 109] It is an abstract class not to be used as is. It encapsulates the initialization, running and termination of a transactional RFC server application. This application is available in R/3 version 3.0 and onward. A transactional RFC server application (built with the framework provided by the class library) must have exactly one object of a class derived from *CRfcTransApp*. We will call this the “transaction application object.

This class inherits the following from the *CRfcServerApp* class:

- the encapsulation of the main server processing loop
- the management of the server function list
- the dispatching of requests from R/3 to appropriate server function objects

In addition there are four virtual functions specific to transactional RFCs that must be overridden by the user.

These four virtual functions are similar to the four callback functions you implement and install in the RFC C-API. They are:

- OnCheckID: [Page 193] This function is called when a local transaction starts. It must be used to check whether the transaction is already running or has already been completed.
- OnCommit: [Page 194] This function is called when the local transaction ends. It commits the transaction locally.
- OnRollBack: [Page 194] This function is called if the local transaction fails because of an error during the processing inside the RFC library. It should be used to roll back a failed local transaction.
- OnConfirmID: [Page 195] This function is called when the local transaction is confirmed. All information stored locally about this transaction can be deleted.

These four function are automatically called at runtime by the RFC Class Library when appropriate.

Getting System Information from the Calling System

After receiving a call from the R/3 system, a server application that implements the function `CRfcServerFunc::Process()` can access system information using the CRfcConnection object attached to the parent CRfcServerApp or CRfcTransApp.

Within each CRfcServerFunc object, a pointer (`m_pServerApp`) is stored. This pointer can be used to access the parent CRfcServerApp or CRfcTransApp object that contain a CRfcConnection object used for accepting the RFC call from R/3.

Accessing that CRfcConnection object (using `CRfcServerApp::GetConnection()`), the server application can then obtain information about the calling R/3 by calling `CRfcConnection::GetR3Release()`, `CRfcConnection::GetReleaseString()`, `CRfcConnection::GetR3SystemInfo()`, `CRfcConnection::R3SystemInfoCount()`, and two overloaded `CRfcConnection::R3SystemInfo()` functions.

Setting and Getting Values of Parameter Objects

For all parameter classes (CRfcSimpleParam [Page 127], CRfcStructParam [Page 149], CRfcTableParam [Page 160]), setting or getting individual field values is done through use of an auxiliary class, *CRfcData*.

A CRfcData class object [Page 72] represents a single value of an RFC data type. It carries in itself some type-related information, such as the data type, the number of bytes the type needs, and the number of decimal points. The *CRfcData* object also carries a pointer to a block of memory where the value of the data it represents resides.

The *CRfcData* class has two sets of operators to enable it to represent any RFC data type and to allow automatic type conversion when setting or getting values of the data it represents.

One set of operators are assignment operators (*operator* =) that take various types of data as right-hand-side values. These include:

- *const char** for all character string types
- RFC_INT for all integer types
- RFC_FLOAT for all floating point types
- RFC_CHAR[] for all data array types.

These operators allow values to be written into the data represented by the *CRfcData* object. Data type-checking is done automatically. For example, *operator* = with *const char** on the right-hand-side is only allowed when the data type carried with the *CRfcData* object is of a character string type.

The other set of operators are conversion operators that take various types of variables as left-hand-side parameters:

- *const char** for all character string types
- RFC_INT for all integer types
- RFC_FLOAT for all floating point types

These operators allow values to be read from the data represented by the *CRfcData* object and assigned to the user-provided left-hand-side variable. Data type-checking is again automatically

done. For example, operator RFC_INT is only allowed when the data type carried with the *CRfcData* object is of the RFC_INT type.

The method names for setting and getting actual values of the three basic types of parameters are:

- *Value* for *CRfcSimpleParam*
- *Operator[ ]* for *CRfcStructParam*
- *Cell* or *Field* for *CRfcTableParam*

Each of these methods finds out the necessary type information related to the parameter/field in question, sets up an *CRfcData* object with the type information, points the *CRfcData* object's inner data pointer to the appropriate memory, and returns a reference to this object, whose appropriate operator is then called depending on the operation.

Removing Parameter Objects

In releases prior to 4.5B, users of the C++ class library could both access parameter objects in CRfcFunction [Page 83] object containers and add parameter objects. However, they could not remove any parameter objects from the containers. As of Release 4.5B, users can use interfaces to remove parameter objects from CRfcFunction object containers. SAP has provided six class CRfcFunction public member functions to remove parameter object functions. These functions are:

- RemoveExportParam (by Index) [Page 92]
- RemoveExportParam (by Name) [Page 93]
- RemoveImportParam (by Index) [Page 93]
- RemoveImportParam (by Name) [Page 94]
- RemoveTableParam (by Index) [Page 95]
- RemoveTableParam (by Name) [Page 95]

If parameter objects are automatically created by the class library, these remove functions not only destroy the parameter objects in the CRfcFunction object, but they also delete them. However, if these parameter objects are created by application programs, they are not deleted. The user's application programs must delete them explicitly.

Replacing Parameter Objects

In some applications, users of the class library need to replace the existing parameter objects in the CRfcFunction [Page 83] object containers with other parameter objects of the same type. As of Release 4.5B, users can use three CRfcFunction public member class functions to replace parameter object functions. These functions include:

- ReplaceExportParam [Page 96]
- ReplaceImportParam [Page 96]
- ReplaceTableParam [Page 97]

Example of Set and Get Value Methods

These functions replace the parameter objects in the CRfcFunction object containers with new parameter objects. If an application automatically creates existing parameter objects, the CRfcFunction object deletes the existing parameter objects when the program destroys the CRfcFunction objects. However, if an application program creates parameter objects for a release earlier than 4.5B, these parameter objects will remain in memory after these replacement operations; the CRfcFunction object will no longer be able to obtain references to these objects. To delete these objects, you will have to keep references to them in their application programs.

Example of Set and Get Value Methods

This example shows how to use the *Value* member function from CRfcSimpleParam [Page 127]. (The use of the member functions *operator[ ]* from CRfcStructParam [Page 149], and *Cell* and *Field* from CRfcTableParam [Page 160] is similar.)

Assuming *AnIntParam* is a simple parameter of type RFC_INT,

```
AnIntParam.Value() = 5;  // operator = (RFC_INT) is called on the
                         // intermediate CRfcData object return by
                         // Value(). The parameter's value
                         // is set to 5.
int anInt = AnIntParam.Value();
                         // operator RFC_INT() is called on the
                         // intermediate RfcData object return by
                         // Value(). The parameter's value is
                         // assigned to anInt.
```

Using Default Values for Simple Import Parameters

From release 4.0A on, the RFC class library lets external applications use default parameter values when making client RFC calls to R/3 functions. That is, callers can use the values provided by the called function, instead of providing a value explicitly in the call.

To do this, the calling application simply refrains from assigning values (using the `Value()` function of `CRfcSimpleParam` class) to the simple import parameters for which the default values are to be used. The RFC class library will also refrain from passing those parameters with no value assigned to R/3 during the call.

If a simple import parameter is assigned a value, and was passed to R/3 for an RFC call, then if the same simple import parameter with the same value is to be passed again to R/3 for the next call, then one of the two following actions can be taken: (1) Assign that same value to that parameter again, or, (2) when calling `CRfcClientFunc::Call(...)` or `CRfcClientFunc::CallReceive(...)`, set the boolean argument `bPassAllParams` to TRUE. Both ways would ensure that simple import parameter to be passed to R/3 again, but the second way would pass all the other simple import parameters.

Row Manipulation of the CRfcTableParam Class

The CRfcTableParam [Page 160] class provides methods for setting and getting individual cell values either by row number and column number, or by row number and column name.

See the *Cell* member functions, including the following:

- Cell (by Row/Column Number) [Page 165]

  Gets or sets individual cell values by row and column numbers

- Cell (by Row/Column Name) [Page 164]

  Gets or sets individual cell values by row number and column name

- Cell (Constant Table, by Row/Column Number) [Page 168]

  Gets or sets individual cell values by row and column numbers from constant table parameter objects

- Cell (Constant Table, by Row/Column Name) [Page 167]

  Gets or sets individual cell values by row and column numbers from constant table parameter objects

In addition, the class provides a set of methods for manipulating table data by rows.

A table object keeps as its data member a “current row pointer” that points to a block of memory storing a particular row’s data. The iterator operator [Page 186] advances the row pointer by one row each time it is called. Upon reaching the end of the table, it is set back to before the beginning of the table, so that after the next time it is called, the current row pointer is set to the zeroth row. (Note that all indexes in the class library start from 0.)

The member function SetCurrentRow [Page 189] forces the current row pointer to point to a specific row. The *Field* member functions are used for setting and getting particular field values of the current row. In addition, AppendEmptyRow [Page 163] and InsertEmptyRow [Page 183] each creates and adds an empty row to the table in append and insert mode, respectively, and sets the current row pointer to point to the newly added row. The RemoveRow [Page 187] member function also updates the current row pointer to point to the next row if the current row happens to be the one that is to be removed.

The remaining four row manipulation member functions are similar to their RFC C-API counterparts: ConstRow [Page 170] returns a *const void* pointer to a memory block for the specified row, which only allows read access, Row [Page 188] returns a void pointer to a memory block for the specified row, which allows both read and write access. CopyLineTo [Page 172] copies the content of a row to a memory block specified by the caller. CopyLineFrom [Page 171] copies the content of a memory block specified by the caller to a row.

Accessing Field Values in a Parameter with Complex Structures

The class library provides methods to access individual field values in a structure parameter (the operator **[ ]**) or a table parameter (the member functions *Field* or *Cell*). These member functions all have a parameter that is either an index or a field name. The user must provide related field information to the parameter objects before these objects can be called to access the target fields. This information includes a field’s name, its column number in the structure or table, its data type, its byte offset from the beginning, its length, and the number of decimal points it contains. Unless the information is available, the parameter object does not know (given a field index or name) how to find the correct memory block to map to a *CRfcData* object.

The classes CRfcStructParam [Page 149] and CRfcTableParam [Page 160] have a common ancestor, CRfcComplexParam [Page 40], whose member functions are designed to allow users

to provide and retrieve individual field information. To provide field information, use either of the *AddFieldInfo* methods:

- AddFieldInfo (Multiple Fields) [Page 41]

  Adds information for multiple fields into the parameter

- AddFieldInfo (Single Field) [Page 41]

  Adds information for a single field into the parameter

As noted, the first method shown above other allows you to fill in an array of the structure RFC_FIELD_INFO [Page 41] for multiple fields and then enter them in the object all at once. The second one will allow you to add one field information item at a time. The RFC_FIELD_INFO [Page 41] structure contains a single field's information, such as name, offset, type, length, and so on. (For either of the *AddFieldInfo* methods, if a field is of a type with fixed-length, then the length information is not required.)

Note that the order of providing field information does not matter. It is also not necessary to provide field information for every field in the structure or table. If you are only going to access certain fields, you need only to provide information for those fields. If you do not need to access any individual field values, then no field information need be provided

Class Library Reference

The RFC Class Library has thirteen classes, each of which has one or more construction commands. Each of the classes also has multiple operations commands. The following topics help you access this information:

Structures and Globals

For information on structures and global definitions, see:

The RFC_USER_INFO structure

Defined in CRfcConn.h

The RFC_USER_INFO structure is used for providing user logon information to connection objects. It contains the following fields:

Field Name	Type	Description
rstrClient	CRfcString	Name of the client

rstrUserName	CRfcString	User name
rstrPassword	CRfcString	Password
rstrLanguage	CRfcString	Language

For a CRfcString variable, a character string can be assigned to it and obtained from it using appropriate operators. See the CRfcString [Page 134] class for details.

The RFC_CONNECT_INFO structure

Defined in CRfcConn.h

The RFC_CONNECT_INFO structure is used for providing connection information to connection objects. It contains the following fields:

Field Name	Type	Description
rfcMode	RFC_MODE	The connection mode
rstrR3release	CRfcString	Release version of R/3 system being connected to
bGetR3SystemInfo	BOOL	Indicates to `CRfcConnection::Open()` and `CRfcConnection::SafeOpen()` whether to automatically call `CRfcConnection::GetR3SystemInfo()`
nSAPGUI	RFC_INT	Whether or not to enable SAP GUI in the RFC call
rstrDestination	CRfcString	Destination name of the target system
rstrHostName	CRfcString	Host machine name of the target system
rstrGatewayHost	CRfcString	Gateway host name
rstrGatewayService	CRfcString	Gateway service name
nSystemNo	int	System number
rstrSystemName	CRfcString	R/3 system name. For load-balancing connection only
rstrMsgServer	CRfcString	Host name of the message server. For load-balancing connection only
rstrGroupName	CRfcString	Name of the specific group of application servers. For load-balancing only
cRfcServerType	char	2, 3, or E for R/2, R/3 system, or an external RFC server

rstrProgramID	CRfcString	Path and name of an external RFC server program if the RFC server will be started by the SAP gateway or Program ID of a registered RFC server program if the RFC server is already registered as a RFC destination on R/3 system using sm59 transaction on R/3.

For a CRfcString variable, a character string can be assigned to it and obtained from it using appropriate operators. See the CRfcString [Page 134] class for details.

The RFC_FIELD_INFO Structure

Defined in CRfcComp.h

The RFC_FIELD_INFO structure is used for providing field information to complex parameter objects. It contains the following fields:

Field Name	Type	Description
strName	char[]	Name of the field
nColumn	int	Column number of the field
nOffset	int	Byte offset of the field from the beginning
nType	unsigned	RFC data type of the field
nLength	int	Length in bytes of the field
nDecimals	int	Number of decimal positions contained in the field

The RFC API RFC_ERROR_INFO_EX Structure

The RFC API (underlying the RFC C++ Class Library) throws the RFC_ERROR_INFO_EX structure containing RFC error message information. The RFC_ERROR_INFO_EX structure is returned by the RFC RfcLastErrorEx function, which describes the last RFC error.

In previous versions of the RFC API, the RfcLastError function served this function, and it returned the RFC_ERROR_INFO structure with error information.

For compatibility with previous versions, RFC C++ Class Library functions that used to throw the RFC_ERROR_INFO structure still do so. However, SAP recommends that you check for the new RFC_ERROR_INFO_EX structure instead in any new code you develop. The use of the RFC_ERROR_INFO structure will be phased out in future releases.

Some global definitions

Defined in RfcGlobl.h

Global constants:

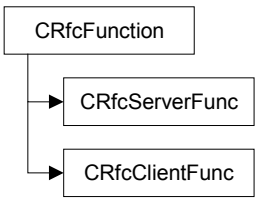
Field Name	Description
FIELDNAME_LEN	Maximum length of a field name
PARAMNAME_LEN	Maximum length of a parameter name
MAX_SAPCOLUMN_LEN	Maximum length of a column or field
_rfcChNil	Null character
TRUE	Logical value for TRUE, equals 1
FALSE	Logical value for FALSE, equals 0
NULL	Value for a null pointer
MEMORY	A string indicating memory allocation failure

Global type defines:

Define	Definition
BYTE	Byte is an eight-bit value, unsigned char
BOOL	Boolean type, int
CSTR	A constant string, const char

The CRfcClientFunc Class

Class Summary



The CrfcClientFunc class is defined in CRfcClnt.h.

The CRfcClientFunc class contains functionality that is needed for making calls to R/3 as a client. Each CRfcClientFunc object represents one remotely-callable function module in R/3.

Users can choose to let this class automatically construct all parameters associated with the RFC function. This includes importing and exporting parameters (simple and structure) and table parameters. To choose automatic construction, you simply set the bAutoCreate argument to TRUE when calling the constructor to instantiate an object.

Automatic generation of the parameters for the client function object cannot be done until a valid connection exists between the client machine and the RFC server. In other words, to construct a client function object, the connection object being passed in to the constructor for this class as argument must have a valid RFC handle (non-zero).

Construction

CRfcClientFunc [Page 33]

Constructs a client function object

Operations

Call [Page 34]

Makes one-way call to R/3; sends import parameters and table data

Receive [Page 38]

Makes one-way call to R/3; receives export parameters and table data

CallReceive [Page 35]

Makes two-way call to R/3; sends and receives import and export parameters and table data

CreateTransID [Page 36]

Obtains a transaction ID from R/3 and stores it in the class (Version 3.0+)

IndirectCall [Page 36]

Makes a one-way transactional call to R/3 (Version 3.0+)

Read [Page 37]

Restores metadata information about this CRfcClientFunc object from persistent stream.

Write [Page 39]

Writes metadata information for this CRfcClientFunc object to persistent stream.

CRfcClientFunc

Purpose

Constructs a client function object.

Syntax

```
CRfcClientFunc (const CRfcConnection* pConnection=NULL,
                        CSTR strName=NULL, BOOL bAutoCreate=FALSE);
```

Parameters

- *pConnection*: Pointer to a connection object.
- *strName*: Name of the remotely-callable function module.
- *bAutoCreate*: TRUE indicates that the constructor itself is responsible for establishing all parameters associated with the function.

Exceptions

Throws const char* for memory exceptions and for failure of automatic parameter creation.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

See “Description” for the constructor of the CRfcFunction class.

Related Information

CRfcFunction [Page 86]

Call

Purpose

Makes a one-way call to R/3, sending import parameters and table data.

Syntax

```
void Call(BOOL bPassAllParams=FALSE);
```

Parameters

bPassAllParams: Indicates whether all simple import parameters should be passed to R/3 for the call.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

Before making this function call, make sure all necessary import and table parameters are added to the function object, and any values to be sent via these parameters are set. Note that when argument bPassAllParams is set to TRUE, then all simple import parameters are passed to R/3

for the call; otherwise only the simple import parameters with values assigned to them are passed.

Related Information

Receive [Page 38]

CallReceive [Page 35]

CallReceive

Purpose

Makes a two-way call to R/3, sending and receiving import and export parameters and table data.

Syntax

```
RFC_RC CallReceive(char*& strException,
                   BOOL bPassAllParams=FALSE);
```

Return Value

Same as Receive.

Parameters

strException: A reference to a string, to be filled with error information when an RFC exception is encountered.

bPassAllParams: Indicates whether all simple import parameters should be passed to R/3 for the call.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

This function is equivalent to calling Call and Receive in a row. Make sure all import, export and table parameters are added to the function object, and all data that need to be sent are set into the corresponding import or table parameters. Returned values are immediately available from the export and table parameter objects upon returning from CallReceive.

Note that when argument bPassAllParams is set to TRUE, then all simple import parameters are passed to R/3 for the call; otherwise only the simple import parameters with values assigned to them are passed.

Related Information

Call [Page 34]

Receive [Page 38]

CreateTransID

Purpose

Obtains a transaction ID from R/3, and stores it in the class. R/3 3.0 onwards only.

Syntax

```
void CreateTransID (void);
```

Exceptions

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

This function has to be called before each successful transactional RFC call. The transaction ID is stored inside the parent class (CRfcFunction::rfcTID) to be used for the subsequent call of a function module in R/3 using the transactional RFC interface, IndirectCall.

If an error occurs (e.g. communication error) during the call of a function module in R/3 via IndirectCall, the RFC client program has to reconnect the RFC connection and repeat the IndirectCall without creating a new TransID.

User is responsible for storing this transaction ID into permanent store when necessary.

Related Information

IndirectCall [Page 36]

IndirectCall

Purpose

Makes a one-way transactional call to R/3. R/3 3.0 onward only.

Syntax

```
void IndirectCall(void);
```

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO and RFC_ERROR_INFO_EX for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

Make sure that a transaction ID has been obtained using CreateTransID, that all import and table parameters are added to the function object, and that all values to be sent over are set into corresponding parameter objects before making this call.

The function module will always be executed exactly once in R/3 even if the client program calls IndirectCall multiple times for the same transaction. You may wish to issue a call to IndirectCall multiple times in the following cases:

- If an error occurs (for example, a communication error) during the call of a function module in R/3 via IndirectCall. The RFC client program has to reconnect the RFC connection and repeat the IndirectCall without creating a new TransID.
- If the error occurs after the first IndirectCall is successfully returned (but before the TID is changed) the client program may repeat an IndirectCall call with the same TID.

Do not create the TID more than once for the same transaction.

Related Information

CreateTransID [Page 36]

Read

Purpose

Restores metadata information about this CRfcClientFunc object from persistent stream.

Syntax

```
BOOL Read(CRfcPersistStream& Pers);
```

Return Value

Returns TRUE when all metadata is successfully restored; FALSE otherwise.

Parameters

Pers: Reference to a CRfcPersistStream object that encapsulates the persistent data stream.

Exceptions

Throws const char* for memory exceptions.

Description

It is important that the given CRfcPersistStream object actually contains the complete metadata for this CRfcClientFunc object. Incomplete metadata causes the restoration to fail.

Receive

Important Note: care must be taken to set the function name of this object using SetFunctionName() before calling Read() to ensure that the Read() function finds the pertinent data inside the persistent stream.

Related Information

The CRfcPersistStream Class [Page 105]

Write [Page 39]

Receive

Purpose

Makes a one-way call to R/3, receiving export parameters and table data.

Syntax

```
RFC_RC Receive(char*& strException);
```

Return Value

The possible return codes are:

RFC_OK

The call was successfully completed and the values of the returned parameters were filled into the memory being supplied by the list of export parameter objects.

RFC_CLOSED

The connection was closed by the other end.

RFC_EXCEPTION

The callee has raised an exception. The string strException contains the name of the exception. No data were transported.

RFC_CALL

The callee has issued an RFC call to the caller of Receive. No data are transported. The call request must be handled by using the CRfcServerApp::Run before another call to Receive can be done.

Parameters

strException: A reference to a string, to be filled with error information when an RFC exception is encountered.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for Receive failure, matching these return codes in the C-API: RFC_SYS_EXCEPTION(local or remote RFC system raised an exception), RFC_FAILURE(internal error). SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

Receive can only be called after a Call is issued. Before making this function call, make sure all necessary export and table parameters are added to the function object. Any values to be received via these parameters can be retrieved right after Receive.

Related Information

Call [Page 34]

CallReceive [Page 35]

Write

Purpose

Writes metadata information for this CRfcClientFunc object to persistent stream.

Syntax

```
BOOL Write(CRfcPersistStream& Pers);
```

Return Value

Returns TRUE when all metadata is successfully written; FALSE otherwise.

Parameters

Pers: Reference to a CRfcPersistStream object that encapsulates the persistent data stream.

Description

This function automatically writes all metadata for this CRfcClientFunc object in its entirety (including metadata for the function itself, the number and types of its parameters, and the metadata for the parameters themselves).

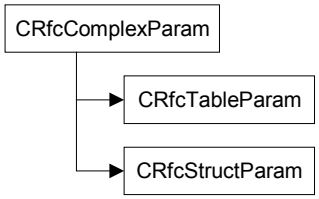
Related Information

The CRfcPersistStream Class [Page 105]

Read [Page 37]

The CRfcComplexParam Class

Class Summary



The CRfcComplexParam class is defined in CRfcComp.h.

The CRfcComplexParam class is an abstract class. It contains common functionality for parameters with complex data structures: structure parameters and table parameters.

Static Variables

BOOL m_bNeedConversion [Page 45]

Indicates if the data received from R/3 needs byte reordering.

Construction

CRfcComplexParam (Default Constructor) [Page 42]

Default constructor

CRfcComplexParam (Copy Constructor) [Page 43]

Copy constructor

Operations

AddFieldInfo (Multiple Fields) [Page 41]

Adds information for multiple fields into the parameter

AddFieldInfo (Single Field) (Defunct) [Page 41]

Adds information for a single field to the parameter

ClearAllFieldInfo [Page 41]

Restores metadata information about this CRfcClientFunc object from persistent stream

CopyAllFieldInfo [Page 42]

Copies the entire list of RFC field information from another CRfcComplexParam object

GetEBCDICCodePage [Page 43]

Retrieves the EBCDIC code page

GetFieldCount [Page 43]

Obtains the number of fields in the parameter structure

GetFieldInfo (by Index/Column Number) [Page 44]

Obtains information about a field by index or column number

GetFieldInfo (by Name) [Page 44]

Obtains information about a field by name

Read [Page 45]

Restores metadata (field information) about this CRfcComplexParam object from persistent stream

SetEBCDICCodePage [Page 46]

Sets the EBCDIC code page

Write [Page 46]

Writes metadata (field information) for this CRfcComplexParam object to persistent stream

AddFieldInfo (Multiple Fields)

Purpose

Adds information for multiple fields into the parameter.

Syntax

```
void AddFieldInfo(const RFC_FIELD_INFO* pFieldsInfo,  int nFieldCount);
```

Parameters

- *pFieldsInfo*: Pointer to an array of RFC_FIELD_INFO structure. See RFC_FIELD_INFO.
- *nFieldCount*: Number of elements contained in the array.

Exceptions

Throws const char* for memory exceptions.

Description

The order in which each field's information appears in the RFC_FIELD_INFO array parameter does not matter. Nor does it matter if the array does not contain information for all fields. It is only necessary to add in information for those fields whose values need to accessed later.

Related Information

AddFieldInfo (Single Field) [Page 41]

AddFieldInfo (Single Field) (Defunct)

ClearAllFieldInfo

Purpose

Restores metadata information about this CRfcClientFunc object from persistent stream.

CopyAllFieldInfo

Syntax

```
void ClearAllFieldInfo();
```

Parameters

None.

Description

This function clears all metadata information about the fields in this parameter.

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

CopyAllFieldInfo

Purpose

Copies the entire list of RFC field information from another CRfcComplexParam object.

Syntax

```
void CopyAllFieldInfo(const CRfcComplexParam pSourceParam);
```

Parameters

pSourceParam: Pointer to the CRfcComplexParam object whose list of field information is to be copied.

Exceptions

Throws const char* for memory exceptions.

Related Information

The RFC_FIELD_INFO Structure [Page 31]

CRfcComplexParam (Default Constructor)

Purpose

Default constructor of CRfcComplexParam.

Syntax

```
CRfcComplexParam ();
```

Related Information

CRfcComplexParam (Copy Constructor) [Page 43]

CRfcComplexParam (Copy Constructor)

Purpose

Copy constructor.

Syntax

```
CRfcComplexParam(const CRfcComplexParam & sourceParam);
```

Parameters

sourceParam: Reference to the parameter to copy from.

Exceptions

Throws const char* for memory exceptions.

GetEBCDICCodePage

This function is obsolete. It is kept only for backward compatibility of applications using the RFC C++ Class Library.

Code pages are now handled per connection, instead of for each parameter. To get the code page that applies to the current connection use the GetCodePage [Page 59] function of the CRfcConnection class.

Purpose

Retrieves EBCDIC code page.

Syntax

```
BOOL GetEBCDICCodePage() const;
```

Parameters

None

Returns

Always returns TRUE.

Exceptions

None

GetFieldCount

Purpose

Obtains the number of fields in the parameter structure.

Syntax

```
int GetFieldCount (void) const;
```

Return Value

Integer count.

Description

For this function to return the correct total number of fields, at least the field information for the last field need to be provided, using either of the AddFieldInfo methods.

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

AddFieldInfo (Single Field) [Page 41]

GetFieldInfo (by Index/Column Number)

Purpose

Obtains information about a field by index or column number.

Syntax

```
const RFC_FIELD_INFO* GetFieldInfo(int nIndex) const;
```

Return Value

Pointer to a constant RFC_FIELD_INFO struct containing the information. NULL if no field information with that column number can be found.

Parameters

nIndex: Column number of the field, beginning from 0.

Related Information

GetFieldInfo (by Name) [Page 44]

The RFC_FIELD_INFO Structure [Page 31]

GetFieldInfo (by Name)

Purpose

Obtains information about a field by name.

Syntax

```
const RFC_FIELD_INFO* GetFieldInfo(CSTR fldName) const;
```

Return Value

Pointer to a constant RFC_FIELD_INFO struct containing the information. NULL if no field information with that name can be found.

Parameters

fldName: Name of the field.

Related Information

GetFieldInfo (by Index/Column Number) [Page 44]

The RFC_FIELD_INFO Structure [Page 31]

m_bNeedConversion

Description

Indicates if the data received from R/3 needs byte reordering. Default setting is FALSE.

Syntax

```
BOOL m_bNeedConversion
```

Read

Purpose

Restores metadata (field information) about this CRfcComplexParam object from persistent stream.

Syntax

```
BOOL Read(CRfcPersistStream& Pers);
```

Return Value

Returns TRUE when all metadata is successfully restored; FALSE otherwise.

Parameters

Pers: Reference to a CRfcPersistStream object that encapsulates the persistent data stream.

Description

Important Note: since CRfcComplexParam is intended to be used as a base class from which usable classes are derived, the metadata is best written and read using the derived classes' (CRfcStructParam and CRfcTableParam) Write() and Read() functions, respectively.

It is important that the given CRfcPersistStream object actually contains the complete metadata for this CRfcComplexParam object. Incomplete metadata causes the restoration to fail.

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

Write [Page 46]

The RFC_FIELD_INFO Structure [Page 31]

The CRfcPersistStream Class [Page 105]

SetEBCDICCodePage

This function is obsolete. It is kept only for backward compatibility of applications using the RFC C++ Class Library.

Code pages are now handled per connection. You no longer need to set them for each parameter. To set the code page for the current connection use the SetCodePage [Page 69] function of the CRfcConnection class.

Purpose

Sets EBCDIC code page.

Syntax

```
void SetEBCDICCodePage(BOOL Value) const;
```

Parameters

None

Return Value

None

Exceptions

None

Write

Purpose

Writes metadata (field information) for this CRfcComplexParam object to persistent stream.

Syntax

```
BOOL Write(CRfcPersistStream& Pers);
```

Return Value

Returns TRUE when all metadata is successfully written; FALSE otherwise.

Parameters

Pers: Reference to a CRfcPersistStream object that encapsulates the persistent data stream.

Description

Important Note: since CRfcComplexParam is intended to be used as a base class from which usable classes are derived, the metadata is best written and read using the derived classes' (CRfcStructParam and CRfcTableParam) Write() and Read() functions, respectively.

This function automatically writes all metadata (field information) for this CRfcComplexParam object to the given persistent stream object.

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

Read [Page 45]

The RFC_FIELD_INFO Structure [Page 31]

The CRfcPersistStream Class [Page 105]

The CRfcConnection Class

Class Summary

CRfcConnection

The *CRfcConnection* class is defined in *CRfcConn.h.*

The CRfcConnection class manages information concerning logon to R/3, and is also responsible for making and terminating connections. A CRfcConnection object with a live connection is needed for any real-time interactions with R/3 through CRfcFunction objects.

Construction

CRfcConnection [Page 55]

Default constructor

CRfcConnection [Page 56]

Constructs a CRfcConnection object

Operations

Abort [Page 49]

Aborts the connection with an error message

Accept (Console Version) [Page 50]

Accept an incoming connection, console version

Accept (Windows Version) [Page 51]

Accept an incoming connection. Windows applications only

The CRfcConnection Class

Abort

Purpose

Abort the connection with an error message.

Syntax

```
void Abort(char* Message);
```

Parameters

Message: A character string error message to be sent to R/3. If it is NULL, only the connection is closed.

Description

This function sends a error message, if possible, and closes the connection.

A given error message cannot be send, if the receiver is not in state, where it expects some RFC data to receive.

Related Information

Close [Page 52]

Accept (Console Version)

Purpose

Accept an incoming connection, console version.

Syntax

```
void Accept(char** argv);
```

Parameters

argv: The command line string containing the logon information. There are two ways to define argv:

1. Working with ‘saprfc.ini’ file:

– argv[0]: -D<destination pointed to an entry in 'saprfc.ini'>

– argv[1]: -t Set RFC-Trace on.

2. Working without the ‘saprfc.ini’ file:

– argv[0]: -a<program ID> e.g. own_host_name.program_name

– argv[1]: -g<host name of the SAP gateway>

– argv[2]: -s<service of the SAP gateway>, e.g. sapgw00

– argv[3]: -t Set RFC-Trace on

We recommend the first way since you can use some RFC features today and will not need to make any changes in your RFC programs. See help on saprfc.ini for details.

The first 3 parameters in the above second way must fit with the configuration in R/3. (Under transaction SM59, connection type T and register mode)

Use at least the host name of the computer where your RFC server program is running as a part of the program ID along with the program name to avoid name conflicts at the SAP gateway.

Exceptions

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failures. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

This function accepts an incoming connection and has to be used, if the program was started by an RFC call issued by an R/3 system or an other program using this API. The command line (argv) has to be passed to this function.

With Release 3.0C onwards this function can be used to register at an SAP gateway and the server program can wait for the next RFC request by invoking the Run member function of the CRfcServerApp object.

Using this functionality an RFC server program can now already run and work as an RFC daemon. A start of any server program by an R/3 application server, by SAPGUI or via remote shell by an SAP gateway is no longer necessary.

Related Information

Connect [Page 54]

SafeConnect [Page 68]

Open [Page 65]

SafeOpen [Page 68]

Accept (Windows Version) [Page 51]

ConnArgv [Page 53]

Accept (Windows Version)

Purpose

Accept an incoming connection, Windows applications only.

Syntax

```
void Accept(char* CommandLine);
```

Parameters

CommandLine: The command line string containing the logon information. See the above member function Accept for detailed explanations.

Exceptions

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failures. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

See the above Accept member function for detailed explanations.

Related Information

Connect [Page 54]

SafeConnect [Page 68]

Open [Page 65]

SafeOpen [Page 68]

ConnArgv [Page 53]

Accept (Console Version) [Page 50]

Clear

Purpose

Close connection and clear user logon and connection information.

Syntax

```
void Clear(void);
```

Description

This member function closes the current connection if there is one, then resets all user and connection information to their respective default values.

Close

Purpose

Closes the connection.

Syntax

```
void Close(void);
```

Description

This function closes the connection if there is a live one established, otherwise does nothing.

Related Information

Abort [Page 49]

ConnArgv

Purpose

Command line parsing, supports 32-bit Windows only

Syntax

```
void  ConnArgv (char** argv);
```

Parameters

argv: Command line to be parsed.

Exceptions

Throws const char* for memory exceptions.

Description

The following tokens are recognized in the argv array, which must be terminated by a NULL entry:

Table Caption

Token	Description or Comment
-d <Destination>:	The name of the destination
-c <NNN>:	Client (sign on data)
-u <User Id>:	User id
-p <Password>:	Password
-l <language>:	Language
-3:	R/3 mode
-2:	CPIC mode
-t:	Turn trace on
-h <Hostname>:	The name of the target host
-s <NN>:	The system number of the target SAP system
-g <Gateway Host>:	The gateway host (if not specified, the h option is used)
-x <Gateway Service>:	The TCP/IP service of the gateway (default is sapgwNN, where NN is the system number (-s)
-gui:	Start 'sapgui' to be able to display SAP Dynpros or Graphics (R/3 mode only).
-debug:	Start communication in debug mode (R/3 mode only).

All tokens that were interpreted by ConnArgv are removed from the argv array.

Connect

Purpose

Opens the connection with load-balancing.

Syntax

```
void Connect(void);
```

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

Opens up a connection with R/3 with load-balancing using the stored user and connection information.

Related Information

SafeConnect [Page 68]

Open [Page 65]

SafeOpen [Page 68]

Accept (Windows Version) [Page 51]

Accept (Console Version) [Page 50]

ConnectInfoFromConnectParam

Purpose

Converts a character string into the RFC_CONNECT_INFO and RFC_USER_INFO objects.

Syntax

```
void ConnectInfoFromConnectParam(char * connect_param);
```

Parameters

Consists of a character string containing the connection and user information.

Exceptions

Throws const char* for memory exceptions.

Description

This method is used to obtain the RFC_CONNECT_INFO and the RFC_USER_INFO objects from a character string and store the objects in the CRfcConnection object. The character string

is in the format of connect_param in RfcOpenEx(char *connect_param, RFC_ERROR_INFO_EX * error_info). Please see the RFC library documentation for details.

Related Information

ConnectParamFromConnectInfo [Page 55]

ConnectParamFromConnectInfo

Purpose

Converts RFC_CONNECT_INFO and RFC_USER_INFO objects into a string that is encapsulated in a CRfcString object.

Syntax

```
void ConnectParamFromConnectInfo(CRfcString & connect_param);
```

Parameters

Consist of a CRfcString object that contains a character string for encapsulating the RFC_CONNECT_INFO and RFC_USER_INFO objects.

Exceptions

Throws const char* for memory exceptions.

Description

This method is used to convert the RFC_CONNECT_INFO and RFC_USER_INFO objects into a a character string and store the objects in a CRfcString object. This character string is in the same format as RfcOpenEx(char * connect_param, RFC_ERROR_INFO_EX * error_info). Please see the RFC Library documentation for details.

Related Information

ConnectInfoFromConnectParam [Page 54]

CRfcConnection

Purpose

CRfcConnection is the default connection constructor.

Syntax

```
CRfcConnection(void);
```

Description

Constructs an empty CRfcConnection object and sets all logon information to default values.

This constructor can be used for RFC server programs. The logon and connection information will be passed through the command line and automatically parsed by the method Accept.

If used by a client program, the user for fills in logon and connection information for the CRfcConnection object by using the ConnArgv, SetUserInfo or SetConnectInfo methods.

Related Information

ConnArgv [Page 53]

SetUserInfo [Page 70]

SetConnectInfo [Page 69]

CRfcConnection

Purpose

Constructs a CRfcConnection object using the information contained in the userInfo structure and specified by the destination string.

Syntax

```
CRfcConnection (const RFC_USER_INFO &userInfo, CSTR  destination =
NULL);
```

Parameters

- *userInfo*: A structure that contains user logon information. See RFC_USER_INFO.
- *destination*: A string that specifies the name of the destination for connection. If the connection is not described completely, or if the RFC_MODE_PARAMETER is used, this name is used as a key for the *sideinfo* file or the *saprfc.ini* file, where the connection should be described. You always have to provide the destination information when making a connection as a client. If not specified here, specify it using the *SetConnectInfo* method or the *ConnArgv* method before trying to establish a connection as a client.

Exceptions

Throws const char* for memory exceptions.

Description

This constructor is only used for RFC client programs.

Related Information

SetUserInfo [Page 70]

SetConnectInfo [Page 69]

Decrypt

Purpose

Restore encrypted information stored in a text string.

Syntax

```
virtual void Decrypt(CRfcString& rfcString);
```

Parameters

rfcString: reference to a CRfcString object containing encryption to be restored.

Description

This function offers a simple algorithm for restoring encrypted information. It is intended to be used in conjunction with ReadRegistry() function to read encrypted connection and user information that had been written to Window's Registry.

This function can be overridden in a derived class to implement the user's own algorithm for decipher encryption.

Related Information

WriteRegistry [Page 71]

ReadRegistry [Page 67]

Encrypt [Page 57]

Encrypt

Purpose

Encrypt information in a text string.

Syntax

```
virtual void Encrypt(CRfcString& rfcString);
```

Parameters

rfcString: reference to a CRfcString object containing a text string to be encrypted.

Description

This function offers a simple algorithm for encrypting information. It is intended to be used in conjunction with WriteRegistry() function to store encrypted connection and user information into Window's Registry.

This function can be overridden in a derived class to implement the user's own algorithm for decipher encryption.

Related Information

WriteRegistry [Page 71]

ReadRegistry [Page 67]

Decrypt [Page 56]

FromHandle

Purpose

Uses the RFC_HANDLE parameter to establish a new CRfcConnection object.

Syntax

```
void FromHandle(RFC_HANDLE RfcHandle, BOOL ISClient);
```

Parameters

RfcHandle: The RFC handle.

IsClient: Boolean parameter that determines if the application is a client program.

Return Value

None.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

This function uses a given RFC handle to establish a new connection.

GetAttributes

Purpose

Retrieves RFC attributes stored in the CrfcConnection object.

Syntax

```
const RFC_ATTRIBUTES& GetAttributes() const;
```

Parameters

None.

Return Value

A reference pointer to the RFC attributes in RFC_ATTRIBUTES.

Exceptions

None.

Description

This function returns a reference pointer to the RFC attributes stored in the object.

GetCodePage

Purpose

Get the current code page.

Syntax

```
char *GetCodePage(void) const;
```

Parameters

None

Return Value

codepage: a pointer to a character string.

Related Information

SetCodePage [Page 69]

GetCommandLineArgument

Purpose

Returns the command line arguments in a character string.

Syntax

```
char** GetCommandLineArgument(void) const;
```

Parameters

None.

Return Value

A pointer to a character string containing command line arguments.

Exceptions

None.

Description

This function returns a pointer to a character string containing command line arguments.

GetConnectInfo

Purpose

Obtains connection information stored in the object.

Syntax

```
const RFC_CONNECT_INFO* GetConnectInfo(void) const;
```

Return Value

A pointer to an RFC_CONNECT_INFO structure. See RFC_CONNECT_INFO.

Related Information

SetConnectInfo [Page 69]

GetHandle

Purpose

Obtains the connection handle stored in the connection object.

Syntax

```
RFC_HANDLE GetHandle(void);
```

Return Value

Returns the RFC handle stored in the connection object without any verification.

Description

Use this method to read the value of the RFC handle stored in the connection object, use GetSafeHandle instead if you want verification.

Related Information

GetSafeHandle [Page 63]

GetR3Release

Purpose

Calls RfcGetAttributes or the Rfc function RFC_GET_SAP_SYSTEM_PARAMETERS to obtain the R/3 release version.

Syntax

```
void GetR3Release(void);
```

Parameters

None.

Return Value

None.

Exceptions

Throws const char* for RFC call failure.

Description

The application program can use this function to call R/3 and obtain R/3 release version. This function is automatically called by `CRfcConnection::Open()` and `CRfcConnection::SafeOpen()` if RFC_CONNECT_INFO. The R3 release is set to RELUNKNOWN.

Related Information

GetReleaseString [Page 62]

The RFC_CONNECT_INFO structure [Page 30]

GetR3SystemInfo

Purpose

Calls RFC function RFC_SYSTEM_INFO to obtain R/3 various system information.

Syntax

```
void GetR3SystemInfo(void);
```

Parameters

None.

Return Value

None.

Exceptions

Throws const char*, RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX (and not RFC_ERROR_INFO). See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

The application program can use this function to call R/3 and obtain R/3 system information. Please use R/3 Transaction SE11 to view the content of the reference structure type RFCSI, and use R/3 Transaction SE37 to view other export parameters of the function module RFC_SYSTEM_INFO. This function is automatically called by `CRfcConnection::Open()` and `CRfcConnection::SafeOpen()`if RFC_CONNECT_INFO.bGetR3SystemInfo is set to TRUE by the application program.

Related Information

R3SystemInfo (by field-name) [Page 65]

R3SystemInfo (by index) [Page 66]

R3SystemInfoCount [Page 67]

The RFC_CONNECT_INFO structure [Page 30]

GetR3SystemParameters

Purpose

Calls the RFC function RFC_GET_SAP_SYSTEM_PARAMETERS to obtain the R/3 release version.

Syntax

```
void GetR3SystemParameters(void);
```

Parameters

None.

Return Value

None.

Exceptions

Throws const char*, RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX (and not RFC_ERROR_INFO). See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

If your application server is running an R/3 release earlier than 4.0A, you must call this function to retrieve the R/3 release version. You can get the release version directly or by using GetR3Release.

GetReleaseString

Purpose

Returns a character string containing the release version of the R/3 system being connected to.

Syntax

```
CSTR GetReleaseString(void);
```

```
string GetReleaseString(void);
```

Parameters

None.

Return Value

If the user chooses to use Standard C++ Library on Windows NT, the template class object ‘string’ is returned. Otherwise, a pointer to a character string is returned.

Exceptions

None.

Description

This function returns a const character string containing the release version of the R/3 system being connected to. The release string should be 3 characters long. For example: “31G”, “31H”, “40A”. The alphabet character in the string is in uppercase.

Related Information

GetR3Release [Page 60]

The RFC_CONNECT_INFO structure [Page 30]

GetSafeHandle

Purpose

Obtains a verified, live connection handle.

Syntax

```
C_HANDLE GetSafeHandle(void);
```

Return Value

Returns a valid RFC handle if connection is alive and user logon information is valid. Otherwise returns RFC_HANDLE_NULL.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFOR for RfcCall failure.

Description

Use this method to obtain a live RFC connection handle and verify user logon information.

Related Information

GetHandle [Page 60]

SafeOpen [Page 68]

SafeConnect [Page 68]

GetTraceLevel

Purpose

Obtains the current trace level.

Syntax

```
int GetTraceLevel(void);
```

Return Value

An integer value indicating the current trace level for this connection.

Related Information

SetTraceLevel [Page 70]

GetUserInfo

Purpose

Obtains user information stored in the object.

Syntax

```
const RFC_USER_INFO* GetUserInfo(void) const;
```

Return Value

A pointer to an RFC_USER_INFO structure. See RFC_USER_INFO.

Related Information

SetUserInfo [Page 70]

IsR3AtLeast40A

Purpose

Determines if the application server is running Release 4.0A or a later version of the R/3 System.

Syntax

```
BOOL isR3AtLeast40A(void) const;
```

Parameters

None.

Return Value

Boolean TRUE or FALSE

Exceptions

None.

Description

This function returns TRUE if the application server is running Release 4.0A or a later version of the R/3 System; otherwise the function returns FALSE.

Open

Purpose

Opens the connection.

Syntax

```
void Open(void);
```

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

Opens up a connection with R/3 using the stored user and connection information.

Related Information

GetHandle [Page 60]

SafeOpen [Page 68]

SafeConnect [Page 68]

Connect [Page 54]

Accept (Windows Version) [Page 51]

Accept (Console Version) [Page 50]

R3SystemInfo (by field-name)

Purpose

Returns value of R/3 system information field.

Syntax

```
const CRfcData& R3SystemInfo(CSTR InfoName);
```

Parameters

InfoName: A character string containing the name of the field.

Return Value

A CRfcData object containing the value of the field whose name is given as argument. This CRfcData object's content can be integer or character string, and should be extracted using one of its unary cast operators.

Exceptions

None.

Description

The name of the field given as argument for this function can be one of both: (1) the name of one of the fields in the export structure parameter RFCSI_EXPORT of RFC_SYSTEM_INFO, or (2) name of one of the export simple parameters of RFC_SYSTEM_INFO.

Related Information

GetR3SystemInfo [Page 61]

R3SystemInfoCount [Page 67]

The RFC_CONNECT_INFO structure [Page 30]

R3SystemInfo (by index)

Purpose

Return value of R/3 systme information field.

Syntax

```
const CRfcData& GetR3SystemInfo(int nIndex);
```

Parameters

nIndex: index into a list of R/3 system information fields.

Return Value

A CRfcData object containing the value of the field whose index is given as argument. This CRfcData object's content can be integer or character string, and should be extracted using one of its unary cast operators.

Exceptions

None.

Description

The system information is stored in a list and each list element can be accessed by index. The list contains all the fields in the structure export parameter RFCSI_EXPORT of RFC_SYSTEM_INFO and the simple export parameters of RFC_SYSTEM_INFO. This function should be used in conjection with CRfcConnection::R3SystemInfoCount() for iterating through the entire list.

Related Information

GetR3SystemInfo [Page 61]

R3SystemInfo (by field-name) [Page 65]

R3SystemInfoCount [Page 67]

The RFC_CONNECT_INFO structure [Page 30]

R3SystemInfoCount

Purpose

Returns the number of fields in the R/3 system information list.

Syntax

```
int R3SystemInfoCount(void);
```

Parameters

None.

Return Value

An integer indicating the number of fields in the R/3 system information list.

Exceptions

None.

Description

This function should be used in conjunction with `CRfcConnection::R3SystemInfo(int)` by index to iterate through the list of R/3 system inforamtion.

Related Information

R3SystemInfo (by field-name) [Page 65]

R3SystemInfo (by index) [Page 66]

The RFC_CONNECT_INFO structure [Page 30]

ReadRegistry

Purpose

Reads from the Windows registry user and connection information and store into the connection object. Windows applications only.

Syntax

```
void ReadRegistry(HKEY hKeyRoot, CSTR regPath);
```

Parameters

- *hKeyRoot*: The registry key handle at root level, e.g., HKEY_LOCAL_MACHINE.
- *regPath*: Path name of the child key under the root key, e.g., “\SOFTWARE\SAP\RFC\BIN”.

Exceptions

Throws const char* for memory exceptions, and registry query failure.

Description

See WriteRegistry for a detailed description of the values that could be associated with the registry key. Note that not all entries may be present under the key, depending on the requirements of the connection mode.

Related Information

WriteRegistry [Page 71]

SafeConnect

Purpose

Opens the connection with load-balancing and user information verification.

Syntax

```
void SafeConnect(void);
```

Exceptions

Throws const char* for memory exceptions. Throws REC_ERROR_INFO for RfcOpen failure and user information verification errors.

Description

Opens up a connection with R/3 with load-balancing using the stored user and connection information, and verify user logon information.

Related Information

Connect [Page 54]

Open [Page 65]

SafeOpen [Page 68]

Accept (Windows Version) [Page 51]

Accept (Console Version) [Page 50]

SafeOpen

Purpose

Opens the connection with user information verification.

Syntax

```
void SafeOpen(void);
```

Exceptions

Throws const char* for memory exceptions. Throws REC_ERROR_INFO for RfcOpen failure and user information verification errors.

Description

Opens up a connection with R/3 using the stored user and connection information, and verify user logon information.

Related Information

Open [Page 65]

SafeOpen [Page 68]

Connect [Page 54]

Accept (Windows Version) [Page 51]

Accept (Console Version) [Page 50]

SetCodePage

Purpose

Set the current code page.

Syntax

```
void SetCodePage(char * codepage);
```

Parameters

codepage: a character string indicating the new code page.

Related Information

GetCodePage [Page 59]

SetConnectInfo

Purpose

Sets connection information stored in the object.

Syntax

```
void SetConnectInfo(const RFC_CONNECT_INFO &connectInfo);
```

Parameters

connectInfo: An RFC_CONNECT_INFO structure. See RFC_CONNECT_INFO.

Exceptions

Throws const char* for memory exceptions.

Related Information

GetConnectInfo [Page 59]

SetTraceLevel

Purpose

Sets the current trace level.

Syntax

```
void SetTraceLevel(int nTraceLevel);
```

Parameters

nTraceLevel: An integer indicating the trace level to be set.

Related Information

GetTraceLevel [Page 63]

SetUserInfo

Purpose

Sets user information stored in the object.

Syntax

```
void SetUserInfo(const RFC_USER_INFO &userInfo);
```

Parameters

userInfo: An RFC_USER_INFO structure. See RFC_USER_INFO.

Exceptions

Throws const char* for memory exceptions.

Related Information

GetUserInfo [Page 64]

WriteRegistry

Purpose

Writes the current user and connection information into the Windows registry. Windows applications only.

Syntax

```
void WriteRegistry (HKEY hKeyRoot, CSTR regPath);
```

Parameters

- *hKeyRoot*: The registry key handle at root level, e.g., HKEY_LOCAL_MACHINE.
- *regPath*: Path name of the child key under the root key, e.g., "\SOFTWARE\SAP\RFC\BIN".

Exceptions

Throws const char* for registry entry creation failure.

Description

This member function writes into the Windows registry current user logon and connection information

The following values names are written into the registry to be associated with the specified registry key. The data type of the values are shown on the right.

Value Name	Data Type	Description
Destination	REG_SZ	Destination name of the R/3 system you are connecting to
User	REG_SZ	R/3 User name
Client	REG_SZ	R/3 Client number
Password	REG_SZ	R/3 User password
Language	REG_SZ	Language of R/3 log on, usually either English or German
Host Name	REG_SZ	Name of the application server host machine
System Number	REG_DWORD	R/3 System number
Gateway Host	REG_SZ	Name of the gateway host
Gateway Service	REG_SZ	Name of the gateway service
Group	REG_SZ	If host name is unspecified, group name is used for R/3 load-balancing connection
Message Server	REG_SZ	Name of the message server
System Name	REG_SZ	The R/3 System name for the load-balancing connection

SAPGUI	REG_DWORD	Flag to indicate whether SAPGUI is enabled or not
RFC Mode	REG_DWORD	Mode of the RFC connection
Encryption	REG_DWORD	Flag 0 = password not encrypted, 1 = password is encrypted

Related Information

ReadRegistry [Page 67]

The CRfcData Class

Class Summary

CRfcData

The CRfcData class is defined in CRfcData.h.

The CRfcData class is the enabling class for getting or setting parameter values. Objects of this class serve as intermediate data “converters” between the left and right hand side of an assignment operation using the parameter’s get or set value member functions. For details see Setting and Getting Values of Parameter Objects [Page 24].

RFC data type TYPP is currently not supported.

Construction

CRfcData [Page 73]

Constructs a CRfdData object

CRfcData (Copy Constructor) [Page 74]

Copy constructor

Operations

SetAttributes [Page 74]

Assigns data type information to object to represent

Init [Page 75]

Assigns data type related information to the object to represent

SetStraightData [Page 75]

Sets flag indicating conversion or byte reordering

operator= (CRfcData Object) [Page 76]

Assignment from a CRfcData object

operator= (Integer) [Page 76]

Assignment from an integer

operator= (String) [Page 77]

Assignment from a string

operator= (Float Number) [Page 77]

Assignment from a floating-point number

operator= (Array) [Page 78]

Assignment from an array

CopyRawDataFrom [Page 79]

Copy row data from the specified memory to the data object's memory

operator RFC_INT [Page 79]

Obtains the integer value of the data

operator RFC_INT1 [Page 80]

Obtains the four-byte value of the data

operator RFC_INT2 [Page 80]

Obtains the single-byte value of the data

operator CSTR [Page 81]

Obtains the double-byte value of the data

operator RFC_FLOAT [Page 81]

Obtains the floating point value of the data

CopyRawDataTo [Page 82]

Copy row data to the specified memory from the data object's memory

ConvertData [Page 82]

Do data conversion or byte reordering

CRfcData

Purpose

Constructs a CRfcData object.

Syntax

```
CRfcData(BOOL bStraightData=TRUE);
```

Parameters

bStraightData: Flag indicating the data to be represented needs conversion or byte re-ordering. If so, pass TRUE, otherwise pass FALSE.

Exceptions

Throws const char* for memory exceptions.

Related Information

SetAttributes [Page 74]

Init [Page 75]

CRfcData (Copy Constructor)

Purpose

Copy constructor for CRfcData.

Syntax

```
CRfcData(const CRfcData& sourceData);
```

Parameters

sourceData: Reference to the object to copy from.

Exceptions

Throws const char* for memory exceptions.

SetAttributes

Purpose

Assigns data type related information to the object to represent.

Syntax

```
void SetAttributes(     void* pData,
                        unsigned nType,
                        int nLen=0,
                        int nDecimal=0);
```

Parameters

pData: Pointer to the memory block where the actual data is stored.

nType: RFC type of the data.

nLen: Length of the data in bytes.

nDecimal: Decimal positions in the data.

Description

The length of the data is not required except for TYPC, TYPX, TYPNUM and TYPP.

Related Information

Init [Page 75]

GetAttributes

Purpose

Obtains data type related information.

Syntax

```
void SetAttributes(     unsigned& nType,
                   int& nLen,
                   int& nDecimal);
```

Parameters

nType: reference to a variable that would receive RFC type of the data.

nLen: reference to a variable that would receive length of the data in bytes.

nDecimal: reference to a variable that would receive decimal positions in the data.

Related Information

SetAttributes [Page 74]

Init

Purpose

Set data to an initial value.

Syntax

```
void Init(void);
```

Description

Make sure data type related information has been provided using SetAttributes. All numeric types are initialized to 0, all character string types are initialized to spaces.

Related Information

SetAttributes [Page 74]

SetStraightData

Purpose

Sets the conversion indicator flag (indicating whether the data represented need conversion or byte re-ordering).

operator= (CRfcData Object)

Syntax

```
void SetStraightData( BOOL bStraightData);
```

Parameters

bStraightData: Pass TRUE if the data does not need conversion, otherwise pass FALSE.

operator= (CRfcData Object)

Purpose

Assignment from a CRfcData object.

Syntax

```
CRfcData& operator=(const CRfcData& sourceData);
```

Parameters

sourceData: Reference to the data object to assign from.

Return Value

Reference to this object itself.

operator= (Integer)

Purpose

Assignment from an integer.

Syntax

```
CRfcData& operator=(RFC_INT nValue);
```

Parameters

nValue: Integer value to set to the data object.

Return Value

Reference to this object itself.

Exceptions

Throws const char* for memory exceptions.

Description

This operator is only allowed on data object with integer or HEX types.

Related Information

operator= (Array) [Page 78]
operator= (CRfcData Object) [Page 76]
operator= (Float Number) [Page 77]
operator= (String) [Page 77]
operator RFC_INT [Page 79]
operator RFC_INT1 [Page 80]
operator RFC_INT2 [Page 80]

operator= (String)

Purpose

Assignment from a string.

Syntax

```
CRfcData& operator=(CSTR strValue);
```

Parameters

strValue: String value to set to the data object.

Return Value

Reference to this object itself.

Exceptions

Throws const char* for memory exceptions.

Description

This operator is only allowed on data object with character string types.

Related Information

operator= (Array) [Page 78]
operator= (CRfcData Object) [Page 76]
operator= (Integer) [Page 76]
operator= (Float Number) [Page 77]

operator= (Float Number)

Purpose

Assignment from a float.

operator= (Array)

Syntax

```
CRfcData& operator=(RFC_FLOAT fValue);
```

Parameters

fValue: Floating point value to set to the data object.

Return Value

Reference to this object itself.

Exceptions

Throws const char* for memory exceptions.

Description

This operator is only allowed on data object of types TYPFLOAT.

Related Information

operator= (Array) [Page 78]

operator= (CRfcData Object) [Page 76]

operator= (Integer) [Page 76]

operator= (String) [Page 77]

operator RFC_FLOAT [Page 81]

operator= (Array)

Purpose

Assignment from an array.

Syntax

```
CRfcData& operator=(RFC_CHAR arrayValue[ ]);
```

Parameters

arrayValue: Array of values to set to the data object.

Return Value

Reference to this object itself.

Description

This operator is only allowed on data object of types TYPC.

Related Information

operator= (Array) [Page 78]
operator= (CRfcData Object) [Page 76]
operator= (Integer) [Page 76]
operator= (Float Number) [Page 77]
operator= (String) [Page 77]

CopyRawDataFrom

Purpose

Copy raw data from the specified memory to the data object's memory.

Syntax

```
void CopyRawDataFrom(void* pDataFrom);
```

Parameters

pDataFrom: Pointer to the source memory.

Description

You must make sure the valid length of the memory block pointed to by pDataFrom matches the length(m_nLength) stored in the object. Only allowed when the data type is TYPX.

Related Information

CopyRawDataTo [Page 82]

operator RFC_INT

Purpose

Obtains the integer value of the data.

Syntax

```
operator RFC_INT(void) const;
```

Return Value

Integer value of the data object.

Exceptions

Throws const char* for memory exceptions.

operator RFC_INT1

Description

Only allowed when the data type is TYPINT, TYPINT1, TYPINT2, or TYPFX with length <= 4.

Related Information

operator RFC_INT1 [Page 80]

operator RFC_INT2 [Page 80]

operator= (Integer) [Page 76]

operator RFC_INT1

Purpose

Obtains the integer value of the data.

Syntax

```
operator RFC_INT1(void) const;
```

Return Value

Integer value of the data object.

Exceptions

Throws const char* for memory exceptions.

Description

Only allowed when the data type is TYPINT1 or TYPFX with length <= 1.

Related Information

operator RFC_INT [Page 79]

operator RFC_INT2 [Page 80]

operator= (Integer) [Page 76]

operator RFC_INT2

Purpose

Obtains the integer value of the data.

Syntax

```
operator RFC_INT2(void) const;
```

Return Value

Integer value of the data object.

Exceptions

Throws const char* for memory exceptions.

Description

Only allowed when the data type is TYPINT1, TYPINT2, or TYPFX with length <= 2.

Related Information

operator RFC_INT [Page 79]

operator RFC_INT1 [Page 80]

operator= (Integer) [Page 76]

operator CSTR

Purpose

Obtains the string value of the data.

Syntax

```
operator CSTR(void) const;
```

Return Value

String value of the data object.

Exceptions

Throws const char* for memory exceptions.

Description

Only allowed when the data type is TYPC, TYPNUM, TYPDATE, or TYPTIME.

Related Information

operator= (String) [Page 77]

operator RFC_FLOAT

Purpose

Obtains the floating point value of the data.

Syntax

```
operator RFC_FLOAT(void) const;
```

Return Value

Floating point value of the data object.

Exceptions

Throws const char* for memory exceptions.

Description

Only allowed when the data type is TYPFLOAT.

Related Information

operator= (Float Number) [Page 77]

CopyRawDataTo

Purpose

Copy row data to the specified memory from the data object's memory.

Syntax

```
void CopyRawDataTo(void* pDataTo);
```

Parameters

pDataTo: Pointer to the destination memory.

Description

User needs to make sure the valid length of the memory block pointed to by pDataFrom matches the length(m_nLength) stored in the object. Only allowed when the data type is TYPX.

Related Information

CopyRawDataFrom [Page 79]

ConvertData

Purpose

Do data conversion or byte re-ordering.

Syntax

```
void ConvertData(void);
```

Description

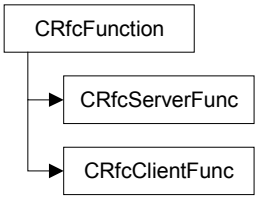
Does nothing if the flag for straight data is set to be TRUE.

Related Information

SetStraightData [Page 75]

The CRfcFunction Class

Class Summary



The *CRfcFunction* class is defined in *CRfcFunc.h.*

The *CRfcFunction* class is an abstract class. It is the main class that directly corresponds to remotely callable function modules. For each remotely callable function, an instance of a *CRfcFunction* derived class (whether *CRfcClientFunc* or *CRfcServerFunc*) can be created. It acts as a container of all parameters required for this function, and packs or unpacks parameter values in order to process the actual RFC calls.

Static Variables

rfcTID [Page 99]

Transaction ID for transactional RFC calls

Construction

CRfcFunction [Page 86]

Constructs a function object

Operations

AddExportParam [Page 85]

Adds an export parameter to the function object

AddImportParam [Page 85]

Adds an import parameter to the function object

AddTableParam [Page 86]

Adds a table parameter to the function object

GetConnection [Page 87]

Obtains connection object assigned to function object

GetConnectionRfcHandle [Page 87]

Retrieves the connection handle stored in the CRfcFunction object.

GetExportCount [Page 88]

Returns number of export parameters.

GetExportParam (by Index) [Page 88]

Retrieves an export parameter by index

GetExportParam (by Name) [Page 88]

Retrieves an export parameter by name

GetFunctionName [Page 89]

Retrieves the name of the remote-callable function

GetImportCount [Page 89]

Returns number of import parameters.

GetImportParam (by Index) [Page 90]

Retrieves an import parameter by index

GetImportParam (by Name) [Page 90]

Retrieves an import parameter by name

GetTableCount [Page 91]

Returns number of table parameters.

GetTableParam (by Index) [Page 91]

Retrieves a table parameter by index

GetTableParam (by Name) [Page 92]

Retrieves a table parameter by name

Listen [Page 97]

Checks whether an RFC request is available

RemoveExportParam (By Index) [Page 92]

Removes an export parameter from a function object by index

RemoveExportParam (By Name) [Page 93]

Removes an export parameter from a function object by parameter name

RemoveImportParam (By Index) [Page 93]

Removes an import parameter from a function object by index

RemoveImportParam (By Name) [Page 94]

Removes an import parameter from a function object by parameter name

RemoveTableParam (By Index) [Page 95]

Removes a table parameter from a function object by index

RemoveTableParam (By Name) [Page 95]

Removes a table parameter from a function object by parameter name

ReplaceExportParam [Page 96]

Replaces an export parameter in a function object

ReplaceImportParam [Page 96]

Replaces an import parameter in a function object

ReplaceTableParam [Page 97]

Replaces a table parameter in a function object

SetConnection [Page 98]

Assigns a connection object to the function object

SetFunctionName [Page 99]

Assigns a name to the function

AddExportParam

Purpose

Add an export parameter to the function object.

Syntax

```
void AddExportParam(CRfcImpExpParam& param);
```

Parameters

param: The parameter to be added, can be either a CRfcSimpleParam or a CRfcStructParam.

Exceptions

Throws const char* for memory exceptions.

Description

A parameter can only be associated with one function, so if *param* has already been added to another function object, adding it again to this function object will not be allowed.

Related Information

AddImportParam [Page 85]

AddTableParam [Page 86]

AddImportParam

Purpose

Add an import parameter to the function object.

Syntax

```
void AddImportParam(CRfcImpExpParam& param);
```

Parameters

param: The parameter to be added, can be either a CRfcSimpleParam or a CRfcStructParam.

Exceptions

Throws const char* for memory exceptions.

Description

A parameter can only be associated with one function, so if *param* has already been added to another function object, adding it again to this function object will not be allowed.

Related Information

AddExportParam [Page 85]

AddTableParam [Page 86]

AddTableParam

Purpose

Add a table parameter to the function object.

Syntax

```
void AddTableParam(CRfcTableParam&  table);
```

Parameters

table: The table parameter to be added.

Exceptions

Throws const char* for memory exceptions.

Description

A table parameter can only be associated with one function, so if *table* has already been added to another function object, adding it again to this function object will not be allowed.

Related Information

AddExportParam [Page 85]

AddImportParam [Page 85]

CRfcFunction

Purpose

Constructs a function object.

Syntax

```
CRfcFunction(const CRfcConnection* pConnection=NULL,
                       CSTR  strFuncName=NULL);
```

Parameters

- *pConnection*: Pointer to a connection object.

- *strFuncName*: Name of the remote-callable function.

Exceptions

Throws const char* for memory exceptions.

Description

The connection object's pointer is stored in the function object for use when making or accepting calls to or from R/3. If no connection object or function name is provided at construction time, they will have to be set with SetConnection and SetFunctionName before making or accepting calls. Once set, these two parameters are used for all subsequent calls.

Related Information

SetConnection [Page 98]

SetFunctionName [Page 99]

GetConnection

Purpose

Obtains the connection object assigned to the function object.

Syntax

```
const  CRfcConnection& GetConnection (void) const;
```

Return Value

A reference to the connection object.

Related Information

SetConnection [Page 98]

GetConnectionRfcHandle

Purpose

Obtains the connection handle stored in the CRfcFunction object.

Syntax

```
RFC_HANDLE GetConnectionRfcHandle(void) const
```

Exceptions

None

GetExportCount

Purpose

Returns number of export parameters.

Syntax

```
unsigned GetExportCount();
```

Return Value

An unsigned integer indicating the number of export parameters encapsulated

Related Information

AddExportParam [Page 85]

GetExportParam (by Index)

Purpose

Retrieves an export parameter by index.

Syntax

```
CRfcImpExpParam* GetExportParam (int  nIndex);
```

Return Value

A pointer to the export parameter object. Can be a CRfcSimpleParam or a CRfcStructParam.

Returns NULL if no export parameter with that index can be found.

Parameters

nIndex: The index of the export parameter in the export parameter list, beginning from 0.

Related Information

GetImportParam (by Index) [Page 90]

GetImportParam (by Name) [Page 90]

GetExportParam (by Name) [Page 88]

GetTableParam (by Index) [Page 91]

GetTableParam (by Name) [Page 92]

GetExportParam (by Name)

Purpose

Retrieves an export parameter by name.

Syntax

```
CRfcImpExpParam* GetExportParam (CSTR strParamName);
```

Return Value

A pointer to the export parameter object. Can be a CRfcSimpleParam or a CRfcStructParam.

Returns NULL if no export parameter with that name can be found.

Parameters

strParamName: The name of the export parameter.

Related Information

GetImportParam (by Index) [Page 90]
GetImportParam (by Name) [Page 90]
GetExportParam (by Index) [Page 88]
GetTableParam (by Index) [Page 91]
GetTableParam (by Name) [Page 92]

GetFunctionName

Purpose

Retrieves the name of the remote-callable function

Syntax

```
CSTR GetFunctionName (void) const;
```

Return Value

A const char* containing the function name.

Related Information

SetFunctionName [Page 99]

GetImportCount

Purpose

Returns number of import parameters.

Syntax

```
unsigned GetImportCount();
```

Return Value

An unsigned integer indicating the number of import parameters encapsulated

Related Information

AddImportParam [Page 85]

GetImportParam (by Index)

Purpose

Retrieves an import parameter by index.

Syntax

```
CRfcImpExpParam* GetImportParam (int  nIndex);
```

Return Value

A pointer to the import parameter object. Can be a CRfcSimpleParam or a CRfcStructParam.

Returns NULL if no import parameter with that index can be found.

Parameters

nIndex: The index of the import parameter in the import parameter list, beginning from 0..

Related Information

GetImportParam (by Name) [Page 90]

GetExportParam (by Index) [Page 88]

GetExportParam (by Name) [Page 88]

GetTableParam (by Index) [Page 91]

GetTableParam (by Name) [Page 92]

GetImportParam (by Name)

Purpose

Retrieves an import parameter by name.

Syntax

```
CRfcImpExpParam* GetImportParam (CSTR strParamName);
```

Return Value

A pointer to the import parameter object. Can be a CRfcSimpleParam or a CRfcStructParam.

Returns NULL if no import parameter with that name can be found.

Parameters

strParamName: The name of the import parameter.

Related Information

GetImportParam (by Index) [Page 90]

GetExportParam (by Index) [Page 88]

GetExportParam (by Name) [Page 88]

GetTableParam (by Index) [Page 91]

GetTableParam (by Name) [Page 92]

GetTableCount

Purpose

Returns number of table parameters.

Syntax

```
unsigned GetTableCount();
```

Return Value

An unsigned integer indicating the number of table parameters encapsulated

Related Information

AddTableParam [Page 86]

GetTableParam (by Index)

Purpose

Retrieves a table parameter by index.

Syntax

```
CRfcTableParam* GetTableParam (int  nIndex);
```

Return Value

A pointer to the table parameter object, NULL if no table parameter with that index can be found..

Parameters

nIndex: The index of the table parameter in the table parameter list, beginning from 0.

Related Information

GetImportParam (by Index) [Page 90]

GetImportParam (by Name) [Page 90]

GetExportParam (by Index) [Page 88]

GetExportParam (by Name) [Page 88]

GetTableParam (by Name) [Page 92]

GetTableParam (by Name)

Purpose

Retrieves a table parameter by name.

Syntax

```
CRfcTableParam * GetTableParam(CSTR strParamName);
```

Return Value

A pointer to the table parameter object, NULL if no table parameter with that name can be found.

Parameters

strParamName: The name of the table parameter.

Related Information

GetImportParam (by Index) [Page 90]

GetImportParam (by Name) [Page 90]

GetExportParam (by Index) [Page 88]

GetExportParam (by Name) [Page 88]

GetTableParam (by Name) [Page 92]

RemoveExportParam (by Index)

Purpose

Removes an export parameter object from a function object.

Syntax

```
BOOL RemoveExportParam (int  nIndex);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

nIndex: The index of the export parameter in the export parameter list; begins at 0.

Description

If the class library creates the parameter internally, it deletes it Otherwise, you must delete the parameter explicitly in any application that uses it.

Related Information

RemoveExportParam (by Name) [Page 93]

RemoveImportParam (by Index) [Page 93]

RemoveImportParam (by Name) [Page 94]

RemoveTableParam (by Index) [Page 95]

RemoveTableParam (by Name) [Page 95]

RemoveExportParam (by Name)

Purpose

Removes an export parameter object from a function object.

Syntax

```
BOOL RemoveExportParam (CSTR strParamName);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

strParamName: The name of the export parameter in the export parameter list.

Description

If the class library creates the parameter internally, it deletes it Otherwise, you must delete the parameter explicitly in any application that uses it.

Related Information

RemoveExportParam (by Index) [Page 92]

RemoveImportParam (by Index) [Page 93]

RemoveImportParam (by Name) [Page 94]

RemoveTableParam (by Index) [Page 95]

RemoveTableParam (by Name) [Page 95]

RemoveImportParam (by Index)

Purpose

Removes an import parameter object from a function object.

RemoveImportParam (by Name)

Syntax

```
BOOL RemoveImportParam (int nIndex);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

nIndex: The index of the import parameter in the import parameter list; begins at 0.

Description

If the class library creates the parameter internally, it deletes it Otherwise, you must delete the parameter explicitly in any application that uses it.

Related Information

RemoveExportParam (by Index) [Page 92]

RemoveExportParam (by Name) [Page 93]

RemoveImportParam (by Name) [Page 94]

RemoveTableParam (by Index) [Page 95]

RemoveTableParam (by Name) [Page 95]

RemoveImportParam (by Name)

Purpose

Removes an import parameter object from a function object.

Syntax

```
BOOL RemoveImportParam (CSTR strParamName);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

strParamName: The name of the import parameter in the import parameter list.

Description

If the class library creates the parameter internally, it deletes it Otherwise, you must delete the parameter explicitly in any application that uses it.

Related Information

RemoveExportParam (by Index) [Page 92]

RemoveExportParam (by Name) [Page 93]

RemoveImportParam (by Index) [Page 93]

RemoveTableParam (by Index) [Page 95]

RemoveTableParam (by Name) [Page 95]

RemoveTableParam (by Index)

Purpose

Removes a table parameter object from a function object.

Syntax

```
BOOL RemoveTableParam (int  nIndex);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

nIndex: The index of the table parameter in the table parameter list; begins at 0.

Description

If the class library creates the parameter internally, it deletes it Otherwise, you must delete the parameter explicitly in any application that uses it.

Related Information

RemoveExportParam (by Index) [Page 92]

RemoveExportParam (by Name) [Page 93]

RemoveImportParam (by Index) [Page 93]

RemoveImportParam (by Name) [Page 94]

RemoveTableParam (by Name) [Page 95]

RemoveTableParam (by Name)

Purpose

Removes a table parameter object from a function object.

Syntax

```
BOOL RemoveTableParam (CSTR strParamName);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

strParamName: The name of the table parameter in the table parameter list.

Description

If the class library creates the parameter internally, it deletes it Otherwise, you must delete the parameter explicitly in any application that uses it.

Related Information

RemoveExportParam (by Index) [Page 92]

RemoveExportParam (by Name) [Page 93]

RemoveImportParam (by Index) [Page 93]

RemoveImportParam (by Name) [Page 93]

RemoveTableParam (by Index) [Page 95]

ReplaceExportParam

Purpose

Replaces an export parameter object in a function object with a new export parameter object of the same type.

Syntax

```
BOOL ReplaceExportParam (CRfcImpExpParam * newParam);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

newParam: A pointer to the new export parameter object.

Description

If an application internally creates a copy of the parameter to be replaced in the class library, it also deletes it. Otherwise, you must explicitly delete the old parameter object in your applications.

Related Information

ReplaceImportParam

[Page 96]ReplaceTableParam [Page 97]

ReplaceImportParam

Purpose

Replaces an import parameter object in a function object with a new import parameter object of the same type.

Syntax

```
BOOL ReplaceImportParam (CRfcImpExpParam * newParam);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

newParam: A pointer to the new import parameter object.

Description

If an application internally creates a copy of the parameter to be replaced in the class library, it also deletes it. Otherwise, you must explicitly delete the old parameter object in your applications.

Related Information

ReplaceExportParam [Page 96]

ReplaceTableParam [Page 97]

ReplaceTableParam

Purpose

Replaces a table parameter object in a function object with a new table parameter object of the same type.

Syntax

```
BOOL ReplaceTableParam (CRfcTableParam * newParam);
```

Return Value

TRUE if successful, FALSE otherwise.

Parameters

newParam: a pointer to the new table parameter object.

Description

If an application internally creates a copy of the parameter to be replaced in the class library, it also deletes it. Otherwise, you must explicitly delete the old parameter object in your applications.

Related Information

ReplaceExportParam [Page 96]

ReplaceImportParam

[Page 96]

Listen

Purpose

Check whether an RFC request is available.

Syntax

```
RFC_RC Listen (void);
```

Return Value

An RFC_RC code indicating the status: RFC_OK, if there is an RFC event pending (call or return), RFC_RETRY, if nothing has arrived yet, or RFC_FAILURE, if an error has occurred or the server process is completed.

Description

For an RFC client program, if one does not always want to wait for the answer to an RFC call by using the Receive member function of CRfcClientFunc, one has to call the function Listen to listen for incoming RFC events. The function always returns immediately. If Listen returns RFC_OK, Receive has to be called to handle the event. It only is possible to listen for one incoming RFC message at a time.

For an RFC server program, it is also possible to use Listen while waiting for a new RFC request by Dispatch with or without register mode (see CRfcConnection::Accept [Page 51]). The CRfcServerApp's Run member function already encapsulates this process.

SetConnection

Purpose

Assigns a connection object to the function object.

Syntax

```
void SetConnection (const CRfcConnection& connection);
```

Parameters

connection: Reference to a connection object.

Exceptions

Throws const char* for memory exceptions.

Description

Different connection objects can be assigned to the function object during its existence. Once assigned, the connection object is used for all subsequent calls until a new connection object is assigned to the function.

Related Information

GetConnection [Page 87]

SetFunctionName

Purpose

Assigns a name to the function.

Syntax

```
void SetFunctionName (CSTR name);
```

Parameters

name: A const character string containing the function name.

Exceptions

Throws const char* for memory exceptions.

Description

The function name can only be set once and for all, it can not be changed over the lifetime of the function object. Therefore, this method can only be used once and only if the function name was not provided at construction time.

Related Information

GetFunctionName [Page 89]

rfcTID

Description

Transaction ID for transactional RFC calls. Supported only with R/3 3.0C or later.

Syntax

```
RFC_TID rfcTID
```

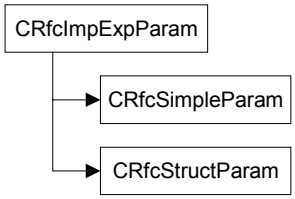
Related Information

CreateTransID [Page 36]

IndirectCall [Page 36]

The CRfcImpExpParam Class

Class Summary



The CRfcImpExpParam class is defined in CRfcImEx.h.

The CRfcImpExpParam class is an abstract class that contains common functionality for all import and export parameters.

Construction

CRfcImpExpParam [Page 101]

Constructs an import or export parameter object

CRfcImpExpParam (Copy Constructor) [Page 102]

Copy constructor

Operations

Clear [Page 101]

Cleans out the data buffer for all values in the parameter

GetLength [Page 102]

Gets the parameter length in bytes

GetParamName [Page 102]

Obtains the parameter name

InhibitParamToR3 [Page 103]

Inhibits the passing of this parameter to R/3 during a client function call (applicable only when this parameter is simple parameter).

IsAutoCreated [Page 103]

Determines if a parameter is automatically created in the class library

IsSimpleParam [Page 104]

Identifies the class of the import or export parameter

PassParamToR3 [Page 104]

Allows (and forces) the passing of this parameter to R/3 during a client function call (applicable only when this parameter is simple parameter).

SetParamName [Page 105]

Assigns a name to the parameter

Clear

Purpose

Clean out the data buffer for all values in the parameter.

Syntax

```
virtual void Clear (void);
```

Related Information

CRfcSimpleParam::Clear [Page 128]

CRfcStructParam::Clear [Page 150]

CRfcImpExpParam

Purpose

Constructs an import parameter or export parameter object.

Syntax

```
CRfcImpExpParam (CSTR strParamName=NULL,  int nLength = 0);
```

Parameters

- *strParamName*: Character string containing the parameter name.
- *nLength*: Length of the parameter value in bytes.

Exceptions

Throws const char* for memory exceptions.

Description

If parameter name or length can not provided at construction time, they can be set using other methods later on.

Related Information

CRfcImpExpParam (Copy Constructor) [Page 102]

SetParamName [Page 105]

CRfcSimpleParam::SetAttributes [Page 132]

CRfcStructParam::SetLength [Page 159]

CRfcImpExpParam (Copy Constructor)

Purpose

Copy constructor for CRfcImpExpParam.

Syntax

```
CRfcImpExpParam (Const CRfcImpExpParam& sourceParam);
```

Parameters

sourceParam: Reference to the parameter object to copy from.

Exceptions

Throws const char* for memory exceptions.

Related Information

CRfcImpExpParam (Copy Constructor) [Page 102]

GetLength

Purpose

Obtains the length of the parameter value in bytes.

Syntax

```
int GetLength(void) const;
```

Return Value

Integer value of the length.

Related Information

CRfcSimpleParam::SetAttributes [Page 132]

CRfcStructParam::SetLength [Page 159]

GetParamName

Purpose

Obtains the parameter name.

Syntax

```
CSTR GetParamName(void) const;
```

Return Value

A const char* containing the parameter name.

Related Information

SetParamName [Page 105]

IsAutoCreated

Purpose

Determines if a parameter is automatically created in the class library.

Syntax

```
virtual BOOL IsAutoCreated(void) const = 0;
```

Return Value

TRUE if automatically created, FALSE otherwise.

Description

This is a pure virtual function defined in the CRfcImpExpParam [Page 100] class. It must be implemented in any derived parameter classes.

InhibitParamToR3

Purpose

Inhibits the passing of this parameter to R/3 during a client function call (applicable only when this parameter is simple parameter).

Syntax

```
void InhibitParamToR3();
```

Parameters

None.

Description

RFC Library design offers the feature that if a parameter is not present (in the list of marshaled parameters) when a client function call reaches the destined function module in R/3, then the function module uses a pre-defined default value for that absent parameter. This feature is relevant only to simple parameters.

Related Information

PassParamToR3 [Page 104]

IsSimpleParam

Purpose

Identifies the class of the import or export parameter.

Syntax

```
virtual BOOL IsSimpleParam (void) const;
```

Return Value

- TRUE if the parameter is a simple parameter, containing a single value.
- FALSE if the parameter is a structure parameter, containing multiple values.

Description

This is a pure virtual function, implemented by the two derived classes: CRfcSimpleParam and CRfcStructParam. Call this function on any import or export parameter objects to find out about the type of the parameter.

Related Information

CRfcSimpleParam::IsSimpleParam [Page 131]

CRfcStructParam::IsSimpleParam [Page 153]

PassParamToR3

Purpose

Allows (and forces) the passing of this parameter to R/3 during a client function call (applicable only when this parameter is simple parameter).

Syntax

```
void PassParamToR3();
```

Parameters

None.

Description

RFC Library design offers the feature that if a parameter is not present (in the list of marshaled parameters) when a client function call reaches the destined function module in R/3, then the function module uses a pre-defined default value for that absent parameter. This function forces the passing of this parameter to R/3 during the call. This feature is relevant only to simple parameters.

Note: the other way of passing the parameter to R/3 during the call is to set a value to the parameter using the CRfcSimpleParam::Value() function. In the case that no particular value is to be set, then this function would also force the parameter to be passed.

Related Information

InhibitParamToR3 [Page 103]

SetParamName

Purpose

Assigns a name to the parameter.

Syntax

```
void SetParamName(CSTR paramName);
```

Parameters

paramName: A character string containing the name of the parameter.

Exceptions

Throws const char* for memory exceptions.

Description

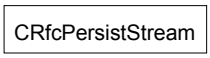
Once the parameter name is set, it can not be changed. Therefore, this function can only be called once, and only if the name was not provided at construction time.

Related Information

GetParamName [Page 102]

The CRfcPersistStream Class

Class Summary



The CRfcPersistStream class is defined in CRfcPers.h.

The CRfcPersistStream class is derived from ‚fstream‘, and is a class that encapsulates the following two functionality: (1) a text stream for storing persistent data of objects, and (2) implementation of specialized knowledge of organizing and retrieving metadata of objects of CRfcClientFunc, CRfcSimpleParam, CRfcStructParam and CRfcTableParam classes. The text stream is intended to be used to store persistent data of objects of user-defined classes along with the metadata of the above-mentioned classes of the RFC Class Library. However, any functionality that is necessary for organizing user-specific data must be user-implemented because this class simply does not have any knowledge of how to organize user data.

CRfcPersistStream

A CRfcPersistStream object is uni-directional. This means that the stream is used for either writing out to file or reading in from file, but cannot be both.

Important Note: for use in the ‚out' direction, the user's program is responsible for opening the file by calling fstream::open() and checking that the operation was executed successfully, before attempting to write the content of the stream to the physical file.

Construction

CRfcPersistStream [Page 106]

Constructs a persistent stream object.

CRfcPersistStream (with filename and direction) [Page 107]

Constructs a persistent stream object with given file name and direction.

Operation

Flush [Page 107]

Commits the content of the stream object to physical file storage.

GetDirection [Page 107]

Returns the STREAM_DIRECTION value indicating the direction of this stream object.

GetFileName [Page 108]

Returns the path/name of the physical disk file associated with this stream object.

SetDirection [Page 108]

Sets the STREAM_DIRECTION value indicating the direction of this stream object.

SetFileName [Page 109]

Sets the path/name of the physical disk file associated with this stream object.

CRfcPersistStream

Purpose

Constructs a persistent stream object.

Syntax

```
CRfcPersistStream() ;
```

Parameters

None.

CRfcPersistStream (with filename and direction)

Purpose

Constructs a persistent stream object with given file name and direction.

Syntax

```
CRfcPersistStream(CSTR filename, STREAM_DIRECTION dir) ;
```

Parameters

filename: pathname of the disk file associated with the stream.

dir: direction of the stream, can be DIR_INPUT, DIR_OUTPUT or DIR_UNDEFINED.

Description

A CRfcPersistStream object is uni-directional. This means that the stream is used for either writing out to file or reading in from file, but cannot be both.

Flush

Purpose

Commits the content of the stream object to physical file storage.

Syntax

```
BOOL Flush();
```

Parameters

None.

Return Value

Returns TRUE if operation is successful; otherwise FALSE.

Description

This function should be called after all data is written to the stream. Error in the physical file would occur if the user writes data to the stream, calls this function, then writes more data to the stream, then calls this function again. Also, the physical disk file must first be opened with fstream::open() before calling this function to avoid operation failure.

GetDirection

Purpose

Returns the STREAM_DIRECTION value indicating the direction of this stream object.

GetFileName

Syntax

```
STREAM_DIRECTION GetDirection() const;
```

Parameters

None.

Return Value

Returns the stored STREAM_DIRECTION value.

GetFileName

Purpose

Returns the path/name of the physical disk file associated with this stream object.

Syntax

```
CSTR GetFileName();
```

Parameters

None.

Return Value

Returns pointer to a constant character array that contains the path/name of the file.

SetDirection

Purpose

Sets the STREAM_DIRECTION value indicating the direction of this stream object.

Syntax

```
void SetDirection(STREAM_DIRECTION dir);
```

Parameters

dir: a STREAM_DIRECTION value to be stored.

SetFileName

Purpose

Sets the path/name of the physical disk file associated with this stream object.

Syntax

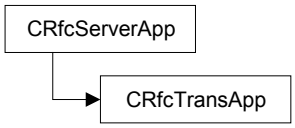
```
void SetFileName(CSTR szName);
```

Parameters

szName: path/name of the physical disk file.

The CRfcServerApp Class

Class Summary



The CRfcServerApp class is defined in CRfcSApp.h. (Copy from User Guide.)

The CRfcServerApp class encapsulates the initialization, running (listening for requests from R/3 and dispatching to corresponding functions) and termination of an RFC server application. An application built with the framework provided by the class library must have one and only one object of the class CRfcServerApp, which is called the application object.

Construction

CRfcServerApp [Page 111]

Constructs a server application object

Operations

AddServerFunction [Page 110]

Adds a server function object to the application object

Clone [Page 111]

Makes a dynamic copy of a CRfcServerApp object or a derived class object

Dispatch [Page 112]

Function to be called to process the next RFC request

GetConnection [Page 113]

Obtains the connection object associated with the application object

GetConnectionRfcHandle [Page 113]

Obtains the connection handle stored in the CRfcServerApp object

AddServerFunction

Purpose

Adds a server function object to the application object.

Syntax

```
void AddServerFunction(CRfcServerFunc& serverFunc);
```

Parameters

serverFunc: Reference to the server function object to be added.

Exceptions

Throws const char* for memory exceptions.

Related Information

RemoveServerFunction (by Index) [Page 117]

RemoveServerFunction (by Name) [Page 117]

Clone

Purpose

Makes a dynamic copy of a CRfcServerApp [Page 109] object or a derived class object.

Syntax

```
virtual CRfcServerApp * clone();
```

Exceptions

Throws const char* for memory exceptions.

Description

This clone() function must be implemented in the derived class of CRfcServerApp to make a multithreading application work. If you do not want to use the class library's multithreading server features, you do not need to implement this function in the derived class.

CRfcServerApp

Purpose

Constructs a server application object.

Syntax

```
CRfcServerApp (CRfcConnection* pConnection = NULL) ;
```

Parameters

pConnection: Pointer to a connection object.

Exceptions

Throws const char* when attempt to create more than 1 CRfcServerApp object.

Related Information

SetConnection [Page 119]

Dispatch

Purpose

Function to be called to process the next RFC request.

Syntax

```
RFC_RC Dispatch(void);
```

Return Value

Returns RFC_OK or other RFC_RC return code.

Description

To be used together with InstallFunction. Use this set of functions to install and dispatch your server functions written in C only. Otherwise, use the AddServerFunction and Run member functions to take advantage of the automatic dispatching provided by the server application class.

Related Information

GetName [Page 114]

InstallFunction [Page 115]

ExitInstance

Purpose

Stops the server application.

Syntax

```
virtual void ExitInstance (void);
```

Description

The default implementation simply closes down the R/3 connection. This results in stopping the Run [Page 118] loop. The user can provide their own ExitInstance implementation.

GetFunctionCount

Purpose

Obtains the total number of server functions registered with this application object.

Syntax

```
int  GetFunctionCount(void) const;
```

Return Value

An integer count of total registered server functions.

Related Information

GetServerFunction (by Index) [Page 114]
GetServerFunction (by Name) [Page 115]

GetConnection

Purpose

Obtains the connection object associated with the application object.

Syntax

```
      const CRfcConnection* GetConnection(void) const;
```

Return Value

A pointer to a const CRfcConnection object.

Related Information

SetConnection [Page 119]

GetConnectionRfcHandle

Purpose

Obtains the connection handle stored in the CRfcServerApp object.

Syntax

```
   RFC_HANDLE GetConnectionRfcHandle(void) const
```

Return Value

Consists of an unsigned integer, RFC_HANDLE, for the RFC handle.

Exceptions

None

GetName

Purpose

Obtains the name of the called function.

Syntax

```
RFC_RC GetName(RFC_FUNCTIONNAME functionName);
```

Return Value

RFC_OK or RFC_FAILURE

Parameters

functionName: Name of the called function module (RFC_FUNCTIONNAME).

Description

For server functions written in pure C only. Besides using Dispatch it is also possible to receive RFC call directly. The function GetName must then be used to get the name of the called function. The calling program then has to determine the interface of the requested function module and to receive the parameters as within a function being installed via InstallFunction.

In register mode (see CRfcConnection::Accept) the server program will wait for the next RFC requests.

Related Information

CRfcConnection::Accept [Page 51]

InstallFunction [Page 115]

Dispatch [Page 112]

GetServerFunction (by Index)

Purpose

Obtains a server function object by index.

Syntax

```
CRfcServerFunc* GetServerFunction(int nIndex) const;
```

Return Value

A pointer to the server function object. Returns NULL if no function with that index can be found.

Parameters

nIndex: The index number of the server function in the application object's server function list, starting from 0.

Related Information

GetServerFunction (by Name) [Page 115]

GetServerFunction (by Name)

Purpose

Obtains a server function object by name.

Syntax

```
CRfcServerFunc* GetServerFunction(RFC_FUNCTIONNAME functionName) const;
```

Return Value

A pointer to the server function object. Returns NULL if no function with that name can be found.

Parameters

functionName: The name of the server function in the application object's server function list.

Related Information

GetServerFunction (by Index) [Page 114]

OnIdle

Purpose

Things to do when waiting for incoming requests.

Syntax

```
virtual void OnIdle();
```

Description

Implement this function if you want to do something when the server is idle or during listening for response. It is automatically called by the Run member function. The default implementation does nothing.

Related Information

Run [Page 118]

InstallFunction

Purpose

Install a server function written in C with the RFC library.

ReceiveData

Syntax

```
RFC_RC InstallFunction (RFC_FUNCTIONNAME functionName,
                                RFC_ONCALL fp, char* Description);
```

Return Value

Returns RFC_OK or RFC_FAILURE, if there is no memory available to register the function.

Parameters

- *functionName*: Name of the function to install or register.
- *fp*: The function pointer a function of type RFC_ONCALL(RFC_RC RFC_ONCALL(void)), which contains the functionality being offered as an RFC function module. RFC_ONCALL
- *Description*: A character string containing the description of the function.

Exceptions

Throws const char* for memory exceptions.

Description

To be used together with Dispatch. Use this set of functions to install and dispatch your server functions written in C only. Otherwise, use the AddServerFunction and Run member functions to take advantage of the automatic dispatching provided by the server application class.

Related Information

GetName [Page 114]

Dispatch [Page 112]

UnInstallFunction [Page 120]

ReceiveData

Purpose

Receives data from R/3 systems.

Syntax

```
RFC_RC ReceiveData(void):
```

Parameters

None.

Return Value

RFC_RC structure data.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

This function receives data from connected R/3 systems. It is a virtual function that you can redefine in the derived class.

RemoveServerFunction (by Index)

Purpose

Removes a server function object from the application object. Also uninstalls the server function (the function written in C/C++ with the RFC class library).

Syntax

```
void RemoveServerFunction (int nIndex);
```

Parameters

nIndex: Index of the server function object to be removed, beginning from 0.

Exceptions

Throws const char* for memory exceptions.

Related Information

AddServerFunction [Page 110]

RemoveServerFunction (by Name) [Page 117]

RemoveServerFunction (by Name)

Purpose

Removes a server function object from the application object. Also uninstalls the server function (the function written in C/C++ with the RFC class library).

Syntax

```
void RemoveServerFunction(RFC_FUNCTIONNAME functionName);
```

Parameters

functionName: Name of the server function object to be removed.

Exceptions

Throws const char* for memory exceptions.

Related Information

AddServerFunction [Page 110]

RemoveServerFunction (by Index) [Page 117]

ReturnCall

Purpose

Sends data to R/3 systems.

Syntax

```
RFC_RC ReturnCall(void);
```

Parameters

None.

Return Value

RFC_RC structure data.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API failure. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Description

This function sends data to connected R/3 systems. It is a virtual function that you can redefine in the derived class.

Run

Purpose

Main loop of the server application.

Syntax

```
RFC_RC Run(void);
```

Win32 Only

```
RFC_RC Run(int nThreads=0);
```

Argument

nThreads, where this is the number of child threads that the class library will create during the main loop of the server application.

Return Value

RFC_RC structure data.

Exceptions

Throws RFC_ERROR_INFO_EX and RFC_ERROR_INFO for RFC API errors. SAP recommends that you check for the RFC_ERROR_INFO_EX. See the description of the RFC_ERROR_INFO_EX structure [Page 31].

Throws const char* for memory exceptions.

Description

Call this function when you are ready to process all incoming calls. You must perform the following tasks before you call this function:

- register all server function objects to the application object
- establish a valid connection with R/3
- associate the connection with the application object.

You can also call this function to run a multi-threading Microsoft Windows 32-bit application. If you do not enter an argument or enter a value of zero in the nThreads parameter, the RFC_RC Run(int nThreads=x) function is equivalent to the Run(void) function.

Related Information

AddServerFunction [Page 110]

OnIdle [Page 115]

SetConnection

Purpose

Associates a connection object with the application object.

Syntax

```
void SetConnection (CRfcConnection& connection);
```

Parameters

connection: Reference to a connection object.

Related Information

GetConnection [Page 113]

UnInstallFunction

Purpose

Uninstalls a server function written in C/C++ with the RFC C++ Class Library.

Syntax

```
RFC_RC UnInstallFunction (RFC_FUNCTIONNAME functionName);
```

Return Value

The RFC_RC return code. Please see the structure definition of RFC_RC for the available values.

Parameters

functionName: Name of the server function to be uninstalled.

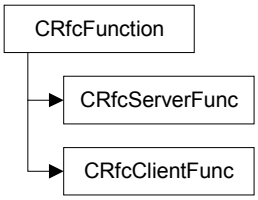
Related Information

InstallFunction [Page 115]

GetName [Page 114]

The CRfcServerFunc Class

Class Summary



The CRfcServerFunc class is d in CRfcServ.h.

The CRfcServerFunc class contains functionality that are needed when making calls to R/3 as a server. Each CRfcServerFunc object represents one remote-callable function from R/3.

Construction

CRfcServerFunc [Page 122]

Constructs a server function object

Operations

CallReceive [Page 121]

Makes a two-way call to R/3, sending and receiving import and export parameters and table data

Clone [Page 122]

Clones a CRfcServerFunc object for multithreading applications

GetDescription [Page 123]

Obtains a text description of the server function

GetServerApp [Page 123]

Returns a pointer to the parent CRfcServerApp or CRfcTransApp object

MultipleThreadDataProcess [Page 124]

Obtains the data, processes it, and then sends the data to the calling client for multithreading applications

Process [Page 124]

Data process logic for the server function.

Raise [Page 124]

Raises an exception while processing an RFC

RaiseTables [Page 125]

Raises an exception if function module being called has tables parameters

ReceiveData [Page 125]

Receives data from R/3 systems after a function call

ReturnCall [Page 126]

Sends data to R/3 systems after a function call

SetServerApp [Page 126]

Accepts and stores pointer to the parent CRfcServerApp or CRfcTransApp object

CallReceive

Purpose

Makes a two-way call to R/3, sending and receiving import and export parameters and table data.

Syntax

```
RFC_RC CallReceive(char*& strException, BOOL bPassAllParams=FALSE);
```

Parameters

strException: Refers to a string that is filled with error information when an RFC exception is encountered.

bPassAllParams: Indicates whether all simple import parameters should be passed to R/3 for the call.

Return value

The RFC_RC return code. Please see the structure definition of RFC_RC for the available values.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX for RFC function call failure.

Clone

Description

This function is equivalent to making sequential Call and Receive calls. Make sure all import, export, and table parameters are added to the function object, and that all required information is contained in the corresponding import or table parameters. Return values are available immediately from the export and table parameter objects upon return from CallReceive.

Note that when the argument bPassAllParams is set to TRUE, CallReceive passes ALL simple import parameters to R/3 for the call; otherwise CallReceive passes only those simple import parameters to R/3 that have values assigned to them.

Clone

Purpose

Clones a CRfcServerFunc object for multithreading applications.

Syntax

```
CRfcServerFunc * clone();
```

Return value

Consists of a pointer to the CRfcServerFunc object.

Exceptions

Throws const char* for memory exceptions.

CRfcServerFunc

Purpose

Constructs a server function object.

Syntax

```
CRfcServerFunc(RFC_FUNCTIONNAME FunctionName);
```

Parameters

FunctionName: Name of the remote-callable function, a character array.

Exceptions

Throws const char* for memory exceptions.

Description

See “Description” for the constructor of the CRfcFunction [Page 86] class.

Related Information

CRfcFunction [Page 86]

GetDescription

Purpose

Obtains a text description of the server function.

Syntax

```
virtual char* GetDescription (void) ;
```

Return Value

A string containing the description.

Description

The default implementation for this function returns a NULL string. The implementer of the derived class is responsible for providing a meaningful description of the particular function this derived class represents.

The server application object calls this function to fetch function descriptions on every function in its function list. This fetch happens automatically when the user requests *System Information → Function List* while configuring RFC destinations under transaction SM59:

1. Request transaction SM59.
2. Select *RFC Destinations → TCP/IP* from the configuration hierarchy.
3. Select any configuration entry in the *TCP/IP* list.
4. In the resulting screen, select the *System Information → Function List*.

The program entered in the *Explicit host* field connects to the relevant host and fetches Function Library descriptions for all the remote functions in the function list.

GetServerApp

Purpose

Returns a pointer to the parent CRfcServerApp or CRfcTransApp object.

Syntax

```
const CRfcServerApp* GetServerApp(void);
```

Parameters

None.

Return Value

A pointer to the parent CRfcServerApp or CRfcTransApp object.

Related Information

SetServerApp [Page 126]

CRfcServerApp [Page 111]

MultipleThreadDataProcess

Purpose

Obtains the data, processes it, and then sends the data to the calling client for multithreading applications.

Syntax

```
Void MultipleThreadDataProcess(void);
```

Exceptions

Throws const char* for memory exceptions.

Process

Purpose

Data process logic for the server function.

Syntax

```
virtual void Process (void);
```

Description

This function is a pure virtual one in the CRfcServerFunc class. It is the function to be implemented by inherited classes. It should contain all necessary function-specific data processing logic.

Do not call this function directly. It is called by Run [Page 118] function in the The CRfcServerApp Class [Page 109].

Upon entering the function, all data that are to be received from R/3 for the function should already have been received and stored inside the corresponding export and table parameters. In addition, upon existing the function, all data that are written into import and table parameters are automatically sent back to R/3.

The implementer of this function's main task is therefore reading values from the export and table parameters, implementing the processing logic of the data, and writing the result into appropriate import and table parameters.

Raise

Purpose

Raises an exception while processing an RFC.

Syntax

```
RFC_RC Raise(char* strException);
```

Return Value

RFC_OK or RFC_FAILURE.

Parameters

strException: Exception to be raised (null terminated string).

Description

If you are having tables parameters in your function module, use RaiseTables instead.

Related Information

RaiseTables [Page 125]

RaiseTables

Purpose

Raises an exception while processing an RFC, if the function module being called has tables parameters.

Syntax

```
RFC_RC RaiseTables(char* strException);
```

Return Value

RFC_OK or RFC_FAILURE.

Parameters

strException: Exception to be raised (null terminated string).

Related Information

Raise [Page 124]

ReceiveData

Purpose

Receives data from R/3 systems after a function call.

Syntax

```
RFC_RC ReceiveData(void);
```

Return value

RFC structure data.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX for Rfc function call failure.

Description

This function receives data from connected R/3 systems. It is a virtual function that you can redefine in the derived class.

Related Information

ReturnCall

[Page 126]

ReturnCall

Purpose

Sends data to R/3 systems after a function call.

Syntax

```
RFC_RC ReturnCall(void);
```

Return value

RFC structure data.

Exceptions

Throws const char* for memory exceptions.

Throws RFC_ERROR_INFO_EX for Rfc function call failure.

Description

This function sends data to connected R/3 systems. It is a virtual function that you can redefine in the derived class.

Related Information

ReceiveData

[Page 125]

SetServerApp

Purpose

Accepts and stores pointer to the parent CRfcServerApp or CRfcTransApp object.

Syntax

```
void SetServerApp(CRfcServerApp *pServerApp);
```

Parameters

pServerApp: pointer to a CRfcServerApp or CRfcTransApp object that would serve as the parent for this CRfcServerFunc object.

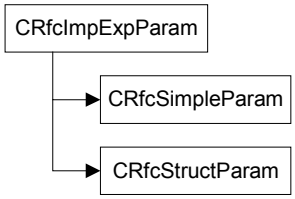
Related Information

GetServerApp [Page 123]

CRfcServerApp [Page 111]

The CRfcSimpleParam Class

Class Summary



The *CRfcSimpleParam* class is defined in *CRfcSimp.h.*

The *CRfcSimpleParam* class represents the simple import parameters or export parameters that contain only a single value. It inherits from the *CRfcImpExpParam* class.

Construction

CRfcSimpleParam [Page 128]

Constructs a simple parameter object

CRfcSimpleParam (Copy Constructor) [Page 129]

Copy constructor

Operations

Clear [Page 128]

Resets the parameter to an appropriate initial value

GetDecimals [Page 129]

Returns the number of decimal places if the parameter type is TYPP (packed)

GetType [Page 130]

Retrieves the RFC data type of the parameter

IsAutoCreated [Page 130]

Determines if a parameter is automatically created in the class library

IsSimpleParam [Page 131]

Identifies the object as a simple parameter

operator= (General) [Page 131]

Clear

Assignment operator

Read [Page 131]

Restores the metadata information about this CRfcSimpleParam object from given persistent data stream.

SetAttributes [Page 132]

Sets the parameter type and length

Value [Page 133]

Function for setting or getting the data value of the parameter

Value (for Constants) [Page 133]

Function for getting the data value of constant simple parameter objects

Write [Page 134]

Writes the metadata information about this CRfcSimpleParam object to the given persistent data stream.

Clear

Purpose

Reset the parameter to an appropriate initial value.

Syntax

```
virtual void Clear (void);
```

Description

Initial value for all numeric types is 0, initial value for all character string types is space.

CRfcSimpleParam

Purpose

Constructs a simple parameter object.

Syntax

```
CRfcSimpleParam ( CSTR paramName=NULL,
                  unsigned nType=TYPC,
                  int nLength=0);
```

Parameters

paramName: Name of the parameter.

nType: RFC type of the parameter.

nLength: Length of the parameter value in bytes. Not required for fixed-length parameter types.

Exceptions

Throws const char* for memory exceptions.

Description

If parameter name, type or length can not provided at construction time, they can be set using the SetParamName and SetAttributes functions later on.

Related Information

SetAttributes [Page 132]

SetParamName [Page 105]

CRfcSimpleParam (Copy Constructor)

Purpose

Copy constructor for CRfcSimpleParam.

Syntax

```
CRfcSimpleParam (const CRfcSimpleParam& sourceParam);
```

Parameters

sourceParam: Reference to the parameter object to copy from.

Exceptions

Throws const char* for memory exceptions.

Related Information

CRfcSimpleParam (constructor) [Page 128]

GetDecimals

Purpose

Returns the number of decimal places if the parameter type is TYPP (packed).

Syntax

```
unsigned GetDecimals(void) const;
```

Parameters

None.

GetType

Return Value

An unsigned integer containing the number of decimal places.

Description

When the type of the parameter is TYPP (PACK, binary-coded decimal), the number of decimal places is returned by this function.

GetType

Purpose

Retrieves the RFC data type of the parameter.

Syntax

```
unsigned GetType (void) const;
```

Return Value

An RFC data type.

Related Information

SetAttributes [Page 132]

RfcImpExpParam::GetLength [Page 102]

RfcImpExpParam::GetParamName [Page 102]

IsAutoCreated

Purpose

Determines if a parameter is automatically created in the class library.

Syntax

```
virtual BOOL IsAutoCreated(void) const {return m_bAutoCreate;}
```

Return Value

TRUE if automatically created, FALSE otherwise.

Description

This is a pure virtual function defined in the CrfcImpExpParam [Page 100] class. It is implemented in this derived parameter class.

IsSimpleParam

Purpose

Identifies that the object is a simple parameter.

Syntax

```
virtual BOOL IsSimpleParam (void) const;
```

Return Value

TRUE.

Related Information

CRfcImpExpParam::IsSimpleParam [Page 104]

CRfcStructParam::IsSimpleParam [Page 153]

operator= (General)

Purpose

Assignment operator.

Syntax

```
CRfcSimpleParam& operator=(const CRfcSimpleParam& sourceParam);
```

Return Value

A reference to the object itself.

Parameters

sourceParam: Reference to the object to assign from.

Exceptions

Throws const char* for memory exceptions.

Description

Copies everything from the source parameter, including name, type, length, and data value.

Read

Purpose

Restores the metadata information about this CRfcSimpleParam object from given persistent data stream.

SetAttributes

Syntax

```
BOOL Read(CRfcPersistStream& Pers);
```

Parameters

Pers: Reference to a CRfcPersistStream object that encapsulates the persistent data stream.

Exceptions

Throws const char* for memory exceptions.

Description

This function reads from the given persistent stream object to restore all metadata information about this CRfcSimpleParam object.

Important Note: Before calling this function, care must be taken to first set the parameter name attribute of this object using the SetParamName() function. This ensures that the Read() function successfully finds the pertinent metadata in the given persistent stream object.

Related Information

Write [Page 134]

The CRfcPersistStream Class [Page 105]

SetAttributes

Purpose

Sets the parameter type and length.

Syntax

```
void SetAttributes(unsigned nType,  int nLength=0);
```

Parameters

nType: RFC data type of the parameter.

nLength: Length of the parameter value in bytes.

Exceptions

Throws const char* for memory exceptions.

Description

Once the parameter type and length is set, they can not be changed. Therefore, this function can only be called once, and only if the type and length were not provided at construction time.

Value

Purpose

Function for setting or getting the data value of the parameter.

Syntax

```
CRfcData& Value (void) ;
```

Return Value

Reference to a CRfcData object containing the data value and type information.

Description

This function can be used to assign and retrieve values from the parameter through the support of CRfcData. User should not be concerned with or make any direct use of the intermediate CRfcData object.

Assuming AnIntParam is a simple parameter of type RFC_INT,

```
AnIntParam.Value() = 5;          //  Sets the parameter's
value to 5.
int anInt = AnIntParam.Value();  //  Retrieves the
parameter's value
                                  //  and assigns to anInt.
```

Related Information

Value (for Constants) [Page 133]

Value (for Constants)

Purpose

Gets the data value of constant simple parameter objects.

Syntax

```
const CRfcData& Value (void) const;
```

Return Value

Reference to a CRfcData object containing the data value and type information.

Description

Similar to the other Value function, but this one allows getting values from constant CRfcSimpleParam objects.

Assuming AnIntParam is a constant simple parameter of type RFC_INT,

```
int anInt = AnIntParam.Value();  //  Retrieves the
parameter's value
                                  //  and assigns it to
anInt.
```

Related Information

Value [Page 133]

Write

Purpose

Writes the metadata information about this CRfcSimpleParam object to the given persistent data stream.

Syntax

```
BOOL Write(CRfcPersistStream& Pers);
```

Parameters

Pers: Reference to a CRfcPersistStream object that encapsulates the persistent data stream.

Return Value

A flag indicating the result of the operation.

Description

This function writes to the given persistent stream object to save all metadata information about this CRfcSimpleParam object.

Related Information

Read [Page 131]

The CRfcPersistStream Class [Page 105]

The CRfcString Class

Class Summary

CRfcString

The CRfcString class is defined in CRfcStrn.h.

The CRfcString class contains some basic string operations.

Construction

CRfcString [Page 137]

Constructor from a character string

CRfcString (Copy Constructor) [Page 137]

Copy constructor

CRfcString (from a CRfcData object) [Page 138]

Constructor from a CRfcData object

CRfcString (with Padding) [Page 138]

This constructor is deprecated. We recommend that you use the CRfcString [Page 137] constructor instead.

Constructor from a character string to be padded with spaces for a designated length

Operations

Clear [Page 136]

Clears out the string buffer

Find [Page 139]

Finds the first occurrence of the given character in this CRfcString object.

GetCapitalize [Page 139]

Gets the flag in the object indicating automatic capitalization

GetLength [Page 140]

Gets the length of the string (without padded spaces)

GetPaddedLength [Page 140]

This function is deprecated. We recommend that you do not use it in new code.

Gets length of the string with padded spaces

GetPaddedString [Page 141]

This function is deprecated. We recommend that you do not use it in new code.

Gets a string padded with spaces

operator [Page 141]

Retrieves or sets the character value in the CRfcString by index

operator CSTR [Page 142]

Gets the string represented by the object

operator= (String object) [Page 144]

Assignment from a CRfcString object

operator= (String) [Page 142]

Assignment from a string

operator != (CRfcString object) [Page 142]

Clear

Compares to the contents of another CRfcString object for inequality.

operator != (character array) [Page 142]

Compares to the contents of a character array for inequality.

operator += (CRfcString object) [Page 143]

Concatenates the content of this and another CRfcString object and stores the result in this CRfcString object.

operator += (character array) [Page 143]

Concatenates the content of this CRfcString object and a character array, and stores the result in this CRfcString object.

operator == (CRfcString object) [Page 145]

Compares to the contents of another CRfcString object for equality.

operator == (character array) [Page 146]

Compares to the contents of a character array for equality.

operator >= (CRfcString object) [Page 146]

Compares lexigraphically to the contents of another CRfcString object for ‚greater than‘.

operator >= (character array) [Page 147]

Compares lexigraphically the contents of this CRfcSting object to the contents of a character array for ‚greater than.

SetCapitalize [Page 147]

Sets the capitalization flag

Substr (with starting position) [Page 148]

Obtains a CRfcString object whose content is the substring of this CRfcString object starting with the character at the given index and extends to the end

Substr (with starting position and length) [Page 148]

Obtains a CRfcString object whose content is the substring of this CRfcString object starting with the character at the given index and extends to the end

ToUpper [Page 149]

Converts all lower-case alphabetical characters to upper-case

Clear

Purpose

Clear out the string buffer.

Syntax

```
void Clear();
```

Description

This function does nothing if the object represents a null string. Otherwise it resets the string represented by the object to an empty string.

(GetPaddedLength, which is deprecated, will return a string with all spaces).

CRfcString

Purpose

Constructor from a character string.

Syntax

```
CRfcString(CSTR aString=NULL);
```

Parameters

aString: A character string.

Exceptions

Throws const char* for memory exceptions.

Description

Note that it is different to construct a CRfcString object using a null string than to do so using an empty string.

Related Information

CRfcString (with Padding) [Page 138]

CRfcString (Copy Constructor) [Page 137]

CRfcString (from a CRfcData object) [Page 138]

CRfcString (Copy Constructor)

Purpose

Copy constructor for CRfcString.

Syntax

```
CRfcString( const CRfcString& sourceString);
```

Parameters

sourceString: Reference to the CRfcString object to copy from.

Exceptions

Throws const char* for memory exceptions.

Related Information

CRfcString [Page 137]

CRfcString (with Padding) [Page 138]

CRfcString (from a CRfcData object) [Page 138]

CRfcString (from a CRfcData object)

Purpose

Constructor from a CRfcData object.

Syntax

```
CRfcString(const CRfcData& data);
```

Parameters

data: A CRfcData object containing data of any of the SAP data types.

Exceptions

Throws const char* for memory exceptions.

Throws const char* for data conversion errors.

Description

Converts data of any SAP data type contained in the CRfcData object in the argument to a character string encapsulated in this CRfcString object.

Related Information

CRfcString [Page 137]

CRfcString (with Padding) [Page 138]

CRfcString (Copy Constructor) [Page 137]

CRfcString (with Padding)

This constructor is deprecated. We recommend that you do not use this constructor in new code. Use the CRfcString constructor [Page 137], instead.

Purpose

Constructor from a character string, to be padded with spaces for a designated length.

Syntax

```
CRfcString(CSTR aString, int nPaddedLength);
```

Parameters

- *aString*: A character string.

- *nPaddedLength*: The string length to padded spaces up to.

Exceptions

Throws const char* for memory exceptions.

Description

This constructor reserves a memory block for the string with padded spaces. If aString is NULL and nPaddedLength is -1, then a null string object is constructed.

Related Information

CRfcString [Page 137]

CRfcString (Copy Constructor) [Page 137]

CRfcString (from a CRfcData object) [Page 138]

Find

Purpose

Finds the first occurrence of the given character in this CRfcString object.

Syntax

```
int Find(char ch) const;
```

Parameters

ch: the character whose position in this string object is to be determine.

Return

An integer index indicating the position of the character. The first position's index is 0. If the character is not found, then `CRfcString::m_nEndPos` is returned.

GetCapitalize

Purpose

Obtains the capitalization flag in the object indicating whether the string stored is automatically capitalized or not.

Syntax

```
BOOL GetCapitalize(void) const;
```

Return Value

TRUE if the string is capitalized, FALSE otherwise.

Description

When the capitalization flag is set, any string value (currently stored as well as subsequently assigned) to the object is automatically capitalized. Once set, capitalization is always done until the flag is reset to FALSE.

Related Information

SetCapitalize [Page 147]

GetLength

Purpose

Returns the length of the string.

Syntax

```
int GetLength(void) const;
```

Return Value

String length without any padding. Returns -1 if the string represented by the object is null.

GetPaddedLength

This function is deprecated. We recommend that you avoid using this function in new code.

Purpose

Length of the string with padded spaces.

Syntax

```
int GetPaddedLength(void) const;
```

Return Value

Padded string length. Returns -1 if the object represents a null string.

Description

If the padded length was not specified at construction time, this function simply returns the regular string length.

Related Information

CRfcString (with Padding) [Page 138]

GetLength [Page 140]

GetPaddedString

This function is deprecated. We recommend that you avoid using this function in new code.

Purpose

Obtains a string padded with spaces.

Syntax

```
CSTR GetPaddedString(void) const;
```

Return Value

A string padded with spaces to the specified padded length. Returns NULL if the object represents a null string.

Exceptions

Throws const char* for memory exceptions.

Description

If the padded length is not specified at construction time, then the function simply returns a regular string with no padded spaces.

Related Information

GetPaddedLength [Page 140]

CRfcString (with Padding) [Page 138]

operator

Purpose

Retrieves or sets the character value in the CRfcString by index.

Syntax

```
char CRfcString::operator[](int nIndex) const;
```

Parameters

NIndex: Indexing integer.

Return Value

A character value.

Exceptions

None.

operator CSTR

Description

This function accesses individual characters within a CRfcString after it receives the position of the characters in the string.

operator CSTR

Purpose

Obtains the read-only string equivalent of the object.

Syntax

```
operator CSTR() const;
```

Return Value

A read-only string. Returns NULL if the object cannot be converted to string.

operator != (CRfcString object)

Purpose

Compares to the contents of another CRfcString object for inequality.

Syntax

```
BOOL operator!=(const CRfcString& thatString);
```

Parameters

thatString: Reference to the other CRfcString object.

Return Value

Returns TRUE when inequality holds; otherwise returns FALSE.

Related Information

operator != (character array) [Page 142]

operator != (character array)

Purpose

Compares to the contents of a character array for inequality.

Syntax

```
BOOL operator!=(CSTR aString);
```

Parameters

aString: pointer to a character array.

Return Value

Returns TRUE when inequality holds; otherwise returns FALSE.

Related Information

operator != (CRfcString object) [Page 142]

operator += (CRfcString object)

Purpose

Concatenates the content of this and another CRfcString object and stores the result in this CRfcString object.

Syntax

```
CRfcString& operator+=(const CRfcString& thatString);
```

Parameters

thatString: reference to the other CRfcString object.

Return Value

Returns reference to this CRfcString object.

Exceptions

Throws const char* for memory exceptions.

Comments

Any existing padding is added to the end of the resulting string.

Related Information

operator += (character array) [Page 143]

operator += (character array)

Purpose

Concatenates the content of this CRfcString object and a character array, and stores the result in this CRfcString object.

operator= (String object)

Syntax

```
CRfcString& operator+=(CSTR aString);
```

Parameters

aString: reference to a character array.

Return Value

Returns reference to this CRfcString object.

Exceptions

Throws const char* for memory exceptions.

Comments

Any existing padding is added to the end of the resulting string.

Related Information

operator += (CRfcString object) [Page 143]

operator= (String object)

Purpose

Assignment from a CRfcString object.

Syntax

```
CRfcString& operator=(const CRfcString& sourceString);
```

Parameters

sourceString: Reference to the object to assign from.

Return Value

Reference to this object itself.

Exceptions

Throws const char* for memory exceptions.

Related Information

operator= (Stringcharacter array) [Page 145]

operator= (String)

Purpose

Assignment from a string.

Syntax

```
CRfcString& operator=(CSTR aString);
```

Parameters

aString: The string to assign from.

Return Value

Reference to this object itself.

Exceptions

Throws const char* for memory exceptions.

Description

If aString is NULL, the resulting object represents a NULL string.

Related Information

operator= (String object) [Page 144]

operator == (CRfcString object)

Purpose

Compares to the contents of another CRfcString object for equality.

Syntax

```
BOOL operator==(const CRfcString& thatString);
```

Parameters

thatString: Reference to the other CRfcString object.

Return Value

Returns TRUE when equality holds; otherwise returns FALSE.

Related Information

operator == (character array) [Page 146]

operator == (character array)

Purpose

Compares to the contents of a character array for equality.

Syntax

```
BOOL operator==(CSTR aString);
```

Parameters

aString: pointer to a character array.

Return Value

Returns TRUE when equality holds; otherwise returns FALSE.

Related Information

operator == (CRfcString object) [Page 145]

operator >= (CRfcString object)

Purpose

Compares lexigraphically to the contents of another CRfcString object for ‚greater than'.

Syntax

```
BOOL operator>=(const CRfcString& thatString);
```

Parameters

thatString: Reference to the other CRfcString object.

Return Value

Returns TRUE when the contents of this CRfcString object is lexigraphically greater than that of the other; otherwise returns FALSE.

Related Information

operator >= (character array) [Page 147]

operator >= (character array)

Purpose

Compares lexigraphically the contents of this CRfcSting object to the contents of a character array for ‚greater than‘.

Syntax

```
BOOL operator>=(CSTR aString);
```

Parameters

aString: pointer to a character array.

Return Value

Returns TRUE when the contents of this CRfcString object is lexigraphically greater than that of the character array; otherwise returns FALSE.

Related Information

operator >= (CRfcString object) [Page 146]

SetCapitalize

Purpose

Sets the capitalization flag indicating whether the string stored is automatically capitalized or not.

Syntax

```
void SetCapitalize(BOOL bCapitalize);
```

Parameters

bCapitalize: The flag value to set to.

Description

When the capitalization flag is set, any string value (currently stored as well as subsequently assigned) to the object is automatically capitalized. Once set, capitalization is always done until the flag is reset to FALSE.

Related Information

GetCapitalize [Page 139]

Substr (with starting position)

Purpose

Obtains a CRfcString object whose content is the substring of this CRfcString object starting with the character at the given index and extends to the end.

Syntax

```
CRfcString Substr(int pos) const;
```

Parameter

pos: the index of the character that marks the beginning of the desired substring.

Return Value

A CRfcString object containing the desired substring.

Exceptions

Throws const char* for memory exceptions or an illegal starting position.

Related Information

Substr (with starting position and length) [Page 148]

Substr (with starting position and length)

Purpose

Obtains a CRfcString object whose content is a substring of this CRfcString object, specifying the starting position for extracting the substring and the length of the substring to extract.

Syntax

```
CRfcString Substr(int pos, int len) const;
```

Parameter

pos: the index of the character that marks the beginning of the desired substring.

len: the length of the desired substring.

Return Value

A CRfcString object containing the desired substring.

Exceptions

Throws const char* for memory exceptions or an illegal combination of starting position and length.

Related Information

Substr (with starting position) [Page 148]

ToUpper

Purpose

Converts all lower-case alphabetical characters to upper-case.

Syntax

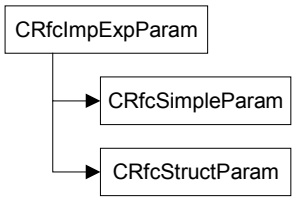
```
void ToUpper();
```

Parameter

None.

The CRfcStructParam Class

Class Summary



The CRfcStructParam class is defined in CRfcStrc.h.

The CRfcStructParam class represents the structure parameters (import or export) that contain multiple fields. This class inherits from both the CRfcImpExpParam class and the CRfcComplexParam class.

You can choose to have the object's constructor automatically establish all field information for the structure parameter. To do this, you must furnish further arguments to the constructor.

For automatic generation of field information for the structure parameter object, a valid connection object (with a valid RFC handle) must be passed to the constructor.

Construction

CRfcStructParam [Page 152]

Constructs a structure parameter object

CRfcStructParam (Copy Constructor) [Page 152]

Copy constructor

Operations

Clear [Page 150]

Clear

Resets each field of the parameter to an appropriate initial value

CopyContentsFrom [Page 151]

Copies contents from another CRfcStructParam object.

CopyContentsTo [Page 151]

Copies contents to another CRfcStructParam object.

GetParamName [Page 153]

Returns the name of the parameter

IsAutoCreated [Page 185]

Determines if a parameter is automatically created in the class library

IsSimpleParam [Page 153]

Verifies that the object is not a simple parameter

operator[ ] (Structure Fields By Index or Column) [Page 154]

Sets or gets the value of a field in a structure parameter by index or column number

operator[ ] (Structure Fields By Name) [Page 155]

Sets or gets the value of a field in a structure parameter by name

operator[ ] (Constant Structure Fields, by Index or Column) [Page 156]

Gets data value of field of a constant structure parameter by index or column number

operator[ ] (Constant Structure Fields, by Name) [Page 157]

Gets data value of field of a constant structure parameter by name

Read [Page 158]

Restores the metadata information about this CRfcStructParam object from given persistent data stream.

SetLength [Page 159]

Sets the total length of the structure parameter

Write [Page 159]

Writes the metadata information about this CRfcStructParam object to the given persistent data stream.

Clear

Purpose

Reset each field of the parameter to an appropriate initial value.

Syntax

```
virtual void Clear (void);
```

Exceptions

Throws const char* for memory exceptions.

Description

Initial value for all numeric types is 0, initial value for all character string types is space. If no type information is provided for a certain field, the memory is simply reset to 0.

CopyContentsFrom

Purpose

Copies contents from another CRfcStructParam object.

Syntax

```
void CopyContentsFrom(void* pSource);
```

Parameters

pSource: pointer to the memory block that holds the data to be copied.

Description

This function does a straight memory copy. The number of bytes to be copied is the length of this CRfcStructParam object.

Related Information

CopyContentsTo [Page 151]

CopyContentsTo

Purpose

Copies contents to another CRfcStructParam object.

Syntax

```
void CopyContentsFrom(void* pDestination);
```

Parameters

pDestination: pointer to the destination memory block.

Description

This function does a straight memory copy. The number of bytes to be copied is the length of this CRfcStructParam object.

Related Information

CopyContentsFrom [Page 151]

CRfcStructParam

Purpose

Constructs a structure parameter object.

Syntax

```
CRfcStructParam ( CSTR paramName=NULL,
                                unsigned nStructLen=0,
                                BOOL bAutoCreate=FALSE,
                                const CRfcConnection *pConnection=NULL,
                                CSTR strTabName=NULL);
```

Parameters

- *paramName*: Name of the parameter.
- *nStructLen*: Total length of the structure parameter value in bytes.
- *bAutoCreate:* TRUE to automatically generate field information for the structure parameter object.
- *pConnection:* pointer to a valid connection object with a valid RFC handle.
- *strTabName:* TABNAME (or type of structure) for the structure parameter to be created.

Exceptions

Throws const char* for memory exceptions and for failure of automatic generation of field information for the structure parameter.

Description

If parameter name or length can not provided at construction time, they can be set using the SetParamName and SetLength functions later on.

Related Information

SetParamName [Page 191]

SetLength [Page 190]

CRfcStructParam (Copy Constructor)

Purpose

Copy constructor for CRfcStructParam.

Syntax

```
CRfcStructParam (const CRfcStructParam & sourceParam);
```

Parameters

sourceParam: Reference to the parameter object to copy from.

Exceptions

Throws const char* for memory exceptions.

Related Information

CRfcStructParam (constructor) [Page 152]

GetParamName

Purpose

Returns the name of the parameter.

Syntax

```
CSTR GetParamName(void) const;
```

Parameters

None.

Return Value

A const char pointer to the character string containing the parameter name.

IsAutoCreated

Purpose

Determines if a parameter is automatically created in the class library.

Syntax

```
virtual BOOL IsAutoCreated(void) const = 0;
```

Return Value

TRUE if automatically created, FALSE otherwise.

Description

This is a pure virtual function defined in the CRfcImpExpParam [Page 100] class. It must be implemented in any derived parameter classes.

IsSimpleParam

Purpose

Identifies that the object is not a simple parameter.

Syntax

```
virtual BOOL IsSimpleParam (void) const;
```

Return Value

FALSE.

Related Information

CRfcImpExpParam::IsSimpleParam [Page 104]

CRfcSimpleParam::IsSimpleParam [Page 131]

operator[ ] (Structure Fields By Index or Column)

Purpose

Function for setting or getting the data value of a field in the structure parameter by index or column number.

Syntax

```
CRfcData& operator[ ] (int nIndex);
```

Parameters

nIndex: Column number of the field, beginning from 0.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified column.

Description

Use this function to assign and retrieve values from any field of the parameter through the support of CRfcData. User should not be concerned with or make any direct use of the intermediate CRfcData object.

To be able to access a field value, information about the field should have been already supplied to the parameter object using either of the CRfcComplexParam::AddFieldInfo functions.

Assuming AStructParam is a structure parameter with column number 0 being type RFC_INT,

```
AStructParam[0] = 5;            //  Sets column(field) 0 of
the parameter
                                //  to value 5.

int anInt = AStructParam[0];    //  Retrieves column(field) 0
of the
                                //  parameter and assigns it
to anInt.
```

Related Information

operator[ ] (Constant Structure Fields, by Index or Column) [Page 156]

operator[ ] (Constant Structure Fields, by Name) [Page 157]

operator[ ] (Structure Fields By Name) [Page 155]

CRfcComplexParam::AddFieldInfo [Page 41]

operator[ ] (Structure Fields By Name)

Purpose

Function for setting or getting the data value of a field in the structure parameter by name.

Syntax

```
CRfcData& operator[ ](CSTR fldName);
```

Parameters

fldName: Name of the field.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified field name.

Description

Use this function to assign and retrieve values from any field of the parameter through the support of CRfcData. User should not be concerned with or make any direct use of the intermediate CRfcData object.

To be able to access a field value, information about the field should have been already supplied to the parameter object using either of the CRfcComplexParam::AddFieldInfo functions.

Assuming AStructParam is a structure parameter with column named *NUMBER* of type RFC_INT,

```
AStructParam["NUMBER"] = 5;          //  Sets NUMBER field of
the
                                     //  parameter to value
5.
int anInt = AStructParam["NUMBER"];  //  Retrieves the value
of the
                                     //  NUMBER field of the
parameter
                                     //  and assigns it to
anInt.
```

Related Information

operator[ ] (Constant Structure Fields, by Index or Column) [Page 156]

operator[ ] (Constant Structure Fields, by Name) [Page 157]

operator[ ] (Structure Fields By Index or Column) [Page 154]

CRfcComplexParam::AddFieldInfo [Page 41]

operator[ ] (Constant Structure Fields, by Index or Column)

Purpose

Function for getting the data value of a field of a constant structure parameter by index or column number.

Syntax

```
const CRfcData& operator[ ] (int nIndex) const;
```

Parameters

nIndex: Column number of the field, beginning from 0.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified column.

Description

Similar to the other operator[ ](int) function, but this one allows getting values from constant CRfcStructParam objects.

Assuming AStructParam is a structure parameter with column number 0 of type RFC_INT,

```
int anInt = AStructParam[0];  //  Retrieves column(field) 0
of the
                              //  parameter and assigns it to
anInt.
```

Related Information

operator[ ] (Constant Structure Fields, by Name) [Page 157]

operator[ ] (Structure Fields By Index or Column) [Page 154]

operator[ ] (Structure Fields By Name) [Page 155]

CRfcComplexParam::AddFieldInfo [Page 41]

operator[ ] (Constant Structure Fields, by Name)

Purpose

Function for getting the data value of a field in the structure parameter by name.

Syntax

```
const CRfcData& operator[ ](CSTR fldName) const;
```

Parameters

fldName: Name of the field.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified field name.

Description

Similar to the other operator[ ](CSTR) function, but this one allows getting values from constant CRfcStructParam objects.

Assuming AStructParam is a structure parameter with column named *NUMBER* of type RFC_INT,

```
int anInt = AStructParam["NUMBER"];    //  Retrieves the value
of the
                                       //  NUMBER field of the
parameter
                                       //  and assigns it to
anInt.
```

Related Information

operator[ ] (Constant Structure Fields, by Index or Column) [Page 156]

operator[ ] (Structure Fields By Index or Column) [Page 154]

operator[ ] (Structure Fields By Name) [Page 155]

CRfcComplexParam::AddFieldInfo [Page 41]

Read

Purpose

Restores the metadata information about this CRfcStructParam object from given persistent data stream.

Syntax

```
BOOL Read(CRfcPersistStream& Pers);
```

Parameters

Pers: Reference to a CRfcPersistStream object that encapsulates the persistent data stream.

Exceptions

Throws const char* for memory exceptions.

Description

This function reads from the given persistent stream object to restore all metadata information about this CRfcStructParam object.

Important Note: Before calling this function, care must be taken to first set the parameter name attribute of this object using the SetParamName() function. This ensures that the Read() function successfully finds the pertinent metadata in the given persistent stream object.

Related Information

Write [Page 159]

The CRfcPersistStream Class [Page 105]

SetLength

Purpose

Sets the total length of the structure parameter.

Syntax

```
virtual void SetLength(int nLength);
```

Parameters

nLength : Length of the parameter value in bytes.

Exceptions

Throws const char* for memory exceptions.

Description

Once the parameter length is set, it can not be changed. Therefore, this function can only be called once, and only if the length information was not provided at construction time.

Write

Purpose

Writes the metadata information about this CRfcStructParam object to the given persistent data stream.

Syntax

```
BOOL Write(CRfcPersistStream& Pers);
```

Parameters

Pers: Reference to a CRfcPersistStream object that encapsulates the persistent data stream.

Return Value

A flag indicating the result of the operation.

Description

This function writes to the given persistent stream object to save all metadata information about this CRfcStructParam object.

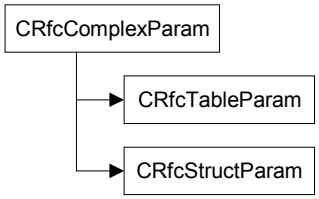
Related Information

Read [Page 158]

The CRfcPersistStream Class [Page 105]

The CRfcTableParam Class

Class Summary



The CRfcTableParam class is defined in CRfcTabl.h.

The CRfcTableParam class represents the table parameters for RFC functions. It inherits from the CRfcComplexParam class.

You can choose to have the object's constructor automatically establish all field information for the table parameter. To do this, you furnish further arguments to the constructor.

For automatic generation of field information for the table parameter object to be constructed, a valid connection object with a valid RFC handle must be passed to the constructor.

Construction

CRfcTableParam [Page 174]

Constructs a table parameter object

CRfcTableParam (Copy Constructor) [Page 175]

Copy constructor

Operations

AppendEmptyRow [Page 163]

Appends an empty row to the table

AppendInitializedRow [Page 163]

Appends a row with initialized fields to the table

Cell (by Row/Column Name) [Page 164]

Gets or sets individual cell values by row number and column name

Cell (by Row/Column Number) [Page 165]

Gets or sets individual cell values by row and column number

Cell (Constant Table, by Row/Column Name) [Page 167]

Gets or sets individual cell values by row number and column name from constant table parameter objects

Cell (Constant Table, by Row/Column Number) [Page 168]

Gets or sets individual cell values by row and column numbers from constant table parameter objects

The CRfcTableParam Class

Gets the name of the table parameter

GetParentFunction [Page 183]

Retrieves the function object to which this table is assigned

GetRowCount [Page 183]

Gets the total number of rows contained in the table parameter object

InsertEmptyRow [Page 183]

Inserts an empty row into the table

InsertInitializedRow [Page 184]

Inserts an empty row before the specified index and moves the cursor to the new row

IsAutoCreated [Page 185]

Determines if the parameter is automatically created in the class library

MoveDataFrom [Page 185]

Moves the data contents from the source table to this table object

operator() [Page 186]

Iterator operator

Read [Page 187]

Reads data from the persistent stream

RemoveRow [Page 187]

Deletes a row from the table

Row [Page 188]

Obtains pointer to a row's memory (Read/Write)

SetAutoConversion [Page 188]

Sets the flag for automatic conversion for the table parameter object

SetCurrentRow [Page 189]

Sets the current row cursor to the specified row

SetLength [Page 190]

Sets the total length of the table parameter row

SetParamName [Page 191]

Assigns the table parameter name

SetTypeHandle [Page 191]

Sets the RFC type handle

Write [Page 192]

Writes data to the persistent stream

AppendEmptyRow

Purpose

Appends an empty row to the table.

Syntax

```
void* AppendEmptyRow(void);
```

Return Value

Pointer to the memory of the new row.

Exceptions

Throws const char* for ItAppLine failure in the RFC C-API.

Description

The current row pointer is reset to point to the newly appended row.

Related Information

Row [Page 188]

ConstRow [Page 170]

RemoveRow [Page 187]

InsertEmptyRow [Page 183]

AppendInitializedRow [Page 163]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

AppendInitializedRow

Purpose

Appends a row with initialized fields to the table.

Syntax

```
void* AppendInitializedRow(void);
```

Return Value

Pointer to the memory of the new row.

Exceptions

Throws const char* for ItAppLine failure in the RFC C-API.

Description

The current row pointer is reset to point to the newly appended row. All fields in the appended row are initialized according to the data type of each field.

Related Information

Row [Page 188]

ConstRow [Page 170]

RemoveRow [Page 187]

InsertEmptyRow [Page 183]

AppendEmptyRow [Page 183]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

Cell (by Row/Column Name)

Purpose

Function for getting or setting individual cell values by row number and column name.

Syntax

```
CRfcData& Cell(int nRow, CSTR strFieldName);
```

Parameters

- *nRow*: Row number for the cell, starting from 0.
- *strFieldName*: Name of the column or field.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified column.

Throws const char* if no such row can be found.

Description

Use this function to assign and retrieve values from any cell through the support of CRfcData. User should not be concerned with or make any direct use of the intermediate CRfcData object.

To be able to access a cell value, information about the corresponding column or field should have been already supplied to the parameter object using either of the CRfcComplexParam::AddFieldInfo functions.

Note that cell by cell operations on the same row should be done through the functions SetCurrentRow and Field to achieve better performance.

Assuming ATableParam is a table parameter with column named *NUMBER* of type RFC_INT:

```
ATableParam.Cell(0, "NUMBER") = 5;          //  Sets table
cell at row 0,
                                            //  column
NUMBER to value 5.
int anInt = ATableParam.Cell(0, "NUMBER");  //  Retrieves
table cell at
                                            //  row 0,
column NUMBER, and
                                            //  assigns it
to anInt.
```

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

AddFieldInfo (Single Field) [Page 41]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

Cell (by Row/Column Number) [Page 165]

Cell (Constant Table, by Row/Column Name) [Page 167]

Cell (Constant Table, by Row/Column Number) [Page 168]

Cell (by Row/Column Number)

Purpose

Function for getting or setting individual cell values by row and column numbers.

Cell (by Row/Column Number)

Syntax

```
CRfcData& Cell(int nRow, int nCol);
```

Parameters

- *nRow*: Row number for the cell, starting from 0.
- *nCol*: Column number for the cell, starting from 0.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified column.

Throws const char* if no such row can be found.

Description

Use this function to assign and retrieve values from any cell through the support of CRfcData. User should not be concerned with or make any direct use of the intermediate CRfcData object.

To be able to access a cell value, information about the corresponding column or field should have been already supplied to the parameter object using either of the CRfcComplexParam::AddFieldInfo functions.

Note that cell by cell operations on the same row should be done through the functions SetCurrentRow and Field to achieve better performance.

Assuming ATableParam is a table parameter with column number 0 of type RFC_INT:

```
ATableParam.Cell(0,0) = 5;          //  Sets table cell at row 0,
column
                                    //  0 to value 5.

int anInt = ATableParam.Cell(0,0);  //  Retrieves table cell at row
0,
                                    //  column 0 and assigns it to
anInt.
```

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

AddFieldInfo (Single Field) [Page 41]

SetCurrentRow [Page 189]

operator() [Page 186]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

Cell (by Row/Column Name) [Page 164]

Cell (Constant Table, by Row/Column Name) [Page 167]

Cell (Constant Table, by Row/Column Number) [Page 168]

Cell (Constant Table, by Row/Column Name)

Purpose

Function for getting or setting individual cell values by row number and column name from constant table parameter objects.

Syntax

```
const CRfcData& Cell(int nRow, CSTR strFieldName) const;
```

Parameters

- *nRow*: Row number for the cell, starting from 0.
- *strFieldName*: Name of the field.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified field name.

Throws const char* if no such row can be found.

Description

Similar to the other Cell(int,CSTR) function, but this one allows getting values from constant CRfcTableParam objects.

Assuming ATableParam is a table parameter with column named *NUMBER* of type RFC_INT:

```
int anInt = ATableParam.Cell(0, "NUMBER");   //  Retrieves
table cell at row
                                             //  0, column
NUMBER, and
                                             //  assigns it
to anInt.
```

Related Information

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

Cell (by Row/Column Name) [Page 164]

Cell (by Row/Column Number) [Page 165]

Cell (Constant Table, by Row/Column Number) [Page 168]

Cell (Constant Table, by Row/Column Number)

Purpose

Function for getting or setting individual cell values by row and column numbers from constant table parameter objects.

Syntax

```
const CRfcData& Cell(int nRow, int nCol) const;
```

Parameters

- *nRow*: Row number for the cell, starting from 0.
- *nCol*: Column number for the cell, starting from 0.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified column.

Throws const char* if no such row can be found.

Description

Similar to the other Cell(int,int) function, but this one allows getting values from constant CRfcTableParam objects.

Assuming ATableParam is a table parameter with column number 0 of type RFC_INT:

```
int anInt = ATableParam.Cell(0,0);  //  Retrieves table cell
at row 0,
                                    //  column 0 and assigns
it to anInt.
```

Related Information

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

Cell (by Row/Column Name) [Page 164]

Cell (by Row/Column Number) [Page 165]

Cell (Constant Table, by Row/Column Name) [Page 167]

Clear

Purpose

Clear contents of the table.

Syntax

```
void Clear (void);
```

Exceptions

Throws const char* for ItFree failure in the RFC C-API.

Description

Only the data contents of the table is cleared, table structure information and the internal table handle are kept.

Related Information

Delete [Page 175]

ConstRow

Purpose

Obtains pointer to a row's memory for reading.

Syntax

```
const void* ConstRow(int nRow);
```

Parameters

nRow: Row number, starting from 0.

Return Value

Pointer to the specified row's memory if found. Otherwise returns NULL.

Description

Writing to the row memory is not permitted. Use the Row or CopyLineFrom functions if you want to write to the row. Use Cell or Field if you want to access individual fields.

Related Information

Row [Page 188]

CopyLineTo [Page 172]

CopyLineFrom [Page 171]

ConvertTable

Purpose

Does data conversion for all data in the table.

Syntax

```
void ConvertTable(void);
```

Exceptions

Throws const char* for ItGetLine failures in the RFC C-API.

Description

This function does data conversion or byte re-ordering field by field and row by row. May be computationally intensive for large tables.

Related Information

GetIsConverted [Page 182]

CopyDataFrom

Purpose

Copies the data contents from the source table to specified table object.

Syntax

```
void CopyDataFrom(const CRfcTableParam& sourceTable);
```

Parameters

sourceTable: Reference to the source table.

Exceptions

Throws const char* for the following C API failures: ItFree, ItCreate, ItRegTable, ItGetLine, and ItAppLine.

Description

The source table's structure should be the same as that of this table.

The old contents of the current table is deleted. The source table is not changed in any way after this operation.

Related Information

CRfcTableParam (Copy Constructor) [Page 175]

CreateEmptyTableCopy [Page 173]

MoveDataFrom [Page 185]

CopyLineFrom

Purpose

Copies the data from the specified memory block pointed into a table line.

Syntax

```
void CopyLineFrom( int nRow, void* pSource);
```

Parameters

- *nRow*: Row number, starting from 0.
- *pSource*: Pointer to the source memory block.

Exceptions

Throws const char* for ItPutLine failure in the RFC C-API.

CopyLineTo

Description

Caller of this function should make sure that the memory block provided contains enough data for the whole row, and that all data are valid.

Related Information

Row [Page 188]

ConstRow [Page 170]

AppendEmptyRow [Page 163]

InsertEmptyRow [Page 183]

CopyLineTo [Page 172]

CopyLineTo

Purpose

Copies the contents of line nRow to the designated memory block.

Syntax

```
void CopyLineTo(int nRow, void* pDestination);
```

Parameters

- *nRow*: Row number, starting from 0.
- *pDestination*: Pointer to the destination memory block.

Exceptions

Throws const char* for ItCpyLine failure in the RFC C-API.

Description

Caller of this function should make sure that the memory block provided contains enough number of bytes for the whole row's data.

Related Information

Row, ConstRow, AppendEmptyRow, InsertEmptyRow, CopyLineFrom

Row [Page 188]

ConstRow [Page 170]

AppendEmptyRow [Page 163]

InsertEmptyRow [Page 183]

CopyLineFrom [Page 171]

Create

Purpose

Creates an internal table handle and stores it inside the CRfcTableParam object.

Syntax

```
void Create(void);
```

Exceptions

Throws const char* for creation failure.

Description

The function does nothing if an internal table handle has already been created.

If the table parameter is used for RFC function calls as a client, then Create has to be called before making any calls to R/3. Otherwise data sent back from R/3 through the table will not be received properly.

Create is not required for tables in server functions.

CreateEmptyTableCopy

Purpose

Creates an empty copy of this table (with the same structure information, but no data contents).

Syntax

```
CRfcTableParam* CreateEmptyTableCopy (CSTR strParamName) const;
```

Parameters

strParamName: The table parameter name to be assigned to the new table parameter object.

Return Value

Pointer to the newly created table parameter objects.

Exceptions

Throws const char* for memory exceptions.

Description

Use the copy constructor, or the CopyDataFrom member function if you want the new table to have the name data content also.

Related Information

CRfcTableParam (Copy Constructor) [Page 175]

CopyDataFrom [Page 171]

MoveDataFrom [Page 185]

CRfcTableParam

Purpose

Constructs a table parameter object.

Syntax

```
CRfcTableParam (  CSTR paramName=NULL,
                              int  nLength=0,
                              BOOL bAutoConversion=FALSE,
                              BOOL bAutoCreate=FALSE,
                              const CRfcConnection *pConnection=NULL,
                              CSTR strTabName=NULL);
```

Parameters

- *paramName*: Name of the table parameter.
- *nLength*: Total length of a row for the table parameter in bytes.
- *bAutoConversion*: Whether or not to have automatic data conversion on the table object when receiving data from R/3. Currently not supported, always set to FALSE.
- *bAutoCreate:* TRUE for automatically establishing all field information for this table parameter.
- *pConnection:* For automatic creation of field information for the table parameter object. This is the connection to be used by the constructor in communicating with R/3. This must be a valid R/3 connection, with a valid RFC handle.
- *strTabName:* TABNAME (or type of table) for the table to be constructed. This argument is passed only for automatic creation of the field information of the table parameter object.

Exceptions

Throws const char* for memory exceptions and for failure of automatic creation of field information.

Description

If parameter name and length can not be provided at construction time, they can be set later when using the SetParamName and SetLength functions.

Before making any RFC calls as a client, make sure Create is called on all table parameter objects related to the RFC.

Related Information

Create [Page 173]

SetParamName [Page 191]

SetLength [Page 190]

CRfcTableParam (Copy Constructor)

Purpose

Copy constructor for CRfcTableParam.

Syntax

```
CRfcTableParam (const CRfcTableParam& sourceTable);
```

Parameters

sourceTable: Reference to the table object to copy from.

Exceptions

Throws const char* for memory exceptions.

Description

The copy constructor constructs a new table parameter object, copying every thing from the source parameter. If the source table contains a non-null internal table handle with or without any data, a new internal table handle is created for the new table parameter, and contents of the source table is copied to the new table, if any.

Related Information

Create [Page 173]

CRfcTableParam [Page 174]

Delete

Purpose

Delete the data contents of the table as well as the internal table handle.

Syntax

```
void Delete(void);
```

Exceptions

Throws const char* for ItDelete failure in the RFC C-API.

Description

Only the table content and internal table handle are deleted, table structure information is kept.

Related Information

Clear [Page 169]

Field (by Row/Column Number)

Purpose

Function for getting or setting individual field values in the current row by column number.

Syntax

```
CRfcData& Field(int nCol);
```

Parameters

nCol: Column number for the field, starting from 0.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified column.

Description

Use this function to assign and retrieve values from any field of the current row through the support of CRfcData. User should not be concerned with or make any direct use of the intermediate CRfcData object.

To be able to access a field value, information about the field should have been already supplied to the parameter object using either of the CRfcComplexParam::AddFieldInfo functions.

Assuming ATableParam is a table parameter with column number 0 of type RFC_INT, and current row set to 0:

```
ATableParam.Field(0) = 5;          //  Sets table cell at row
0, column 0
                                   //  to value 5.

int anInt = ATableParam.Field(0); //  Retrieves table cell at
row 0,
                                   //  column 0 and assigns it
to anInt.
```

Also see examples in operator() and SetCurrentValue

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

AddFieldInfo (Single Field) [Page 41]

SetCurrentRow [Page 189]

operator() [Page 186]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Cell (by Row/Column Name) [Page 164]

Cell (by Row/Column Number) [Page 165]

Cell (Constant Table, by Row/Column Name) [Page 167]

Cell (Constant Table, by Row/Column Number) [Page 168]

Field (by Row/Column/Field Name)

Purpose

Function for getting or setting individual field values in the current row by column or field name.

Syntax

```
CRfcData& Field(CSTR strFieldName);
```

Parameters

strFieldName: Name of the column or field.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified column.

Description

Use this function to assign and retrieve values from any field of the current row through the support of CRfcData. User should not be concerned with or make any direct use of the intermediate CRfcData object.

To be able to access a field value, information about the field should have been already supplied to the parameter object using either of the CRfcComplexParam::AddFieldInfo functions.

Assuming ATableParam is a table parameter with column named *NUMBER* of type RFC_INT, and current row set to 0:

```
ATableParam.Field("NUMBER") = 5;            //  Sets table
cell at row 0,
                                            //  column name
NUMBER to value 5.
int anInt = ATableParam.Field("NUMBER");    //  Retrieves
table cell at row
                                            //  0, column name
NUMBER and
                                            //  assigns it to
anInt.
```

Also see examples in operator() and SetCurrentValue

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

AddFieldInfo (Single Field) [Page 41]

SetCurrentRow [Page 189]

operator() [Page 186]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column Number) [Page 176]

Cell (by Row/Column Name) [Page 164]

Cell (by Row/Column Number) [Page 165]

Cell (Constant Table, by Row/Column Name) [Page 167]

Cell (Constant Table, by Row/Column Number) [Page 168]

Field (Constant Table, by Index/Column Name

Purpose

Function for getting the field value from the current row of a constant table parameter by index or column name.

Syntax

```
const CRfcData& Field(CSTR strFieldName) const;
```

Parameters

strFieldName: Name of the column or field.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified field name.

Description

Similar to the other Field(CSTR) function, but this one allows getting values from constant CRfcTableParam objects.

Assuming ATableParam is a table parameter with column named *NUMBER* of type RFC_INT, and current row set to 0:

```
int anInt = ATableParam.Field("NUMBER");  //  Retrieves table
cell at row
                                          //  0, column name
NUMBER and
                                          //  assigns it to
anInt.
```

Also see examples in operator() and SetCurrentValue

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

AddFieldInfo (Single Field) [Page 41]

SetCurrentRow [Page 189]

operator() [Page 186]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

Cell (by Row/Column Name) [Page 164]

Cell (by Row/Column Number) [Page 165]

Cell (Constant Table, by Row/Column Number) [Page 168]

Field (Constant Table, by Index/Column Number)

Purpose

Function for getting the field value from the current row of a constant table parameter by index or column number.

Syntax

```
const CRfcData& Field(int nCol) const;
```

Parameters

nCol: Column number of the field, beginning from 0.

Return Value

Reference to a CRfcData object containing the data value and type information.

Exceptions

Throws const char* if no field information can be found for the specified column.

Description

Similar to the other Field(int) function, but this one allows getting values from constant CRfcTableParam objects.

Assuming ATableParam is a table parameter with column number 0 of type RFC_INT, and current row set to 0:

```
int anInt = ATableParam.Field(0);  //  Retrieves table cell
at row 0,
                                   //  column 0 and assigns
it to anInt.
```

Also see examples in operator() and SetCurrentValue

Related Information

AddFieldInfo (Multiple Fields) [Page 41]

AddFieldInfo (Single Field) [Page 41]

SetCurrentRow [Page 189]

operator() [Page 186]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

Cell (by Row/Column Name) [Page 164]
Cell (by Row/Column Number) [Page 165]
Cell (Constant Table, by Row/Column Name) [Page 167]

GetAutoConversion

Purpose

Obtains the flag for automatic conversion for the table parameter object.

Syntax

```
BOOL GetAutoConversion(void) const;
```

Return Value

TRUE if the automatic conversion flag is set, FALSE otherwise.

Description

This function is reserved for later use, since automatic data conversion is currently not supported.

Related Information

SetAutoConversion [Page 188]

GetHandle

Purpose

Obtains the internal table handle contained in the table parameter object.

Syntax

```
ITAB_H GetHandle(void) const;
```

Return Value

An internal table handle. ITAB_HANDLE_NULL if not created.

Related Information

Create [Page 173]

GetIsConverted

Purpose

Obtains the table's conversion status.

Syntax

```
BOOL GetIsConverted(void) const;
```

Return Value

TRUE if all data contained in the table do not need byte re-ordering. FALSE if otherwise.

Description

Use this function to check the status of the table data if you are retrieving data using row operations, since these operations do not do any necessary data conversion or byte re-ordering. If the function returns FALSE, you will need to do the data conversion yourself.

You do not need to do the above if you are retrieving data using field operations (member functions Cell or Field).

Related Information

ConvertTable [Page 170]

GetLength

Purpose

Obtains the length of the table row in bytes.

Syntax

```
int GetLength(void) const;
```

Return Value

Integer value of the length.

Related Information

SetLength [Page 190]

GetParamName

Purpose

Obtains the name of the table parameter.

Syntax

```
CSTR GetParamName (void) const;
```

Return Value

Name of the table parameter.

Related Information

SetParamName [Page 191]

GetParentFunction

Purpose

Retrieves the function object this table is assigned to.

Syntax

```
const CRfcFunction* GetParentFunction (void) const;
```

Return Value

Pointer to the containing function object.

GetRowCount

Purpose

Obtains the total number of rows contained in the table parameter object.

Syntax

```
unsigned GetRowCount (void) const;
```

Return Value

Total row count.

InsertEmptyRow

Purpose

Inserts an empty row into the table.

Syntax

```
void* InsertEmptyRow(int nRow);
```

InsertInitializedRow

Parameters

nRow: Row number (starting from 0) before which the new row is to be inserted.

Return Value

Pointer to the memory of the new row.

Exceptions

Throws const char* for ItInsLine failure in the RFC C-API.

Description

The current row pointer is reset to point to the newly appended row.

Related Information

Row, ConstRow, AppendEmptyRow, RemoveRow, Field

Row [Page 188]

ConstRow [Page 170]

AppendEmptyRow [Page 163]

RemoveRow [Page 187]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Field (by Row/Column/Field Name) [Page 177]

Field (by Row/Column Number) [Page 176]

InsertInitializedRow

Purpose

Inserts an empty row before the specified index and moves the cursor to the new row.

Syntax

```
void* InsertInitializedRow(int nRow);
```

Parameters

NRow: An integer; specifies the row number.

Return Value

None.

Exceptions

Throws const char* for ItInsLine failure.

Description

InsertInitializedRow inserts an empty row before the specified index and moves the cursor to the new row. It then initializes all fields within the row according the data type of each field.

IsAutoCreated

Purpose

Determines if the parameter is automatically created in the class library.

Syntax

```
BOOL IsAutoCreated(void) const {return m_bAutoCreate;}
```

Return Value

TRUE if automatically created, FALSE otherwise.

Description

This IsAutoCreated function determines if the parameter is automatically created in the class library. The function belongs to the complex parameter class.

MoveDataFrom

Purpose

Moves the data contents from the source table to this table object.

Syntax

```
void MoveDataFrom(CRfcTableParam& sourceTable);
```

Parameters

sourceTable: Reference to the source table.

Exceptions

Throws const char* for ItDelete failure in the C API.

Description

The source table's structure should be the same as that of this table.

The old contents of the current table is deleted. The source table no longer contains any data after this operation.

Related Information

CRfcTableParam (Copy Constructor) [Page 175]

CreateEmptyTableCopy [Page 173]

CopyDataFrom [Page 171]

operator()

Purpose

Iterator operator.

Syntax

```
BOOL operator()(void);
```

Return Value

Returns TRUE if successfully moved to the next row, FALSE if end of the table is reached.

Description

The iterator operator reads the next row and stores it as the current row. Returns FALSE if the end of table is reached, in which case the iterator is reset back to before the beginning of the table.

Note that the member functions SetCurrentRow, RemoveRow, AppendEmptyRow and InsertEmptyRow also affects that current row setting.

Assuming ATableParam is a CRfcTableParam object, the following code iterates from the beginning of the table to the end, setting each row's column 0 to some integer value:

```
int i=0;
while( ATableParam())
      ATableParam.Field(0) = i++;
```

Related Information

SetCurrentRow [Page 189]

RemoveRow [Page 187]

AppendEmptyRow [Page 163]

InsertEmptyRow [Page 183]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Cell (by Row/Column Name) [Page 164]

Cell (by Row/Column Number) [Page 165]

Read

Purpose

Reads data from the persistent stream.

Syntax

```
BOOL Read(CRfcPersistStream& ps, CSTR funcname, int nIndex);
```

Parameters

ps: Refers to the CRfcPersistStream object.

funcname: The RFC function name.

nIndex: The table index.

Return Value

Boolean value, TRUE or FALSE.

Exceptions

None.

Description

Reads data from the persistent stream.

RemoveRow

Purpose

Deletes a row from the table.

Syntax

```
void RemoveRow(int nRow);
```

Parameters

nRow: Row number, starting from 0.

Exceptions

Throws const char* if the row is not found.

Throws const char* for ItDelLine failure in the RFC C-API.

Description

The current row pointer stored in the table is updated if necessary to still point to the same row before RemoveRow. If the current row is removed, the current row pointer is moved to point to the next row.

Related Information

Row [Page 188]

ConstRow [Page 170]

AppendEmptyRow [Page 163]

InsertEmptyRow [Page 183]

Row

Purpose

Obtains pointer to a row's memory for reading and writing.

Syntax

```
void* Row(int nRow);
```

Parameters

nRow: Row number, starting from 0.

Return Value

Pointer to the specified row's memory if found. Otherwise returns NULL.

Description

Writing to the row memory is permitted. To be safe, use the ConstRow function if you only need to read from the row. Use Cell or Field if you want to access individual fields.

Related Information

ConstRow [Page 170]

CopyLineTo [Page 172]

CopyLineFrom [Page 171]

SetAutoConversion

Purpose

Sets the flag for automatic conversion for the table parameter object.

Syntax

```
void SetAutoConversion(BOOL bAutoConversion);
```

Parameters

bAutoConversion: Value of the flag to set to.

Description

Set the auto conversion flag to TRUE if you want the class library to do byte-reordering for the whole table immediately upon receiving table data from R/3. If set to FALSE, the conversion is done only when data values are retrieved from the table.

This function is reserved for later use, since automatic data conversion is currently not supported.

Related Information

GetAutoConversion [Page 181]

SetCurrentRow

Purpose

Sets the current row cursor to the specified row.

Syntax

```
BOOL SetCurrentRow(int nRow);
```

Return Value

TRUE if cursor is successfully set. FALSE otherwise.

Description

The iterator operator reads the next row and stores it as the current row. Returns FALSE if the end Parameters

nRow: The row number, beginning from 0.

Description

Note that the member functions operator(), RemoveRow, AppendEmptyRow and InsertEmptyRow also affects that current row setting.

Assuming ATableParam is a CRfcTableParam object and its row structure contains fields all of type TYPC, the following code sets some fields of the first row of the table.

```
ATableParam.SetCurrentRow(0);

ATableParam.Field(0) = value1;  //  sets cell at row 0,
column 0,
                                //  equivalent to
                                //  ATableParam.Cell(0,0) =
value1

ATableParam.Field(1) = value2;  //  sets cell at row 0,
column 1,
                                //  equivalent to
                                //  ATableParam.Cell(0,1) =
value2;
```

Related Information

operator() [Page 186]

RemoveRow [Page 187]

AppendEmptyRow [Page 163]

InsertEmptyRow [Page 183]

Field (Constant Table, by Index/Column Name [Page 178]

Field (Constant Table, by Index/Column Number) [Page 180]

Cell (by Row/Column Name) [Page 164]

Cell (by Row/Column Number) [Page 165]

SetLength

Purpose

Sets the total length of the table parameter row.

Syntax

```
virtual void SetLength(int nLength);
```

Parameters

nLength : Length of a row in bytes.

Description

Once the row length is set, it can not be changed. Therefore, this function can only be called once, and only if the length information was not provided at construction time.

SetParamName

Purpose

Assigns the table parameter name.

Syntax

```
void SetParamName (CSTR paramName);
```

Parameters

paramName: Name of the table parameter.

Exceptions

Throws const char* for memory exceptions.

Description

Once the table parameter name is set, it can not be changed. Therefore this function can only be called once, and only if the parameter name was not provided at construction time.

Related Information

GetParamName [Page 182]

SetTypeHandle

Purpose

Sets the RFC type handle.

Syntax

```
void SetTypeHandle(RFC_TYPEHANDLE nTypeHandle);
```

Parameters

nTypeHandle: RFC type handle.

Return Value

None.

Exceptions

None.

Description

Sets the RFC type handle.

Write

Purpose

Writes data to the persistent stream.

Syntax

```
BOOL Write(CRfcPersistStream& ps);
```

Parameters

ps: Refers to the CRfcPersistStream object.

Return Value

Boolean value; returns TRUE (successful), or FALSE (failure).

Exceptions

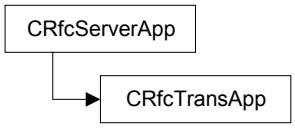
None.

Description

Writes data to the persistent stream.

The CRfcTransApp Class

Class Summary



The CRfcTransApp class is defined in CRfcTran.h.

The CRfcTransApp class is an abstract class. It encapsulates the initialization, running and termination of a transactional RFC server application, available from R/3 3.0 onwards. As the CRfcServerApp, a transactional RFC server application built with the framework provided by the class library must have one and only one object of a class derived from CRfcTransApp.

Construction

CRfcTransApp [Page 193]

Constructs a transactional server application object

Operations

OnCheckID [Page 193]

Called when local transaction starts

OnCommit [Page 194]

Called when local transaction ends

OnRollBack [Page 194]

Called when local transaction ends with failure

OnConfirm [Page 195]

Called when local transaction is completed

CRfcTransApp

Purpose

Constructs a transactional server application object.

Syntax

```
CRfcTransApp (CRfcConnection* pConnection = NULL) ;
```

Parameters

pConnection: Pointer to a connection object.

Exceptions

Throws const char* when attempt to create more than 1 CRfcTransApp object.

Related Information

SetConnection [Page 119]

OnCheckID

Purpose

This function is called when local transaction starts.

Syntax

```
virtual  int OnCheckID(RFC_TID TransID);
```

Return Value

0: transaction ID stored, transaction can be started.

1: transaction already done, skip the request.

< 0: cannot lock transaction, internal error.

Parameters

transID: The transaction ID for the local transaction.

Description

This function is a pure virtual function to be implemented by derived classes. It is called when a local transaction is starting. Since a transactional RFC call can be issued many times by the client system, the function is responsible for storing the transaction ID in permanent storage. If the client system tries starting the same transaction a second time, the function has to return 1.

Do not call this function yourself, it is automatically invoked during Run.

Related Information

OnCommit [Page 194]

OnRollBack [Page 194]

OnConfirm [Page 195]

OnCommit

Purpose

The function is called when local transaction starts.

Syntax

```
virtual  void OnCommit(RFC_TID TransID);
```

Parameters

transID: The transaction ID for the local transaction.

Description

This function is a pure virtual function to be implemented by derived classes. It is to be used to commit the local transaction, if necessary. If this functions has access to a SQL database, it should perform a operation like Commit work;

Do not call this function yourself, it is automatically invoked during Run.

Related Information

OnCheckID [Page 193]

OnRollBack [Page 194]

OnConfirm [Page 195]

OnRollBack

Purpose

This function is called when a local transaction ends with failure.

Syntax

```
virtual  void OnRollBack(RFC_TID TransID);
```

Parameters

transID: The transaction ID for the local transaction.

Description

This function is a pure virtual function to be implemented by derived classes. It is to be used to roll back the local transaction, if necessary. If this functions has access to a SQL database, it should perform a operation like Rollback work;

Do not call this function yourself, it is automatically invoked during Run.

Related Information

OnCheckID [Page 193]

OnCommit [Page 194]

OnConfirm [Page 195]

OnConfirm

Purpose

This function is called when local transaction is completed.

Syntax

```
virtual  void OnConfirm(RFC_TID TransID);
```

Parameters

transID: The transaction ID for the local transaction.

Description

This function is a pure virtual function to be implemented by derived classes. In general it can be used to delete the transaction ID from permanent storage. If this functions has access to a SQL database, it should perform a operation like

Delete From SomeTable where key = :transactionId; Commit Work;

where the table 'SomeTable' should have a unique index over the transaction ID.

Do not call this function yourself, it is automatically invoked during Run.

Related Information

OnCheckID [Page 193]

OnCommit [Page 194]

OnRollBack [Page 194]

Programming Examples

The sample program files for the RFC C++ Class Library reside in the following subdirectory under the directory where you have installed SAP Automation:

`\crfcsdk\samples`

The following table lists the sample program files:

Program Files	Description of Sample Program
crclient.h, crclient.cpp	Client program
crautocl.hpp, crautocl.cpp	Client program with parameters automatically created in the class library
crautsrv.h, crautsrv.cpp	Server program with an included ABAP client in the header file for testing the auto-creating server feature
crpersst.cpp	Program testing persistent features of the class library
crserver.h, crserver.cpp	Server program
crservth.h, crservth.cpp	Multithreading server program
crtstasv.h, crtstasv.cpp	Client program for testing the auto-creating server feature
sapinfo.hpp, sapinfo.cpp	Client program for getting R/3 system info
srfcserv.hpp, srfcserv.cpp	Server program registering multiple RFC functions with R/3 systems.
trfcserv.hpp, trfcserv.cpp	Server program for transactional RFC
trfctest.hpp, trfctest.cpp	Client program for transactional RFC
crclntth.h, crclntth.cpp	Multithreading client program
Crcltdlg.cpp, crcltdlg.h, crcltmfc.cpp, crcltmfc.h, crcltmfc.rc, crhnderr.h, crimpmdg.cpp, crimpmdg.h, crresrce.h, crrfcclt.cpp, crrfcclt.h, crshexpm.cpp, crshexpm.h, crshtbpm.cpp, crshtbpm.h, crstdafx.cpp, crstdafx.h	Client program using MFC together with the class library to provide a user interface. For use on Microsoft Windows systems supporting MFC in Visual C++ 5.0 or 6.0 only.

For more information on the sample programs please see the detailed comments in each program and the documentation for each class library function.

All non-multithreading class library programs can run on multiple platforms, including Windows, UNIX, and IBM AS/400.

Multithreading Examples

All multithreading examples require a 32-bit Windows platform (Windows NT or Windows 95/98).