SAP .NET Connector 3.0 Programming Guide

HELP.BCMIDCONF

Release 3.0

Copyright

services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

Icons in Body Text

Icon	Meaning
⚠	Caution
❖	Example
💡	Note
⬆	Recommendation
◇	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help → General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Typographic Conventions

Type Style	Description
Example text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options.
	Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
Example text	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
Example text	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
<Example text>	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
EXAMPLE TEXT	Keys on the keyboard, for example, F2 or ENTER.

The SAP .NET Connector 3.0 Programming Guide

This documentation gives you an overview of the SAP .Net Connector 3.0 API and shows how to use this API for RFC Client and Server programming, including some programming samples.

Note

For a detailed description of the .Net Connector API (API Reference) please check the file *NCo30APIDocumentation.chm*, which is included in the Nco distribution.

Metadata and Data Container Classes

The basic concepts that you need to be familiar with for both RFC client programming as well as RFC server programming, are the metadata classes and the data container classes.

Note

Metadata classes are descriptions of a function module's importing, exporting, changing and tables parameters and descriptions of the field layout of structures and tables used in function modules. Data container classes specify objects with the correct data layout of a certain function module, structure or table, into which you can fill your input data or from which you can read the function module's output data.

In most cases you don't need to work with the metadata classes yourself: the .NET Connector runtime will use them internally and will already give you the corresponding data container classes to work with. You only need to tell the .NET Connector, where to get the metadata information from. For this end you use the concept of a *repository*.

The Repository

In most common use cases the .NET Connector will read the necessary metadata information from the data dictionary (DDIC) of a connected SAP backend system. An SAP backend system is represented by the class `RfcDestination`. Consequently, you can obtain a repository from an `RfcDestination` object. The repository object then gives you access to both metadata descriptions (needed only, if you want to explore the layout of unknown function modules, structures and tables dynamically) and data containers (needed for executing RFC client calls, as well as for processing incoming RFC requests in a server program). A repository is represented by the class `RfcRepository.`

Working with Hard-Coded Metadata

An alternative to looking up the metadata dynamically from a backend is to hard-code the structure and function module descriptions in your source code. However, this should be necessary only in exceptional cases, for example because you want to write a server application, that either can be run without logon credentials for the backend or wants to provide function modules that do not exist in the backend's DDIC. For using hard-coded metadata you have to create a “custom repository”. A custom repository can be used to store your home-made metadata. Optionally, you can assign a *destination* to the custom repository. This allows you to provide only a few function descriptions of your own and have the custom repository lookup the remaining function modules from the backend's DDIC using the given destination object.

The Metadata Classes

In case you really need to work with the metadata objects, here they are. Basically there are two kinds of metadata:

- **Element metadata**, which are used for fields of structures, parameters of function modules and attributes of ABAP classes. They contain information like the data type of a field (CHAR, FLOAT, INT, etc.) and its length.
- **Container metadata,** which are used to describe composite types like structures (which are collections of various fields) and function modules (which are collections of their parameters and exceptions)

Element Metadata

The base class for all element metadata classes is:

```
public abstract class RfcElementMetadata
```

Three classes are derived from that base class:

- `RfcFieldMetadata` describing a field of a structure. It contains information about the field’s name, data type, its offset inside the structure and - where appropriate - about its length and decimals value.
- `RfcParameterMetadata` describing an importing/exporting/tables parameter of a function module.
- `RfcAttributeMetadata` describing an attribute of an ABAP object.

Container Metadata

After we defined the necessary metadata classes for the simple element types, we can now proceed to the metadata definitions for composite types (structure, table, class).

This is the base class for all container metadata classes:

```
public abstract class RfcContainerMetadata<T>
where T : RfcElementMetadata
```

Like in the element case, there are three derivations of this class:

- `RfcStructureMetadata:RfcContainerMetadata<RfcFieldMetadata>` describing a structure. It contains information about the DDIC name, the total length and the various fields of a structure.
- `RfcTableMetadata:RfcContainerMetadata<RfcFieldMetadata>` describing a table. Very similar to a structure metadata.
- `RfcFunctionMetadata:RfcContainerMetadata<RfcParameterMetadata>` describing a function module. It contains general DDIC information, information about the function module’s exceptions and parameter metadata for each of the FM’s parameters.
- `RfcAbapObjectMetadata:RfcContainerMetadata<RfcAttributeMetadata>` describing an ABAP object. At the moment used only for transporting class-based exceptions.

Data Container Classes

Data containers are objects into which you can fill your application data (input data in case of RFC client calls, output data in case your program acts as an RFC server) or from which you can read the SAP system's data (output data in case of RFC client calls, input data in case your program acts as an RFC server).

The most commonly used data containers are structures, tables and function modules, but there are also special data containers for tRFC, qRFC and bgRFC logical units of work (LUWs), which will be discussed in the corresponding sections on tRFC, qRFC and bgRFC.

These classes are defined as interfaces, so that future changes to the internal data layout will not affect the applications.

You should almost never need to create a data container yourself. Just call the `GetStructure()/GetTable()` methods of `RfcFunction` or of the parent container and the .NET Connector will create the appropriate container (using lazy initialization). But for those rare cases where you really need to create one of your own, you can use the Create-APIs on the corresponding Metadata classes.

The abstract base class for all data containers is:

`public interface IRfcDataContainer`

Similar to the metadata classes there are also four derived classes implementing the actual data containers:

- **`IRfcStructure`**
- **`IRfcTable`**
- **`IRfcFunction`**
- **`IRfcAbapObject`**

I think it's pretty self-explanatory what these classes stand for.

Normally you will never need to create a data container yourself, except for function modules. The obvious way of creating a function is

```
RfcRepository.CreateFunction("BAPI_CUSTOMER_GETLIST");
```

If you happen to have an instance of `RfcFunctionMetadata` at your disposal, you can also use `RfcFunctionMetadata.CreateFunction()`.

The other data containers can then be obtained from the function module via `IRfcFunction.GetStructure()` or `IRfcFunction.GetTable()`. The `CreateX()` functions of `RfcStructureMetadata` and `RfcTableMetadata` will be needed only rarely.

Two special features of an IRfcFunction deserve additional attention: the flags describing whether a certain parameter is "active" or not, which controls the behavior of optional parameters, and the flag describing whether class-based exceptions shall be transferred (if the function module uses any). In the following, I'll explain these features in more detail.

Optional Parameters – Client Case

In an RFC client application, by default all parameters are active in the beginning, except for the optional parameters. Only active importing, changing and tables parameters are sent to the backend during `Invoke()`, i.e. they are "supplied" to the backend system. And only active exporting, changing or tables parameters are "requested" from the backend, i.e. the backend will return values for them.

So the “active” flag controls two things: for parameters that are input to the function module, it determines whether a value for that parameter is transferred to the backend system, and consequently whether the backend system will use this parameter’s default value as defined in SE37 or not. And for parameters that are output of the function module, it controls whether a value for the parameter will be returned to you.

This feature can be used to significantly improve a function module’s runtime performance as well as the network time: if a function module has several output tables, for which it potentially selects several thousand lines, but you are interested in only one of these tables, then you can set the other tables to inactive. The function module’s ABAP code can test for this condition via the keyword “IS SUPPLIED” (or “IS REQUESTED” in older releases) and can then omit a time-consuming algorithm for selecting this unneeded data. And the backend’s RFC runtime can omit the work of serializing and sending that data over the network.

Another use case is: if you know that a default value is defined for this parameter in SE37 and you want to have the backend use that default value, you can set the parameter to inactive (if it isn’t already). Otherwise the .NET Connector would send the initial value corresponding to this parameter’s data type, and the backend system will use that instead of the parameter’s default value.

Although optional parameters are inactive in the beginning, once you set a value for one via one of the many `SetValue()` functions, it is automatically activated. (Same for non-optional parameters that have previously been deactivated using `SetParameterActive(false)`.) However, if you call `SetParameterActive(false)`, after you have set a value for that parameter, that value is not sent. Or in other words: the last activation/deactivation ‘wins’.

RFC client applications will usually not need `IsParameterActive()`.

Note:

SE37 allows defining default values also for non-optional parameters.

Optional Parameters – Server Case

In an RFC server application, by default all those parameters are active, which the backend system has supplied or requested. So in your application you can test, which of the input parameters the backend system has supplied (and use default values for the ones that are not supplied) and which of the output parameters the backend system has requested (and omit producing values for the ones that are not requested).

So basically by using `IsParameterActive()`, you can simulate the behavior of the ABAP keyword “IS SUPPLIED” (or “IS REQUESTED”).

RFC server applications will usually not need `SetParameterActive()`

Class-based Exceptions

Using the property

```
RfcAbapClassException.Mode IRfcFunction.AbapClassExceptionMode
```

an RFC client application can control the way the RFC protocol handles ABAP class-based exceptions for this particular function call. “OFF” means that any class-based exception the function module may throw, is converted into a standard SYSTEM_FAILURE and reported to you in that format. If you want to get a representation of the real ABAP object corresponding to the exception, set this property to “EXCEPTION_CHAIN” (only scalar attributes of the ABAP exception object are transferred via RFC) or “FULL” (all object attributes are transferred, even if these attributes are structures, tables or objects themselves – with possible further sub-objects and sub-sub-objects). The default is “OFF”.

For ABAP objects (as of today used in “class-based exceptions”) the following container is used:

public interface IRfcAbapObject:IRfcDataContainer

CodeSample: Accessing the Fields of a Table

The following sample code illustrates the usage of field access in the “JCo way” and in the “ADO .NET way”.In the given example a function module returned a table named ADDRESSES and we want to loop over it and print the column STREET. Here is first how this would look like in the JCo way:

```
IRfcFunction function = …;
IRfcTable addresses = function.GetTable(“ADDRESSES”);

Console.WriteLine(“STREET”);
for (addresses.CurrentIndex = 0;
            addresses.CurrentIndex < addresses.RowCount;
            ++(addresses.CurrentIndex)){
      Console.WriteLine(addresses.GetString(“STREET”));
}
```

And here is how the coding would look like in the alternative ADO .NET way:

```
IRfcFunction function = …;
IRfcTable addresses = function[“ADDRESSES”].GetTable();

Console.WriteLine(“STREET”);
for (int index = 0;
            index < addresses.RowCount;
            ++index){
      Console.WriteLine(addresses[index][“STREET”].GetString());
}
```

The following three classes are helper classes for the alternative way of reading/writing field and parameter values (the ADO .NET way).

public interface IRfcElement

For structure/table fields:

public interface IRfcField:IRfcElement

For function module parameters:

```
public interface IRfcParameter:IRfcElement
```

For ABAP object attributes:

```
public interface IRfcAttribute:IRfcElement
```

Objects of these types are returned by the string-indexer operations in the above example. For instance the expression `function[“ADDRESSES”]` returns an object of type `IRfcParameter`, and then calling `GetTable()` on this object returns an `IRfcTable`, if the parameter ADDRESSES is indeed a tables parameter.

However, applications will probably never need to use the `IRfcElement` classes directly.

RFC Client Programming

The process of executing a function module in the SAP backend system usually looks as follows:

1. Provide the necessary logon parameters and obtain an `RfcDestination` object corresponding to the SAP system in which you want to call the function module.
2. Use the destination’s repository to create an `IRfcFunction` object for the function module you want to call
3. Fill your input data into the importing, changing and tables parameters of the `IRfcFunction` object, using the data container API as outlined in section *Data Container Classes*.
4. Execute the function on the given destination.
5. Read the function module’s output data from the exporting, changing and tables parameters.

These five steps are now described in detail.

Destination Management

All RFC destinations of an application are configured centrally using one (singleton) class, the `RfcDestinationManager`. It manages the logon parameters for all SAP systems to which the application needs access. Basically, there are three mutually exclusive ways for an application to provide the logon parameters needed for SAP system access:

- Provide the parameters during runtime (e.g. if you get them from a UI).
- Define the parameters in the standard .NET configuration file of the executable
- Implement and provide a configuration object, which the `RfcDestinationManager` can “ask”, whenever it needs logon parameters for a particular system. This configuration object can then do whatever it wants (e.g. lookup parameters in a central LDAP system or a database), as long as it comes up with the required parameters.

The base class for destination management is:

```
public class RfcDestinationManager
```

Applications that don’t want to use a central configuration, can simply call

```
RfcDestinationManager.GetDestination(RfcConfigParameters parameters);
```

and provide the necessary logon parameters in the parameters collection. The `RfcConfigParameters` class contains constants for all parameter names like application server host, system number, user, password, etc. However, this approach is recommended only for test programs or small applications. (See also *Appendix A*: List of Possible Logon and Configuration Parameters.)

A different approach is to provide all logon parameters in the app.config file of the application. See the file sample_configuration_destinations.config in the tutorial ZIP archive for examples on how to set up the app.config file. The parameter names to be used are the same as the string constants defined in `RfcConfigParameters`. (See also *Appendix A*: List of Possible Logon and Configuration Parameters.)

And finally, applications that want to use the “configuration approach” need to implement the following interface

`public interface IDestinationConfiguration`

and then register an instance of this interface with the RfcDestinationManager during application startup:

```
RfcDestinationManager.RegisterDestinationConfiguration(myConfig);
```

(Alternatively, they can provide AssemblyName and TypeName of the implementation in the executable’s configuration file, and the .NET Connector will then try to load an instance of the given class from the Global Assembly Cache. For an example of how to do this, see the file sample_configuration_types.config in the tutorial ZIP archive.)

This object then provides access to the application’s central SAP system configuration. This approach is certainly the most flexible and most secure one.

Code Sample: Implementing a Destination Configuration

For a code sample showing how to implement an `IDestinationConfiguration` within the .Net Connector please refer to the example `SampleDestinationConfiguration.cs` included in the tutorial ZIP archive.

Via `RfcDestinationManager.GetDestination()` you can now get an object of the following type, which is a reference to a particular SAP system with particular user and logon information:

`public class RfcDestination`

It encapsulates functionality like connection pooling and the invocation mechanism for synchronous RFCs, tRFCs, qRFCs and bgRFCs.

Additionally, for a set of most important parameters, it provides *read-only* properties.

The remaining steps 2. – 5. are now illustrated in the following code sample.

Code Sample: Executing synchronous RFCs

The tutorial ZIP archive provides a code sample (`StepByStepClient.cs`) showing how to implement a .Net Connector RFC client program performing the following steps:

- Create a destination and a function object
- Execute a (synchronous stateless) call: see method `ExampleSimpleCall()`
- Read the data returned from the function module: see method `ExampleWorkWithStructuresAndTables()`
- Handle exceptions: see method `ExampleCatchErrors()`

Code Sample: Session Management

If you want to perform stateful client calls (function module invocations that keep the SAP system user session across several RFC calls), you need to think about "Session Management" and the requirements you place upon your session management.

The most important class used for session management is the `ISessionProvider`:

`public interface ISessionProvider`

This class encapsulates everything needed for session management (client & server). The application needs to provide the .NET Connector with an object of this type, which then creates (or returns an existing) session ID, whenever the .NET Connector needs one. Session IDs are required for stateful RFC communication in both, client and server scenarios.

The `ISessionProvider` implementation needs to be registered with the central class for session management:

`public class RfcSessionManager`

Applications that have no need for a sophisticated user session management can simply use the following default implementation of `ISessionProvider`. It is a simple implementation using the current thread's thread ID as session ID, so basically everything running in one thread is considered as belonging to one user session. This implementation should be sufficient for the majority of standalone applications. If you don't register your own session provider object, the default implementation will automatically be used by NCo, making stateful RFC very simple (as shown in step 4 of the step-by-step client example above).

`public class RfcSessionProvider:ISessionProvider`

The two methods `ExampleSimpleStatefulCalls()` and `ExampleMultiThreadedStatefulCallsWithDefaultSessionProvider()` in `StepByStepClient.cs` illustrate, how easy it is to achieve stateful calls (i.e. calls which keep the ABAP user session and thus all local ABAP memory on SAP system side) in this case.

However, applications that have more advanced requirements as to session/user management (e.g. because user sessions span across several threads) can implement the above interface in order to provide a "bridge" between their own session management (for example in application server environments like a web server that keeps user sessions with logged in browsers via Cookies) and the ABAP backend's user session management.

During startup of your application, you then need to pass an instance of your `ISessionProvider` implementation to `RfcSessionManager.RegisterSessionProvider()`.

The method `ExampleMultiThreadedStatefulCallsWithCustomSessionProvider()` together with the `ExampleSessionProvider` shows a very basic example of how to do this. It just creates one user session and then executes several stateful calls from different threads as part of the same user session.

Executing Client Side tRFCs, qRFCs and bgRFCs

This section describes how an RFC client application can send tRFC, qRFC or bgRFC units (LUWs) into an SAP system. First, we need a couple of helper classes for managing the necessary unique identifiers required for all three types of communication:

```
public class RfcTID
public class RfcUnitID
public enum RfcUnitType {TRANSACTIONAL, QUEUED}
```

The following class is a special kind of data container that stores the `RfcFunction` objects comprising one logical unit of work (LUW). When you commit the `RfcTransaction`, all contained `RfcFunctions` will be executed by the SAP backend as an atomic unit in the order in which they were added to the `RfcTransaction` object. If you send the transaction via qRFC instead of tRFCs, then the backend first puts the LUW into an inbound queue, thus guaranteeing the relative execution order of several LUWs that were sent to the same queue.

```
public class RfcTransaction
```

Here is the corresponding class for bgRFC communication:

```
public class RfcBackgroundUnit
```

In the bgRFC case, `RfcUnitAttributes` can be used to modify unit behavior in the backend system or to provide additional detail information about the sender:

```
public struct RfcUnitAttributes
```

The methods `ExampleTrfcClient()` and `ExampleBgrfcClient()` plus their related helper methods and helper classes illustrate a quite complex example of how to achieve end-to-end transactional security. For understanding the logic of tRFC/qRFC/bgRFC calls, you don’t need to understand, how the `TidStore` class works internally. Just consider it as a “poor man’s database”, that was added in order to be able to show a fully functional command line example. However, in your own applications you should let a different class based on a real database take over that task.

RFC Server Programming

The process of receiving an RFC request from an SAP backend system and processing that request looks as follows:

- Provide the necessary gateway connection parameters and obtain an `RfcServer` object corresponding to the backend from which you want to receive calls.
- Implement the business logic for the function module(s) you intend to process and register these implementations as C# delegates of type `RfcServerFunction` with the `RfcServer` object.
- Either create an `RfcCustomRepository` containing the metadata for the function modules you want to process, or configure an RfcDestination (as described in chapter *RFC Client Programming*), whose repository you can use. Register this repository with the `RfcServer`.
- Optionally implement an `IServerSecurityHandler` and register it with the `RfcServer`, if you want to protect your function implementations from unauthorized access by any SAP system user.
- Optionally implement an `ISessionProvider` and register it with the `RfcSessionManager` as described in section *Code Sample: Session Management*, if you want to process stateful RFC calls.
- Optionally implement an `ITransactionIDHandler` and register it with the `RfcServer`, if you want to process tRFC/qRFC calls.
- Optionally implement an `IUnitIDHandler` and register it with the `RfcServer`, if you want to process bgRFC calls.
- Optionally register event handlers for server state changes or server errors, if you are interested in these events for monitoring or tracing reasons.
- Finally start the `RfcServer`.

The necessary APIs for these steps are now described in detail.

Server Management

The following class manages the connection parameters for all servers and allows the application runtime to shutdown all running servers at once.

In analogy to destination management (section *Destination Management*) there are three mutually exclusive ways for an application to provide the connection parameters needed for starting an RFC server:

1. Provide the parameters during runtime (e.g. if you get them from a UI).
2. Define the parameters in the standard .NET configuration file of the executable.
3. Implement and provide a configuration object, which the `RfcServerManager` can “ask”, whenever it needs logon parameters for a particular system. This configuration object can then do whatever it wants (e.g. lookup parameters in a central LDAP system or a database), as long as it comes up with the required parameters.

The central class for server management is:

`public class RfcServerManager`

Applications that want to use the “configuration approach” need to implement the following interface and then set an instance on the `RfcDestinationManager` during application startup:

`public interface IServerConfiguration`

Alternatively, they can use the App.config file and provide AssemblyName and TypeName of the implementation and the .NET Connector will try to load an instance of the given class from the Global Assembly Cache. This object then provides access to the application’s central RFC server configuration.

A list of the configuration parameters necessary for starting an RFC server is available in *Appendix A*: List of Possible Logon and Configuration Parameters.

One interesting feature to mention here probably is: if your server configuration object says it supports configuration change events, then the .NET connector will register a delegate of type `ConfigurationChangeHandler` on your object in order to receive notifications for this event. Then whenever your application wants to change one or more connection parameters for a given `RfcServer` (e.g. using a different hostname or changing the number of parallel listening connections), it needs to trigger this event, and the .NET Connector will on the fly change the corresponding `RfcServer` object correspondingly.

Processing Incoming RFC Calls

The `RfcServer` class finally is the class receiving and processing the incoming RFC requests:

```
public class RfcServer
```

In addition to the configuration object, the `RfcServer` class provides read-only Properties for a set of most important parameters.

For processing an incoming function call, you need to implement a class (of arbitrary type) and add a method of same type as the delegate `RfcServerFunction` to this class.

```
public delegate void RfcServerFunction(
    RfcServerContext ctx, IRfcFunction function)
```

Annotate the method with an attribute of type `RfcServerFunctionAttribute`.

```
public class RfcFunctionAttribute :
                          System.Attribute
```

There are two ways to set this attribute:

- `[RfcServerFunction(Name = "STFC_CONNECTION")]`
  In this case NCo executes this server function, whenever a request for function module STFC_CONNECTION comes in. Or more general, this allows providing a server function for one particular function module.
- `[RfcServerFunction(Default = true)]`
  In this case NCo executes this server function, whenever a request for a function module, which does not have an explicit server function bound to its name, comes in.

It is possible to mix both variants. So you could have three server functions explicitly designed to handle function modules A, B and C, and a fourth “default” server function, which handles all other incoming function module requests. Also note that NCo already contains nine server functions to handle the following system level function modules:

- RFC_PING, RFC_SYSTEM_INFO, RFC_DOCU – used by SM59
- ARFC_DEST_SHIP, ARFC_DEST_CONFIRM, API_CLEAR_TID – used by the backend for transmitting tRFC/qRFC LUWs.

- BGRFC_DEST_SHIP, BGRFC_DEST_CONFIRM, BGRFC_CHECK_UNIT_STATE_SERVER – used by the backend for transmitting bgRFC LUWs.

If a function module request comes in, for whose name no server function has been registered, and if no default server function is present either, then NCo sends a corresponding error message back to the SAP system.

Server functions can be static methods or instance methods. In most cases it does not matter, which one you choose, but in general static methods are used for stateless RFC servers, meaning the function module does not need to preserve state information between different invocations. If you use instance methods, the NCo runtime creates a new object instance of the surrounding class for each user session on ABAP system side and keeps it as long as that user session (and thus the RFC connection between that user session and our .NET program) is alive. That way the .NET application can use the instance variables of that object for keeping state between several calls of the function module. You can even keep and use state information between calls of different function modules, if the implementing methods for these function modules are provided by the same object. However, if you don't like to have that many objects getting created, you can just as well implement your function modules as a static method and keep any state information in your session related storage.

Server functions can be spread across several classes. This can be useful again in stateful scenarios, where you want to "group" several sets of function modules into separate business objects. For example, you could have one class named `SalesOrder`, which keeps all state information for a sales order and provides three server function implementations for the function modules BAPI_SALESORDER_CREATEFROMDAT2, BAPI_SALESORDER_CHANGE and BAPI_SALESORDER_GETSTATUS, all acting on the member variables of the `SalesOrder` object, and a second class named `Customer`, whose internal variables keep all information necessary to simulate a customer and which offers three server functions for BAPI_CUSTOMER_CREATE, BAPI_CUSTOMER_DELETE and BAPI_CUSTOMER_EDIT.

Once you are finished implementing all server functions, you hand them over to the `RfcServer` when creating it:

```
Type[] handlers = new Type[2] { typeof(SalesOrder), typeof(Customer) };
RfcServer server = RfcServerManager.GetServer("PRD_REG_SERVER", handlers);
```

"PRD_REG_SERVER" must be the name of a server configuration present in either the App.config file or in the registered `IServerConfiguration`, as described in section *Server Management*.

Adding a Repository to the Server

Once you have your `RfcServer` object, you need to tell it, where it can find the necessary function module descriptions. The easiest way is to just add the configuration parameter `RepositoryDestination` to the configuration of your server. Then you need to do nothing here. (Of course the referenced destination needs to exist, and its DDIC needs to contain the function modules used by your server.)

Alternatively, you can assign the repository of some appropriate backend system to your server:

```
server.Repository =
                RfcDestinationManager.GetDestination("PRD_000").Repository;
```

And finally, if you don't want to keep user information about some backend system in your server program, or if you want to provide function modules, that do not exist in the backend system, you can implement hard-coded metadata using the class `RfcCustomRepository` as described in section *Working with Hard-Coded Metadata*.

The sample program `StepByStepServer.cs` contains an example of a stateless server: see the method `ExampleSimpleServer()` and its helper methods/classes, like the function module implementation `StfcConnectionStaticImpl`, which uses a static method as server function.

The method `ExampleGenericServer()` shows how to add a "default server function" to the server, which handles all function modules different from STFC_CONNECTION.

Protecting the Server from Unauthorized Access

If you want to perform access checks in order to protect the server's functionality from access by unauthorized users, there are basically two levels on which this can be done:

- "Logon Check". – When an ABAP work process/user session establishes a connection to your RFC server, you can check, whether you want to allow access to your server in general for that user from that SAP backend system.
- "Authorization Check". – When a request for a particular function module comes in, you can check, whether you want to allow the given user to execute this particular function module.

In a stateless server, you always have exactly one logon check directly followed by one authority check, since there is always only one function request running in one user session. In a stateful server, however, there may be multiple function requests being executed inside the same user session, so you'll get one logon check and multiple authorization checks, one for each function module being called by the user.

In order to make use of this functionality, you need to implement the following interface:

`public interface IServerSecurityHandler`

and then register an instance of your class with your `RfcServer`, before starting it:

```
public class MySecurityHandler : IServerSecurityHandler{
   ...
}
server.SecurityHandler = new MySecurityHandler();
```

If such a security handler is registered with your `RfcServer` object, the NCo runtime will call the method `LogonCheck()`, whenever a SAP system opens a fresh connection to the `RfcServer`, and the method `AuthorizationCheck()`, whenever a new function module is about to be executed. If no security handler is installed, access is always allowed.

In both methods, a lot of information is available about the incoming call, and you can perform all kinds of checks. A common scenario is for example:

- In the logon check you verify the backend's system ID (`RfcServerContext.SystemAttributes.SystemID`) ("only the PROD system is allowed to use my server, but not DEV and QA"), you check whether the user ID (`RfcSecurityData.UserName`) is in the list of users allowed to use this server, and you may check that the connection is secured with SNC (`RfcSecurityData.Type`) and the `SncPartnerKey` is from a system you trust.

  If the corresponding destination in SM59 has the flag "Send SAP Logon Ticket" is activated, you can even verify the user's identity using the SAPSSOEXT library: http://help.sap.com/saphelp_nw73/helpdata/en/12/9f244183bb8639e10000000a1550b0/frameset.htm. ~~By combining SNC with the validation of the logon ticket/SSO2 ticket/assertion~~ ticket, you can secure your server against malicious attacks forging the identity of a backend system and/or user.

- In the authorization check, you simply take the current function module name and user name and check whether the given user is authorized to execute the given function module.

If you don't need to assign different levels of authorization for different function modules, you can also just perform the logon check and then always return true from the authorization check. A convenient alternative to checking the backend system yourself in the logon check, is to provide the server configuration parameter `SystemIDs` and give a list of allowed system IDs here.

Processing Stateful RFC Calls

Most of the time, stateful RFC servers are not necessary. But if you want to keep user state in your server program across several function calls, there are two basic ways to do it.

Your .NET program does not have its own user session concept

In this case it is probably best to use the instance method approach and use the function handler object for storing state information for as long as the backend user session is still alive. The .NET Connector runtime will handle the necessary session management for you.

Let's look at a little example here. Assume this is your server function implementation for the function module STFC_CONNECTION:

```
public class MyServerHandler{
   private int callCount = 0;

   [RfcServerFunction(Name = "STFC_CONNECTION", Default = false)]
   public void StfcConnection(RfcServerContext context,
                IRfcFunction function){
      Console.WriteLine("Received function call {0} from system {1}.",
            function.Metadata.Name,
            context.SystemAttributes.SystemID);

      String reqtext = function.GetString("REQUTEXT");
      Console.WriteLine("REQUTEXT = {0}\n", reqtext);

      ++callCount;

      function.SetValue("ECHOTEXT", reqtext);
      function.SetValue("RESPTEXT", "You have called this function "
         + callCount.ToString() + " times so far");

      if (!context.Stateful)
          context.SetStateful(true);
   }
}
```

The important point here is the line

```
    context.SetStateful(true);
```

This statement tells NCo to keep the connection open until

- either you set it to false in one of the following incoming function module invocations
- or the ABAP side closes the connection. The SAP system closes the connection, if one of the following three conditions is fulfilled:
  - The ABAP code explicitly closes the connection by using the (local) function module RFC_CLOSE_CONNECTION, passing the name of the corresponding SM59 destination as input.
  - The internal mode ends (e.g. the user enters “/n” into the OK code field).
  - There is a work process or gateway timeout.

This is already everything that needs to be done in order to achieve a simple stateful server. Sometimes it is useful to make the connection stateful, even if you don’t want to keep user state in your server: it can improve the performance, if thousands of RFC calls are send to the server in a loop within the same ABAP internal mode. For example, when the server is receiving 100.000 IDocs that have been kicked off by a batch job.

Please complete the program above (by adding necessary configuration parameters and a `Main()` method that creates and starts the server), and then test it from SE37. Execute STFC_CONNECTION against the corresponding RFC destination, and you will notice how the counter keeps increasing. Then in your SAPGui session execute a “/n” in the OK-code field. This ends your current ABAP user session and starts a new one. When you now execute the function module again, you will notice that the counter starts at 1 again, so a new instance of the `MyServerHandler` class has been created on .NET side!

A simple test of how NCo handles several parallel backend user sessions is the following: start a second SAPGui session (for instance by executing “/ose37” in the ok-code field) and then keep calling STFC_CONNECTION first a couple of times from one SAPGui session, then from the other one. You will notice how two different counters are getting incremented, depending on which SAPGui session is being used. This means that the NCo runtime has created two different instances of the `MyServerHandler` class and is using one for each of the two SAPGui sessions.

The sample program `StepByStepServer.cs` contains another fully functional example of this type of simple stateful server: see method `ExampleStatefulServer()` and the server function implementation class `StfcConnectionImpl`.

Your .NET program has its own user session concept

In this case you need a “bridge” between the ABAP system’s user session management and the user session management of your .NET program. Whenever an ABAP backend user “logs in” to your server application, a new user session of your .NET application needs to be created and associated with the ABAP backend user session. In order to achieve this, you need to implement the interface `ISessionProvider` and register an object of that type with the `RfcSessionManager`.

This is similar to what we did in section *Code Sample: Session Management* for the advanced stateful RFC client scenario. The difference is only that now you need to implement the remaining methods of the `ISessionProvider` interface. The .NET Connector runtime will then interact with the session management of your application in the following way:

`public String CreateSession()`

Whenever the ABAP backend system opens a new connection to the external server program, the NCo runtime calls this method and requests your application's session management to create a new user session and attach the current thread to that new session.

Note:

This is partly related to `IServerSecurityHandler`. However, we decided to keep the two notions of "session management" on one side and "user logon check" on the other side separated from each other, so that those applications that are not interested in session management and stateful communication, can still protect their server application by logon and user validation mechanisms. See the interface `IServerSecurityHandler` for more details.

`public void PassivateSession(String sessionID)`

The .NET Connector calls this method, whenever an RFC request, which is processed as part of a stateful server connection, has been finished and the corresponding stateful server connection is now idle (waiting for the next RFC request from the backend). The ISessionProvider should detach the given user session from the current thread.

`public void ActivateSession(String sessionID)`

The .NET Connector calls this method, whenever a new RFC request comes in over an already existing stateful server connection. The ISessionProvider should attach the given user session to the current thread.

`public void DestroySession(String sessionID)`

The .NET Connector calls this method, whenever the ABAP backend system (or the implementation of a server function) closes a stateful server connection. The application's session management framework should now delete the corresponding user context.

`public bool IsAlive(String sessionID)`

Allows the .NET Connector to test, whether a particular application session is still alive.

Once you have implemented an object like this, all you need to do to make your RFC servers interact with your application's session management framework is to add a line like

```
RfcSessionManager.RegisterSessionProvider(new MySessionProvider();
```

to the startup code in the Main() method of the previous example.

Processing Incoming tRFCs, qRFCs and bgRFCs

In order to be able to process tRFC/qRFC and bgRFC LUWs and to guarantee transactional security (exactly once execution), your server needs to fulfill certain requirements. In particular, it needs to have an object of type `ITransactionIDHandler` (for tRFC/qRFC) and of type `IUnitIDHandler` (for bgRFC). These objects should store each TID/UnitID together with its current processing status in a database (or at least in some fail-proof file based status-keeping component).

The first tasks for implementing a tRFC server are the same as for every RFC server: you need to implement the necessary handler function(s) for the function module(s) contained in the tRFC LUWs that you want to receive. Please note that one tRFC LUW may contain several function module calls. These are then supposed to be processed one after the other, and they are considered as one single atomic unit, i.e. they should be committed all at once, or not at all. (However, in 99% of the cases a tRFC LUW contains only one single function call.)

Next you need to implement the interface `ITransactionIDHandler` and install an instance of it in your `RfcServer`:

`public interface ITransactionIDHandler`

For processing incoming bgRFCs, you need to implement this interface and install an instance of it in your `RfcServer`:

`public interface IUnitIDHandler`

But this is pretty much identical to the tRFC/qRFC case, except that it uses 32-digit GUIDs instead of 24-digit GUIDs and except that it offers a few additional features.

This transaction ID handler class will interact with your status management component and that way guarantee transactional security (execution "exactly once") of the LUWs your program will receive. In the four methods of that class you need to do the following:

- `CheckTransactionID`

  A new tRFC/qRFC LUW arrived from the backend and we need to check, whether the corresponding TID is already known on our server. The Check-function should now search the status database for the given TID. This search can lead to the following three results:

  - The connection to the database is currently down, or some other internal error happens. In this case just throw an `RfcInternalError`. The .NET Connector will then send a SYSTEM_FAILURE back to the ABAP system, and the ABAP system can retry the same transaction sometime later.
  - The TID does not yet exist in the TID database, or it exists and has the status Created or Rolled back. In this case, the Check-function should create an entry for this TID (if not already existent), set the Status to Created and return "true". The .NET Connector will then proceed executing the function modules contained in the tRFC/qRFC LUW, i.e. it will call the function handlers for the corresponding function module names.
  - The TID already exists in the TID database and has the status Committed. The Check-function should simply return "false" in that case. The .NET Connector will then return an OK message to the ABAP system without executing the LUW a second time. The ABAP system then knows that the transaction has already been processed successfully and can continue with the ConfirmTID step.
- `Commit`

  After all function modules contained in the current LUW have been executed successfully, the .NET Connector calls this method. It should persist all changes done for this TID in the previous function module handler functions. If this is successful, it should set the status to Committed.
- `Rollback`

  If one of the function modules contained in the current LUW ended with an exception, the .NET Connector calls this API. It should rollback all changes done for this TID in the previous function modules. Afterwards, it should set the status to Rolled back.
- `ConfirmTransactionID`
  When the backend system finds out that the LUW corresponding to this TID has been executed completely and successfully on the RFC server side, it will trigger a ConfirmTID event. When receiving this event, the .NET Connector will call this API. It should simply delete the entry for this TID from the database.

Once you have implemented a class that does all the necessary work (e.g. `MyTidHandler`), you enable your RFC server for tRFC/qRFC processing by simply adding a line like

```
server.TransactionIDHandler = new MyTidHandler();
```

before the `server.Start()`.

The sample program `StepByStepServer.cs` contains two fully functional examples, one for a tRFC server, and a similar one for a bgRFC server. See the two methods `ExampleTRfcServer()` and `ExampleBgRfcServer()` including their helper methods and necessary function module implementations, and see the two classes `MyUnitIDHandler` and `MyTidHandler`. You don't need to understand the class `TidStore` in order to understand tRFC/bgRFC processing. You can view it as a black box that simulates a poor man's database.

Monitoring the State of the Server

While your RFC server is running, you can get information about the state of your server and about possible problems. For this purpose `RfcServer` provides three events, on which you can register your event handler, if you are interested in this information.

public event RfcServerErrorEventHandler RfcServerError;

This event is triggered, whenever NCo encounters a problem in lower level runtime components. Example for this are: the network connection between SAP System and your RFC server got broken; a user tried to logon or invoke a certain function module, but an `IServerSecurityHandler` denied access, you are processing tRFCs, but your `ITransactionIDHandler` was not able to process TIDs, e.g. because its database is currently down, etc.

public event RfcServerErrorEventHandler RfcServerApplicationError;

This event is triggered, whenever one of your server function implementations (the "application logic") throws an Exception. This can be either a normal ABAP Exception or a "serious" Exception that results in a SYSTEM_FAILURE raised to the backend.

For both of the above events you need to implement a delegate of type

```
public delegate void RfcServerErrorEventHandler(Object server,
                          RfcServerErrorEventArgs errorEventData);
```

The server object can be cast to `RfcServer` and gives you information about which of your running servers encountered the problem (in case there are several of them). The event arguments give you the original Exception (e.g. an `RfcServerAuthorizationException` in case a user logon attempt was denied by your `IServerSecurityHandler`, or an `RfcCommunicationException` in case the connection to the backend was destroyed by network problems, etc) as well as the current server context, so you can read information about the current backend user, the name of the calling ABAP program, the name of the function module the user tried to invoke, etc.

The `StepByStepServer.cs` example provides a simple implementation of these two event handles: `OnRfcServerError()` and `OnRfcServerApplicationError()`.

The third event notifies you about changes in the server's runtime state.

`public event RfcServerStateChangedEventHandler RfcServerStateChanged;`

If you register a delegate of type

```
public delegate void RfcServerStateChangedEventHandler(Object server,
                     RfcServerStateChangedEventArgs stateEventData);
```

the server object can again be cast to `RfcServer` and gives you a reference to the server, whose state changed, while the event arguments will give you the server's old state and new state in form of an `RfcServerState` enum. `RfcServerState` can take these values:

`public enum RfcServerState` `{STARTING, RUNNING, BROKEN, STOPPING, STOPPED}`

These state changes show you, when servers get started or stopped. As long as at least one registration of the server is still alive, the state will remain in `RUNNING`. If the last gateway registration breaks down (e.g. because of network problems or because the backend is being shutdown), the server will go into the `BROKEN` state. It will automatically try to reestablish the connection at periodic intervals, and once the backend system (or the network) is back up, the state will go into `RUNNING` again.

Exception Handling

Base class for all our exceptions:

`public abstract class RfcBaseException:ApplicationException`

The following exception is thrown, whenever you try an operation that is not supported at the current point of time. For example when trying to re-initialize a configuration after it has already been initialized.

`public class RfcInvalidStateException:RfcBaseException`

The next class collects everything that can go wrong on a technical layer, like OutOfMemory, LZ- or GZIP-decompression error, xRFC- or basXML-deserialization error, type conversion error, codepage conversion error. If the problem has been caused by a .NET runtime exception, the original exception will be passed as nested exception:

`public class RfcSerializationException:RfcBaseException`

If you try to get/set a non-existing FM parameter or structure/table field, you will get this one:

`public class RfcInvalidParameterException:RfcBaseException`

The next exception indicates that your external RFC client program tries to invoke a function module on an external server program. Extern–extern communication is not supported by the .NET Connector.

```
public class
RfcUnsupportedPartnerException:RfcBaseException
```

The following corresponds to RFC_COMMUNICATION_FAILURE (problems on network level) and is thrown by all APIs that communicate with a backend system.

```
public class
RfcCommunicationException:RfcBaseException
```

Corresponds to RFC_SYSTEM_FAILURE (technical problem in the backend system or in the external server program):

```
public class RfcSystemException:RfcBaseException
```

For some reason the backend system (or external server program) refused to login the current user (Invalid user credentials, unsupported logon language, wrong SNC or SSO2 settings, etc.):

```
public class RfcLogonException:RfcBaseException
```

Base class for all problems happening inside ABAP function modules (or inside C# implementations of function modules) or inside the backend’s ABAP runtime:

```
public abstract class
RfcAbapBaseException:RfcBaseException
```

The following class is the base class for all “classic” ways to throw an error in ABAP, i.e. the “non-class-based” exceptions and messages.

```
public abstract class
RfcAbapClassicException:RfcAbapBaseException
```

For more details see also RfcAbapClassicException:

```
public class
RfcAbapException:RfcAbapClassicException
```

The next class represents a classic ABAP Message thrown via the ABAP statement **MESSAGE …**. For more details see also RfcAbapClassicException.

```
public class
RfcAbapMessageException:RfcAbapClassicException
```

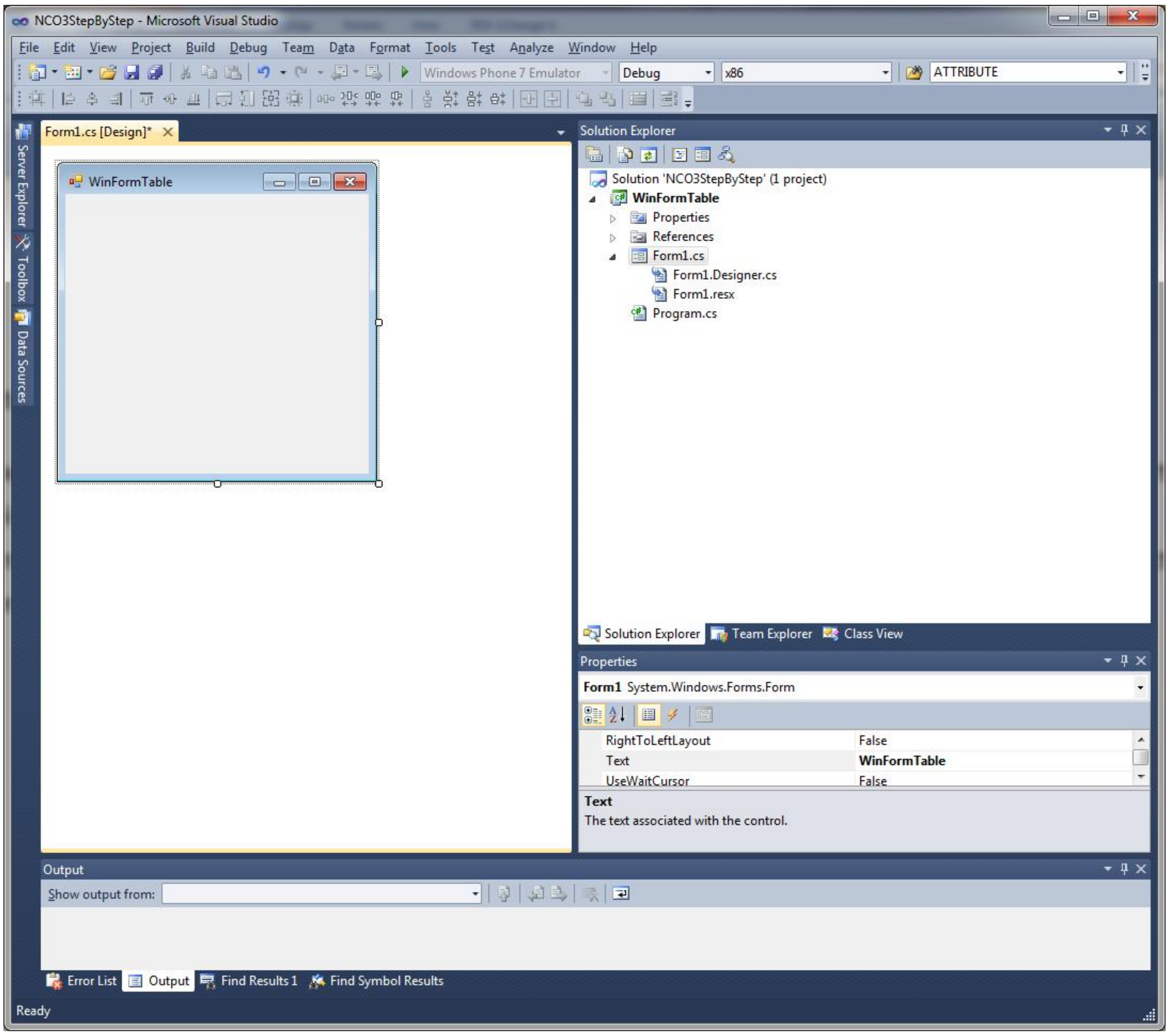
The final class now represents an ABAP class-based exception, which can be used starting with SAP_BASIS release 7.20:

`public class RfcAbapClassException:RfcAbapBaseException`

Using Data Binding in Windows Forms

Beginning with release 3.0.3, the SAP .NET Connector 3.0 (or NCo 3 for short) supports data binding with RfcTable in both, Windows Forms and ASP.NET web applications. This example gives a step by step introduction to how to bind a DataGridView control on a Windows Form to an NCo 3 RfcTable object.

1. Create a Windows Forms application project named WinFormTable in Visual studio

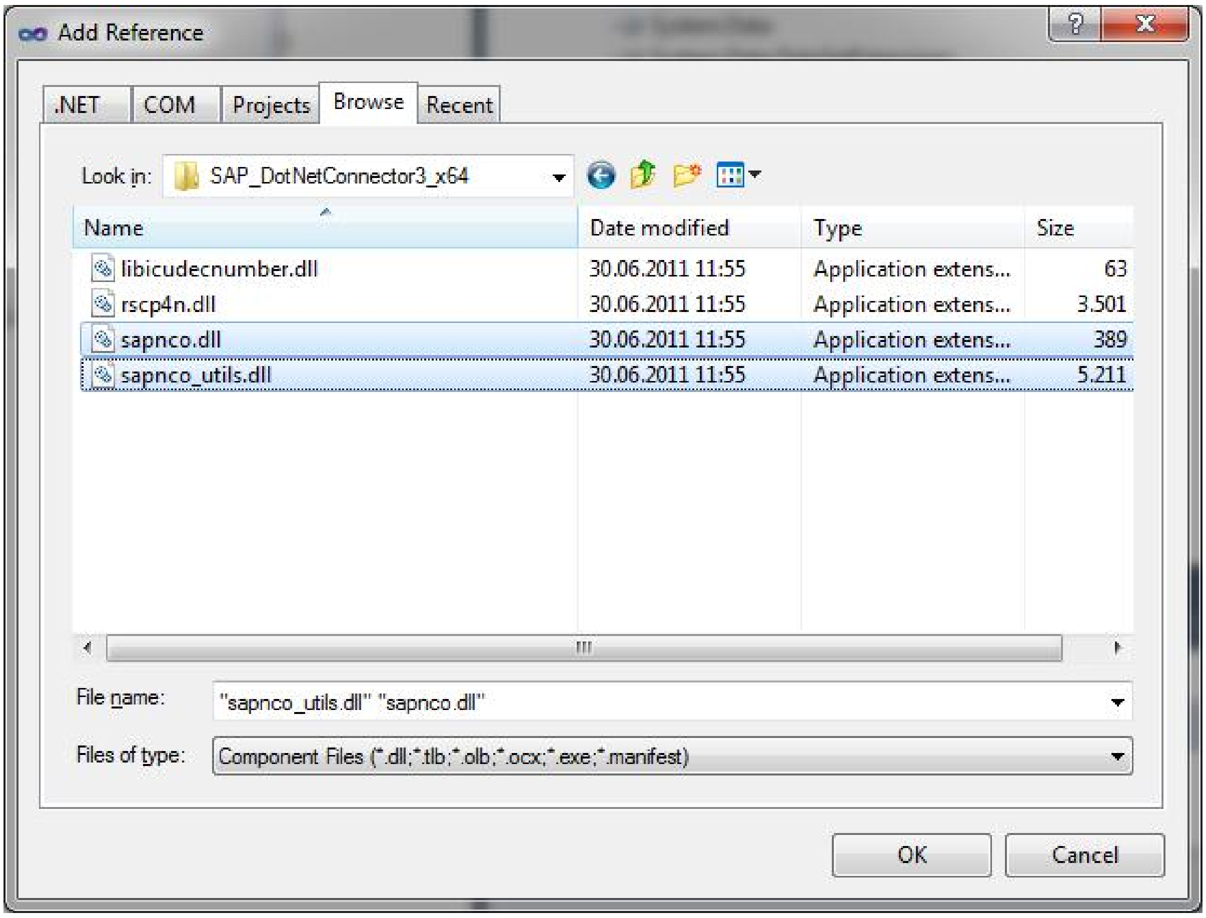


Form1 System.Windows.Forms.Form

RightToLeftLayout	False
Text	WinFormTable
UseWaitCursor	False

Text
The text associated with the control.

2. Add references to the two NCo 3 assemblies (sapnco.dll and sapnco_utils.dll located in the installation folder) to the project. Additionally, you need to manually copy the dependent unmanaged DLLs rscp4n.dll and libicudecnumber.dll to the Output folder of the project.



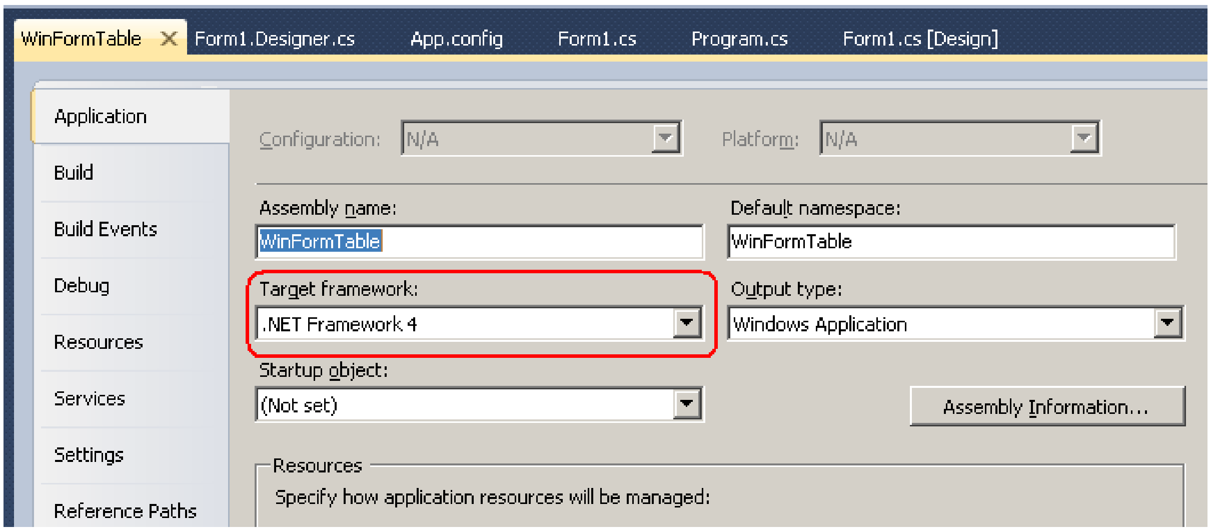
Look in: SAP_DotNetConnector3_x64

Name	Date modified	Type	Size
libicudecnumber.dll	30.06.2011 11:55	Application extens...	63
rscp4n.dll	30.06.2011 11:55	Application extens...	3.501
sapnco.dll	30.06.2011 11:55	Application extens...	389
sapnco_utils.dll	30.06.2011 11:55	Application extens...	5.211

File name: "sapnco_utils.dll" "sapnco.dll"

Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

3. In the Application Properties of the project, change the “Target framework” from ".NET Framework 4 Client Profile" to ".NET Framework 4" if necessary.



Configuration: N/A

Platform: N/A

Assembly name: WinFormTable

Default namespace: WinFormTable

Target framework: .NET Framework 4

Output type: Windows Application

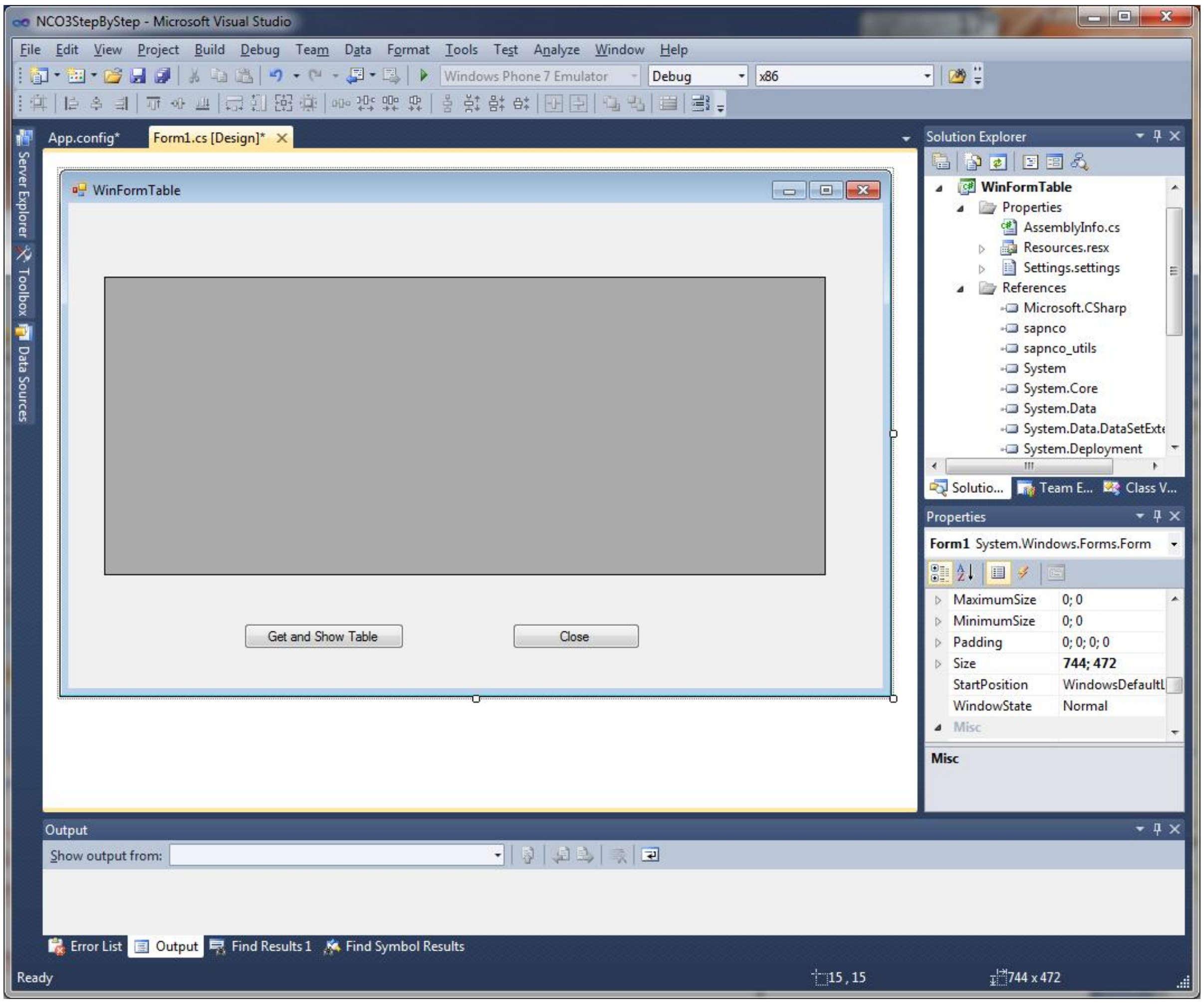
Startup object: (Not set)

Specify how application resources will be managed:

4. Add an application configuration file app.config to the project and populate it with the flowing settings, for example:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="SAP.Middleware.Connector">
      <sectionGroup name="ClientSettings">
              <section name="DestinationConfiguration"
              type="SAP.Middleware.Connector.RfcDestinationConfigurat
              ion, sapnco"/>
      </sectionGroup>
    </sectionGroup>
  </configSections>
  <SAP.Middleware.Connector>
    <ClientSettings>
      <DestinationConfiguration>
        <destinations>
                <add NAME="NCO_TESTS" USER="MYUSER"
       PASSWD="1234567" CLIENT="100" LANG="EN" ASHOST="hostname"
       SYSNR="00" POOL_SIZE="5" MAX_POOL_SIZE="10"/>
        </destinations>
      </DestinationConfiguration>
    </ClientSettings>
```

4. Add a DataGridView control and two buttons to the Windows form:



MaximumSize	0; 0
MinimumSize	0; 0
Padding	0; 0; 0; 0
Size	744; 472
StartPosition	WindowsDefaultL
WindowState	Normal

5. Double click the “Get and Show Table” and “Close” buttons to add OnClick event handler for each of the buttons:

```
using SAP.Middleware.Connector;

namespace WinFormTable
{
    public partial class Form1 : Form
    {
     public Form1()
        {
            InitializeComponent();
        }

        private void GetAndShow_Click(object sender, EventArgs e)
        {
         // TODO:  add code here to get a RfcTable and then bind it to the
      DataGridView control

        }

        private void Close_Click(object sender, EventArgs e)
        {
            // just call Close method of the base class
```

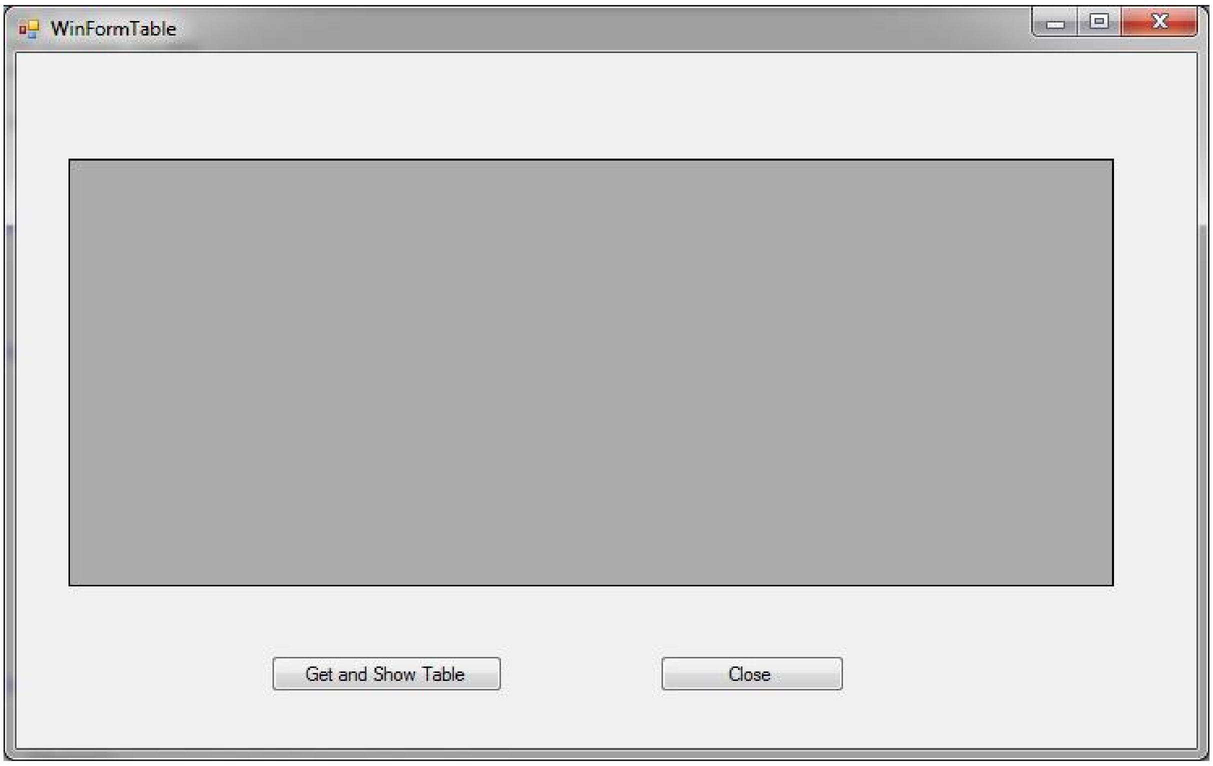
6. Implement the method `GetAndShow_Click in two steps:`
   6.1 Implement the static method GetTableByRfcCall to get an RfcTable instance by making an RFC call to the test function module STFC_STRUCTURE using destination “NCO_TESTS”

```
static IRfcTable GetTableByRfcCall(string destName, int rowCount)
{
    // get the destination
    RfcDestination dest = RfcDestinationManager.GetDestination(destName);
    // create a function object
    IRfcFunction func = dest.Repository.CreateFunction("STFC_STRUCTURE");

    //prepare input parameters
    IRfcStructure impStruct = func.GetStructure("IMPORTSTRUCT");
    impStruct.SetValue("RFCFLOAT", 12345.6789);
    impStruct.SetValue("RFCCHAR1", "A");
    impStruct.SetValue("RFCCHAR2", "AB");
    impStruct.SetValue("RFCCHAR4", "NCO3");
    impStruct.SetValue("RFCINT4", 12345);
    impStruct.SetValue("RFCHEX3", new byte[] { 0x41, 0x42, 0x43 });
    impStruct.SetValue("RFCDATE", DateTime.Today.ToString("yyyy-MM-dd"));
    impStruct.SetValue("RFCDATA1", "Hello World");

    // fill the table parameter
    IRfcTable rfcTable = func.GetTable("RFCTABLE");
    for (int i = 0; i < rowCount; i++)
    {
        // make a copy of impStruct
        IRfcStructure row = (IRfcStructure)impStruct.Clone();
        // make such changes to the fields of the cloned structure
        impStruct.SetValue("RFCFLOAT", 12345.6789 + i);
        row.SetValue("RFCINT1", i);
        row.SetValue("RFCINT2", i * 2);
        row.SetValue("RFCINT4", i * 4);
        impStruct.SetValue("RFCTIME", DateTime.Now.ToString("T"));
        row.SetValue("RFCDATA1", i.ToString() + row.GetString("RFCDATA1"));
        rfcTable.Append(row);
    }
    // submit the RFC call
    func.Invoke(dest);
    // Return the table. The backend has added one more line to it.
    rfcTable = func.GetTable("RFCTABLE");
    return rfcTable;
}
```

6.2 Inside GetAndShow_Click, call the method GetTableByRfcCall to get an RfcTable and then bind it to the DataGridView control. Please note that the interface IRfcTable doesn't support data binding and, therefore, cannot be directly bound to the data grid view control. What we need is to get an ISupportTableView interface out of IRfcTable and bind the property DefaultView as data source to the control.

```
private void GetAndShow_Click(object sender, EventArgs e)
{
    startover:
    try {
        // Call GetTableByRfcCall to get an Rfc Table
        IRfcTable table = GetTableByRfcCall("NCO_TESTS", 100);
        // Get the bindable view of the RFC table
        IRfcTableView view = (table as ISupportTableView).DefaultView;
        // bind the table view to the data grid control
        this.dataGridView1.DataSource = view;
    }
    catch (Exception err) {
        DialogResult result = MessageBox.Show(err.ToString(), "Error",
            MessageBoxButtons.AbortRetryIgnore, MessageBoxIcon.Error);

        switch (result) {
            case DialogResult.Abort:
                Environment.Exit(0);
                break; // Stupid compiler....
```

7. Build the project and then start the program.

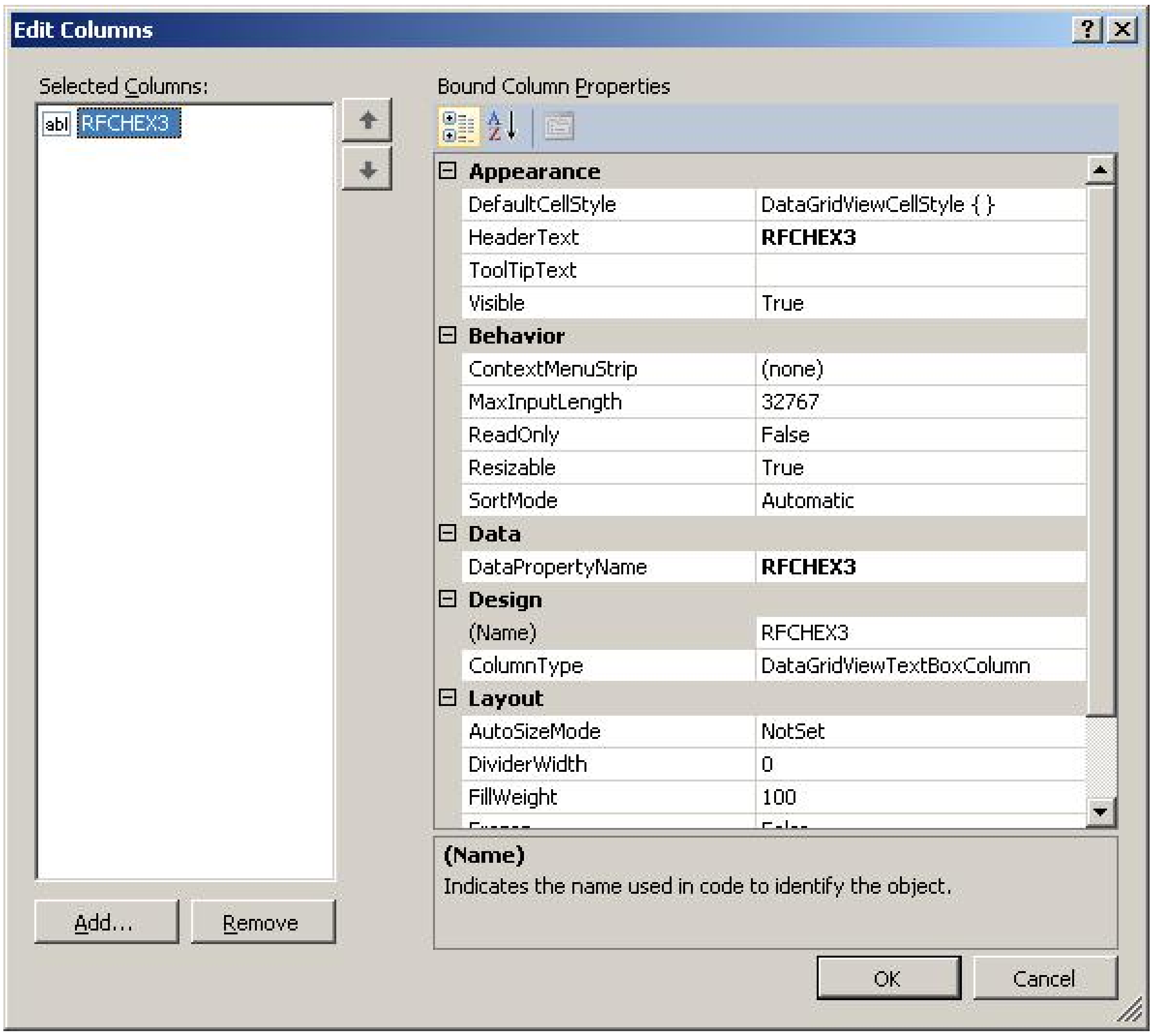


8. If you have made your DataGridView large enough, you will now get dozens of error message popups of type “The following exception occurred in the DataGridView: System.ArgumentException: Parameter is not valid.”

   This is, because there is one column in RFCTABLE, which is of type RAW. NCo 3 treats this as a byte[] and by default DataGridView tries to display byte arrays as jpegs, leading to the above error…

   If your tables don’t contain binary data, everything is ok, but in this case we need to do some extra work in order to process and display the GridView successfully. For this, open the Form1 Design, mark the DataGridView, so that its Properties get displayed, and then select the Columns collection at the end of the Properties. The Column Editor will pop up.

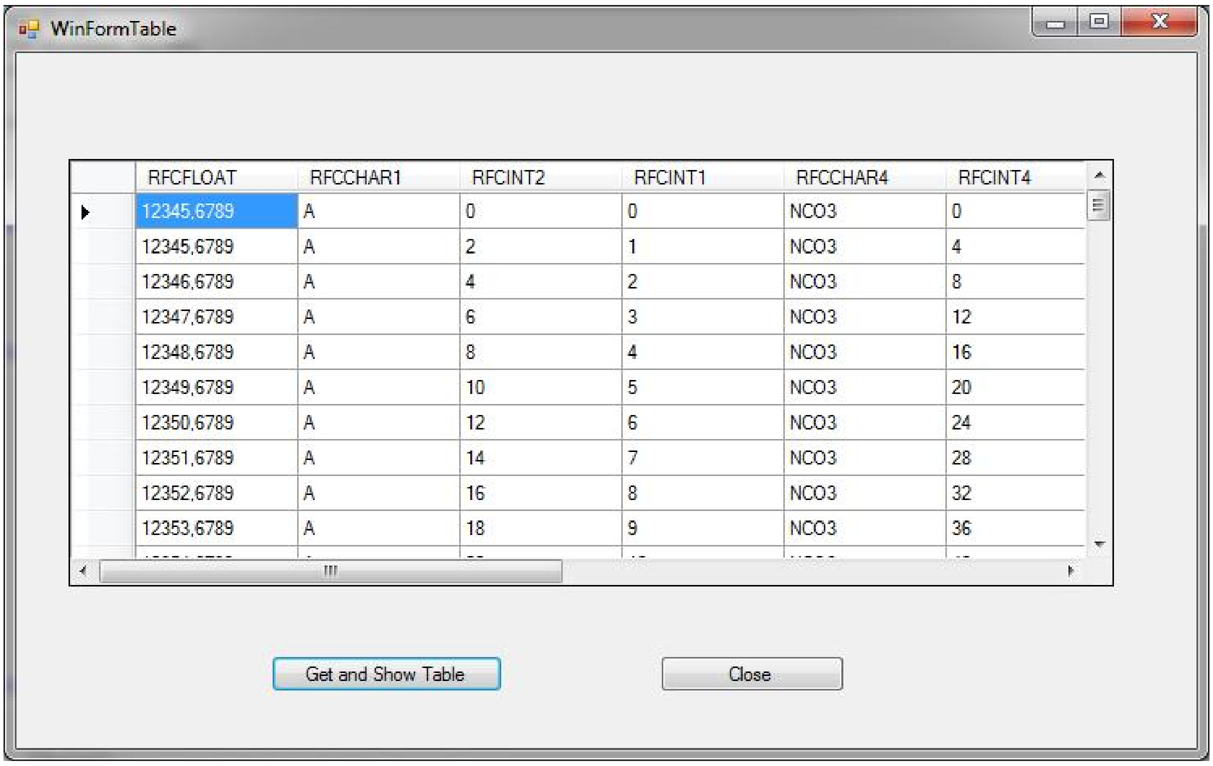
   Here add one Column with the name RFCHEX3 and make sure that the “DataPropertyName” is set to “RFCHEX3” as well and that the “ColumnType” is set to “DataGridViewTextBoxColumn” as in the following example:



Appearance	
DefaultCellStyle	DataGridViewCellStyle { }
HeaderText	**RFCHEX3**
ToolTipText	
Visible	True
Behavior	
ContextMenuStrip	(none)
MaxInputLength	32767
ReadOnly	False
Resizable	True
SortMode	Automatic
Data	
DataPropertyName	**RFCHEX3**
Design	
(Name)	RFCHEX3
ColumnType	DataGridViewTextBoxColumn
Layout	
AutoSizeMode	NotSet
DividerWidth	0
FillWeight	100

(Name)
Indicates the name used in code to identify the object.

9. Recompile the project, and now the data is finally displayed successfully without any exceptions. (The contents of the RFCHEX3 field are not displayed, but that’s as good as it gets…)



RFCFLOAT	RFCCHAR1	RFCINT2	RFCINT1	RFCCHAR4	RFCINT4
12345,6789	A	0	0	NCO3	0
12345,6789	A	2	1	NCO3	4
12346,6789	A	4	2	NCO3	8
12347,6789	A	6	3	NCO3	12
12348,6789	A	8	4	NCO3	16
12349,6789	A	10	5	NCO3	20
12350,6789	A	12	6	NCO3	24
12351,6789	A	14	7	NCO3	28
12352,6789	A	16	8	NCO3	32
12353,6789	A	18	9	NCO3	36

10. Sorting and filtering

    The interface IRfcTableView supports both sorting and filtering. Sorting works immediately after the table is bound to the grid view control. Clicking on a column header, the data in the grid view will be sorted by this column, ascending or descending.

    To make use of filtering feature, we need to do a little more:

```
10.1 Provide a filter handler of type RfcTableFilterHandler
```

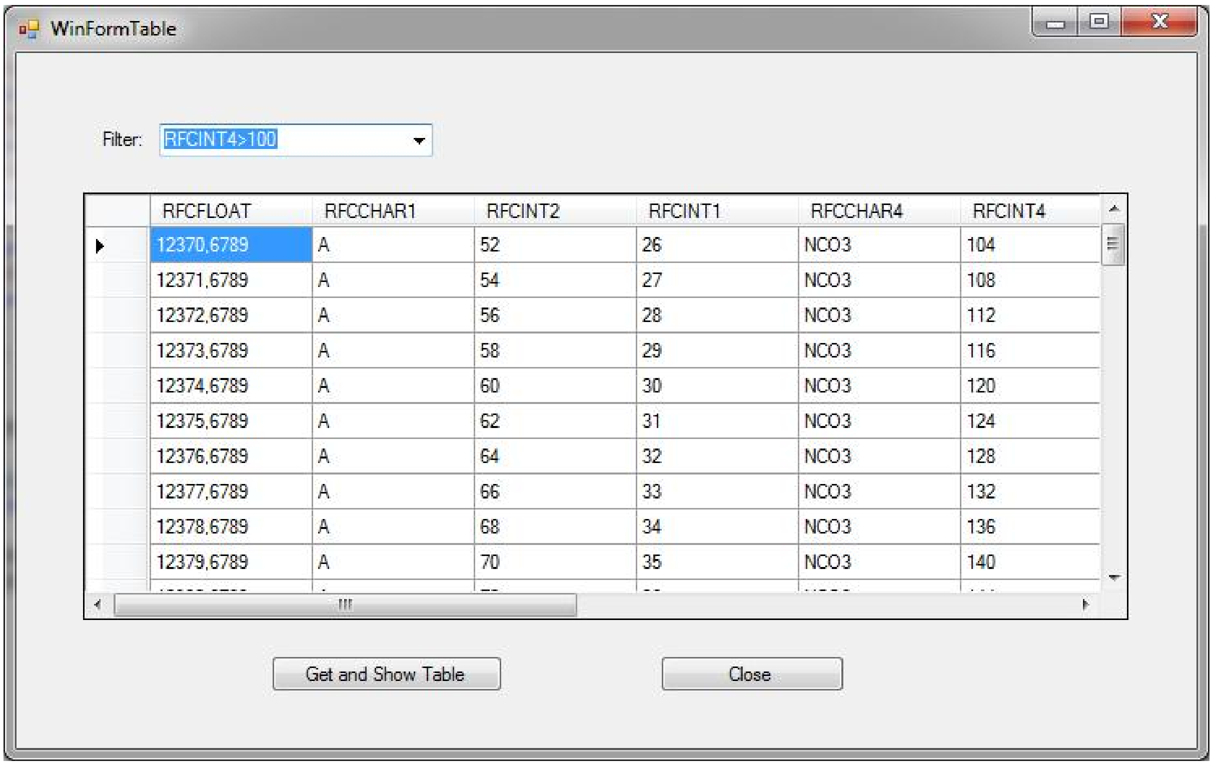
```
delegate bool RfcTableFilterHandler(string filter, IRfcStructure row);
```

```
    How to define the syntax of the filter string and how to implement
    the filter handler is totally up to you or your application’s
    requirements. To keep the thing simple, we implement a handler that
    just understands “RFCINT4>100” and “RFCINT2<50” as filtering
    criteria:
```

```
        private bool MyFilterHandler(string filter, IRfcStructure row)
        {
            // I am too lazy to parse the filter criteria, so just handle the "hard-
coded" filter string
            switch(filter)
            {
                case "RFCINT4>100":
                    return row.GetInt("RFCINT4") > 100;
                case "RFCINT2<50":
                    return row.GetInt("RFCINT2") < 50;
            }
```

10.2 Add a ComboBox (named “Filter”) to the Windows form to allow the user to select filter criteria, and add three items to the ComboBox: “*”, “RFCINT4>100” and “RFCINT2<50”. Set the “DropDownStyle” to “DropDownList”. Finally double-click on the ComboBox to add an event handler to it that reacts to the event of the selection getting changed:

```
        private void Filter_SelectedIndexChanged(object sender, EventArgs e)
        {
            IRfcTableView view = this.dataGridView1.DataSource as IRfcTableView;
            if (view != null)
            {
                view.FilterHandler = MyFilterHandler;
                view.Filter = this.Filter.Text;
            }
```



Filter: RFCINT4>100

RFCFLOAT	RFCCHAR1	RFCINT2	RFCINT1	RFCCHAR4	RFCINT4
12370,6789	A	52	26	NCO3	104
12371,6789	A	54	27	NCO3	108
12372,6789	A	56	28	NCO3	112
12373,6789	A	58	29	NCO3	116
12374,6789	A	60	30	NCO3	120
12375,6789	A	62	31	NCO3	124
12376,6789	A	64	32	NCO3	128
12377,6789	A	66	33	NCO3	132
12378,6789	A	68	34	NCO3	136
12379,6789	A	70	35	NCO3	140

Support for Extended Passport

Applications that need to evaluate or provide statistical data via “Extended Passport” (EPP) can do so by plugging in an IRfcClientPassportHandler. The .NET Connector will call the OnCallStart() method and hand the received passport over to the application, whenever an RFC request arrives from the backend system, and it will call the OnCallStart() method whenever it is about to send an RFC request into the backend system and needs a passport for sending it along with the request. After the processing OnCallEnd() will be invoked.

The interface for *EPP* support is:

`public interface IRfcClientPassportHandler`

.NET Connector 3.0 in High Availability Environments

The .NET Connector itself has no special needs for the setup in a HA environment. Actually, the main parts that need to be cared for of are part of the application server setup, in which the NCo is embedded. Basically, the following topics that need to be taken into consideration:

- Configuration data should be available uniquely for *all* instances. Make sure that your `IDestinationConfiguration` implementation is taking such a setup into account.
- In case a stateful scenario is executed, the containing session should be treated in a special way by the environment: a load balancer that is distributing requests should make sure that session stickiness is used at least from then on. This is necessary, because the sessions of NCo in the ABAP backend are bound to the physical connection. And such a connection cannot be moved from one machine to the other.

Appendix A: List of Possible Logon and Configuration Parameters

All parameters used in the .NET Connector are defined as constants in the class `RfcConfigParameters`. When providing parameters programmatically – for example when calling `RfcDestinationManager.GetDestination(RfcConfigParameters)` or when returning parameters from your own implementation of `IDestinationConfiguration` – you just use these defined constants. When providing parameters in the App.config file, you need to use the string values of these constants. In the following you find a list of all these parameters sorted by areas: parameters for configuring the "general behavior" of NCo, parameters used in Client programs, and parameters used in Server programs. Finally, we'll give a detailed description of the XML format of the App.config section corresponding to NCo configuration.

General Configuration Parameters

These parameters influence some aspects of NCo's general behavior.

Constant **(to be used in RfcConfigParameters)**	**String Value** **(to be used in App.config)**	**Description**	**Possible Values**
None – set `RfcTrace.DefaultTraceLevel` directly (note that this property has type `uint`, so be sure not to confuse the `uint 3` with the string `"3"`, for instance)	defaultTraceLevel	This trace level is used for all destinations and servers, which do not define their own trace level. In addition it is used for writing traces in those deeper level layers of NCo, which are not directly related to a particular destination or server and consequently cannot use a destination/server trace level for their trace output.	When setting the value `RfcTrace.DefaultTraceLevel` programmatically, use the Enum values of `RfcTracing`. These can be combined using bitwise OR in order to activate the desired components simultaneously. When setting the value as a string in App.config, you can specify a comma-separated list of the Enum-names of the components to be traced. In addition, the following convenience Enums/strings are available: None/"0": no tracing Level1/"1": trace all function calls Level2/"2": function calls and public API calls Level3/"3": function calls, public API calls and internal method calls Level4/"4": function calls, public API calls, internal method calls and network hex dump

None – set `RfcTrace.TraceDirectory` directly	traceDir	Changes the directory, into which all trace files are written, from the current working directory to the given directory.	Any valid directory string. Non-existing directories are attempted to be created.
None – set `RfcTrace.TracePerThread` directly	traceType	Defines whether NCo should keep a separate file for each thread (default), or keep only one file for the entire process.	When setting the value programmatically via `RfcTrace.TracePerThread`, use true or false. When setting the value in the config file, set traceType = “PROCESS”, if you want to get a single file for the entire process.
None – set `RfcTrace.TraceEncoding` directly	traceEncoding	Sets the encoding, in which the trace files should be written. Default is UTF-8.	Any encoding name valid for the API `System.Text.Encoding.GetEncoding()`.
None – set `GeneralConfiguration.CPICTraceLevel` directly	cpicTraceLevel	Sets the CPIC trace level.	CPIC tracing can be helpful when troubleshooting the underlying CPIC communication layer. The CPIC trace level can be set to 0, 1, 2 or 3. The higher the CPIC trace level, the more detailed the tracing information will be that is collected in the CPIC trace file nco_cpic_xxxxx.trc, which is located in the same directory as the NCo trace files are. Setting the trace level to 0 will switch off CPIC tracing. The default value is specified by the environment variable CPIC_TRACE, if this variable has been set. Otherwise the default value is 0.
None – set `RfcRepository.UseRoundtripOptimization` directly	useRoundtrip Optimization	Sets the general use of roundtrip optimization for repository metadata lookups.	Use `on` or `true` to generally switch on the use of roundtrip optimization and `off` or `false` to switch it off (the string values are interpreted in a case-insensitive manner). Any other value (or no value) will have no effect and result in the default behavior. For details check the API documentation of `RfcRepository.UseRoundtripOptimization`.

None – set `GeneralConfiguration.DNSCacheTTL` directly	dnsCacheTTL	Sets the TTL (time to live) in seconds for positive records in the DNS cache.	The default, and also the maximal supported value, is 24 * 60 * 60 seconds which is 24 hours.Use value 0 to switch off DNS caching. A low value will enable frequent updates of the records in the DNS cache. However, updating the DNS cache too frequently may harm performance.
None – set `GeneralConfiguration.DNSNegativeCacheTTL` directly	dnsNegative CacheTTL	Sets the TTL (time to live) in seconds for negative records in the DNS cache.	The default value is 5 * 60 seconds which is 5 minutes. The maximal supported value is 24 * 60 * 60 seconds which is 24 hours. Use value 0 to switch off DNS caching. A low value will enable frequent updates of the records in the DNS cache. However, updating the DNS cache too frequently may harm performance.

General Connection Parameters

The following parameters can be used in RFC client programs and RFC server programs alike.

Constant (to be used in RfcConfigParameters)	**String Value** (to be used in App.config)	**Description**	**Possible Values**
Name	NAME	Each destination and server needs to be given a name. The application can then access the destination/server via this name *N* by calling `RfcDestinationManager.GetDestination(`*N*`)` or `RfcServerManager.GetServer(`*N*`)` respectively.	Arbitrary string.
PartnerCharSize	PCS	"Partner character size". In 99.9% of the cases you don't need to bother with that. During the initial handshake, NCo obtains the correct value from the backend and uses it from then on. One rare use case is as follows: you know that the backend is Unicode	1: Non-Unicode 2: Unicode

		and want to use a non-ISO-Latin-1 username or password for the initial logon. As the initial handshake is done with ISO-Latin-1, the characters in username/passwd would break, resulting in a refused logon. In that case set PCS=2 and NCo will use Unicode for the initial handshake.	
Codepage	CODEPAGE	Similar to PCS above. You only need it if you want to connect to a non-Unicode backend using a non-ISO-Latin-1 username or password. NCo will then use that codepage for the initial handshake, thus preserving the characters in username/password. Another use case is, if you logon to a backend, whose “system codepage” is different from 1100, and an error happens during the initial handshake before the partners can exchange their supported codepages. The backend will return an error message in its system codepage (e.g. Japanese Shift-JIS) and NCo will try to interpret the error message as 1100, making it illegible.	A few common values are: 1401: ISO-Latin-2 1500: ISO-Latin-5/Cyrillic 1600: ISO-Latin-3 1610: ISO-Latin-9/Turkish 1700: ISO-Latin-7/Greek 1800: ISO-Latin-8/Hebrew 1900: ISO-Latin-4/Lithuanian/Latvian 8000: Japanese 8300: Traditional Chinese 8400: Simplified Chinese 8500: Korean 8600: Thai 8700: ISO-Latin-6/Arabic However, please note that these values can be customized in the backend. Better consult the backend sysadmin first, as otherwise things may go terribly wrong.
OnCharacterConversionError	ON_CCE	“Character Conversion Error” What shall NCo do when it encounters a character that does not exist in the target codepage, a broken character, or a control character (0x00 - 0x19)?	This parameter can take three values: 0: Abort with an error message (default behavior). Note that in this case control characters (e.g. tabulator, carriage return, or linefeed characters) are not considered "illegal" and will therefore not cause an abort. 1: Copy the character in a "round-trip compatible way". The resulting output character may be "garbage"

			in the target codepage, but when converted back to the source codepage, it will be the original character. 2: Replace the character with a substitute symbol (usually a # character). Note that in this case the control characters are replaced as well. If you need the control characters, then you'll have to use option 0 or 1, depending on whether you want the NW RFC Lib to abort the call in case of broken characters or not.
CharacterFaultIndicatorToken	CFIT	“Conversion Fault Indicator Token" The substitute symbol used if ON_CCE=2.	Needs to be given as hexadecimal value of a Unicode codepoint. The default is 0x0023 ("# character").
UseSAPCodepages	USE_SAP_CODEPAGES	By default, NCo uses Microsoft’s codepage converters contained in the .NET Framework. However, for certain SAP Codepages this may lead to incorrectly translated and broken characters, for example, if the backend is running a “blended codepage” or an East-Asian codepage. In this case you can specify a list of codepages for which you want NCo to use the SAP codepage converters.	Comma-separated list of SAP codepages, e.g. 6100,8000,8340
RepositoryDestination	REPOSITORY_DESTINATION	Provides the name of a destination that should be used for retrieving metadata information (function module and structure descriptions). For a destination, you will not need this in most of the cases. Usually a destination uses itself for the necessary DDIC lookups. However, if you are communicating with a large number of different backend	The “Name” of a destination

		systems, all of which have the same release (and consequently identical metadata information), it would be wasteful to cache that identical metadata information several times. Instead you can use only one system and its corresponding DDIC cache and assign this system to the remaining n-1 destinations as a “repository destination”. Then we store all DDIC information in memory only once, and the other destinations reuse the information from the first destination. For a server, a repository is mandatory, but you do not need to use the “repository destination” parameter. Instead you can also set the repository programmatically, for example when using a hard-coded repository (`RfcCustomRepository`).	
Trace	TRACE	Sets a trace level for this specific destination or server.	Same values as for the default trace level.
SncMode	SNC_MODE	Determines whether connections will be secured with SNC.	0: do not use SNC (default) 1: use SNC
SncMyName	SNC_MYNAME	In most cases this can be omitted. The installed SNC solution usually knows its own SNC name. Only for solutions supporting “multiple identities”, you may need to specify the identity to be used for this particular destination/server.	Varies depending on the installed SNC solution (Secude, Kerberos, NTLM, etc). Example for Secude: p/secude:CN=ALEREMOTE, O=Mustermann-AG, C=DE
SncPartnerName	SNC_PARTNERNAME	The backend’s SNC name.	See SncMyName
SncLibraryPath	SNC_LIB	Full path and name of the SNC shared library to be used.	

SncQOP	SNC_QOP	Quality of Service to be used for SNC communication of this particular destination/server.	One of the following values: 1: Digital signature 2: Digital signature and encryption 3: Digital signature, encryption, and user authentication 8: Default value defined by back-end system 9: Maximum value that the current security product supports
SAPRouter	SAPROUTER	If the connection needs to be made through a firewall via a SAPRouter, specify the SAPRouter parameters here.	A list of host names and service names / port numbers in the following format: /H/hostname/S/portnumber
NoCompression	NO_COMPRESSION	By default the RFC protocol compresses tables when they reach a size of 8KB or more. On very rare occasions you may want to turn this off, for example if you are transporting huge integer/binary tables with "random" data, where compression would have no notable effect except for wasting CPU cycles. Or if you are trouble-shooting a certain problem and want to see the table in the trace file in human-readable format.	0: Compress tables (default) 1: Do not compress tables

Client Parameters

The following parameters are used for defining an RFC destination.

Constant **(to be used in RfcConfigParameters)**	**String Value** **(to be used in App.config)**	**Description**	**Possible Values**
Client	CLIENT	The backend’s client (or “Mandant”) into which to log in.	000 - 999
Language	LANG	The logon language to be used.	DE, EN, JA, …
User	USER	The username to be used for log in.	

AliasUser	ALIAS_USER	Optional. Consult ABAP documentation for details on the usage of this parameter.	
Password	PASSWD	The password to be used for log in.	
SAPSSO2Ticket	MYSAPSSO2	Instead of logging in with user and password, you can also log in with an SSO2 ticket or Assertion ticket. However, as an external application it is quite difficult to get one. Currently the only known way of obtaining an SSO/Assertion ticket is, if you are an RFC server program, and the backend sends you one. You can then use this ticket for logins to further systems that have the same user base.	Use this parameter instead of USER&PASSWD to log in with an SSO2 ticket (Single-Sign-On) or with an "Assertion" ticket (starting with backend release 7.00).
X509Certificate	X509CERT	Another alternative way to user/password login is login via an X.509 certificate. The backend system needs to be set up accordingly and map the certificate to the corresponding user.	X.509 certificate in Base64 encoded form
LogonCheck	LCHECK	Determines whether the login procedure should be executed when opening an RFC connection. Not very useful, however, because you cannot do much with an RFC connection that is not logged in as a particular user.	0: do not perform the login procedure 1: perform the login procedure (default)
ExternalIDType	EXTIDTYPE	“External ID” is an old login mechanism that is no longer recommended. By default, this mechanism is disabled in the SAP system. If you want to enable it you have to set the profile parameter snc/extid_login_rfc=1. The external ID type defines what kind of data the external ID data parameter will contain.	NT: NTLM/ Windows Domain User ID: Microsoft .NET passport DN: certificate
ExternalIDData	EXTIDDATA	See ExternalIDType. Data which somehow identifies a SAP backend user. The exact format of this value depends on the value of the external ID type. For all values of ExternalIDType, the external ID data needs to be mapped to a SAP system user in the table	Some data identifying the external user, depending on the type of external ID

		VUSREXTID.	
UseSAPGui	USE_SAPGUI	Determines whether a SAPGui session should be attached to the RFC connections of this destination. This can be useful if you want to call old BAPIs that produce Dynpro output and would otherwise dump with a DYNPRO_SEND_IN _BACKGROUND (“Screen output without connection to user”).	0: no SAPGui (default) 1: attach a visible SAPGui 2: attach a hidden SAPGui, which just receives and ignores the screen output. Note that for values other than 0 a SAPGui needs to be installed on the machine where the client program is running. This can be either a normal Windows SAPGui or a Java Gui. Additionally, the backend needs to fulfill the requirements listed in SAP note 1258724.
AbapDebug	ABAP_DEBUG	Can be used for R/3 systems with release < 6.20, where "external breakpoints" are not yet available. The connections are opened in debug mode and the invoked function module can be stepped through in the debugger.	0: no debugging (default) 1: attach a visible SAPGui and break at the first ABAP statement of the invoked function module. Note that for debugging a SAPGui needs to be installed on the machine where the client program is running. This can be either a normal Windows SAPGui or a JavaGui. For backend releases >= 6.20 use "external breakpoints" instead (see note 668256), as this is more convenient and allows the debugger to run on any host, not only the host on which the RFC client program is running. Again the prerequisites of SAP note 1258724 need to be satisfied.

PasswordChangeEnforced	PASSWORD_CHANGE_ENFORCED	When a backend user still has an initial password that user may or may not be able to logon via RFC, depending on the value of the profile parameter rfc/reject_expired_passwd. See SAP note 161146 for details. If the backend system rejects expired/initial passwords (the above profile parameter is set to "1"), then there is nothing you can do about this: you need to change the user's password via SAPGui. If the backend has the above profile parameter set to "0", you have two choices: either you don't care and just keep logging in with the initial password, or you want to force your application to change the user's password to a safe one. In that case you need to activate the PasswordChangeEnforced flag and in addition implement a delegate of type `PasswordChangeHandler` and register it at the `RfcDestination`. This delegate will be invoked during the initial login process and allows you to change the user's password on the fly.	If the backend's profile parameter rfc/reject_expired_passwd is set to "0", this flag has the following effect: 0: NCo logs in using the provided initial password. 1: NCo checks, whether a `PasswordChangeHandler` is registered for the current RfcDestination. If there is one, NCo invokes it and tries to change the user's password to the new one. If no `PasswordChangeHandler` is registered, login will fail with an `RfcLogonException`. If the profile parameter is set to "1", login will fail with an `RfcLogonException`, no matter how you set the PasswordChangeEnforced flag.
PoolSize	POOL_SIZE	Gives the maximum number of RFC connections that this destination will keep in its pool. More connections can be opened (until PeakConnectionsLimit is reached), but they are closed immediately after usage.	Default is 10.
PeakConnectionsLimit	MAX_POOL_SIZE *Note*: This name is inappropriate but had to be retained to ensure downward compatibility. It may be adapted in future releases.	In order to prevent an unlimited number of connections to be opened (which from a certain point on would cause the entire machine's performance to deteriorate dramatically), you can set the PeakConnectionsLimit parameter. NCo will not open further connections for this destination when this limit is	Default is 10.

		reached.	
PoolIdleTimeout	POOL_IDLE_TIMEOUT	If a pool has been idle for more than PoolIdleTimeout seconds (i.e., no connections were drawn from the pool) it will be destroyed upon checking for idle connections or pools.	Time in seconds. Default is 3600s (one hour).
MaxPoolWaitTime	MAX_POOL_WAIT_TIME	When requesting a connection although the peak connection limit has been reached, the request will wait for at most the given number of milliseconds. If a connection is returned to the pool during that time the request can be satisfied. Otherwise a RfcResourceException is thrown.	Time in milliseconds (default is 0ms)
ConnectionIdleTimeout	IDLE_TIMEOUT *Note*: This name is not adequate but had to be retained to ensure downward compatibility. It may be adapted in future releases.	If a connection has been idle for more than ConnectionIdleTimeout seconds, it will be closed and removed from the connection pool upon checking for idle connections or pools.	Time in seconds. Default is 600s.
RepositoryConnectionIdle Timeout	REP_CONN_IDLE_TIMEO UT	With this parameter ConnectionIdleTimeout (or its default) can be overridden specifically for connections that are used for metadata lookup.	The value specified through ConnectionIdleTimeout (or its default)
IdleCheckTime	IDLE_CHECK_TIME	This parameter defines how often NCo should check pools for connections that exceeded their respective idle timeout as well as pools themselves for exceeding the pool timeout.	Time in seconds. Default is 600s.
RepositoryUser	REPOSITORY_USER	If you do not want the same user to be used for both the “application level” function calls and the calls that lookup repository information from the backend’s DDIC you can configure a separate repository user here. This allows you to separate the “application user” from the technical “DDIC user” and the respective set of RFC authorizations.	
RepositoryPassword	REPOSITORY_PASSWD	Password for the above repository user.	
RepositorySncMyName	REPOSITORY_SNC _MYNAME	Can be used instead of RepositoryUser/ RepositoryPasssword.	

RepositoryX509Certificate	REPOSITORY_X509CERT	Can be used instead of RepositoryUser/ RepositoryPasssword.	

In addition to these parameters you need to define one of the following set of parameters, depending on whether you want to logon directly to one specific SAP application server or whether you want to use group logon via a SAP message server.

Application Server Logon

Constant **(to be used in RfcConfigParameters)**	**String Value** **(to be used in App.config)**	**Description**	**Possible Values**
AppServerHost	ASHOST	The hostname of the specific SAP application server, to which all connections shall be opened.	
AppServerService	ASSERV	The service name (as defined in etc/services) or the port number, under which the application server's gateway process is listening for RFC requests. Note: usually this parameter can be omitted. By default, RFC uses the port number "33XY", where XY is the system number of the SAP system.	e.g. sapgw00, 3300
SystemNumber	SYSNR	The SAP system's system number	00-99

Group Logon

Constant **(to be used in RfcConfigParameters)**	**String Value** **(to be used in App.config)**	**Description**	**Possible Values**
MessageServerHost	MSHOST	The hostname of the SAP system's message server (central instance).	
MessageServerService	MSSERV	The service name (as defined in etc/services) or the port number under which the message server is listening for load-balancing requests. Note: usually this parameter can be omitted, if SystemID is specified. By default RFC uses the service name "sapmsABC", where ABC is the system ID of the SAP system. If specified, this default behavior is overridden	e.g. sapmsPRD, 3600
SystemID	SYSID	The SAP system's three-letter system ID. Mandatory if MessageServerService is not present	e.g. PRD
LogonGroup	GROUP	The logon group, from which the message server shall select an application server. Please note, that though being optional, it is always a good idea to specify this parameter.	e.g. PUBLIC

Server Parameters

The following parameters are used for registering an RFC server at a backend system.

Constant **(to be used in RfcConfigParameters)**	**String Value** **(to be used in App.config)**	**Description**	**Possible Values**
GatewayHost	GWHOST	The hostname of the gateway at which you want to register. Needs to be the same as defined in SM59 (if the RFC destination defines an explicit gateway).	
GatewayService	GWSERV	The service name (as defined in etc/services) or the port number under which the gateway is listening.	sapgw00 – sapgw99, 3300 - 3399
ProgramID	PROGRAM_ID	The RFC destination's program ID as defined in SM59.	
SystemIDs	SYS_IDS	If you want to restrict your RFC server to be accessible only from certain SAP systems (e.g. only from PROD, but not from DEV), you can provide here a list of system IDs of those systems which shall be allowed to send requests to the RFC server.	A comma-separated list of IDs of those systems that are granted access. By default all systems are admitted.

RegistrationCount	REG_COUNT	Defines the number of (initial) registrations on the SAP gateway and thus determines how many (stateless) RFC requests our server is able to process simultaneously.	Default is 1.
MaxRegistrationCount	MAX_REG_COUNT	Whenever the ABAP side (or our server function implementation) decides to turn a connection stateful, this connection is reserved for private usage of this particular SAP user session (“internal mode”). This can potentially last for a very long time. If several users do this at the same time, all or a large number of server connections will go into busy state and become unavailable for “normal” RFC request processing. In order to prevent an RFC server from becoming unresponsive, NCo opens a new registration whenever one of the existing ones goes into stateful state. However, too many registrations would have a severely adverse impact on the overall system performance. Therefore NCo stops making new registrations once the MaxRegistrationCount is reached. Also note that once connections return from their stateful state and become available for the “public” again, NCo will close some of the idle registrations until the number is down to RegistrationCount again.	Default is 1.
MaxShutdownWaitTime	MAX_SHUTDOWN_ WAIT _TIME	When shutting down an RFC server via `RfcServer.Shutdown(false)`, the RFC server will wait for currently being processed RFC requests to finish. However, it will wait at most for MaxShutdownWaitTime milliseconds and will then abort any RFC requests that did not finish by then.	A time in milliseconds. Default is 30000 ms.
ServerRestartTimeout	SERVER_RESTART _TIMEOUT	When a server is broken (i.e., all connections are dysfunctional) periodic restart attempts are executed until the timeout is reached (or indefinitely if the value of the timeout is negative)	A timeout in seconds. Default is -1 meaning no timeout. For further details consult API documentation.

The App.config File

In an application based on SAP NCo, the App.config file can be used in two ways:

- You do not want to implement your own instances of `IDestinationConfiguration` and/or `IServerConfiguration` and instead want to supply all necessary configuration parameters in the App.config file.
- You have implemented your own instances of `IDestinationConfiguration` and/or `IServerConfiguration`, but do not want to register them programmatically. Instead you want to ensure that instances of your configuration classes are registered with NCo before any of your coding is executed. You provide the names of your classes in the App.config file, and NCo registers them right when the sapnco.dll is loaded by the operating system. This guarantees that no other coding can inject its own implementations before your coding gets a chance to register your implementations. (The downside from a security perspective is that you now need to ensure that the App.config file is well protected.)

The format of a App.config file can become quite complex and confusing, and if you make a subtle error somewhere it can be rather tedious to locate the cause of the problem. Therefore here is an example of how to create an App.config file for NCo in the simplest way.

First of all, all NCo parameters need to be given in a sectionGroup of name "`SAP.Middleware.Connector`".

- If you want to supply some of the “general configuration parameters”, you need to add a section of name “`GeneralSettings`” to that sectionGroup. This group needs to be of type “`SAP.Middleware.Connector.RfcGeneralConfiguration`”.
- If you want to supply destination parameters, add another sectionGroup of name “`ClientSettings`” to that sectionGroup.

  This sectionGroup may then contain one of the following two sections:

  If you want to supply connection parameters directly, add a section of name “`DestinationConfiguration`” and type “`SAP.Middleware.Connector.RfcDestinationConfiguration`”. (It will contain both, the “general connection parameters” as well as the “client parameters”.)

  If you want to supply the type name of your own `IDestinationConfiguration` implementation for NCo to load on process startup, add a section of name “`DestinationTypeConfiguration`” and type “`SAP.Middleware.Connector.RfcTypeConfiguration`”.
- If you want to supply server parameters, add another sectionGroup of name “`ServerSettings`” to that sectionGroup.

  This sectionGroup may then contain one of the following two sections:

  If you want to supply connection parameters directly, add a section of name “`ServerConfiguration`” and type “`SAP.Middleware.Connector.RfcServerConfiguration`”. (It will contain both, the “general connection parameters” as well as the “server parameters”.)

  If you want to supply the type name of your own `IServerConfiguration` implementation for NCo to load on process startup, add a section of name “`ServerTypeConfiguration`” and type “`SAP.Middleware.Connector.RfcTypeConfiguration`”.

Putting everything together, here is now a full example showing some general settings and direct logon parameters for a few destinations and servers:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <sectionGroup name="SAP.Middleware.Connector">
            <section name="GeneralSettings" type=
                   "SAP.Middleware.Connector.RfcGeneralConfiguration,
                    sapnco" />

            <sectionGroup name="ClientSettings">
                <section name="DestinationConfiguration" type=
                             "SAP.Middleware.Connector.
                              RfcDestinationConfiguration, sapnco" />
            </sectionGroup>

            <sectionGroup name="ServerSettings">
                <section name="ServerConfiguration" type=
                             "SAP.Middleware.Connector.
                              RfcServerConfiguration, sapnco" />
            </sectionGroup>
        </sectionGroup>
    </configSections>

    <SAP.Middleware.Connector>
        <GeneralSettings defaultTraceLevel="1" traceDir="C:\Temp"
                         traceEncoding="UTF-8" traceType="PROCESS" />

        <ClientSettings>
            <DestinationConfiguration>
                <destinations>
                    <add NAME="PRD_DirectLogon" USER="rfctest"
                         PASSWD="1234" CLIENT="800" LANG="DE"
                         ASHOST="hostname" SYSNR="36"
                         POOL_SIZE="15" MAX_POOL_SIZE="25" />
                    <add NAME="PRD_GroupLogon_with_SNC" SNC_MODE="1"
                         SNC_PARTNERNAME="p/secude:CN=PRD,
                                          O=Mustermann-AG, C=DE"
                         SNC_QOP="9" CLIENT="000" LANG="DE"
                         MSHOST="prdmain" SYSID="PRD" GROUP="PUBLIC"
                         REPOSITORY_DESTINATION="PRD_DirectLogon" />
                </destinations>
            </DestinationConfiguration>
        </ClientSettings>

        <ServerSettings>
            <ServerConfiguration>
                <servers>
                    <add NAME="TaxCalculator" GWHOST="prdmain"
                         GWSERV="sapgw36" PROGRAM_ID="Tax_Server"
                         REPOSITORY_DESTINATION="PRD_DirectLogon"
                         REG_COUNT="5" />
                    <add NAME="InventorySystem" GWHOST="prdappserv2"
                         GWSERV="sapgw36" PROGRAM_ID="InvControl"
                         REG_COUNT="10" MAX_REG_COUNT="20"
                         SNC_MODE="1" SNC_PARTNERNAME=
                         "p/secude:CN=PRD, O=Mustermann-AG, C=DE"
                         SNC_QOP="9" TRACE="2" SYS_IDS="PRD"/>
                </servers>
            </ServerConfiguration>
        </ServerSettings>
    </SAP.Middleware.Connector>
```

```
</configuration>
```

An example for specifying type name and assembly name of your own `IDestinationConfiguration` and `IServerConfiguration` to be loaded by NCo at startup looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <sectionGroup name="SAP.Middleware.Connector">
            <section name="GeneralSettings" type=
                  "SAP.Middleware.Connector.RfcGeneralConfiguration,
                   sapnco" />

            <sectionGroup name="ClientSettings">
                <section name="DestinationTypeConfiguration" type=
                      "SAP.Middleware.Connector.RfcTypeConfiguration,
                       sapnco" />
            </sectionGroup>

            <sectionGroup name="ServerSettings">
                <section name="ServerTypeConfiguration" type=
                      "SAP.Middleware.Connector.RfcTypeConfiguration,
                       sapnco" />
            </sectionGroup>
        </sectionGroup>
    </configSections>

    <SAP.Middleware.Connector>
        <GeneralSettings defaultTraceLevel=
                   "RemoteFunctionCall,RfcData" traceDir="C:\Temp" />

        <ClientSettings>
            <DestinationTypeConfiguration assemblyName="myapp.exe"
                        typeName="MyNamespace.MyDestinationConfig" />
        </ClientSettings>

        <ServerSettings>
            <ServerTypeConfiguration assemblyName="myapp.exe"
                        typeName="MyNamespace.MyServerConfig" />
        </ServerSettings>
    </SAP.Middleware.Connector>
   </configuration>
```

Appendix B: Installation of different NCo versions and their prerequisites

NCo 3.0 requires a SAP backend release of R/3 4.0B or higher. R/3 3.1I is no longer supported.

The supported Microsoft Windows versions are listed in SAP note 856863, point 3.

NCo comes in four different versions:

- x86 (32bit), compiled with Visual Studio 2005

  (.NET Framework 2.0 – 3.5 and Visual C++ runtime 8.0)
- x86 (32bit), compiled with Visual Studio 2010

  (.NET Framework 4.0 and Visual C++ runtime 10.0)
- x64 (64bit), compiled with Visual Studio 2005

  (.NET Framework 2.0 – 3.5 and Visual C++ runtime 8.0)
- x64 (64bit), compiled with Visual Studio 2010

  (.NET Framework 4.0 and Visual C++ runtime 10.0)

The NCo versions compiled with Visual Studio 2005 need the exact Visual C++ runtime version 8.0.50727.4053. The corresponding MSI installers automatically install this C++ runtime version, if not yet present on the target machine.

The NCo versions compiled with Visual Studio 2010 should work with any 10.0 C++ runtime.

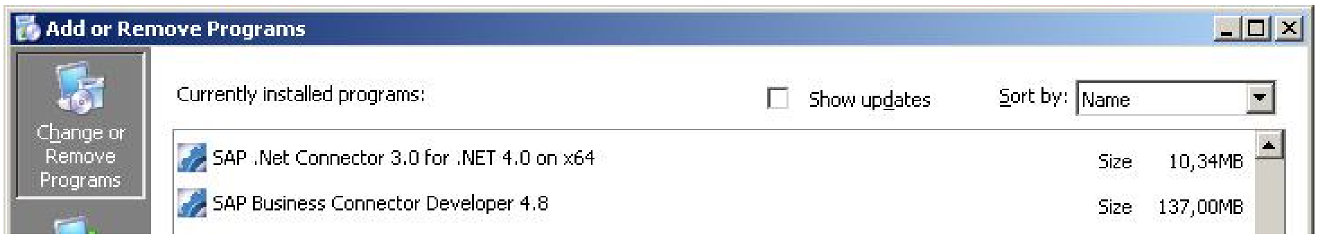
In general, only one version of NCo needs to be installed on a machine, and then any number of different .NET applications can use that single NCo installation. (You may want to register sapnco.dll and sapnco_utils.dll in the GAC, while libicudecnumber.dll, which is a plain unmanaged C/C++ DLL, should be available in the PATH.)

However, in some situations it may be necessary to install two different versions of NCo, for example if you have some .NET applications that need to run in 32bit mode, while others need to run in 64bit mode. In this case you will need to install an x86 as well as an x64 version of NCo.

Starting with NCo 3.0.3 this is possible out-of-the-box and without any problems. First of all, the various MSI installers carry the necessary information about .NET version (2.0 or 4.0) and about CPU architecture (x86 or x64) in their names, in order to prevent an accidental confusion of the different versions.

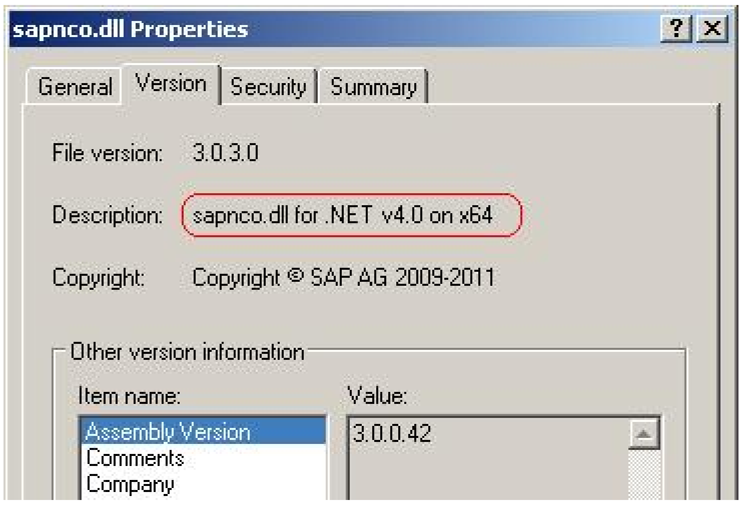
Beginning with NCo 3.0.6, the setup program for the .NET 4.0 runtime allows the user to optionally install the NCo3 assemblies into the system GAC with or without registering the NCo3 WMI provider implemented within NCo3. Registering the NCo3 WMI provider allows applications that use NCo3 to publish the instrumented NCo3 object instances to WMI infrastructure. Please note that only applications that run as administrator can publish NCo3 WMI object instances.

Next, the entry the installer creates in the machine's "Software" overview carries the same information. This allows both quickly checking which version has been installed and installing several different versions side-by-side. (In previous patch levels of NCo 3.0 this was a problem, as all installers used the same name in the "Software" overview, so only one version could be installed, and any following installation attempts of a different version would run into an error.) Example from "Software" overview (or "Add/Remove Programs" on Windows XP):



SAP .Net Connector 3.0 for .NET 4.0 on x64 Size 10,34MB
SAP Business Connector Developer 4.8 Size 137,00MB

And even if you have only the bare DLLs and can no longer relate it to a particular NCo installation, you can still right-click on the DLL and find the precise version in the properties:



File version: 3.0.3.0

Description: sapnco.dll for .NET v4.0 on x64

Copyright: Copyright © SAP AG 2009-2011

Assembly Version: 3.0.0.42

In case you run the installer in an unattended mode, you can specify certain properties to determine how to install:

- TARGETDIR: the location where to install the .NET Connector
- GAC_WMI (only .NET 4.0 installer): determines how to deal with GAC and WMI
  - 1: Only install into target directory
  - 2: Install the assemblies of NCo also into GAC
  - 3. Install the assemblies of NCo also into GAC an register the WMI provider

The example below installs NCo into C:\TEMP\NCo307 and into GAC:

msiexec /passive /i NCo307_Net40_x64.msi TARGETDIR=C:\TEMP\NCo307 GAC_WMI=2

Prerequisites for using NCo in your .NET application

As NCo 3.0 contains unmanaged C/C++ components, a bit care needs to be taken when referencing it in your .NET applications, especially when developing on x64 platforms. Otherwise you can easily run into one of the following two problems:

- "System.BadImageFormatException: Could not load file or assembly 'sapnco, Version=3.0.0.42, Culture=neutral, PublicKeyToken=50436dca5c7f7d23' or one of its dependencies. An attempt was made to load a program with an incorrect format."

  This happens when you mix components for different processor architectures (for example, the installed .NET Framework is x86 (32bit) and the installed NCo is x64 (64bit)), or when you try to build a project with "Platform=Any CPU".

- "System.IO.FileLoadException: Could not load file or assembly 'sapnco_utils, Version=3.0.0.42, Culture=neutral, PublicKeyToken=50436dca5c7f7d23' or one of its dependencies. This application has failed to start because the application configuration is incorrect. Reinstalling the application may fix this problem. (Exception from HRESULT: 0x800736B1)"

This happens when the required Visual C++ runtime is not installed. (In addition to the correct version, you also need the correct platform, namely x86 or x64.)

In order to prevent these problems, make sure to adhere to the following steps when setting up your Visual Studio project:

- Decide for which target platform you want to develop your application, x86 or x64.
- Make sure the necessary .NET Framework is installed for that target platform.
- Make sure the necessary Visual C++ runtime is installed for that target platform. (For C++ 8.0 see SAP note 1375494. For C++ 10.0, any runtime version should be ok.)
- Install NCo 3.0 for the desired target platform.
- In your Visual Studio project select "Platform=<the desired target platform>". Selecting "Any CPU" will not work.
- Add two assembly references to your Visual Studio project, one for sapnco.dll and one for sapnco_utils.dll. The other two libraries "libicudecnumber.dll" and "rscp4n.dll" do not need to be added as an assembly reference, because they are plain unmanaged C libraries. In most cases they are not required. Only if you use the ABAP type DECFLOAT, your project will need the libicudecnumber.dll, and only if you use the "UseSAPCodepages" parameter, your project will need the rscp4n.dll. In that case make sure they can be found in the PATH environment variable.

Appendix C: Required Network Settings on Hosts Running NCo Applications

In some scenarios, NCo requires certain entries in the "Services" configuration to be present, as otherwise the establishment of a TCP/IP connection to the backend may fail with an error message like

```
SAP.Middleware.Connector.RfcCommunicationException:
LOCATION    CPIC (TCP/IP) on local host BLABLABLA with Unicode
ERROR       service 'sapmsABC' unknown
TIME        Wed Mar 27 17:14:47 2013
RELEASE     720
COMPONENT   NI (network interface)
VERSION     40
RC          -3
MODULE      nixxsl.cpp
LINE        184
DETAIL      NiSrvLGetServNo: service name cached as unknown
COUNTER     7
```

If SAPLogon is installed on the machine, where the NCo application is running, it usually creates all necessary entries for the SAP backend systems in your network. However, if SAPLogon is not installed (or you need to connect to a backend system, for which SAPLogon does not have the necessary information), you need to create these entries manually as follows.

Open the file "C:\Windows\System32\drivers\etc\services" in a text editor that can deal with Unix-style line-endings properly, e.g. Write/Wordpad, Notepad++ or some other editor of your choice. Make sure to start the editor with Administrator privileges, because otherwise Windows won't let you save your changes.

Add an entry like the following to the file and save:

```
sapmsABC       3600/tcp
```

How this entry has to look like, is now described in detail:

- **Application Server Logon**

  When logging on to an SAP backend via direct Application Server logon, you usually provide these two logon parameters to NCo:
  ASHOST=application-server hostname
  SYSNR=XY

  NCo then automatically uses the port 33XY for connecting to the backend. This is the default port for RFC communication, so in most cases there is nothing to do for you. Only in the rare cases where the default gateway port of that system has been changed for some reason, you need to add the entry
  ```
  sapgwXY       1234/tcp
  ```
  to the services file as described above, and then replace the logon parameter SYSNR=XY with GWSERV=sapgwXY. NCo will then automatically get the correct port from the operating system.

- **Message Server Logon**

  In this case NCo performs an additional step, before logging on to an application server: it connects to the message server and asks for the hostname/IP-address of an application server to use belonging to the given logon group. For this case you usually pass the following parameters to NCo:
  MSHOST=message-server hostname
  SYSID=ABC
  GROUP=PUBLIC

  For the connection to the message server, NCo then tries to obtain the port number corresponding to the service sapmsABC from the operating system. So an entry like
  ```
  sapmsABC       3600/tcp
  ```
  needs to exist in the services file.
  Tip: If you can’t change the services file (e.g. because of missing administrator priviledges), you can for example provide the logon parameter MSSERV=3600 to NCo instead of the SYSID parameter, and NCo will use that port for opening the connection.

- **Registered Server**

  For registering an RFC server at a gateway, you usually provide parameters like
  GWHOST=gateway hostname
  GWSERV=sapgwXY
  PROGRAM_ID=program ID

  Similar to the application server logon case, NCo opens a connection on the port corresponding to sapgwXY to the gateway, so an entry for this needs to exist in the services file. (XY is again the backend’s system number.)
  Again, if you can’t change the services file, you may pass the port number directly like this:
  GWSERV=3301