SAP NetWeaver RFC SDK

SAP NetWeaver Release 7.1

Copyright

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`.

Version

Version 1.8.1	**2011/07/18**

Contents

The SAP NetWeaver RFC API 5
- Architecture 7
- Data Processing and Transfer 8
- Security Issues 10
- Supported Platforms 11
- Contents of the NW RFC SDK 11
- The SAPNWRFC.INI File 15
- Parameter Overview 17
- RFC API and SAP Router 21

Programming with the RFC API 28
- Working with the NW RFC SDK Files 29
- Getting Connected 30
- Passing Parameters 31
- RFC Client Programs 32
- RFC Server Programs 42
- tRFC between SAP and external Systems 55
- Error Handling 60
- Compiling 65

NetWeaver RFC SDK Reference 66

The SAP NetWeaver RFC API

Purpose

The RFC API enables you to establish RFC communication between an SAP system and external systems. Via the RFC API, an external system can communicate as client or server with the SAP system.

Using the RFC SDK (Software Development Kit) you can implement a bundle of C++ functions controlling RFC communication on an external client.

This documentation provides information on the SAP NetWeaver RFC SDK only. You can find information on the classic RFC SDK in the SAP Library.

Implementation Considerations

The RFC API exists in two different versions:

- Classic RFC API
- SAP NetWeaver RFC API

You can use the SAP NetWeaver RFC API with all SAP systems being currently maintained by SAP.

However, the classic and NetWeaver version of the RFC SDK are *incompatible* to each other. You cannot use both versions in parallel for a given external application.

The migration from the classic version to SAP NetWeaver RFC API requires development effort for adapting the C/C++ application environment.

Features

The two versions of the RFC API are characterized by their different functionality:

- Classic RFC API (ASCII or Unicode version)
  - Support of the classic RFC protocol only
  - No dynamic meta data retrieval
  - Not free of redundant functions
  - The ASCII version supports the SAPGUI protocol
- SAP NetWeaver RFC API contains a restructured set of functions and thus offering extended functionality like:
  - Dynamic meta data retrieval
  - Support of all SAP single-byte code pages
  - Support of the classic RFC protocol and a new binary protocol (basXML) that allows a considerable reduction of data volume when using complex parameter types.
  - No redundant functions
  - Does not support SAPGUI protocol or start of SAPGUI
  - Does not support communication between two external systems

SAP generally recommends the use of the NetWeaver RFC API due to its functional enhancements.

Further Information

You can find detailed information on security issues of the RFC API in the SAP Library:

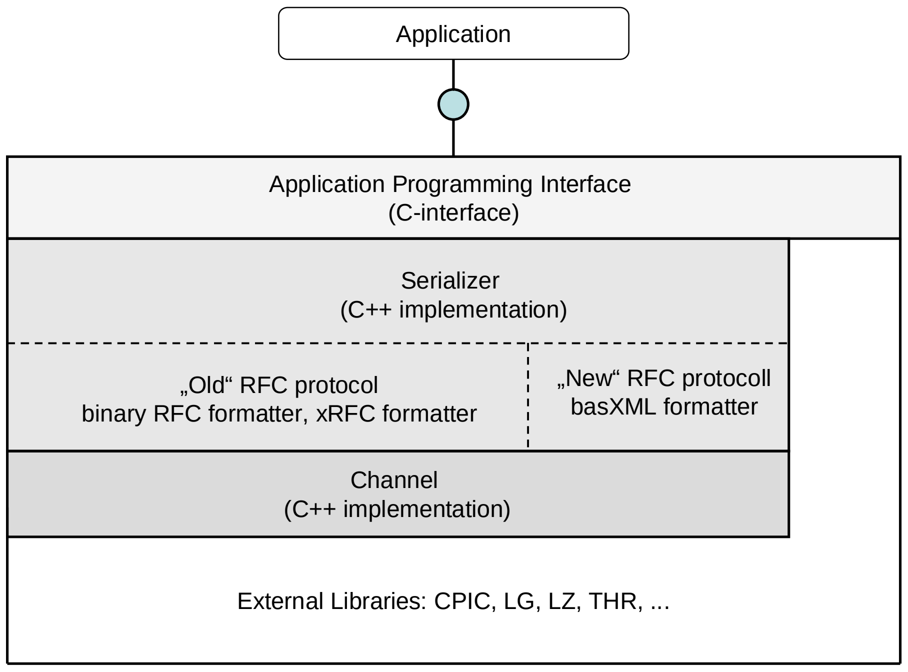
- RFC/ICF Security Guide

Architecture

The structure of the SAP NW RFC SDK consists of different layers, shown in the figure below:

- API Layer
- Serializer
- Transport Layer

Structure of the RFC SDK



Each layer executes a specified function:

- **RFC Application Programming Interface (API)**

  This is the entry point for RFC-Library. Any interaction between application and RFC-Library takes place over API only. Additionally represents API the RFC data model at the user point of view.

- **RFC serialization layer**

  is responsible for connection management, serialization and deserialization of the RFC payload into sap-proprietary format.

- **RFC Channel**

  is the transport layer. This part of the RFC library is responsible for the transport of the serialized RFC stream to the remote partner. RFC channel uses CPIC as the underlying network data transfer protocol.

Data Processing and Transfer

Data exchange of the RFC API is based on the request response model. During request (this is the phase of RFC call) the name of the called remote function module will be serialized into an RFC buffer first. All available parameter (exporting, changing, tables) will be serialized in a second step.

During response (receive phase), all received data will be deserialized and converted into its local representation.

The serialization layer supports the following RFC protocols:

- Classic RFC protocol.

  In this case the classic RFC format will be used for serialization of elementary and complex data types.

- basXML RFC protocol.

  In this version RFC data will generally be serialized using the basXML format.

  You can use the same API calls for calling remote function modules at the ABAP side regardless data serialization type you choose.

Data Conversion

The RFC API supports all SAP single-byte code pages and Unicode. Code page processing between the SAP system and the RFC API does not require any special treatment regardless of whether you use a single-byte code page or Unicode. The code pages for the external RFC program *always* have to be converted to Unicode in case of using any single-byte code page, regardless of whether it corresponds to a SAP single-byte code page or not.

This means that the RFC API *always* requires Unicode.

Data Compression

Data is compressed before sending, if both client and server system are capable of this. Otherwise, only the blank spaces at the end of a table line will be truncated.

Data Types

The NW RFC API supports all elementary and complex ABAP data types. The following list shows the supported data types and their specifications:

ABAP Data Types Supported by the RFC API

ABAP type	Typedef	Length (in Bytes)	Description
c	RFC_CHAR	1-65535	Characters, blank padded at the end
n	RFC_NUM	1-65535	Digits, fixed size, leading '0' padded.
x	RFC_BYTE	1-65535	Binary data
p	RFC_BCD	1-16	BCD numbers (Binary Coded Decimals)
i	RFC_INT	4	Integer
b	RFCTYPE_INT1	1	1-byte integer, not

			directly supported by ABAP
s	RFCTYPE_INT2	2	2-byte integer, not directly supported by ABAP
f	RFC_FLOAT	8	Floating point
d	RFC_DATE	8	Date ("YYYYMMDD")
t	RFC_TIME	6	Time ("HHMMSS")
decfloat16	RFC_DECF16	8	Decimal floating point 8 bytes (IEEE 754r)
decfloat34	RFC_DECF34	16	Decimal floating point 16 bytes (IEEE 754r)
g	RFC_STRING		Variable-length, zero terminated string
y	RFC_XSTRING		Variable-length raw string, length in bytes

Security Issues

In this section you can find general information on security-critical issues when working with the SAP NW RFC SDK.

Programming Issues

- **RFC client receiving call-backs from an SAP system**

  If you want to enable an external RFC client to receive call-backs from an SAP system you have to implement the corresponding RFC server functionality in addition to the proper client functions of the external program. This means that the server-related security issues may affect your external RFC client as well.

- **Logon Check for registered RFC servers**

  If an RFC server is registered on an RFC gateway, it is generally possible to send calls from other SAP systems (not relevant to this gateway) or from external RFC clients to this server. If, for security reasons, the server should only be able to be called by specified systems or users, the server must implement its own logon data check and reject unwanted initiators.

  For detailed information on executing this logon check see Registering Server Programs on the SAP Gateway (section *Security*).

Administration Issues

- **Using the sapnwrfc.ini file**

  You can generally use the `sapnwrfc.ini` file as a repository for connection parameters that can be referenced by the corresponding functions in order to relieve programming activities. As the information included in this file is stored on the server's hard disc it may be subject to external attacks. Therefore it is strongly recommended to avoid the storage of security-related data in this file. Security-critical parameters are mainly *User* and *Password*, but also information about message server names, program ID or gateway information may be affected.

- **Configuring registered RFC servers via transaction SM59**

  If you specify an external RFC server as RFC destination via transaction SM59 you need to enter the corresponding program ID of the RFC server. This program ID can – if known – generally be used by other external servers (not related to the SAP Gateway) to establish a connection to an SAP system. Therfore, it is generally recommended to:

  - Implement a dynamic (changeable) token for this program ID in the external RFC server.
  - Choose a value for this program ID that exhausts the provided number of digits in order to make it as secure as possible.

Further Information

You can find general information on RFC security issues in the SAP Library:

- SAP Netweaver Security Guide:

  Security Guides for Connectivity and Interoperability Technologies ->

  - RFC/ICF Security Guide
  - Security Settings in the SAP Gateway

Supported Platforms

The NW RFC SDK is available for various platforms. In the following list you will some of the most frequently used platforms that are currently supported:

- Windows x86 (32 bit)
- Windows x86 (64 bit)
- Windows Itanium (64 bit)
- HP-UX (64 bit)
- Solaris (64 bit)
- Linux (x86, 64 bit)
- AIX

For the complete list of platforms currently supported see SAP Note **1025361**.

Further Information

For an overview of the RFC SDK elements (header files and libraries) per platform see next chapter: *Contents of the NW RFC SDK*.

Contents of the NW RFC SDK

The RFC API mainly consists of library routines you can call to communicate with an RFC partner. These routines (implemented in C) perform the RFC calls and other administrative tasks needed to handle the external system's communication. The integrated RFC interface of the AS ABAP processes the counterpart communication within the SAP System.

The RFCSDK contains the following includes and platform-specific libraries:

File	Description		
sapnwrfc.h	This include file contains all data types and structures required and the prototypes (declarations) of the RFC calls.		
sapuc.h	Header file containing global Unicode specifications (part of **`sapnwrfc.h`)**		
sapucx.h	File with containing detailed Unicode specifications		
sapdecf.h	Specifies functions for working with the data type decfloat [8].		
<RFC library>	Depending on the platform, the following libraries are required:		
	Windows (valid for all available variants)		
	/lib	**sapnwrfc.lib**	Contains the RFC library functions.
		libsapucum.lib	Contains the standard c functions in unicode version.

	sapdecfICUlib.lib	Contains functions for working with the data type decfloat.
	sapnwrfc.dll	Specifies the RFC library functions.
	libsapucum.dll	Contains the standard c functions in unicode version.
	libicudecnumber.dll	Contains functions for working with data type decfloat.
	icudt34.dll	Internal code page converter. Do not use !
	icuuc34.dll	Internal code page converter. Do not use !
	icuin34.dll	Internal code page converter. Do not use !
HP-UX		
/bin	**sapnwrfc.sl**	Specifies the RFC library functions.
	libsapucum.sl	Contains the standard c functions in unicode version.
	libicudecnumber.sl	Contains functions for working with data type decfloat.
	libicuuc.sl.34	Internal code page converter. Do not use !
	libicudata.sl.34	Internal code page converter. Do not use !
	libicui18n.sl.34	Internal code page converter. Do not use !

Solaris/Linux		
/bin	**libsapnwrfc.so**	Specifies the RFC library functions.
	libsapucum.so	Contains the standard c functions in unicode version.
	libicudecnumber.so	Contains functions for working with data type decfloat.
	libicuuc.so.34	Internal code page converter. 💡 Do not use !
	libicui18n.so.34	Internal code page converter. 💡 Do not use !
	libicudata.so.34	Internal code page converter. 💡 Do not use !
AIX		
/bin	**libsapnwrfc.so**	Specifies the RFC library functions.
	libsapucum.so	Contains the standard c functions in unicode version.
	libicudecnumber.so	Contains functions for working with data type decfloat.
	libicuuc34.a	Internal code page converter. 💡 Do not use !
	libicui18n34.a	Internal code page converter. 💡 Do not use !
	libicudata34.a	Internal code page converter. 💡 Do not use !

The SAPNWRFC.INI File

Introduction

If you want to change or add parameters for your external RFC connections, you need not change the program code. These parameters can be specified the file *sapnwrfc.ini*.

The RFC library will read the *sapnwrfc.ini* file to find out the connection type and all RFC-specific parameters needed to connect to an SAP system, or to register an RFC server program at an SAP Gateway and wait for RFC calls from any SAP system.

All RFC-specific parameters, both currently known (e.g. load balancing) or becoming available in the future, can be used without changing the RFC programs.

The *sapnwrfc.ini* file must be in the same directory as the RFC client/server program.

Using the *sapnwrfc.ini* file

- an *RFC client* program must issue the function **RfcOpenConnection**. The destination parameter of this function must point to an entry in the *sapnwrfc.ini* file.
- An *RFC server* program must issue the **RfcRegisterServer**. The destination parameter of this function must point to an entry in the *sapnwrfc.ini* file.

Examples

The following listing shows some example parameter settings for the *sapnwrfc.ini file* (depending on the connection type and on the required features):

Parameter Settings for sapnwrfc.ini

Connection Type / Feature	Settings

RFC server program registered at an SAP Gateway	`DEST=rfctest`
	`PROGRAM_ID =hw1145.rfcexec`
	`GWHOST=hs0311`
	`GWSERV=sapgw53`
	`RFC_TRACE=1`

SAP System using Load Balancing	`DEST=BIN`
	`R3NAME=BIN`
	`MSHOST=hs0311`
	`GROUP=PUBLIC`
	`RFC_TRACE=0`

SAP System using a specific application server	`DEST=BIN_HS0011`
	`ASHOST=hs0011`
	`SYSNR=53`
	`RFC_TRACE=0`

Parameter Overview

In this section you will find an overview of all connection parameters that can be used in the saprfc.ini file.The parameters are divided into 3 different categories:

- General Connection parameters. These can be used with client and server programs.
- Parameters used in client programs
- Parameters depending on the connection type you use

General Connection parameters:

Parameter	Description
SAPROUTER	If the connection needs to be made through a firewall using a SAPRouter, specify the SAPRouter parameters in the following format: /H/hostname/S/portnumber/H/
SNC_LIB	Full path and name of third-party security library to use for SNC communication (authentication, encryption and signatures). Alternatively you can set the environment variable SNC_LIB.
SNC_MYNAME	Token/identifier representing the external RFC program.
SNC_PARTNERNAME	Token/identifier representing the backend system
SNC_QOP	Use 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 backend system 9: Maximum value that the current security product supports
TRACE	Use one of the following values: 0: off 1: brief 2: verbose 3: full
PCS	Partner character size. The use cases for this parameter are not very frequent. During the initial handshake the RFC library 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 and want to use a non-ISO-Latin-1 user name or password for the initial logon. As the initial handshake is done with ISO-Latin-1, the characters in USERNAME/PASSWD could not be read and logon would be denied. In that case set PCS=2 and the RFC library will use Unicode for the initial handshake.
CODEPAGE	Similar to PCS above. Only needed it if you want to connect to a non-Unicode backend using a non-ISO-Latin-1 user name or password. The RFC library will then use that codepage for the initial handshake, thus preserving the characters in username/password.

	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 The values can be customized in the backend system. Contact your backend system administrator before modifying these settings. End of the note.
NO_COMPRESSION	By default the RFC protocol compresses tables, when they reach a size of 8 KB or more. In very special situations it may be useful to turn this off, for example if you are transporting very large integer/binary tables with "random" data, where compression would have no effect, while consuming CPU resources.

Parameters used in client programs:

Parameter	Description
USER	User name
PASSWD	Password
CLIENT	The Client to which to logon.
LANG	Logon Language. Either specify the two-character ISO code (like 'EN' for English, 'KO' for Korean) or the one-character SAP-specific code (like 'E' for English, '3' for Korean). ISO codes are case-insensitiv, while the SAP codes are not. For example 'D' logs you on in German, while 'd' logs you on in Serbian, if that language is installed in your system. End of the note.
MYSAPSSO2	Use this parameter instead of USER and PASSWD to log on with an SSO2 ticket (Single-Sign_On).
GETSSO2	Set this to 1, if the backend should generate an SSO2 ticket for your user.

X509CERT	Use this parameter instead of USER and PASSWD, if you want to logon with an X.509 certificate. The certificate needs to be Base64 encoded and needs to be mapped to a valid user in the backend's user configuration.
EXTIDDATA	Old logon mechanism similar to SSO. No longer recommended.
EXTIDTYPE	See EXTIDDATA.
LCHECK	If you set this to 0, RfcOpenConnection() only opens a network connection but does not perform the logon procedure. I.e. no user session will be created inside the backend system. This parameter is intended for executing the function module RFC_PING only. End of the note.
USE_SAPGUI	Specifies whether a SAPGUI should be attached to the connection. Some (old) BAPIs need this, because they try to send screen output to the client while executing. Possible values are: 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 same machine the client program is running on. This can be a Windows SAPGUI or a Java GUI on Linux/Unix systems. `USE_SAPGUI` = 2 has a negative performance impact. If you are using only one function module that needs a GUI, and a large number of "normal" function modules, you should consider the following alternatives: a) Open the connection with `USE_SAPGUI` = 1, and immediately after each RFC call that may invoke SAPGUI, execute an RFC call to the FM `SYSTEM_INVISIBLE_GUI` on the same connection to hide the SAPGUI. Call your other FMs as usual. b) Open two connections, one without `USE_SAPGUI` parameter and one with `USE_SAPGUI` = 2. Use the first connection for executing all "normal" FMs and the second connection whenever you need to call the FM that uses the GUI.
ABAP_DEBUG	Can be used for SAP systems with release < 6.20, where external breakpoints are not yet availble. The connection is opened in debug mode and the invoked function module can be stepped through in the debugger. Possible values: 0: no debugging (default) 1: attach a visible SAPGUI and break at the first ABAP statement of the invoked function module. For debugging, a SAPGUI needs to be installed on the same machine the client program is running on. This can be either a normal Windows SAPGUI

or a Java GUI on Linux/Unix systems.

For backend releases >= 6.20 use external breakpoints instead (see SAP 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.

End of the note.

Parameters per Connection Type:

Connection Type	Parameter
RFC server program registered at an SAP gateway and waiting for RFC calls by an SAP system.	• DEST = <destination in RfcRegisterServer/RfcStartServer> • TPNAME or PROGRAM_ID = <program ID, optional; default: destination > • GWHOST = <host name of the SAP gateway> • GWSERV = <service name of the SAP gateway> In an SAP system, the program ID and this SAP gateway must be specified in the Destination entry defined with transaction SM59: use connection type T and register mode. End of the note.
SAP system using load balancing. The application server will be determined at runtime.	• DEST = <destination in RfcOpenConnection> • R3NAME or SYSID= <name of SAP system, optional; default: destination> • MSHOST = <host name of the message server> • MSSERV = Not needed in most cases. Specify this parameter only, if the message server does not listen on the standard service sapms<SysID>, or if this service is not defined in the services file and you need to specify the network port directly. on Unix the services are defined in /etc/services, while on Windows they are defined in C:\WINDOWS\system32\drivers\etc\services. End of the note. • GROUP = <group name of the application servers, optional; default: PUBLIC>
Logon to specific SAP application server.	• DEST = <destination in RfcOpenConnection> • ASHOST = <host name of a specific SAP application server> • SYSNR = <SAP system number> • GWHOST = <optional; default: gateway on application server> • GWSERV = <optional; default: gateway on

application server>

RFC API and SAP Router

SAProuter is an SAP software product which is available on all SAP-based UNIX platforms as well as on Windows and Linux. It acts like a firewall system by regulating access from/to your network.

SAProuter can be used

- to establish an indirect connection between two programs running on different machines. The network configuration does not allow a direct communication between these machines due to missing IP addresses (or the same IP addresses as well) or firewall restrictions.
- to improve network security by allowing accesses from/to your network with or without password-protection only from a specified machine where the SAProuter is running.
- to control and log all connections between your network and the rest of the world.

Important SAProuter Commands

`saprouter`	Online help (display list of all supported options)
`saprouter -r`	Start SAProuter with default values
`saprouter -s`	Stop SAPRouter

Route String

A route string can have one or more substrings. Each substring contains parameters how to reach the next SAProuter or the target host or program on the target host.

Such parameters are:

- name or IP-address of the target host
- service (port number) of the program running on the target host
- password for this connection, if needed

Example of one substring: **/H/host/S/service/P/password**

H:	Identifier for host name
S:	Identifier for service (port number)
P:	Identifier for password

Route Permission Table

The SAProuter regulates access to your network via the route permission table in form of a file. You can start your SAProuter with this file name.

An entry in a route permission table has the following structure:

```
<P/D> <source host> <target host> <target service> <password>
```

P(ermit):	allows connection
D(eny):	prevents connection
<source host>:	host name or IP-address, could be a SAProuter
<target host>:	host name or IP-address, could be a SAProuter
<target service>:	service (port number) of the program of the target host The default service of SAProuter is ‘3299’.

If no route permission table was explicitly assigned to the SAProuter while starting (option -R <name of a route permission table>), the file ‘saprouttab’ in the current directory will be used. If this file is not available, all connections are allowed.

You can use wildcards (‘*’) to define hosts, services and passwords in your route permission table.

Further Information

You can find more information of SAProuter in:

- the SAP Library:

  SAProuter
- SAP note 30289.

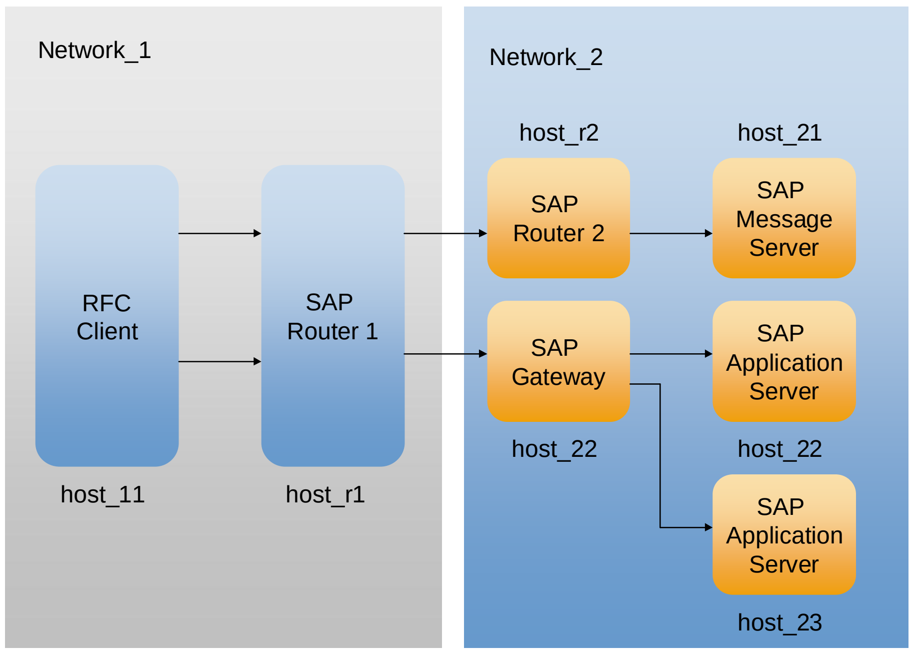
RFC Client Program and SAP Router

Any RFC client program can connect to an SAP System via SAProuter.

One or more SAProuters can be used (depending on your network architecture). The following example shows how the client program can work with two SAProuters and a message server or one SAProuter and an SAP Gateway.

If you decide to use a SAProuter you must also allow the corresponding SAP dispatcher to work with SAProuters.

SAProuter Example



Route Permission Tables

In the route permission table of **SAProuter on host_r1 in Network_1** only one entry is necessary:

P	host_11	host_r2	3299

The entries in the route permission table of **SAProuter on host_r2 in Network_2** are dependent on the type of RFC connection which is established with the SAP system. The RFC client program must inform the RFC library about all used SAProuters for connecting to the SAP system via the parameter ‘host name’ in **RfcOpenConnection**.

1. Using Load Balancing

The host name of the message server must contain the route string. For the example described in the last section, the RFC client must set the host name of the message server as follows:
MS-Host: **/H/host_r1/H/host_r2/H/host_21**

Entries in the route permission table of SAProuter on host_r2 in Network_2:

P	host_r1	host_21	sapms<SAP name>
P	host_r1	<SAP AS host 1>	sapgw<SAP system number>
...			
P	host_r1	<SAP AS host n>	sapgw<SAP system number>

2. Specified SAP Application Server and Default SAP Gateway

The host name of the specified application server must contain the route string. For the example described in the last section, the RFC client must set the host name of the application server as follows:
AS-Host: **/H/host_r1/H/host_r2/H/host_22**

Entries in the route permission table of SAProuter on host_r2 in Network_2:

P	host_r1	host_22	sapgw<SAP system number>

3. Specified SAP Application Server and Specified SAP Gateway

If you are working with a specified SAP Gateway, the host name of the SAP Gateway must contain the route string. The host name of the application server may not contain the route string. For the example described in the last section, the RFC client must set the host name of the SAP Gateway as follows:
GW Host: **/H/host_r1/H/host_r2/H/host_22**
AS Host: **host_23**

Entries in the route permission table of SAProuter on host_r2 in Network_2:

P	host_r1	host_22	sapgw<GW service number>

Starting an RFC Server Program Via SAProuter

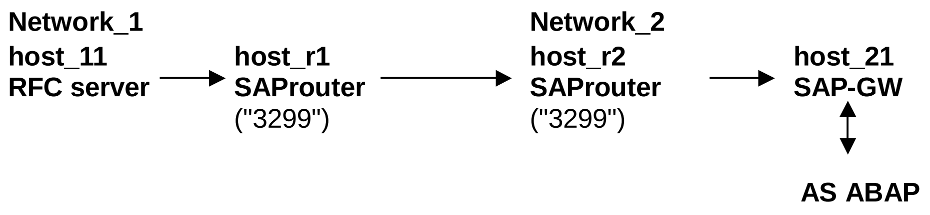
An RFC server program can be started in different ways:

- as a registered server
- started by an application server
- started by an SAP Gateway

In this section you will find information on how to start an RFC server via SAProuter depending on these starting methods.

Using the Registering Feature

One or more SAProuters can be used. The following example shows how a server program can work with two SAProuters using the registering feature:



Route Permission Tables

Entry in the route permission table of **SAProuter on host_r1 in Network_1:**

P	host_11	host_r2	3299

Entry in the route permission table of **SAProuter on host_r2 in Network_2:**

P	host_r1	host_21	sapgw<GW service number>

The host name of the SAP Gateway must contain the route string. For the example above, the RFC server must set the host name of the SAP Gateway as follows:
GW-Name: /H/host_r1/H/host_r2/H/host_21

A destination in transaction SM59 can be defined as follows:

Connection type:	**T**
Activate type:	**Registering**
Program-ID:	**host_11.srfcserv**
Gateway host:	**host_21**
Gateway service:	**sapgw<GW service number>**

Program Start by Application Server

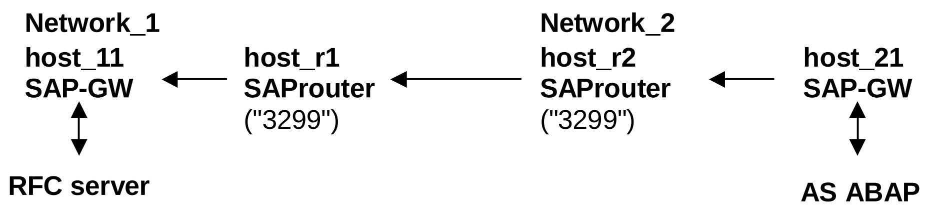
Program start by application server means that the RFC server program will be run on the same machine as the application server. In this case, you have different application servers of an SAP system running on different networks connected via SAProuter.

Program Start by SAP Gateway

Because a SAP Gateway cannot start an RFC server program with remote shell on another machine via SAProuter, it is necessary to install a SAP Gateway on a machine in the network where the RFC server program will be run. It should be the same machine for a better performance.

One or more SAProuters can be used. The following example shows

- two different networks with two SAProuters and
- how an RFC server program can be started by an SAP Gateway and how it communicates with an SAP system running on another network.



Route Permission Tables

Entry in the route permission table of **SAProuter on host_r2 in Network_2:**

P	host_21	host_r1	3299

Entry in the route permission table of **SAProuter on host_r1 in Network_1:**

P	host_r2	host_11	sapgw<GW service number>

A destination in transaction SM59 can be defined as follows:

Connection type:	**T**
Activate type:	**Start**
Program location:	**explicit host**
Program:	**/rfctest/srfcserv**
Target host:	**host_11**

⚠

The maximum length of the gateway host (transaction SM59) is 100 bytes.

Programming with the RFC API

The following section contains important guidelines for writing RFC programs.

The programs you write can call or be called by ABAP programs of an SAP system. You can find a detailed description of the following issues here:

- Working with the NW RFC SDK Files
- Getting Connected
- Passing Parameters
- RFC Client Programs
- RFC Server Programs
- tRFC between SAP and external Systems
- Error Handling

Working with the NW RFC SDK Files

Use

In this section you will find information on how to use the files that are part of the NW RFC SDK. You can use these files to:

- Call a function of the RFC library
- Use the Unicode macros und Unicode versions of the standard C Library
- Execute calculations using the data type decfloat

Procedure

- **Call a function of the RFC library**

  Include the file `sapnwrfc.h` and link it against `sapnwrfc.lib` for Windows (respectively `.so/.sl` on Unix). Add a pre-processor directive –DSAPwithUNICODE and the corresponding Unicode directive for your platform e.g. (_UNICODE on Windows).

- **Use the Unicode macros und Unicode versions of the standard C Library**

  Include `sapuc.h` and `sapucx.h` and link it against `libsapucum.lib` (`.so/.sl` for Unix).

- **Execute calculations using the data type decfloat**

  Include `sapdecf.h` and link it against `sapdecfICUlib.lib` (respectively `libicudecnumber.so/.sl` on Unix).

Getting Connected

RFC and programming information for RFC client/server programs can be summarized as follows:

- An RFC connection is always initiated by an RFC client program.
- An RFC connection is always built up in two steps:
  - Connection from an RFC client to the SAP Gateway
  - Connection from the SAP Gateway to an RFC server
- There are different ways of starting or connecting to an RFC server program:
  - An RFC server program can be registered at an SAP Gateway. The program waits for RFC requests from SAP systems.

    The registered server is the standard method for connecting an RFC server.
  - You can start an RFC server program via the SAP Gateway or the currently running application server.

    Starting the RFC server via an application server is not recommended if you are expecting frequent calls on this server.

Passing Parameters

With the NW RFC SDK the procedure for passing parameters has been considerably simplified.

Basically there are two ways to determine the parameters to be used:

- Dynamic retrieval from the SAP system
- Static specification in the initialization procedure
- You can now use an extensive "get/set" function set to specify parameters for any purpose instead of filling the former structures `RF_PARAMETER` or `RFC_TABLE`.
- The procedure for passing import, export, changing and table parameters (respectively structures) has been unified so you can use similar functions for each parameter type.

Creating and Manipulating Table Parameters

Tables passed as parameters to an RFC function must match SAP internal tables in their structure and handling.

As a result:

- To create a table parameter for sending, use the routine `RfcCreateTable`.
  This routine creates a control structure (defined as `RFC_TYPE_DESC_HANDLE`) for a table just like the one the SAP System creates automatically for internal tables.
- To manipulate a table passed to or from the SAP system (for example: to access, append or delete rows etc.), you can use the table-handling routines provided by the API.

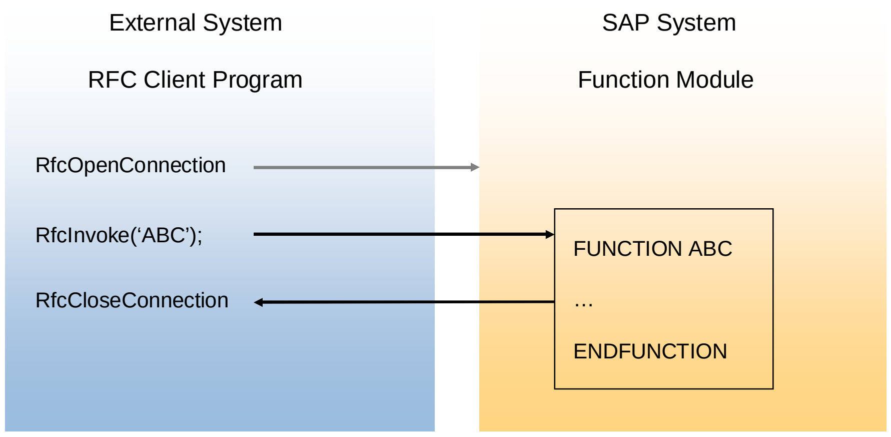
Further Information

You can find a description of all functions used for passing parameters in the

- NW RFC SDK Function Reference

RFC Client Programs

All RFC client programs have to establish an RFC connection to an SAP system:



Getting Connected with the RFC Library

- The connection can be established via an entry in the sapnwrfc.ini file.
- An RFC function is called by **`RfcInvoke`**. After sending the call, **`RfcInvoke`** waits until the returned answer will be received.

Calling a Function in SAP Systems

In the following example program an external RFC client connects to an SAP system and calls the function module `BAPI_COMPANY_GETDETAIL`:

```
#include <stdlib.h>
#include <stdio.h>
#include "sapnwrfc.h"

void errorHandling(RFC_RC rc, SAP_UC description[], RFC_ERROR_INFO*
errorInfo, RFC_CONNECTION_HANDLE connection){
   printfU(cU("%s: %d\n"), description, rc);
   printfU(cU("%s: %s\n"), errorInfo->key, errorInfo->message);

/* It's better to close the TCP/IP connection cleanly, than to
just let the backend get a "Connection reset by peer" error...*/

   if (connection != NULL) RfcCloseConnection(connection, errorInfo);
   exit(1);
}

int mainU(int argc, SAP_UC** argv){
   RFC_RC rc = RFC_OK;
   RFC_CONNECTION_PARAMETER loginParams[6];
   RFC_ERROR_INFO errorInfo;
   RFC_CONNECTION_HANDLE connection;
   RFC_FUNCTION_DESC_HANDLE bapiCompanyDesc;
   RFC_FUNCTION_HANDLE bapiCompany;
   RFC_STRUCTURE_HANDLE returnStructure;
   SAP_UC message[221];
   RFC_BYTE buffer[1105];
   unsigned utf8Len = 1105, resultLen;
   FILE* outFile;

-----------------------------------------------
/*OPEN CONNECTION*/
-----------------------------------------------

/*Create logon parameter list*/

   loginParams[0].name = cU("ashost"); loginParams[0].value =
cU("hs0023");
   loginParams[1].name = cU("sysnr");  loginParams[1].value =
cU("05");
   loginParams[2].name = cU("client"); loginParams[2].value =
cU("800");
   loginParams[3].name = cU("user");   loginParams[3].value =
cU("alice");
   loginParams[4].name = cU("lang");   loginParams[4].value =
cU("EN");
   loginParams[5].name = cU("passwd"); loginParams[5].value =
cU("secret");

/*Open connection*/

   connection = RfcOpenConnection(loginParams, 6, &errorInfo);
   if (connection == NULL) errorHandling(rc, cU("Error during
logon"),
&errorInfo, NULL);
```

```
----------------------------------------------
/* DYNAMIC METADATA retrieval (can also be executed using STATIC
functions)*/
----------------------------------------------

   bapiCompanyDesc = RfcGetFunctionDesc(connection,
cU("BAPI_COMPANY_GETDETAIL"), &errorInfo);
   if (bapiCompanyDesc == NULL) errorHandling(rc, cU("Error during
metadata lookup"), &errorInfo, connection);

----------------------------------------------
/*FUNCTION CALL/*
----------------------------------------------

/*Create function instance*/

bapiCompany = RfcCreateFunction(bapiCompanyDesc, &errorInfo);

   /*Parameter setting*/

   RfcSetChars(bapiCompany, cU("COMPANYID"), cU("000007"), 6,
&errorInfo);

   /*Call*/

   rc = RfcInvoke(connection, bapiCompany, &errorInfo);
   if (rc != RFC_OK) errorHandling(rc, cU("Error calling
BAPI_COMPANY_GETDETAIL"), &errorInfo, connection);

   /*Getting parameters*/

   RfcGetStructure(bapiCompany, cU("RETURN"), &returnStructure,
&errorInfo);
   RfcGetString(returnStructure, cU("MESSAGE"), message, 221,
&resultLen, &errorInfo);

   /*After having finished the procedure: release memory only if a
‘create’ function was invoked before*/

   RfcDestroyFunction(bapiCompany, &errorInfo);
   /*Final UTF8-UTF16 conversion*/

   windows.h utf8Len = WideCharToMultiByte(CP_UTF8, 0, message,
strlenU(message), buffer, 1105, NULL, NULL);
   RfcSAPUCToUTF8(message,  strlenU(message), buffer, &utf8Len,
&resultLen, &errorInfo);

   outFile = fopen("message.xml", "w");
   fputs("<?xml version=\"1.0\"?>\n<message>", outFile);
   fputs(buffer, outFile);
   fputs("</message>", outFile);
   fclose(outFile);
   return 0;
}
```

Starting a SAP GUI

When calling a function module in the SAP system it may be necessary to start a SAP GUI on your client.

Some (older) BAPIs need this, because they try to send screen output to the client while executing.

If you want to start a SAP GUI on an external client, your SAP backend system must meet some requirements. You will find detailed information in SAP note **1258724**.

To start a SAP GUI from your client program, proceed as follows:

Windows:

Set the *property* `USE_SAPGUI` to 1 (hidden) oder 2 (visible).

Prerequisite: You have installed the Windows SAP GUI on your system.

Possible values are:

0: no SAPGUI (default)

1: attach a "hidden" SAPGUI, which just receives and ignores the screen output

2: attach a visible SAPGUI.

Unix:

For Unix systems a Java SAP GUI is required.

Set the envrionment variable via:

```
setenv SAPGUI <pfad zum SAPGUI startskript> (tcsh) oder

set SAPGUI=<pfad zum SAPGUI startskript> (bash)
```

For Mac OS the path name could look like this:

```
"/Applications/SAP Clients/SAPGUI 7.10rev7.3/SAPGUI
7.10rev7.3.app/Contents/MacOS/SAPGUI"
```

For Linux and other Unix systems the following path would be valid:

```
/opt/SAPClients/SAPGUI7.10rev7.3/bin/sapgui
```

You must not add *any* parameter after the script name of the environment variable.

Start the external application from the *same* shell to propagate the environment variable to the program.

Passing Structures and Tables

This programming example shows how to pass structures and tables from the ABAP function module STFC_STRUCTURE to an external RFC Client program:

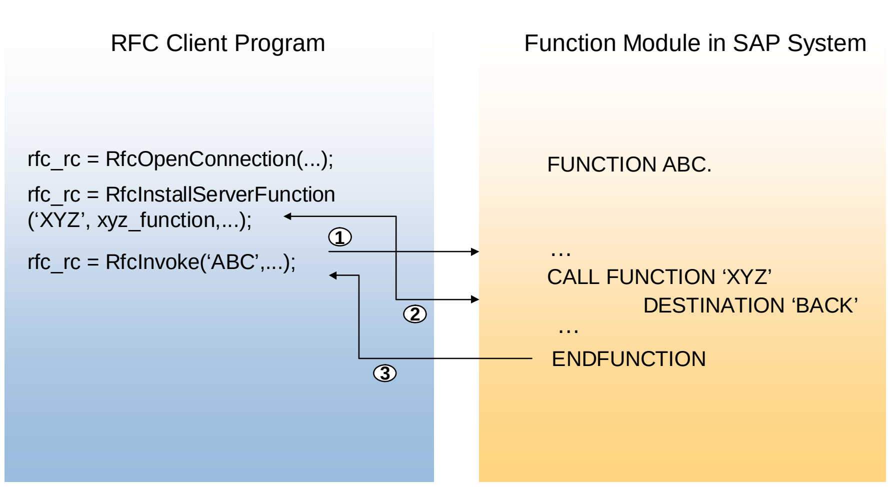
```
RFC_RC STFC_STRUCTURE(RFC_CONNECTION_HANDLE rfcHandle,
RFC_FUNCTION_DESC_HANDLE funcDesc)
{
    RFC_ERROR_INFO error;
    RFC_RC rc = RFC_OK;

    RFC_FLOAT test_float = 12345.6789, veri_float;
    RFC_INT test_int = 12345, veri_int;
    RFC_BYTE test_rfchex3[3] = {0x41, 0x42, 0x43};  // "ABC"
    RFC_BYTE veri_rfchex3[3] = {0, 0, 0};
    const SAP_UC* test_data1 =  cU("Hello World");
    SAP_UC veri_data1[50 + 1] = iU("");



/*Create a function handle*/

    RFC_FUNCTION_HANDLE funcHandle = NULL;
    funcHandle = RfcCreateFunction(funcDesc, &error);


/*Create a structure handle: do not create using RfcCreateStructure,
but just call*/

    RFC_STRUCTURE_HANDLE structHandle = NULL;
    rc = RfcGetStructure(funcHandle,cU("IMPORTSTRUCT"),
&structHandle, &error);

    RfcSetFloat(structHandle, cU("RFCFLOAT"), test_float, 0);
    RfcGetFloat(structHandle, cU("RFCFLOAT"), &veri_float, 0);

    RfcSetInt(structHandle, cU("RFCINT4"), 12345, 0);
    RfcGetInt(structHandle, cU("RFCINT4"), &veri_int, 0);

    RfcSetBytes(structHandle, cU("RFCHEX3"), test_rfchex3,
sizeofR(test_rfchex3), 0);
    RfcGetBytes(structHandle, cU("RFCHEX3"), veri_rfchex3,
sizeofR(veri_rfchex3), 0);
```

```
    RfcSetChars(structHandle, cU("RFCDATA1"), test_data1,
strlenU(test_data1), 0);
    RfcGetChars(structHandle, cU("RFCDATA1"), veri_data1,
sizeofU(veri_data1), 0);



/*Create a structure handle: do not create using RfcCreateTable, but
just call*/

    RFC_TABLE_HANDLE tableHandle = NULL;
    rc = RfcGetTable(funcHandle,cU("RFCTABLE"), &tableHandle, 0);

    for(unsigned i = 0; i < count; i++)
    {
       RfcAppendRow(tableHandle, structHandle, 0);
    }

    SAP_UC buffer[256] = iU("");
    printfU( cU("RFC test with STFC_STRUCTURE\n"));
    RfcInvoke(rfcHandle, funcHandle, &error);

    RfcGetChars(funcHandle, cU("RESPTEXT"), buffer, 255, 0);
    EndTrim(buffer, cU(' '), 255);
    printfU( cU("RESPTEXT : %s\n"), buffer);

    RfcGetChars(funcHandle, cU("RESPTEXT"), buffer, 255, 0);

    structHandle = NULL;
    RfcGetStructure(funcHandle, cU("ECHOSTRUCT"), &structHandle, 0);
    RfcGetFloat(structHandle, cU("RFCFLOAT"), &veri_float, 0);
    RFC_INT rfcint4 = 0;
      RfcGetInt(structHandle, cU("RFCINT4"), &veri_int, 0);
      RfcGetBytes(structHandle, cU("RFCHEX3"), veri_rfchex3,
sizeofR(veri_rfchex3), 0);
      RfcGetChars(structHandle, cU("RFCDATA1"), veri_data1,
sizeofU(veri_data1)-1, 0);
      EndTrim(veri_data1, cU(' '), sizeofU(veri_data1));

      printfU( cU("ECHOSTRUCT.RFCFLOAT : %f\n") ,veri_float);;
      printfU( cU("ECHOSTRUCT.RFCINT4 : %d\n") , veri_int );
      printfU( cU("ECHOSTRUCT.RFCHEX3 : %p\n") ,veri_rfchex3 );
      printfU( cU("ECHOSTRUCT.RFCDATA1 : %s\n") , veri_data1);

        tableHandle = NULL;
        RfcGetTable(funcHandle, cU("RFCTABLE"), &tableHandle, 0);
      RfcGetRowCount(tableHandle, &count, 0);  //<-- moving table
cursor
      RfcMoveToFirstRow(tableHandle, 0);
      for(unsigned i = 0; i < count; i++)
      {
         RFC_STRUCTURE_HANDLE row = NULL;
                  row = RfcGetCurrentRow(tableHandle, 0);
         RfcGetChars(row, cU("RFCDATA1"), veri_data1,
sizeofU(veri_data1), 0);
         EndTrim(veri_data1, cU(' '), sizeofU(veri_data1));

         printfU( cU("RFCTABLE[%d].RFCDATA1 : %s\n"),i,
veri_data1 );
         RfcMoveToNextRow(tableHandle, 0); //<-- moving table
cursor
      }

   /*Clean up*/

    RfcDestroyFunction(funcHandle, 0);
   return rc;
}
```

If a table is too large to be transmitted with a single request, only a suitable data portion will be transmitted by the original request. In this case, your application coding must provide the appropriate handling of table transfer in several steps.

Executing a Call-Back from an ABAP Function Module

The NW RFC API generally enables call-backs from RFC servers, both external and SAP. The following scheme shows you how to use this feature for a call-back from the ABAP server:


RFC Client Program
rfc_rc = RfcOpenConnection(...);
rfc_rc = RfcInstallServerFunction
('XYZ', xyz_function,...);
rfc_rc = RfcInvoke('ABC',...);
①
②
③
Function Module in SAP System
FUNCTION ABC.
…
CALL FUNCTION 'XYZ'
DESTINATION 'BACK'
…
ENDFUNCTION

1. The external RFC client calls a function in the ABAP system
2. The ABAP function module contains another remote function call with destination *BACK*. Within the RFC client, the call-back is directed to the calling program that returns the requested data. This process can be repeated as often as required.
3. After the ABAP function originally called has finished the ABAP function sends a confirmation to the RFC client.

Load Balancing

An RFC client program can call a function module in an SAP system without specifying the application server for establishing the connection, so you can make use of the Load Balancing feature.

This is done by means of the function **`RfcOpenConnection`**: The system first builds up a connection to the Message Server of the SAP system and tries to find the application server with the least load (*load balancing principle*). On the basis of this information, the RFC library internally builds up the connection to the selected application server.

This load balancing feature has the following advantages:

- The load in an SAP system is distributed to different SAP application servers. The RFC connection is always established to an application server with the least load.
- Using load balancing, the RFC server will be determined at run time from the application servers available. Therefore, RFC connections are independent of a specific application server.
- Only the host name of the SAP message server and its port number are required in the *hosts* and *services* file. Information about the SAP Gateway, application server, system number for the **`RfcOpenConnection`** entry and entries for the application server and SAP Gateway are no longer required there.

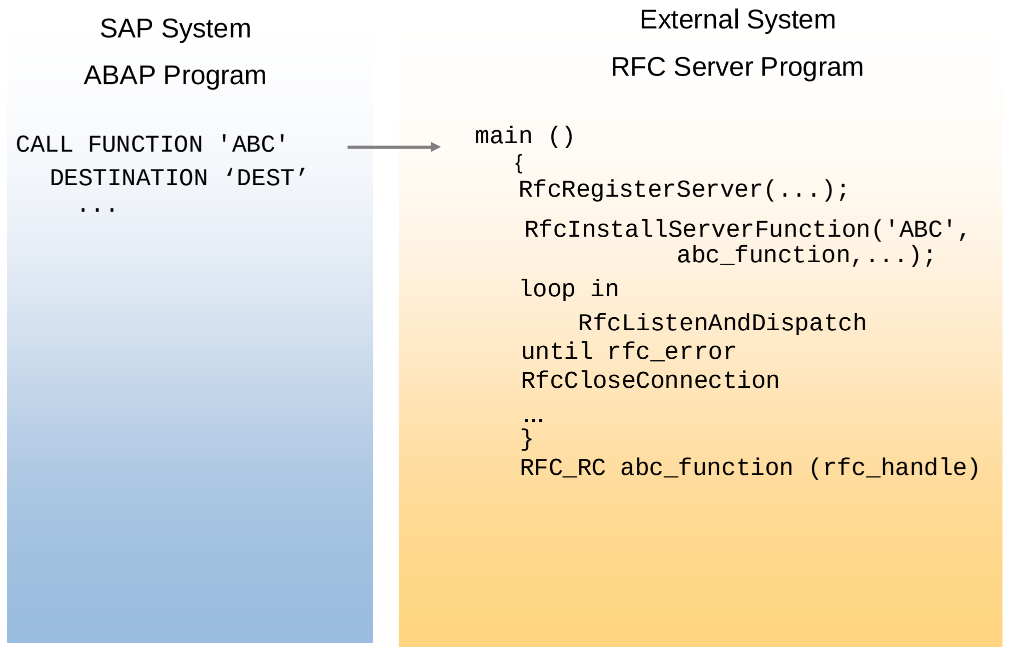
RFC Server Programs

An RFC server program is a program containing RFC functions to be called by ABAP programs.

The RFC API provides routines for implementing RFC server programs. After having started, the RFC server programs must inform the RFC library about all RFC functions that can be called within this server program. It then waits for incoming call requests, and the RFC library dispatches the requested calls (using **`RfcListenAndDispatch`** in a loop).

A typical server looks as follows:

RFC Server Program



```
CALL FUNCTION 'ABC'
   DESTINATION 'DEST'
     ...
```

```
main ()
   {
    RfcRegisterServer(...);
    RfcInstallServerFunction('ABC',
                 abc_function,...);
    loop in
        RfcListenAndDispatch
    until rfc_error
    RfcCloseConnection
    …
    }
    RFC_RC abc_function (rfc_handle)
```

There are two methods of receiving an RFC call. The most simple way of receiving an RFC call in an external program is to register a C function to be called when a call request is received. The function **`RfcInstallServerFunction`** registers a C function to be called when receiving the request for an RFC call. After **`RfcRegisterServer`** (**`RfcStartServer`**), the program must use **`RfcListenAndDispatch`** to internally call the corresponding registered function. The return code of the registered function is again returned by **`RfcListenAndDispatch`**.

There are always some standard functions which are installed automatically. Apart from some internally-used functions, the function modules are as follows:

- RFC_PING

  Is used to test the connection.

- RFC_SYSTEM_INFO

  Returns some information about the library and its environment.

- RFC_DOCU

Returns the function documentation which was installed during the calls of **`RfcInstallServerFunction`**.

Further Information

Information on security issues you can find in the SAP Library:

- RFC/ICF Security Guide

Establishing an RFC Connection from an SAP System

Call from an ABAP program

To call an external RFC server from an ABAP program you generally use the following statement:

```
CALL FUNCTION “ABC” DESTINATION “RFCEXTERNAL”

   IMPORT...
   EXPORT...
   TABLES...
   EXCEPTIONS...
```

RFC Destination

The destination `RFCEXTERNAL` identifies an entry in the table `RFCDES`.

The AS ABAP can find the connection parameter only if the destination has been stored and configured in the system via transaction SM59.

Connection Options

There are basically two different ways to establish a connection between an SAP system and an external RFC server. You can use:

- A registered RFC server
- A server started by the application server or by an SAP Gateway.

Registered RFC Server

If you want to run the RFC server as a registered server, the destination must be specified as *Registered RFC Server* with a corresponding program ID.

Program ID

The program ID is an identifier of the RFC server program for the SAP Gateway. When defining an entry via transaction SM59 you should specify the complete name of the RFC server program (including the full path name).

It is recommended to use both the name of the RFC server program and the host name of the RFC server program.

It is also recommended to define the SAP Gateway explicitly, because an RFC server program usually registers at a specific SAP Gateway. If nothing is specified the SAP Gateway of the relevant application server will be used.

Started RFC Server

As an alternative to the registered RFC server the RFC server program can also be started by the currently running application server or by an SAP Gateway. In both cases it has to communicate via a specified SAP Gateway. Consequently, the following prerequisites must be met:

- The user under which the application server or the SAP Gateway runs must have access rights for the RFC server program.
- Both SAP Gateway and RFC server program are running on the same computer:
  - The IP address of this computer must be specified in the *hosts* file.
  - The service name of the SAP Gateway must be specified in the *services* file.
- The SAP Gateway and the RFC server program are running on different computers:

- The IP addresses of both computers must be specified in both *hosts* files.
- The service name of the SAP Gateway must be specified in the *services* file.
- The SAP Gateway must be authorized to start the RFC server program on the target computer via remote shell:
    i. The user of the SAP Gateway must be defined on the target computer.
    ii. The.*rhosts* file which contains the host name of the gateway computer must exist in this user's home directory on the target computer
    iii. Since the remote shell command is different on different UNIX platforms (remsh, rsh, etc.), the command can be defined in the gateway profile parameter gw/remsh, if necessary (e.g. gw/remsh=/usr/ucb/remsh). The default is '*remsh*'.

Registering Server Programs on the SAP Gateway

An RFC server program can be registered with the SAP Gateway and wait for incoming RFC call requests.

This new registering feature has the following features:

- An RFC server program registers itself under a *program ID* at an SAP Gateway and *not* for a specific SAP system.
- In an *SAP system*, the destination must be defined with transaction SM59, using connection type *T* and *Register Mode*. Moreover, this entry must contain information on the SAP Gateway at which the RFC server program is registered.
- After having executed an RFC function, the RFC connection will be closed. If this RFC server program works with **`RfcListenAndDispatch`** in a loop (this procedure is recommended), it will be automatically registered again at the same SAP Gateway under the same program ID and can then wait for further RFC call requests from the same SAP system or from other SAP systems.

Procedure

AS ABAP

The following properties must be specified for the RFC destination in transaction SM59:

- Connection Type *T*
- *Register Mode*

RFC Server

As a registered server the RFC server uses the initial function `RfcRegisterServer`.

If the RFC server is started by the application server, it must use the function `RfcStartServer`.

Security

If an RFC server is registered on an RFC gateway, it is generally possible to send calls from other SAP systems (not relevant to this gateway) or from external RFC clients to this server.

If, for security reasons, the server should only be able to be called by specified systems or users, the server must implement its own logon data check and reject unwanted initiators.

In contrast to the ‘classic’ RFC SDK (see SAP note **934507**) you would not implement this authorization check as a general callback function (which would be executed for every function call – if required or not) but as an inherent element of a specified function. Like this, the logon check will only be performed if needed.

A typical authorization check within a specified function could look like this:

```
RFC_RC SAP_API myServerFunction(RFC_CONNECTION_HANDLE rfcHandle,
            RFC_FUNCTION_HANDLE funcHandle, RFC_ERROR_INFO*
errorInfo){
    RFC_RC rc;
    RFC_ATTRIBUTES attributes;
    RFC_FUNCTION_DESC_HANDLE metadata;
    RFC_ABAP_NAME funcName;
    int denied = 1;

    /* Error checking omitted for simplicity: */
    rc = RfcGetConnectionAttributes(rfcHandle, &attributes,
errorInfo);
    metadata = RfcDescribeFunction(funcHandle, errorInfo);
    rc = RfcGetFunctionName(metadata, funcName, errorInfo);
    /* Now based on the values of attributes.user, attributes.sysId,
attributes.client,
       attributes.partnerHost and funcName you can decide, whether
this user shall be
       allowed to execute the given function module or not. denied =
0/1.
    */

    if (denied){
        strcpyU(errorInfo.message, cU("Access denied"));
        errorInfo.code = RFC_EXTERNAL_FAILURE;
        return RFC_EXTERNAL_FAILURE;
    }

    /* Otherwise continue with the function module processing */

    return RFC_OK;
}
```

Further Information

You can find general information on security issues of the RFC API in the SAP library:

- SAP NetWeaver Security Guide:

  Security Guides for Connectivity and Interoperability Technologies ->

  - RFC/ICF Security Guide
  - Security Settings in the SAP Gateway

RFC Server Programming Example

In the following section you can find a programming example using basic elements of RFC Server programs like registering the server, installing a server function and error handling:

```
#include <stdlib.h>
#include <stdio.h>
#include "sapnwrfc.h"

static int listening = 1;

/*
Unfortunately SE37 does not yet allow to set complex inputs, so in order to test
this example, you will probably need to write a little ABAP report, which sets a few
input lines for IMPORT_TAB and then does a

CALL FUNCTION 'STFC_DEEP_TABLE' DESTINATION 'MY_SERVER'
  EXPORTING
    import_tab      = imtab
  IMPORTING
    export_tab      = extab
    resptext        = text
  EXCEPTIONS
    system_failure  = 1  MESSAGE mes.

This also allows to catch the detail texts for SYSTEM_FAILURES.
Note: STFC_DEEP_TABLE exists only from SAP_BASIS release 6.20 on.
*/

void errorHandling(RFC_RC rc, SAP_UC description[], RFC_ERROR_INFO* errorInfo,
RFC_CONNECTION_HANDLE connection){
	printfU(cU("%s: %d\n"), description, rc);
	printfU(cU("%s: %s\n"), errorInfo->key, errorInfo->message);
	// It's better to close the TCP/IP connection cleanly, than to just let the
	// backend get a "Connection reset by peer" error...
	if (connection != NULL) RfcCloseConnection(connection, errorInfo);

	exit(1);
}
```

```
RFC_RC SAP_API stfcDeepTableImplementation(RFC_CONNECTION_HANDLE rfcHandle,
RFC_FUNCTION_HANDLE funcHandle, RFC_ERROR_INFO* errorInfoP){
    RFC_ATTRIBUTES attributes;
    RFC_TABLE_HANDLE importTab = 0;
    RFC_STRUCTURE_HANDLE tabLine = 0;
    RFC_TABLE_HANDLE exportTab = 0;
    RFC_ERROR_INFO errorInfo ;
    RFC_CHAR buffer[257]; //One for the terminating zero
    RFC_INT intValue;
    RFC_RC rc;
    unsigned tabLen = 0, strLen;
    unsigned  i = 0;
    buffer[256] = 0;

    printfU(cU("\n*** Got request for STFC_DEEP_TABLE from the following system: ***\n"));
```

/*If you want to include an authorization check for your RFC server, insert it here (see also the section *Security* in Registering Server Programs on the SAP Gateway):*/

```
    RfcGetConnectionAttributes(rfcHandle, &attributes, &errorInfo);
    printfU(cU("System ID: %s\n"), attributes.sysId);
    printfU(cU("System No: %s\n"), attributes.sysNumber);
    printfU(cU("Mandant  : %s\n"), attributes.client);
    printfU(cU("Host     : %s\n"), attributes.partnerHost);
    printfU(cU("User     : %s\n"), attributes.user);

    //Print the Importing Parameter
    printfU(cU("\nImporting Parameter:\n"));
    RfcGetTable(funcHandle, cU("IMPORT_TAB"), &importTab, &errorInfo);

    RfcGetRowCount(importTab, &tabLen, &errorInfo);
    printfU(cU("IMPORT_TAB (%d lines)\n"), tabLen);
    for (i=0; i<tabLen; i++){
        RfcMoveTo(importTab, i, &errorInfo);
        printfU(cU("\t\t-line %d\n"), i);

        RfcGetInt(importTab, cU("I"), &intValue, &errorInfo);
        printfU(cU("\t\t\t-I:\t%d\n"), intValue);
        RfcGetString(importTab, cU("C"), buffer, 11, &strLen, &errorInfo);
        printfU(cU("\t\t\t-C:\t%s\n"), buffer);
        // Check for the stop flag:
        if (i==0 && strncmpU(cU("STOP"), buffer, 4) == 0) listening = 0;
        RfcGetStringLength(importTab, cU("STR"), &strLen, &errorInfo);
        if (strLen > 256) printfU(cU("STRING length bigger than 256: %d. Omitting the STR
field...\n"), strLen);
        else{
            RfcGetString(importTab, cU("STR"), buffer, 257, &strLen, &errorInfo);
            printfU(cU("\t\t\t-STR:\t%s\n"), buffer);
        }
        RfcGetStringLength(importTab, cU("XSTR"), &strLen, &errorInfo);
        if (strLen > 128) printfU(cU("XSTRING length bigger than 128: %d. Omitting the
XSTR field...\n"), strLen);
        else{
            RfcGetString(importTab, cU("XSTR"), buffer, 257, &strLen, &errorInfo);
            printfU(cU("\t\t\t-XSTR:\t%s\n"), buffer);
        }
    }

    //Now set the Exporting Parameters
    printfU(cU("\nSetting values for Exporting Parameters:\n"));
    printfU(cU("Please enter a value for RESPTEXT:\n> "));
    getsU(buffer);
    RfcSetChars(funcHandle, cU("RESPTEXT"), buffer, strlenU(buffer), &errorInfo);
    printfU(cU("\nPlease enter the number of lines in EXPORT_TAB:\n> "));
    getsU(buffer);
    tabLen = atoiU(buffer);
    RfcGetTable(funcHandle, cU("EXPORT_TAB"), &exportTab, &errorInfo);
    for (i=0; i<tabLen; i++){
        tabLine = RfcAppendNewRow(exportTab, &errorInfo);
        printfU(cU("Line %d\n"), i);
        printfU(cU("\tPlease enter a value for C [CHAR10]:> "));
        getsU(buffer);
        RfcSetChars(tabLine, cU("C"), buffer, strlenU(buffer), &errorInfo);
        printfU(cU("\tPlease enter a value for I [INT4]:> "));
        getsU(buffer);
        RfcSetInt(tabLine, cU("I"), atoiU(buffer), &errorInfo);
        printfU(cU("\tPlease enter a value for STR [STRING]:> "));
        fgetsU(buffer, 257, stdin); // For these fields better make sure, the user doesn't bust
our buffer...
        strLen = strlenU(buffer) - 1;
        // In contrast to gets, fgets includes the linebreak... Very consistent...
        RfcSetString(tabLine, cU("STR"), buffer, strLen, &errorInfo);
        mark: printfU(cU("\tPlease enter a value for XSTR [XSTRING]:> "));
        fgetsU(buffer, 257, stdin);
        strLen = strlenU(buffer) - 1;
        // In contrast to gets, fgets includes the linebreak... Very consistent...
        rc = RfcSetString(tabLine, cU("XSTR"), buffer, strLen, &errorInfo);
        if (rc != RFC_OK){
            printfU(cU("\tInvalid value for XSTR. Please only use hex digits 00 - FF.\n"));
            goto mark;
        }
    }
    printfU(cU("**** Processing of STFC_DEEP_TABLE finished ***\n\n"));

    return RFC_OK;
}

int mainU(int argc, SAP_UC** argv){
    RFC_RC rc;
    RFC_FUNCTION_DESC_HANDLE stfcDeepTableDesc;
    RFC_CONNECTION_PARAMETER repoCon[8], serverCon[3];
    RFC_CONNECTION_HANDLE repoHandle, serverHandle;
    RFC_ERROR_INFO errorInfo;

    serverCon[0].name = cU("program_id"); serverCon[0].value = cU("MY_SERVER");
    serverCon[1].name = cU("gwhost");  serverCon[1].value = cU("binmain");
    serverCon[2].name = cU("gwserv");  serverCon[2].value = cU("sapgw53");

    repoCon[0].name = cU("client"); repoCon[0].value = cU("000");
    repoCon[1].name = cU("user");   repoCon[1].value = cU("user");
    repoCon[2].name = cU("passwd");    repoCon[2].value = cU("****");
    repoCon[3].name = cU("lang");   repoCon[3].value = cU("DE");
    repoCon[4].name = cU("ashost");      repoCon[4].value = cU("binmain");
    repoCon[5].name = cU("sysnr"); repoCon[5].value = cU("53");

    printfU(cU("Logging in..."));
    repoHandle = RfcOpenConnection (repoCon, 6, &errorInfo);
    if (repoHandle == NULL) errorHandling(errorInfo.code, cU("Error in
RfcOpenConnection()"), &errorInfo, NULL);
    printfU(cU(" ...done\n"));
    printfU(cU("Fetching metadata..."));
    stfcDeepTableDesc = RfcGetFunctionDesc(repoHandle, cU("STFC_DEEP_TABLE"),
&errorInfo);
    // Note: STFC_DEEP_TABLE exists only from SAP_BASIS release 6.20 on
    if (stfcDeepTableDesc == NULL) errorHandling(errorInfo.code, cU("Error in Repository
Lookup"), &errorInfo, repoHandle);
    printfU(cU(" ...done\n"));


    printfU(cU("Logging out..."));
    RfcCloseConnection(repoHandle, &errorInfo);
    printfU(cU(" ...done\n"));


    rc = RfcInstallServerFunction(NULL, stfcDeepTableDesc, stfcDeepTableImplementation,
&errorInfo);
    if (rc != RFC_OK) errorHandling(rc, cU("Error Setting "), &errorInfo, repoHandle);


    printfU(cU("Registering Server..."));
    serverHandle = RfcRegisterServer(serverCon, 3, &errorInfo);
    if (serverHandle == NULL) errorHandling(errorInfo.code, cU("Error Starting RFC Server"),
&errorInfo, NULL);
    printfU(cU(" ...done\n"));


    printfU(cU("Starting to listen...\n\n"));
    while(RFC_OK == rc || RFC_RETRY == rc || RFC_ABAP_EXCEPTION == rc){
        rc = RfcListenAndDispatch(serverHandle, 120, &errorInfo);
        printfU(cU("RfcListenAndDispatch() returned %s\n"), RfcGetRcAsString(rc));
        switch (rc){
            case RFC_RETRY: // This only notifies us, that no request came in within the
timeout period.
                                    // We just continue our loop.
                printfU(cU("No request within 120s.\n"));
                break;
            case RFC_ABAP_EXCEPTION: // Our function module implementation has
returned RFC_ABAP_EXCEPTION.
                                        // This is equivalent to an ABAP function module throwing an
ABAP Exception.
                                        // The Exception has been returned to R/3 and our
connection is still open.
                                        // So we just loop around.
                printfU(cU("ABAP_EXCEPTION in implementing function: %s\n"),
errorInfo.key);
                break;
            case RFC_NOT_FOUND:    // R/3 tried to invoke a function module, for which we
did not supply
                                    // an implementation. R/3 has been notified of this through a
SYSTEM_FAILURE,
                                    // so we need to refresh our connection.
                printfU(cU("Unknown function module: %s\n"), errorInfo.message);
            case RFC_EXTERNAL_FAILURE:   // Our function module implementation raised
a SYSTEM_FAILURE. In this case
                                    // the connection needs to be refreshed as well.
                printfU(cU("SYSTEM_FAILURE has been sent to backend.\n\n"));
            case RFC_ABAP_MESSAGE:         // And in this case a fresh connection is
needed as well
                serverHandle = RfcRegisterServer(serverCon, 3, &errorInfo);
                rc = errorInfo.code;
                break;
        }

        // This allows us to shutdown the RFC Server from R/3. The implementation of
STFC_DEEP_TABLE
        // will set listening to false, if IMPORT_TAB-C == STOP.
        if (!listening){
            RfcCloseConnection(serverHandle, NULL);
            break;
        }
    }

    return 0;
}
```

Implemtenting an Auto-Reconnect Mechanism

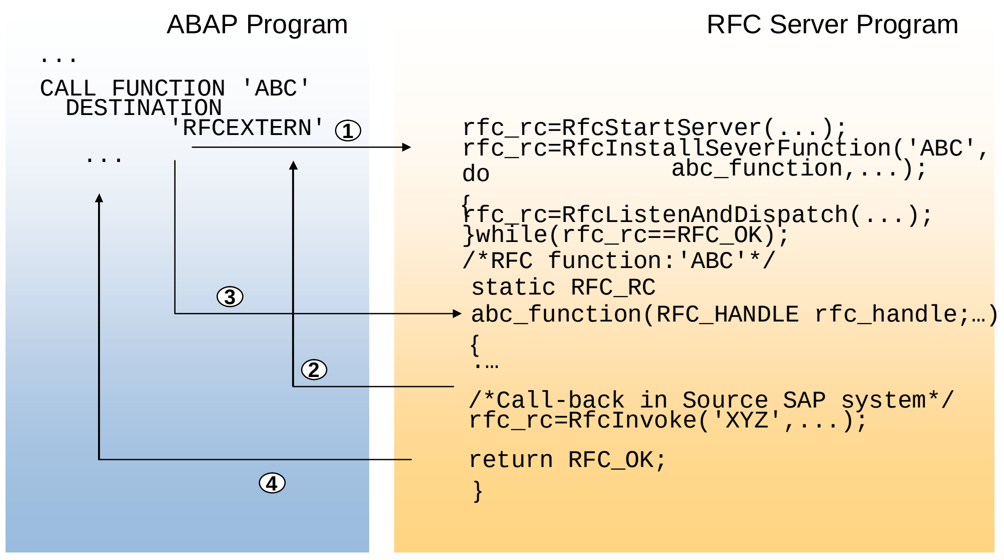
If the server connection has been interrupted for any reason you can implement an auto-reconnect mechanism for registering a new connection at the SAP gateway. This mechanism could look like this:

```
int main (int argc, char** agrv)
{

RFC_HANDLE hRfc = NULL;
RFC_RC rc = RFC_OK;
bool autoReconnect = true;

Reconnect:
hRfc = NULL;

// register RFC server connection
while(!hRfc)
{
     hRfc = RfcRegisterServer( argv);

//have a wait for 1 or more seconds to avoid taking too much CPU away in case that the gateway is
temporarily down
    sleep(...);
}
....

while(RFC_OK ==rc || RFC_RETRY==rc || RFC_EXCEPTION==rc)
{
    rc = RfcListenAndDispatch(hRfc);
}

// The connection has been broken when we reach here

if(autoReconnect)
     goto Reconnect;

return rc;

}
```

The time between two calls of `RfcREgisterServer` can be made configurable according to your needs.

Executing a Call-Back from an RFC Server Program

The NW RFC API generally enables call-backs from RFC servers, both external and SAP. The following scheme shows you how to use this feature from the external RFC server's perspective:



```
...
CALL FUNCTION 'ABC'
  DESTINATION
          'RFCEXTERN'
   ...
```

```
rfc_rc=RfcStartServer(...);
rfc_rc=RfcInstallSeverFunction('ABC',
do              abc_function,...);
{
rfc_rc=RfcListenAndDispatch(...);
}while(rfc_rc==RFC_OK);
/*RFC function:'ABC'*/
 static RFC_RC
 abc_function(RFC_HANDLE rfc_handle;…)
 {
 .…
 /*Call-back in Source SAP system*/
 rfc_rc=RfcInvoke('XYZ',...);
 return RFC_OK;
 }
```

1. An ABAP function module calls a function in the RFC server program. Function module 'XYZ' is any RFC-supported function module in this source SAP system.
2. The RFC server function contains a call-back to the ABAP client. The call-back is directed to the calling program by the ABAP runtime.
3. The ABAP program returns the requested data. This process can be repeated as often as required.
4. After the RFC server function has finished, a return code is sent back to the ABAP system.

tRFC between SAP and external Systems

On external systems, the transactional RFC cannot be fully implemented in the RFC library, because of the following reasons:

- A database is not always available in external systems.
- The RFC library cannot always repeat the RFC call in case of errors, such as network errors.

Therefore, the transactional RFC interface from external systems to an SAP system is implemented as follows:

- **RFC library**

  The RFC library provides some special RFC calls, such as **`RfcGetTransactionID`**, **`RfcInvokeInTransaction`** and **`RfcSetTransactionHandler`** for working with tRFC. It will pack and unpack RFC data between RFC format and tRFC format.

  For an SAP system, there is no difference whether these calls are requested from another SAP system or from an external system. For an RFC server program, the RFC function itself (only the RFC function, not the whole RFC server program) can be executed normally, as it is called via ‘normal’ RFC with RfcGetData and RfcSendData.

- **RFC client programs and RFC server programs**

  Both programs have to manage the TIDs themselves for checking and executing the requested RFC functions **exactly once** as the tRFC component in an SAP system does.

- **SAP systems**

  In SAP systems, no additional changes are necessary in ABAP programs working with external RFC programs which use the tRFC interface. For ABAP programs, such as RFC client programs, the destination defined in CALL FUNCTION must have **‘T’** as connection type.

  As of AS ABAP Release 710 you can use the re-designed variant of tRFC and qRFC, now called background RFC (bgRFC) for the RFC communication. The NW RFC SDK also supports bgRFC. However, the NW RFC SDK itself treats a bgRFC call like a tRFC call. Within the SDK there is no visible difference between tRFC/qRFC and bgRFC processing.

Transactional RFC Client Programs

After having been connected to an SAP system (via **`RfcOpenConnection`**), an RFC client program must use the following two RFC calls for working with the tRFC interface:

- **RfcGetTransactionID**

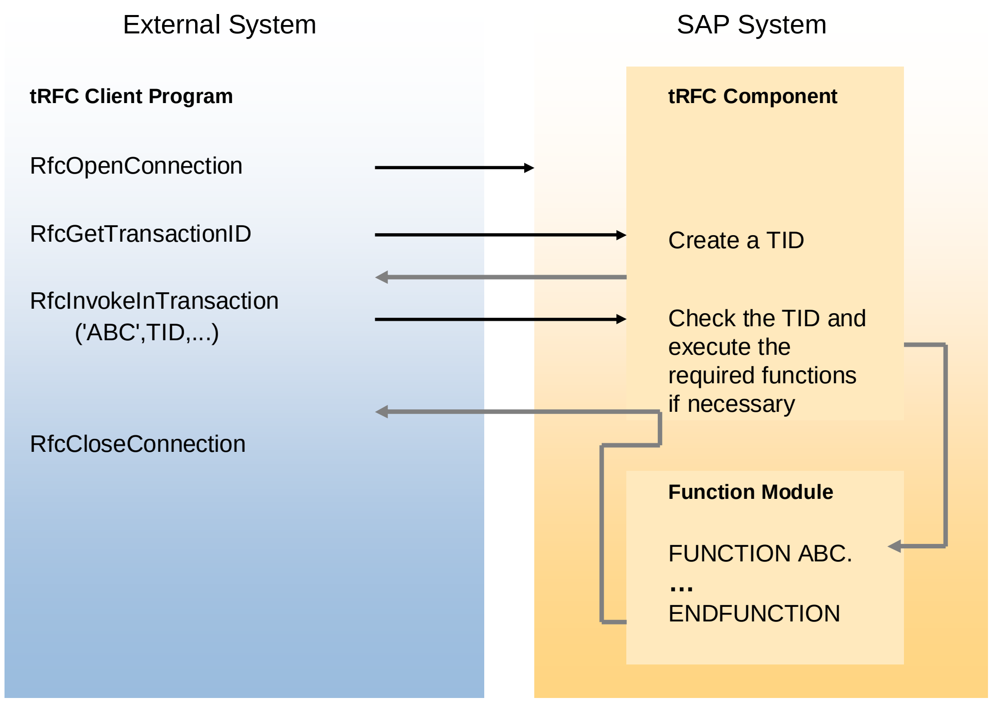
  With this call, the RFC library tries to get a TID created by the SAP system. If errors occur, the RFC client program has to reconnect later and must try to repeat this call. Otherwise, the RFC client program can assign this TID with the RFC data, and if the next call is not successful, it can be repeated later.

- **RfcInvokeInTransaction**

  With this call, the RFC library will pack all RFC data belonging to an RFC function together with the TID and send them to the SAP system using the tRFC protocol.
  If an error occurs, the RFC client program has to reconnect later and must try to repeat this RFC call (**`RfcInvokeInTransaction`**). In this case, it has to use the old TID and *must not* get a new TID with **`RfcGetTransactionID`**. Otherwise, there is no guarantee that this RFC function will be executed *exactly once* in the SAP system.
  After this call is executed successfully, the transaction will be completed once and for all. The RFC client program can then update its own TID management (e.g. delete the TID entry).

Contrary to tRFC between SAP systems, a transaction from an RFC client program contains *only one* RFC function.

tRFC Communication with External Systems



Transactional RFC Server Programs

After having connected to an SAP system (via **`RfcRegisterServer`** or **`RfcStartServer`**) and after having installed the supported RFC functions, the RFC server program has to use the RFC call **`RfcSetTransactionHandlers`** for working with the TIDs to check and execute the real RFC function it supports before entering in the loop with **`RfcListenAndDispatch`**.

This function installs the following four functions to control transactional behavior:

- **`RFC_ON_CHECK_TRANSACTION`**

  This function will be activated if a tRFC is called from an SAP system. The current TID is passed. The function has to store this TID in permanent storage and return 0.
  If the same function will be called again later with the same TID, it has to make sure that it will return a value <>0. If the same TID is already running within another process but is not completed, the function has to wait until the transaction finishes or the server sends a corresponding return code.

- **`RFC_ON_COMMIT_TRANSACTION`**

  This function will be called if all RFC functions belonging to this transaction are done and the local transaction can be completed. It should be used to locally commit the transaction if necessary (working with database).

- **`RFC_ON_ROLLBACK_TRANSACTION`**

  This function is called instead of the function **`RFC_ON_COMMIT_TRANSACTION`** if there is an error occurring in the RFC library while processing the local transaction. This function can be used to roll back the local transaction (working with database).

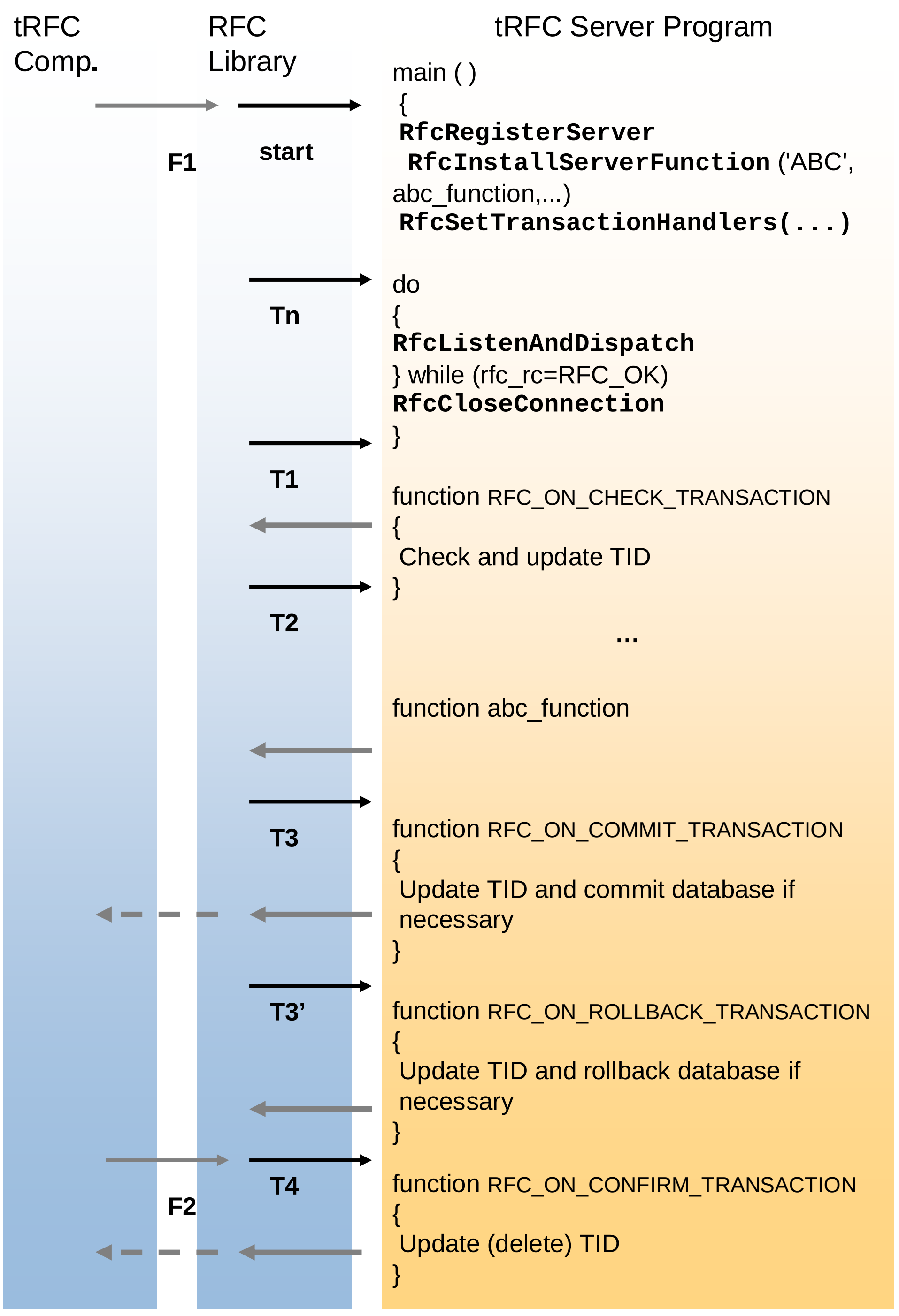
- **`RFC_ON_CONFIRM_TRANSACTION`**

  This function will be called if the local transaction is completed. All information about this TID can be deleted.

  These four 'technical' functions must be implemented once in a tRFC server program. They can be reused by any business function that is to be executed as tRFC function.

  If you do not install and work with these four functions it can not be guaranteed that an RFC call issued by `CALL FUNCTION... IN BACKGROUND TASK` is executed by this RFC server program *exactly once*.

tRFC between tRFC component, RFC library and tRFC server program



```
main ( )
 {
 RfcRegisterServer
  RfcInstallServerFunction ('ABC',
abc_function,...)
 RfcSetTransactionHandlers(...)

do
{
RfcListenAndDispatch
} while (rfc_rc=RFC_OK)
RfcCloseConnection
}

function RFC_ON_CHECK_TRANSACTION
{
 Check and update TID
}
                    ...

function abc_function


function RFC_ON_COMMIT_TRANSACTION
{
 Update TID and commit database if
 necessary
}

function RFC_ON_ROLLBACK_TRANSACTION
{
 Update TID and rollback database if
 necessary
}

function RFC_ON_CONFIRM_TRANSACTION
{
 Update (delete) TID
}
```

The tRFC component, the RFC library and the tRFC server communicate with each other in two phases:

- Phase 1/F1: Function transfer
- Phase 2/F2: Confirmation

F1: Function Transfer

After tRFC data is received, the RFC library will activate the tRFC server program (action: **start**). Using **`RfcDispatch`** in a loop, the library will call up the following functions within this function transfer phase:

- **T1**: `RFC_ON_CHECK_TRANSACTION`
- **T2**: the required RFC function 'abc_function'

  (via `RFC_SERVER_FUNCTION`)
- **T3**: `RFC_ON_COMMIT_TRANSACTION`

As described above, the function transfer phase will be repeated by the tRFC component in the SAP system if any CPI-C error (network errors, for example) occurs during this phase. The maximum number of tries and the time between two tries can be defined by using transaction **SM59** and *TRFC options*.

- **T3'**: `RFC_ON_ROLLBACK_TRANSACTION`

  If the call finally fails, the transaction will be rolled back completely.

F2: Confirmation

After the RFC library informs the tRFC components in the SAP system about a successful T3, it will immediately receive confirmation of the current transaction. The RFC library will then call up the function

- **T4**: `RFC_ON_CONFIRM_TRANSACTION`

After this phase, the current transaction is successfully completed on both sides.

Error Handling

In this section you can find information on the different types of error handling for the NW RFC SDK:

- Debugging
- Error handling in ABAP
- Error handling in RFC Server Programs

Debugging

Use

To debug an ABAP program communicating with an externe RFC program you can can use the *External Debugging* feature. External debugging enables you to access ABAP code within an AS ABAP from any local or remote (web) client (browser). This feature works independently of the RFC API.

Further Information

You can find detailed information on external debugging in the SAP Library:

- External Debugging

Error Handling in ABAP

Causes

There are different causes for ABAP errors, depending on the program. Causes include:

- Incorrect or incomplete entries in the RFC destination (transaction SM59)
- Network problems
- Authorization problems
- Error in the RFC program

Solution

Check the following:

1. Check all the entries made in transaction SM59. Do all the entries match the settings of the target system?
2. Is the hostname correct?
3. Is the service/system number correct?
4. Is the user/password set correctly?
5. Is the target system active?
6. Did you save the destination?
7. Make sure that the target machine can be addressed by the host on which the gateway process is running.
8. Check that the gateway processes are running on both systems.
9. Check the network connection.

Error Handling in RFC Server Programs

This section describes the NW RFC SDK return codes and RFC error groups. You can also find an overview of the most common error causes and their solutions.

NW RFC SDK Return Codes and Error Groups

Return Code	Description
RFC_OK	A function call was processed successfully.
RFC_COMMUNICATION_FAILURE	The connection broke down while processing the function call. No response has been sent to the SAP system.
RFC_LOGON_FAILURE	Unable to logon to SAP system. Invalid password, user locked, etc.
RFC_ABAP_RUNTIME_FAILURE	SAP system runtime error: Shortdump or the called function module raised an X Message.
RFC_ABAP_MESSAGE	A function call was started to be processed, but was aborted with an ABAP A- or E-Message within the implementing application. The message parameters have been returned to the SAP system (and can be evaluated there via the sy-msgid, sy-msgtype, sy-msgno, sy-msgv1, ..., sy-msgv4 parameters).
RFC_ABAP_EXCEPTION	A function call was processed and ended within the implementing application with a defined ABAP Exception, which has been returned to the SAP system.
RFC_CLOSED	Connection closed by the other side.
RFC_CANCELED	An RFC connection has been cancelled by the SAP system.
RFC_TIMEOUT	Time out
RFC_MEMORY_INSUFFICIENT	Memory insufficient
RFC_VERSION_MISMATCH	Version mismatch
RFC_INVALID_PROTOCOL	The received data has an unsupported format.
RFC_SERIALIZATION_FAILURE	A problem while serializing or deserializing RFM parameters.
RFC_INVALID_HANDLE	An invalid handle was passed to an API call.
RFC_RETRY	RfcListenAndDispatch did not receive an RFC during the timeout period.
RFC_EXTERNAL_FAILURE	Error in external custome code (e.g. in the tRFC Handlers).
RFC_EXECUTED	Inbound tRFC call already executed (needs to be returned from RFC_ON_CHECK_TRANSACTION in case the TID is already known).
RFC_NOT_FOUND	Function or structure definition not found (Metadata API).
RFC_NOT_SUPPORTED,	The operation is not supported on that handle.
RFC_ILLEGAL_STATE	The operation is not supported on that handle at the current point of time (e.g. trying a callback on

	a server handle, while not in a call).
RFC_INVALID_PARAMETER	An invalid parameter was passed to an API call, (e.g. invalid name, type or length).
RFC_CODEPAGE_CONVERSION_FAILURE	Codepage conversion error.
RFC_CONVERSION_FAILURE	Error while converting a parameter to the correct data type
RFC_BUFFER_TOO_SMALL	The given buffer was to small to hold the entire parameter. Data has been truncated.
RFC_TABLE_MOVE_BOF	Trying to move the current position before the first row of the table.
RFC_TABLE_MOVE_BOF	Trying to move the current position before the first row of the table.
RFC_TABLE_MOVE_EOF	Trying to move the current position after the last row of the table.

RFC Error Groups	Description
OK	OK
ABAP_APPLICATION_FAILURE	ABAP Exception raised in ABAP function modules within the implementing application.
ABAP_RUNTIME_FAILURE	ABAP Message raised in ABAP function modules or in ABAP runtime of the SAP system (e.g. Kernel).
LOGON_FAILURE	Error message raised when logon fails.
COMMUNICATION_FAILURE	Problems with the network connection (or SAP system broke down and killed the connection).
EXTERNAL_RUNTIME_FAILURE	Problems in the RFC runtime of the external program (i.e. the present library).
EXTERNAL_APPLICATION_FAILURE	Problems in the external program (e.g. in the external server implementation).

Causes

There are different causes for ABAP errors, depending on the program. Causes include:

- Incorrect or incomplete entries in the RFC destination (transaction SM59)
- Network problems
- Authorization problems
- Error in the RFC program

Solution

Check the following:

1. Check all the entries made in transaction SM59. The destination used must be entered with category T. Enter a complete path name, if possible, for the program to be started. Make sure to save the destination.
2. Configuration on operating system level
    I. The host name specification in the destination

a. A name is entered into ‘host name’ or a non-standard gateway is stored (in gateway options). In the latter case, the program is started by the standard gateway program of the system or by the gateway (**gwrd**) specified explicitly via ‘remote shell’.

Make sure that the entered computer can be addressed by the computer of gateway process. Enter the following command on this computer:

/etc/ping <host name> or **ping <host name>**

To start a program with ‘remote shell’ on another computer, it is necessary that the user-ID of the gateway process exists on the target system and that the **HOME** directory of this user contains a file **.rhosts** in the target system. The name of the calling computer must be stored in this file. To test this, log in on the computer of the gateway process under the user-ID of this process and call the command:

remsh <host name> <program name>

In this case, you have to enter the same as in transaction SM59 for **<host name>** and **<program name>**.

(If an RFC server program is called without parameters, the call **RfcRegisterServer** returns an error code in any case and the program should terminate immediately.)

b. No entry is made in ‘*host name*’

In this case, the program is started from the SAP application server.

- Make sure that the program can be addressed from the SAP application server.
- Make sure that the SAP application server has the authorization for starting the program.
- To do this, log on under the user ID of the SAP application server. Go into the *work* directory of the application server (**/usr/sap/.../D.../work**), if possible, and start the RFC server program manually from there.

(If an RFC server program is called without parameters, the call **RfcRegisterServer** returns an error code in any case and the program should terminate immediately.)

II. Problems in the RFC server program itself

To pick up **stderr** output of the RFC server program, you can enter a control program instead of the actual server program into the destination which in turn starts the actual server program with the same command line and which in this case redirects the standard output of the program into a file.

RFC server program is /xxx/xxxx.

However, the C script is called (do not forget the specification of the shell in the first line):

```
#!/bin/csh

date >> /tmp/rfclog

/xxx/xxxx $* >>& /tmp/rfclog

echo $status >>& /tmp/rfclog
```

Display the log file **/tmp/rfclog** for further error analysis.

You can activate the trace flag in the destination (remember to save). A file **`dev_rfc`** is then written by the RFC server program in its current directory containing all data received, operations and errors that occurred.

Compiling

For compiling your C/C++ programs against the SAP NW RFC SDK you have to use an ANSI C compatible C-compiler and set the include and library search path to the installed NW RFC SDK include and lib directory.

It is recommended to use the same compiler that has been used to compile the NW RFC SDK. The exact compiler versions and possible problems and restrictions when using other compilers are listed in SAP Note **1056696**.

NetWeaver RFC SDK Reference

Purpose

This section describes the data types and functions used by the SAP NetWeaver RFC SDK.

Further Information

For more information see:

- NetWeaver RFC API Data Types
- NetWeaver RFC API Functions

Please use the bookmark tree on the left for fast navigation in this area.

NetWeaver RFC API Data Types

The RFC API defines the following data types that are used by various API functions described in the next section:

- RFC API Handles
- RFC API Enumerations
- RFC API Structures

RFC API Handles

In this section you can find the list of handles used by the NetWeaver RFC SDK:

Handle name	Description
handle RFC_CONNECTION_HANDLE	Handle for an internal RFC connection object.
handle RFC_TYPE_DESC_HANDLE	Handle for an internal type descriptor that contains the metadata of a structure type.
handle RFC_STRUCTURE_HANDLE	Handle for an internal structure data object.
handle RFC_FUNCTION_DESC_HANDLE	Handle for an internal RFC function interface descriptor that contains the metadata of a RFC function.
handle RFC_FUNCTION_HANDLE	Handle for an internal RFC function object that works as a client proxy or server stub.
handle RFC_TABLE_HANDLE	Handle for an internal table object that is a collection of structure data objects.
handle RFC_TRANSACTION_HANDLE	Handle for an internal transaction object required by tRfc or qRfc.

RFC API Enumerations

In this section you can find a list of Enumeration data types used by the NetWeaver RFC SDK:

- enum RFCTYPE
- enum RFC_DIRECTION
- enum RFC_RC

enum RFCTYPE

Use

Defines RFC parameter types for each corresponding ABAP types.

Structure

```
typedef enum _RFCTYPE
{
  RFCTYPE_CHAR   = 0,      /* 1-byte or multibyte character, fixed
sized, blank padded */
  RFCTYPE_DATE   = 1,      /* date ( YYYYYMMDD ) */
  RFCTYPE_BCD    = 2,      /* packed number, any length between 1 and
16 bytes */
  RFCTYPE_TIME   = 3,      /* time (HHMMSS)  */
  RFCTYPE_BYTE   = 4,      /* raw data, binary, fixed length, zero
padded.*/
  RFCTYPE_TABLE   = 5,     /* internal table */
  RFCTYPE_NUM    = 6,      /* digits, fixed size, leading '0' padded.
*/
  RFCTYPE_FLOAT  = 7,      /* floating point, double precission */
  RFCTYPE_INT    = 8,      /* 4-byte integer */
  RFCTYPE_INT2   = 9,      /* 2-byte integer, obsolete, not directly
supported by ABAP/4 */
  RFCTYPE_INT1   = 10,     /* 1-byte integer, obsolete, not directly
supported by ABAP/4 */
  RFCTYPE_NULL  = 14,      /* not supported data type. */
  RFCTYPE_STRUCTURE = 17,  /* abap structure */
  RFCTYPE_DECF16  = 23,      /* 16-bytes decimal float point */
  RFCTYPE_DECF32  = 24,      /* 32-bytes decimal float point */
  RFCTYPE_XMLDATA = 28,      /* variable-length, zero terminated
string representing ABAP data   with hierarchical structure (so-
called TYPE2) as XML */
  RFCTYPE_STRING = 29,      /* variable-length, zero terminated
string */
  RFCTYPE_XSTRING = 30,     /* variable-length raw string, length in
bytes */
  RFCTYPE_EXCEPTION = 98,
  _RFCTYPE_max_value      /* the max. value of RFCTYPEs */
}RFCTYPE;
```

enum RFC_DIRECTION

Use

Enumeration used for indicating the direction of a RFC function parameter.

Structure

```
typedef enum _RFC_DIRECTION
{
   RFC_IN = 0x0001,  /* input parameter, corresponding to ABAP
IMPORTING parameter */
   RFC_OUT= 0x0002,  /* output parameter, corresponding to ABAP
EXPORTING parameter */
   RFC_INOUT = RFC_IN | RFC_OUT /* input and output parameters, this
is corresponding to ABAP
                                    CHANGING or TABLES parameter */
}RFC_DIRECTION;
```

enum RFC_RC

Use

Defines RFC API return codes.

Structure

```
typedef enum _RFC_RC
{
    RFC_OK,                 /* OK.                              */
    RFC_FAILURE,                   /* Error occurred
*/
    RFC_COMMUNICATION_FAILURE,  /* Error in Network & Communications
*/
    RFC_LOGON_FAILURE,             /* SAP logon error */
    RFC_SYSTEM_FAILURE,            /* e.g. SAP system runtime error */
    RFC_CODEPAGE_CONVERSION_FAILURE,    /* codepage conversion error
*/
    RFC_CONVERSION_FAILURE,        /* An error during conversion has
been detected            */
    RFC_APPLICATION_ERROR,         /* The called function module raised
a message     */
    RFC_APPLICATION_EXCEPTION,  /* The called function module raised
an exception */
    RFC_EXCEPTION,                 /* Exception raised
*/
    RFC_CALL,                      /* Call received
*/
    RFC_CLOSED,                    /* Connection closed by the other
side                     */
    RFC_RETRY,                     /* No data yet (RfcListen or
RfcWaitForRequest only)      */
    RFC_NO_TID,                    /* No Transaction ID available
*/
    RFC_EXECUTED,                  /* Function already executed
*/
    RFC_SYNCHRONIZE,               /* Synchronous Call in Progress (only
for Windows)        */
    RFC_MEMORY_INSUFFICIENT,       /* Memory insufficient
*/
    RFC_NOT_FOUND,                 /* Function not found (internal use
only)                  */
    RFC_NOT_SUPPORTED,             /* This call is not supported
*/
    RFC_NOT_INITIALIZED,           /* RFC not yet initialized
*/
    RFC_INVALID_HANDLE,            /* An invalid handle was passed to an
API call    */
    RFC_INVALID_PARAMETER,         /* An invalid parameter was passed to
an API */
    RFC_CANCELED,                  /* An rfc call has been canceled by
user                   */
```

```
    RFC_VERSION_MISMATCH,       /* Version mismatch
*/
    RFC_INVALID_PROTOCOL,       /* Invalid RFC protocaol detected*/
    RFC_TIMEOUT            /* time out*/
    RFC_BUSY             = 110  /* @emem System is busy, try later
*/
}RFC_RC;
```

RFC API Structures

In this section you can find a list of Enumeration data types used by the NetWeaver RFC SDK:

- struct RFC_ERROR_INFO
- struct RFC_CONNECTION_PARAMETER
- struct RFC_PARAMETER_DESC
- struct RFC_FUNCTION_DESC
- struct RFC_FIELD_DESC
- struct RFC_STRUCTURE_DESC
- struct RFC_ATTRIBUTES

struct RFC_ERROR_INFO

Use

Structure used by the RFC API to return detailed error information.

Structure

```
typedef struct _RFC_ERROR_INFO
{
    RFC_RC code;          /* Error code   */
    SAP_UC key[128];      /* Error key or error group in short text
*/
    SAP_UC message[1024]; /* Detailed error message */
}RFC_ERROR_INFO, *P_RFC_ERROR_INFO;
```

struct RFC_CONNECTION_PARAMETER

Use

Structure used for defining connection parameter name-value pair.

Structure

```
typedef struct _RFC_CONNECTION_PARAMETER
{
    SAP_UC * name;           /* parametr name   */
    SAP_UC * value;          /* parameter value */
}RFC_CONNECTION_PARAMETER,*P_RFC_CONNECTION_PARAMETER;
```

struct RFC_PARAMETER_DESC

Use

Structure used to describe a RFC parameter, i.e. it contains the metadata of the parameter.

Structure

```
typedef struct _RFC_PARAMETER_DESC
{
    SAP_UC const *   name;     /* parameter name, null terminated
string*/
    RFCTYPE    type;     /* parameter type */
    RFC_TYPE_DESC_HANDLE typeDesc;
    unsigned   length1;  /* parameter length in bytes in a
non_unicode SAP system */
    unsigned   length2;  /* parameter length in bytes in a unicode
SAP system */
    unsigned      decimals;
    RFC_DIRECTION direction;   /* if the parameter is a input, output
or both */
    SAP_UC const * defaultValue;
}RFC_PARAMETER_DESC,*P_RFC_PARAMETER_DESC;
```

struct RFC_FUNCTION_DESC

Use

Structure used to describe the interface of a RFC function, i.e. It contains the metadata of the RFC function.

Structure

```
typedef struct _RFC_FUNCTION_DESC
{
    SAP_UC const* name;    /*function name*/
    RFC_PARAMETER_DESC *parameters; /*an arry of RFC_PARAMETER_DESC
in which each
                                             element describes a
parameter of the function*/
    unsigned   count;         /* number of parameters */
    unsigned short type;             /* 0 - classic RFC, 1 - BAsXML
*/
}RFC_FUNCTION_DESC,*P_RFC_FUNCTION_DESC;
```

struct RFC_FIELD_DESC

Use

Structure used to describe a field inside a structure, i.e. It contains the metadata of the field.

Structure

```
typedef struct _RFC_FIELD_DESC
{
   SAP_UC const*  name;    /* field name, null terminated string */
   RFCTYPE     type;    /* field type */
   unsigned    decimals;
   unsigned length1; /* field length in bytes in a non_unicode SAP
system */
   unsigned offset1; /* field offset in bytes in a non_unicode SAP
system */
   unsigned length2; /* field length in bytes in a unicode SAP system
*/
   unsigned offset2; /* field offset in bytes in a unicode SAP system
*/
   RFC_TYPE_DESC_HANDLE typeDesc;  /* pointer to a RFC_STRUCTURE_DESC
structure for the
                                        nestesd sub-structure or
embedded table if the field type
                                        is RFCTYPE_STRUCTURE or
RFCTYPE_TABLE */
   unsigned char include_flag;  /* a flag indicating the beginning or
the end of a included
                                        structre */
       unsigned       length;       /* actual length for STRING and
XSTRING types */
}RFC_FIELD_DESC,*P_RFC_FIELD_DESC;
```

struct RFC_STRUCTURE_DESC

Use

Structure used to describe a structure, i.e. it contains the metadata of the structure.

Structure

```
typedef struct _RFC_STRUCTURE_DESC
{
    SAP_UC const* name;    /* structure or table name, null
terminated string */
   SAP_UC const* lineTypeName;      /* for tables only: the name of
the line type used for the
                                    table */
    unsigned   length1;    /* total length in bytes in a 1-byte-per-
SAP_CHAR system */
    unsigned   length2;    /* total length in bytes in a 2-byte-per-
SAP_CHAR system */
   RFC_FIELD_DESC *fields;    /* an arry of RFC_FIELD_DESC in which
each element describes
                                    a field in the structre */
    unsigned   count;            /* number of fields of the
structure */
    unsigned char layout_flag;       /* a flag indicating if the
structure is a normal, type1 or
                                      type2 structure */
}RFC_STRUCTURE_DESC, *P_RFC_STRUCTURE_DESC;
```

struct RFC_ATTRIBUTES

Use

Structure returned by `RfcGetConnectionAttributes()` giving some information about the partner system on the other side of this RFC connection.

Structure

```
typedef struct _RFC_ATTRIBUTES
{
    SAP_UC dest[64+1];           ///< RFC destination
    SAP_UC host[100+1];          ///< Own host name
    SAP_UC partnerHost[100+1];      ///< Partner host name
    SAP_UC sysNumber[2+1];       ///< R/3 system number
    SAP_UC sysId[8+1];           ///< R/3 system ID
    SAP_UC client[3+1];          ///< Client ("Mandant")
    SAP_UC user[12+1];           ///< User
    SAP_UC language[2+1];        ///< Language
    SAP_UC trace[1+1];           ///< Trace level (0-3)
    SAP_UC isoLanguage[2+1];     ///< 2-byte ISO-Language
    SAP_UC codepage[4+1];           ///< Own code page
    SAP_UC partnerCodepage[4+1]; ///< Partner code page
    SAP_UC rfcRole[1+1];         ///< C/S: RFC Client / RFC Server
    SAP_UC type[1+1];            ///< 2/3/E/R: R/2,R/3,Ext,Reg.Ext
    SAP_UC partnerType[1+1];     ///< 2/3/E/R: R/2,R/3,Ext,Reg.Ext
    SAP_UC rel[4+1];          ///< My system release
    SAP_UC partnerRel[4+1];         ///< Partner system release
    SAP_UC kernelRel[4+1];       ///< Partner kernel release
    SAP_UC cpicConvId[8 + 1];    ///< CPI-C Conversation ID
    SAP_UC progName[128+1];         ///< Name of the calling APAB
program (report, module pool)
    SAP_UC reserved[86];         ///< Reserved for later use
}RFC_ATTRIBUTES, *P_RFC_ATTRIBUTES;
```

NetWeaver RFC API Functions

Use

In the following section you will find a description of the functions offered by the RFC API.

The list of functions is sorted by functional areas.

Functional Areas

- Administrative Functions
- RFC Server Functions
- Transactional Functions
- Function-Handling Functions
- Table-Handling Functions
- Field-Handling Functions
- Metadata and Repository Functions

Administrative Functions

In the following section you will find a description of the RFC SDK functions you can use for administrative purposes.

Functions

- RfcInit
- Connection Handling

RfcInit

Use

Initializes the RFC library.

On platforms other than Windows, this function should be called before calling any other API functions.

Structure

```
    RFC_RC SAP_API RfcInit();
```

Connection Handling

In the following section you will find a description of the functions you can use for connection-handling activities.

Functions

- RfcOpenConnection
- RfcCloseConnection
- RfcGetConnectionAttributes
- RfcInvoke
- RfcStartServer
- RfcRegisterServer

RfcOpenConnection

Use

Opens a client connection to an SAP System. The connection parameters may contain the following name-value pairs:

client, *user*, *password*, *lang*, *trace* and additionally one of

- Direct application server logon: *ashost*, *sysnr*.
- Logon with load balancing: *mshost*, *msserv*, *sysid*, *group*.

msserv is only needed, if the service of the message server is not defined as *sapms<SYSID>* in */etc/services*.

When logging on with SNC, *user* and *password* are to be replaced by *snc_mode (0/1)*, *snc_qop*, *snc_myname*, *snc_partnername*, *snc_lib*.

When logging on with SSO Ticket, you can use *mysapsso* instead of

User and *password* or *mysapsso2* instead of *password*.

Alternatively the connection parameters can be defined in the configuration file

`sapnwrfc.ini`. In this case you just pass the parameter *dest*=... and the

user credentials into `RfcOpenConnection`.

If the logon was ok, `RfcOpenConnection` returns a client connection handle, that can be used in `RfcInvoke`.

Otherwise the return value is `NULL` and `errorInfo` contains a detailed error description.

errorInfo → *code* will be one of:

- `RFC_INVALID_PARAMETER`: One of the connection parameters was invalid.
- `RFC_COMMUNICATION_FAILURE`: Problems in the network or network settings.
- `RFC_LOGON_FAILURE`: Invalid user/password/ticket/certificate.
- `RFC_ABAP_RUNTIME_FAILURE`: Problems in the SAP backend system.

Structure

```
RFC_CONNECTION_HANDLE SAP_API RfcOpenConnection
(RFC_CONNECTION_PARAMETER const * connectionParams, unsigned
paramCount, RFC_ERROR_INFO* errorInfo);
```

RfcCloseConnection

Use

Closes a RFC client or server connection.

Structure

```
   RFC_RC SAP_API RfcCloseConnection(RFC_HANDLE rfcHandle,
RFC_ERROR_INFO* info);


       rfcHandle       : connection handle;
```

RfcGetConnectionAttributes

Use

Gets the attributes of an opened RFC connection.

Structure

```
   RFC_RC SAP_API RfcGetConnectionAttributes(RFC_HANDLE rfcHandle,
RFC_ATTRIBUTES * attri);
```

RfcInvoke

Use

Executes a function module in the backend system so far. The return codes have the following meaning:

Return Code	Description
RFC_OK	The function call was executed successfully
RFC_ABAP_EXCEPTION	The function call was executed and ended with a defined ABAP Exception. The key of the exception can be obtained from *abapException* → *exception_key*. In the above two cases `rfcHandle` is still open and can be used to execute further function call.
RFC_ABAP_MESSAGE	The function call was started to be processed, but was aborted with an ABAP Message. The message parameters can be obtained from *abapException* → *message_class*, *abapException* → *message_type*, *abapException* → *message_number*, *abapException* → *message[0]*, ..., *abapException* → *message[3]*
RFC_ABAP_RUNTIME_FAILURE	The function call was started to be processed, but was aborted with a `SYSTEM_FAILURE` (e.g. division by zero, unhandled exception, etc in the backend system). Detailes can be obtained from *errorInfo* → *message*
RFC_COMMUNICATION_FAILURE	The connection broke down while processing the function call. Details can be obtained from *errorInfo* → *message*. In these three cases the connection has been closed, so the `rfcHandle` needs to be refreshed via `RfcRegisterServer`.
RFC_INVALID_HANDLE	`rfcHandle` is invalid or points to a connection that has already been closed

Structure

```
RFC_RC SAP_API RfcInvoke (RFC_CONNECTION_HANDLE rfcHandle,
RFC_FUNCTION_HANDLE funcHandle, RFC_ERROR_INFO* errorInfo);


rfcHandle       : connection handle;
funcHandle      : function handle;
errorInfo       : error messages;
```

RfcStartServer

Use

Starts the RFC server (and initializes the RFC library implicitly).

This function needs to be called, if the server program is to be started by the SAP application server (RFC destination of type "T" *in startup mode*).

`argc` and `argv` are the inputs of the mainU function. The SAP application server passes the correct command line to the program, when starting it up, so you only need to forwards these two parameters to `RfcStartServer`.

`connectionParams` is optional and is only needed, if you want to add additional logon parameters to the ones coming from the command line, e.g. for activating traces.

Like `RfcRegisterServer` the function returns a server connection handle that can be used in `RfcListenAndDispatch`.

The mechanism of this kind of RFC destination thus works as follows:

1. The SAP application server opens a telnet connection to the host, where your server program is located, and starts the program with the necessary logon parameters (or creates a child process, if the startup method is "Start on application server".)
2. Your server program calls `RfcStartServer`, which opens an RFC connection back to the SAP system.
3. The SAP system then makes the function call over that RFC connection.

The main differences *startup mode* compared to the *registration mode* are:

Advantage: no logon parameters need to be maintained in the server program (unless you want to open an additional client connection for looking up function module metadata (`RFC_FUNCTION_DESC_HANDLE`) in the SAP DDIC).

Disadvantage: every single function call creates a new process and a telnet connection in addition to the actual RFC connection.

Structure

```
RFC_CONNECTION_HANDLE SAP_API RfcStartServer (int argc, SAP_UC
**argv, RFC_CONNECTION_PARAMETER const * connectionParams, unsigned
paramCount, RFC_ERROR_INFO* errorInfo);
```

RfcRegisterServer

Use

Opens a RFC server connection to a SAP Gateway and registers a program ID at the gateway.

Structure

```
RFC_HANDLE SAP_API RfcRegisterServer (RFC_CONNECTION_PARAMETER const
*connectionParams,
                unsigned paramCount);
connectionParams: an array of connection parameter name-value
                  pairs;
paramCount      : number of the elements of the above array
 return          : a RFC connection handle
```

RFC Server Functions

In the following section you will find a description of the functions you can use within the RFC Server.

Functions

- RfcListenAndDispatch
- RfcInstallServerFunction
- RfcInstallGenericServerFunction

RfcListenAndDispatch

Use

Starts to listen on the given RFC server connection and dispatches incoming RFC calls to the insatlled server functions.

This function will block the current thread until an incoming call returns or times out.

Process

The mechanism for dispatching incoming function calls works as follows:

First `RfcListenAndDispatch` checks, whether for the current combination of SAP System ID and function module name a callback function has been installed via `RfcInstallServerFunction`. If not, it checks, whether a global callback function has been installed via `RfcInstallGenericServerFunction`.

If a callback function has been found, the RFC call will be dispatched to that function for processing, and `RfcListenAndDispatch` returns the return code of the callback function.

Otherwise `RfcListenAndDispatch` returns a `SYSTEM_FAILURE` to the SAP backend and the return code `RFC_NOT_FOUND` to the caller.

In general the return codes of `RfcListenAndDispatch` have the following meaning:

- `RFC_OK`: A function call was processed successfully.
- `RFC_RETRY`: No function call came in within the specified timeout period. (*timeout* is given in seconds.)
- `RFC_ABAP_EXCEPTION`: A function call was processed and ended with a defined ABAP Exception, which has been returned to the backend.

In the above three cases `rfcHandle` is still open and can be used to listen for the next request.

- `RFC_ABAP_MESSAGE`: A function call was started to be processed, but was aborted with an ABAP A- or E-Message. The message parameters have been returned to the backend (and can be evaluated there via the *sy-msgid*, *sy-msgtype*, *sy-msgno*, *sy-msgv1*, ..., *sy-msgv4* parameters).
- `RFC_ABAP_RUNTIME_FAILURE`: A function call was started to be processed, but was aborted with a `RABAX`,

which has been returned to the backend.

- `RFC_COMMUNICATION_FAILURE`: The connection broke down while processing the function call. No response has been sent to the backend.

In these three cases the connection has been closed, so the `rfcHandle` needs to be refreshed via `RfcRegisterServer`.

- `RFC_INVALID_HANDLE`: `rfcHandle` is invalid or points to a connection that has already been closed.

Structure

```
RFC_RC SAP_API RfcListenAndDispatch (RFC_CONNECTION_HANDLE rfcHandle,
int timeout, RFC_ERROR_INFO* errorInfo);
```

`rfcHandle`	: RFC server connection handle;
`timeout`	: time-out period in seconds

RfcInstallServerFunction

Use

Installs a callback function of type `RFC_SERVER_FUNCTION`, which will be triggered when a request for the function module corresponding to `funcDescHandle` comes in from the SAP system corresponding to *sysId*.

The main inputs of `RFC_SERVER_FUNCTION` are as follows:

- `RFC_CONNECTION_HANDLE`

  A connection handle, which can be used to query logon information about the current (backend) user or to make callbacks into the backend.

- `RFC_FUNCTION_HANDLE`

  A data container that represents the current function call. Read the importing parameters, which came from the backend, from this container via the `RfcGetX` functions and write the exporting parameters, which are to be returned to the backend, into this container using the `RfcSetX` functions.The memory of that container is automatically released by the RFC Runtime after the `RFC_SERVER_FUNCTION` returns.

- `RFC_ERROR_INFO`

  If you want to return an ABAP Exception or ABAP Message to the backend, fill the

  parameters of that container and return `RFC_ABAP_EXCEPTION` or `RFC_ABAP_MESSAGE` from your `RFC_SERVER_FUNCTION` implementation.

  If you want to return a `SYSTEM_FAILURE` to the backend, fill the message parameter of this container and return `RFC_EXTERNAL_FAILURE` from your `RFC_SERVER_FUNCTION` implementation.

  If your `RFC_SERVER_FUNCTION` implementation processed the function call successfully, you should return `RFC_OK`.

Structure

```
RFC_RC SAP_API RfcInstallServerFunction(SAP_UC const *sysId,
RFC_FUNCTION_DESC_HANDLE funcDescHandle,RFC_SERVER_FUNCTION
serverFunction, RFC_ERROR_INFO* errorInfo);
```

RfcInstallGenericServerFunction

Use

Installs a generic callback function of type `RFC_SERVER_FUNCTION`. This callback will be called by the RFC Runtime, whenever a combination of SAP System ID & function module name comes in, for which no matching callback function has been installed via `RfcInstallServerFunction`.

In addition to the handler function you need to provide a second callback function:

- `RFC_FUNC_DESC_CALLBACK`.

  This callback will be called to obtain a `RFC_FUNCTION_DESC_HANDLE` for the current function module from you. So this function either needs to return hard-coded meta data or needs to be able to perform a DDIC lookup using a valid client connection and `RfcGetFunctionDesc`.

Structure

```
RFC_RC SAP_API RfcInstallGenericServerFunction(RFC_SERVER_FUNCTION
serverFunction, RFC_FUNC_DESC_CALLBACK funcDescProvider,
RFC_ERROR_INFO* errorInfo);
```

Transactional Functions

In the following section you will find a description of the functions you can use for transaction handling.

Functions

- RfcGetTransactionID
- RfcCreateTransaction
- RfcInvokeInTransaction
- RfcSubmitTransaction
- RfcConfirmTransaction
- RfcDestroyTransaction
- RfcInstallTransactionHandlers

RfcGetTransactionID

Use

Retrieves a unique 24-digit transaction ID from the SAP backend system.

Structure

```
   RFC_RC SAP_API RfcGetTransactionID(RFC_HANDLE rfcHandle, RFC_TID
tid);


       rfcHandle       : connection handle;
       tid             : buffer for returned trasanction ID;
```

RfcCreateTransaction

Use

Creates a container for executing a (multi-step) transactional call.

Structure

```
RFC_TRANSACTION_HANDLE SAP_API RfcCreateTransaction(RFC_HANDLE
rfcHandle, RFC_TID tid, SAP_UC const *queueName);
```

```
       rfcHandle       : connection handle;
       tid             : trasanction ID used to identify the
                         transaction;
       queueName       : queue name associated with the
                         transaction. If it is NULL or empty,
                         the transaction is for tRfc
```

RfcInvokeInTransaction

Use

Puts a tRfc or qRfc call into the given transaction. Can be used multiple times within one transaction handle.

Structure

```
   RFC_RC SAP_API RfcInvokeInTransaction(RFC_TRANSACTION_HANDLE
tHandle,
                       RFC_FUNCTION_HANDLE funcHandle );


       tHandle       : transaction handle;
       funcHandle    : function handle that identifies the Rfc
function;
```

RfcSubmitTransaction

Use

Executes all function modules in the backend system, that have accumulated in the transaction so far.

This step can be repeated until it finally succeeds (RFC_OK). The transaction handling in the backend system protects against duplicates.

Structure

```
   RFC_RC SAP_API RfcSubmitTransaction(RFC_TRANSACTION_HANDLE
tHandle);

       tHandle       : transaction handle;
```

RfcConfirmTransaction

After `RfcSubmitTransaction` has finally succeeded, `RfcConfirmTransaction` cleans up the transaction handling table in the backend.

After this call the backend is no longer protected against this TID. So another `RfcSubmitTransaction` would result in a duplicate.

RfcDestroyTransaction

Use

Releases the memory of the transaction container.

Structure

```
RFC_RC SAP_API RfcDestroyTransaction(RFC_TRANSACTION_HANDLE tHandle,
RFC_ERROR_INFO* errorInfo);


tHandle:  transaction handle;
```

RfcInstallTransactionHandlers

Use

Installs the necessary callback function for being able to process incoming tRFC/qRFC calls.

Process

1. The `RFC_ON_CHECK_TRANSACTION` function is called when a local transaction is starting. Since a transactional RFC call can be issued many times by the client system, the function is responsible for storing the transaction ID in permanent storage. The return value should be one of the following:
   - `RFC_OK`

     Transaction ID stored, transaction can be started.
   - `RFC_EXECUTED`

     This transaction has already been processed successfully in an earlier attempt. Skip the execution now.
   - `RFC_EXTERNAL_FAILURE`

     Currently unable to access my permanent storage. Raise an exception in the sending system, so that the sending system will try to resend the transaction at a later time.
2. The next step will be the execution of the `RFC_SERVER_FUNCTION` for that particular function module.
3. Depending on the result of the `RFC_SERVER_FUNCTION` implementation, next one of `RFC_ON_COMMIT_TRANSACTION` or `RFC_ON_ROLLBACK_TRANSACTION` is called. Do a `COMMIT WORK` or `ROLLBACK WORK` here, so that the changes, which the `RFC_SERVER_FUNCTION` made, are now either persisted or rolled back.
4. In the end `RFC_ON_CONFIRM_TRANSACTION` will be called. All information stored about that transaction can now be discarded by the server, as it no longer needs to protect itself against duplicates.

In general this function can be used to delete the transaction ID from permanent storage.

Structure

```
RFC_RC SAP_API RfcInstallTransactionHandlers (SAP_UC const
*sysId, RFC_ON_CHECK_TRANSACTION onCheckFunction,
RFC_ON_COMMIT_TRANSACTION onCommitFunction,
RFC_ON_ROLLBACK_TRANSACTION onRollbackFunction,
RFC_ON_CONFIRM_TRANSACTION onConfirmFunction, RFC_ERROR_INFO*
errorInfo);
```

Function-Handling Functions

In the following section you will find a description of the functions you can use for function handling.

Functions

- RfcDescribeFunction
- RfcGetFunctionDesc
- RfcCreateFunction
- RfcDestroyFunction
- RfcSetParameterActive

RfcDescribeFunction

Returns the meta data description for the current function module.

Structure

```
RFC_FUNCTION_DESC_HANDLE SAP_API
RfcDescribeFunction(RFC_FUNCTION_HANDLE funcHandle, RFC_ERROR_INFO*
info);
```

RfcGetFunctionDesc

Use

Queries the metadata of a function interface from the SAP system and returns the handle to the descriptor.

Structure

```
   RFC_FUNCTION_DESC_HANDLE SAP_API
RfcGetFunctionDesc(RFC_CONNECTION_HANDLE rfcHandle,
                SAP_UC const * funcName, RFC_ERROR_INFO* errorInfo);

       rfcHandle:      connection handle;
       funcName:       function name;
```

RfcCreateFunction

Use

Creates a data container that can be used to execute function calls in the backend via `RfcInvoke`. The importing parameters can be set using the `RfcSetX` functions. After the RfcInvoke call returned successfully, the exporting parameters can be read from this data container via the `RfcGetX` functions.

Structure

```
RFC_FUNCTION_HANDLE SAP_API
RfcCreateFunction(RFC_FUNCTION_DESC_HANDLE funcDescHandle,
RFC_ERROR_INFO* errorInfo);
```

```
       funcDescHandle:      function interface descriptor;
```

RfcDestroyFunction

Releases all memory used by the data container.

If you have obtained a handle to a structure or table parameter of that function module

(RFC_STRUCTURE_HANDLE or RFC_TABLE_HANDLE), that handle will be invalid afterwards, as the corresponding memory has been released as well!

Structure

```
RFC_RC SAP_API RfcDestroyFunction(RFC_FUNCTION_HANDLE funcHandle,
RFC_ERROR_INFO* errorInfo);


funcHandle: function handle;
```

RfcSetParameterActive

Allows to deactivate certain parameters in the function module interface.

This is particularly useful for BAPIs which have many large tables, in which you are not interested. Deactivate those and leave only those tables active, in which you are interested. This reduces network traffic considerably.

Structure

```
RFC_RC SAP_API RfcSetParameterActive(RFC_FUNCTION_HANDLE funcHandle,
SAP_UC const* paramName, int isActive, RFC_ERROR_INFO* info);
```

Table-Handling Functions

In the following section you will find a description of the functions you can use for table handling.

Functions

- RfcDescribeType
- RfcCreateTable
- RfcCloneTable
- RfcDestroyTable
- RfcAppendRow
- RfcAppendNewRow
- RfcAddRow
- RfcAddNewRow
- RfcInsertRow
- RfcInsertNewRow
- RfcDeleteCurrentRow
- RfcDeleteAllRows
- RfcMoveToFirstRow
- RfcMoveToLastRow
- RfcMoveToNextRow
- RfcMoveTo
- RfcMovetoPreviousRow
- RfcGetCurrentRow
- RfcGetRowCount

RfcDescribeType

Returns the meta data descriptions of a structure or table (`RFC_STRUCTURE_HANDLE` or `RFC_TABLE_HANDLE`).

Structure

```
RFC_TYPE_DESC_HANDLE SAP_API RfcDescribeType(DATA_CONTAINER_HANDLE
dataHandle, RFC_ERROR_INFO* info);
```

RfcCreateTable

Use

Creates a internal table object and returns the handle to the table.

Structure

```
   RFC_TABLE_HANDLE SAP_API RfcCreateTable(RFC_TYPE_DESC_HANDLE
typeDescHandle, RFC_ERROR_INFO* info);


           typeDescHandle : handle to the type descriptor of the
table
```

RfcCloneTable

Use

Creates a copy of the given table and returns the handle to the new table.

Structure

```
   RFC_TABLE_HANDLE SAP_API RfcCloneTable(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* info);


           tableHandle:  handle to the original table
```

RfcDestroyTable

Use

Deletes the given table object.

Structure

```
   RFC_RC SAP_API RfcDestroyTable(RFC_TABLE_HANDLE tableHandle,
RFC_ERROR_INFO* errorInfo);

           tableHandle:  handle to the to be deleted table;
```

RfcAddNewRow

Use

Creates a new row and appends it at the end of the table.

Structure

```
   RFC_STRUCTURE_HANDLE SAP_API RfcAddNewRow(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* info);


           tableHandle:  handle to the to be deleted table;
```

RfcInsertNewRow

Use

Creates a new row and inserts it at the current position of the table cursor.

Structure

```
   RFC_STRUCTURE_HANDLE SAP_API RfcInsertNewRow(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* info);


           tableHandle:  handle to the to be deleted table;
```

RfcAddRow

Use

Appends the given structure at the end of the table.

Structure

```
   RFC_RC SAP_API RfcAddRow(RFC_TABLE_HANDLE tableHandle,
RFC_STRUCTURE_HANDLE structHandle, RFC_ERROR_INFO* info);

          tableHandle:  handle to the table;
          structHandle: handle to the to be added structure ;
```

RfcAppendRow

Use

Appends the given structure at the end of the table.

Structure

```
   RFC_RC SAP_API RfcAppendRow(RFC_TABLE_HANDLE tableHandle,
RFC_STRUCTURE_HANDLE structHandle, RFC_ERROR_INFO* errorInfo);

          tableHandle:  handle to the table;
          structHandle: handle to the to be added structure ;
```

RfcAppendNewRow

Use

Creates a new row and appends it at the end of the table.

Structure

```
   RFC_STRUCTURE_HANDLE SAP_API RfcAppendNewRow(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* errorInfo);

           tableHandle:  handle to the to be deleted table;
```

RfcInsertRow

Use

Inserts the given structure at the current position of the table cursor.

Structure

```
   RFC_RC SAP_API RfcInsertRow(RFC_TABLE_HANDLE tableHandle,
RFC_STRUCTURE_HANDLE ructHandle, RFC_ERROR_INFO* info);


           tableHandle:  handle to the table;
           structHandle: handle to the to be inserted structure ;
```

RfcDeleteCurrentRow

Use

Deletes the current row from the table.

Structure

```
   RFC_RC SAP_API RfcDeleteCurrentRow(RFC_TABLE_HANDLE tableHandle,
RFC_ERROR_INFO* errorInfo);


           tableHandle:  handle to the table;
```

RfcDeleteAllRows

Use

Deletes all rows from the table.

Structure

```
   RFC_RC SAP_API RfcDeleteAllRows(RFC_TABLE_HANDLE tableHandle,
RFC_ERROR_INFO* info);

           tableHandle:  handle to the table;
```

RfcMoveToFirstRow

Use

Moves the current position to the first row of the table.

Structure

```
   RFC_TABLE_MOVE_RC SAP_API RfcMoveToFirstRow(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* info);


           tableHandle:  handle to the table;
```

RfcMoveToLastRow

Use

Moves the current position to the last row of the table.

Structure

```
   RFC_TABLE_MOVE_RC SAP_API RfcMoveToLastRow(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* info);


           tableHandle:  handle to the table;
```

RfcMoveToNextRow

Use

Moves the cursor from the current position to the next row.

Structure

```
   RFC_TABLE_MOVE_RC SAP_API RfcMoveToNextRow(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* info);


           tableHandle:  handle to the table;
```

RfcMoveTo

Use

Moves the cursor from the current position to the position specified by the index.

Structure

```
   RFC_TABLE_MOVE_RC SAP_API RfcMoveTo(RFC_TABLE_HANDLE tableHandle,
unsigned index, RFC_ERROR_INFO* info);

           tableHandle:  handle to the table;
```

RfcMoveToPreviousRow

Use

Moves the cursor from the current position to the prevours row.

Structure

```
   RFC_TABLE_MOVE_RC SAP_API RfcMoveToPreviousRow(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* info);


           tableHandle:  handle to the table;
```

RfcGetCurrentRow

Use

Gets the structure handle to the current row.

Structure

```
   RFC_STRUCTURE_HANDLE SAP_API RfcGetCurrentRow(RFC_TABLE_HANDLE
tableHandle, RFC_ERROR_INFO* info);
```

```
           tableHandle:  handle to the table;
```

RfcGetRowCount

Use

Gets the number of the rows contained in the table.

Structure

```
   unsigned SAP_API RfcGetRowCount(RFC_TABLE_HANDLE tableHandle,
RFC_ERROR_INFO* info);


           tableHandle:  handle to the table;
```

Structure-Handling Functions

In the following section you will find a description of the functions you can use for sructure handling.

Functions

- RfcGetStructureDescHandle
- RfcCreateStructure
- RfcCloneStructure
- RfcDestroyStructure

RfcGetStructureDescHandle

Use

Queries the metadata of a ABAP type the SAP system and returns the handle of the descriptor.

Structure

```
   RFC_TYPE_DESC_HANDLE SAP_API RfcGetStructureDescHandle(RFC_HANDLE
rfcHandle, SAP_UC const * typeName);
```

RfcCreateStructure

Use

Creates an internal structure data object and returns the handle to the structure object.

Structure

```
   RFC_STRUCTURE_HANDLE SAP_API
RfcCreateStructure(RFC_TYPE_DESC_HANDLE typeDescHandle ,
RFC_ERROR_INFO* errorInfo);

      typeDescHandle : handle to the type descriptor;
```

RfcCloneStructure

Use

Creates a copy of the given structure and returns the handle to the new object.

Structure

```
    RFC_STRUCTURE_HANDLE SAP_API
RfcCloneStructure(RFC_STRUCTURE_HANDLE structureHandle,
RFC_ERROR_INFO* errorInfo);
```

```
    structureHandle : handle to the original structure data object.
```

RfcDestroyStructure

Use

Deletes the given structure data object.

Structure

```
   RFC_RC SAP_API RfcDestroyStructure(RFC_STRUCT_HANDLE structHandle,
RFC_ERROR_INFO* errorInfo);


      structHandle : handle to the to be deleted structure data
object .
```

Field-Handling Functions

In the following section you will find a description of the functions you can use for accessing the fields of a data container.

Functions

- RfcGetChars
- RfcGetNum
- RfcGetDate
- RfcGetTime
- RfcGetString
- RfcGetBytes
- RfcGetXString
- RfcGetInt
- RfcGetInt1
- RfcGetInt2
- RfcGetFloat
- RfcGetDecF16
- RfcGetDecF34
- RfcGetStructure
- RfcGetTable
- RfcGetStringLength
- RfcSetChars
- RfcSetNum
- RfcSetDate
- RfcSetTime
- RfcSetString
- RfcSetBytes
- RfcSetXString
- RfcSetInt
- RfcSetInt1
- RfcSetInt2
- RfcSetFloat
- RfcSetDecF16
- RfcSetDecF34
- RfcSetStructure

- RfcSetTable

RfcGetChars

Returns the value of the specified field as char array. The charBuffer will be filled by string representation. The remaining places in the buffer will be filled with spaces. In case the buffer too small the function will return `RFC_BUFFER_TOO_SMALL`.

Supported field types:

- `RFCTYPE_CHAR`
- `RFCTYPE_STRING`
- `RFCTYPE_NUM`
- `RFCTYPE_DATE`
- `RFCTYPE_TIME`
- `RFCTYPE_INTx`
- `RFCTYPE_FLOAT`
- `RFCTYPE_BCD`
- `RFCTYPE_DECFxx`

Structure

```
RFC_RC SAP_API RfcGetChars(DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, RFC_CHAR *charBuffer, unsigned bufferLength,
RFC_ERROR_INFO* info);
```

RfcGetNum

Use

Returns the value of the specified field as num-char array. The `charBuffer` will be filled by string representation. The remaining places in the buffer will be filled with spaces. In case the buffer too small the function will return `RFC_BUFFER_TOO_SMALL`.

Supported field types:

- `RFCTYPE_CHAR`
- `RFCTYPE_STRING`
- `RFCTYPE_NUM`
- `RFCTYPE_DATE`
- `RFCTYPE_TIME`
- `RFCTYPE_INTx`

Structure

```
RFC_RC SAP_API RfcGetNum (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, RFC_NUM *charBuffer, unsigned bufferLength,
RFC_ERROR_INFO* info);
```

RfcGetDate

Supported field type:

- `RFCTYPE_DATE`

Structure

```
RFC_RC SAP_API RfcGetDate (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, RFC_DATE emptyDate, RFC_ERROR_INFO* info);
```

RfcGetTime

Supported field type:

- `RFCTYPE_TIME`

Structure

```
RFC_RC SAP_API RfcGetTime  (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, RFC_TIME emptyTime, RFC_ERROR_INFO* info);
```

RfcGetString

Use

Returns the value of the specified field as null-terminated string. The `charBuffer` will be filled by string representation. In case the buffer is too small e.g. no place for string termination, the function will return `RFC_BUFFER_TOO_SMALL`.

Supported field types:

- `RFCTYPE_CHAR`
- `RFCTYPE_STRING`
- `RFCTYPE_NUM`
- `RFCTYPE_DATE`
- `RFCTYPE_TIME`
- `RFCTYPE_INTx`
- `RFCTYPE_FLOAT`
- `RFCTYPE_BCD`
- `RFCTYPE_DECFxx`

Structure

```
RFC_RC SAP_API RfcGetString  (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, SAP_UC *stringBuffer, unsigned bufferLength,
unsigned* stringLength, RFC_ERROR_INFO* info);
```

RfcGetBytes

Use

Returns the value of the specified field as byte array. In case the buffer is too small the function will return `RFC_BUFFER_TOO_SMALL`. In case the buffer is longer than the field it will be filled with a 0x00 value.

Supported field types:

- `RFCTYPE_BYTE`
- `RFCTYPE_XSTRING`
- `RFCTYPE_CHAR`
- `RFCTYPE_NUM`
- `RFCTYPE_DATE`
- `RFCTYPE_TIME`
- `RFCTYPE_INTx`
- `RFCTYPE_FLOAT`
- `RFCTYPE_BCD`
- `RFCTYPE_DECFxx`

Structure

```
RFC_RC SAP_API RfcGetBytes  (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, SAP_RAW *byteBuffer, unsigned bufferLength,
RFC_ERROR_INFO* info);
```

RfcGetXString

Use

Returns the value of the specified field as byte array. In case the buffer is too small the function will return RFC_BUFFER_TOO_SMALL. xstringLength contains the number of written bytes. The remaining buffer wont be set and can contain random values.

Supported field types:

- RFCTYPE_BYTE
- RFCTYPE_XSTRING
- RFCTYPE_CHAR
- RFCTYPE_NUM
- RFCTYPE_DATE
- RFCTYPE_TIME
- RFCTYPE_INTx
- RFCTYPE_FLOAT
- RFCTYPE_BCD
- RFCTYPE_DECFxx

Structure

```
RFC_RC SAP_API RfcGetXString  (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, SAP_RAW *byteBuffer, unsigned bufferLength,
unsigned* xstringLength, RFC_ERROR_INFO* info);
```

RfcGetInt

Use

Returns the value of the specified field as `RFC_INT`

Supported field types:

- `RFCTYPE_INT`
- `RFCTYPE_INT2`
- `RFCTYPE_INT1`
- `RFCTYPE_BYTE`
- `RFCTYPE_CHAR`
- `RFCTYPE_NUM`

Structure

```
RFC_RC SAP_API RfcGetInt      (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, RFC_INT  *value, RFC_ERROR_INFO* info);
```

RfcGetInt1

Supported field types:

- `RFCTYPE_INT`
- `RFCTYPE_INT2`
- `RFCTYPE_INT1`

Structure

```
RFC_RC SAP_API RfcGetInt1     (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, RFC_INT1 *value, RFC_ERROR_INFO* info);
```

RfcGetInt2

Supported field types:

- `RFCTYPE_INT`
- `RFCTYPE_INT2`
- `RFCTYPE_INT1`

Structure

```
RFC_RC SAP_API RfcGetInt2 (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, RFC_INT2 *value, RFC_ERROR_INFO* info);
```

RfcGetFloat

Supported field types:

- RFCTYPE_FLOAT
- RFCTYPE_BCD

Structure

```
RFC_RC SAP_API RfcGetFloat    (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, RFC_FLOAT *value, RFC_ERROR_INFO* info);
```

RfcGetDecF16

Supported field types:

- RFCTYPE_DECF16
- RFCTYPE_DECF34
- RFCTYPE_FLOAT
- RFCTYPE_BCD
- RFCTYPE_INT
- RFCTYPE_INT2
- RFCTYPE_INT1
- RFCTYPE_CHAR
- RFCTYPE_BYTE

Structure

```
RFC_RC SAP_API RfcGetDecF16   (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, RFC_DECF16 *value, RFC_ERROR_INFO* info);
```

RfcGetDecF34

Supported field types:

- RFCTYPE_DECF16
- RFCTYPE_DECF34
- RFCTYPE_FLOAT
- RFCTYPE_BCD
- RFCTYPE_INT
- RFCTYPE_INT2
- RFCTYPE_INT1
- RFCTYPE_CHAR
- RFCTYPE_BYTE

Structure

```
RFC_RC SAP_API RfcGetDecF34 (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, RFC_DECF34 *value, RFC_ERROR_INFO* info);
```

RfcGetStructure

Supported type:

- `RFCTYPE_STRUCTURE`

Structure

```
RFC_RC SAP_API RfcGetStructure(DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, RFC_STRUCTURE_HANDLE* structHandle,
RFC_ERROR_INFO* info);
```

RfcGetTable

Supported field type:

- RFCTYPE_TABLE

Structure

```
RFC_RC SAP_API RfcGetTable    (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, RFC_TABLE_HANDLE* tableHandle, RFC_ERROR_INFO*
info);
```

RfcGetStringLength

Returns the length of the value of a STRING or XSTRING parameter, so an appropriately sized buffer can be allocated before reading the value.

Structure

```
RFC_RC SAP_API RfcGetStringLength(DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, unsigned* stringLength, RFC_ERROR_INFO* info);
```

RfcSetChars

Sets the given char value (`charValue/valueLength`) into the field.

Supported field types:

- `RFCTYPE_CHAR`
- `RFCTYPE_STRING`
- `RFCTYPE_NUM`
- `RFCTYPE_DATE`
- `RFCTYPE_TIME`
- `RFCTYPE_INTx`
- `RFCTYPE_FLOAT`
- `RFCTYPE_BCD`
- `RFCTYPE_DECFxx`
- `RFCTYPE_BYTE`
- `RFCTYPE_XSTRING`

If the target field has type `BYTE` or `XSTRING` the char value will treated as hex encoded string representation of the bytes.

Structure

```
RFC_RC SAP_API RfcSetChars    (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, const RFC_CHAR *charValue, unsigned valueLength,
RFC_ERROR_INFO* info);
```

RfcSetNum

Structure

```
RFC_RC SAP_API RfcSetNum   (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_NUM *charValue, unsigned valueLength,
RFC_ERROR_INFO* info);
```

RfcSetDate

Supported field type:

- `RFCTYPE_DATE`

Structure

```
RFC_RC SAP_API RfcSetDate  (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_DATE date, RFC_ERROR_INFO* info);
```

RfcSetTime

Structure

```
RFC_RC SAP_API RfcSetTime (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_TIME time, RFC_ERROR_INFO* info);
```

RfcSetString

Structure

```
RFC_RC SAP_API RfcSetString (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const SAP_UC *stringValue, unsigned valueLength,
RFC_ERROR_INFO* info);
```

RfcSetBytes

Sets the given char value (`charValue/valueLength`) into the field.

Supported field types:

- `RFCTYPE_BYTE`
- `RFCTYPE_XSTRING`
- `RFCTYPE_CHAR`
- `RFCTYPE_STRING`
- `RFCTYPE_INTx`
- `RFCTYPE_FLOAT`
- `RFCTYPE_BCD`
- `RFCTYPE_DECFxx`

If the target field has type `CHAR` or `STRING` the byte value will be stored as a hex representation of the bytes.

If the target field has number type, the byte value will be assigned only if the given length fits to the field length.

Structure

```
RFC_RC SAP_API RfcSetBytes (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const SAP_RAW *byteValue, unsigned valueLength,
RFC_ERROR_INFO* info);
```

RfcSetXString

Structure

```
RFC_RC SAP_API RfcSetXString  (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, const SAP_RAW *byteValue, unsigned valueLength,
RFC_ERROR_INFO* info);
```

RfcSetInt

Structure

```
RFC_RC SAP_API RfcSetInt    (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_INT  value, RFC_ERROR_INFO* info);
```

RfcSetInt1

Structure

```
RFC_RC SAP_API RfcSetInt1 (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_INT1 value, RFC_ERROR_INFO* info);
```

RfcSetInt2

Structure

```
RFC_RC SAP_API RfcSetInt2  (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_INT2 value, RFC_ERROR_INFO* info);
```

RfcSetFloat

Supported field types:

- `RFCTYPE_FLOAT`
- `RFCTYPE_BCD`

Structure

```
RFC_RC SAP_API RfcSetFloat (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_FLOAT value, RFC_ERROR_INFO* info);
```

RfcSetDecF16

Supported field types:

- RFCTYPE_DECF16
- RFCTYPE_DECF34
- RFCTYPE_FLOAT
- RFCTYPE_BCD
- RFCTYPE_INT
- RFCTYPE_INT2
- RFCTYPE_INT1
- RFCTYPE_CHAR
- RFCTYPE_BYTE

Structure

```
RFC_RC SAP_API RfcSetDecF16 (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_DECF16 value, RFC_ERROR_INFO* info);
```

RfcSetDecF34

Supported field types:

- `RFCTYPE_DECF16`
- `RFCTYPE_DECF34`
- `RFCTYPE_FLOAT`
- `RFCTYPE_BCD`
- `RFCTYPE_INT`
- `RFCTYPE_INT2`
- `RFCTYPE_INT1`
- `RFCTYPE_CHAR`
- `RFCTYPE_BYTE`

Structure

```
RFC_RC SAP_API RfcSetDecF34   (DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, const RFC_DECF34 value, RFC_ERROR_INFO* info);
```

RfcSetStructure

Supported field type:

- `RFCTYPE_STRUCTURE`

Structure

```
RFC_RC SAP_API RfcSetStructure(DATA_CONTAINER_HANDLE dataHandle,
SAP_UC const* name, const RFC_STRUCTURE_HANDLE value, RFC_ERROR_INFO*
info);
```

RfcSetTable

Supported field type:

- `RFCTYPE_TABLE`

Structure

```
RFC_RC SAP_API RfcSetTable (DATA_CONTAINER_HANDLE dataHandle, SAP_UC
const* name, const RFC_TABLE_HANDLE value, RFC_ERROR_INFO* info);
```

Metadata and Repository Functions

In the following section you will find a description of the functions you can use for accessing the fields of a data container.

Functions

- RfcGetFunctionDesc
- RfcGetCachedFunctionDesc
- RfcAddFunctionDesc
- RfcXXXTypeDesc()
- RfcCreateTypeDesc
- RfcAddTypeField
- RfcSetTypeLength
- RfcGetTypeName
- RfcGetFieldCount
- RfcGetFieldDescByIndex
- RfcGetFieldDescByName
- RfcGetTypeLength
- RfcDestroyTypeDesc
- RfcCreateFunctionDesc
- RfcAddParameter
- RfcEnableBASXML
- RfcGetFunctionName
- RfcGetParameterCount
- RfcGetParameterDescByIndex
- RfcGetParameterDescByName
- RfcDestroyFunctionDesc
- RfcGetRcAsString
- RfcGetTypeAsString
- RfcGetDirectionAsString

RfcGetFunctionDesc

Use

Returns the function description that is valid for the system to which `rfcHandle` points to. If the function description is already in the repository cache for that system ID, it will be returned immediately (from the cache), otherwise it will be looked up in the system's DDIC using the `rfcHandle`. The result from the DDIC lookup will then be placed in the cache. The RFC Runtime maintains a cache for every SAP System ID, as the meta data may be different depending on the SAP release. This is the main API that should be used.

Structure

```
RFC_FUNCTION_DESC_HANDLE SAP_API
RfcGetFunctionDesc(RFC_CONNECTION_HANDLE rfcHandle, SAP_UC const *
funcName, RFC_ERROR_INFO* info);
```

RfcGetCachedFunctionDesc

Returns a cached function description. If `repositoryID` is `NULL`, the "default storage" is used.

This API should be used with care and is only for special scenarios, for example:

1. You know for sure, that a function description has already been cached via `RfcGetFunctionDesc()`, and don't want to open an extra `rfcHandle` that will never be used. In this case simply use the SAP System ID as the `repositoryID`.
2. You have created a hard-coded repository via `RfcAddFunctionDesc()`, which contains functions that do not exist in the backend's DDIC.

Structure

```
RFC_FUNCTION_DESC_HANDLE SAP_API RfcGetCachedFunctionDesc(SAP_UC
const * repositoryID, SAP_UC const * funcName, RFC_ERROR_INFO* info);
```

RfcAddFunctionDesc

Use

Adds a function description to the cache of the specified repository.

If `repositoryID` is `NULL`, the "default storage" is used.

This API should be used with care and is only for special scenarios.

Structure

```
RFC_RC SAP_API RfcAddFunctionDesc(SAP_UC const * repositoryID,
RFC_FUNCTION_DESC_HANDLE funcHandle, RFC_ERROR_INFO* info);
```

RfcXXXTypeDesc()

Use

The `RfcXXXTypeDesc()` functions

- `RfcGetTypeDesc`
- `RfcGetCachedTypeDesc`
- `RfcAddTypeDesc`

are analog to the corresponding `RfcXXXFunctionDesc()` functions.

Structure

```
RFC_TYPE_DESC_HANDLE SAP_API RfcGetTypeDesc(RFC_CONNECTION_HANDLE
rfcHandle, SAP_UC const * typeName, RFC_ERROR_INFO* info);


RFC_TYPE_DESC_HANDLE SAP_API RfcGetCachedTypeDesc(SAP_UC const *
repositoryID, SAP_UC const * typeName, RFC_ERROR_INFO* info);


RFC_RC SAP_API RfcAddTypeDesc(SAP_UC const * repositoryID,
RFC_TYPE_DESC_HANDLE typeHandle, RFC_ERROR_INFO* info);
```

RfcCreateTypeDesc

Add a new field to the type description. After the handle was used any modifications are forbidden.

Structure

```
RFC_RC SAP_API RfcAddTypeField(RFC_TYPE_DESC_HANDLE typeHandle, const
RFC_FIELD_DESC* fieldDescr, RFC_ERROR_INFO* info);
```

RfcAddTypeField

Use

Adds a new field to the type description. After the handle was used any modifications are forbidden.

Structure

```
RFC_RC SAP_API RfcAddTypeField(RFC_TYPE_DESC_HANDLE typeHandle, const
RFC_FIELD_DESC* fieldDescr, RFC_ERROR_INFO* info);
```

RfcSetTypeLength

Structure

```
RFC_RC SAP_API RfcSetTypeLength(RFC_TYPE_DESC_HANDLE typeHandle,
unsigned nucByteLength, unsigned ucByteLength, RFC_ERROR_INFO* info);
```

RfcGetTypeName

Structure

```
RFC_RC SAP_API RfcGetTypeName(RFC_TYPE_DESC_HANDLE typeHandle,
RFC_ABAP_NAME bufferForName, RFC_ERROR_INFO* info);
```

RfcGetFieldCount

Structure

```
RFC_RC SAP_API RfcGetFieldCount(RFC_TYPE_DESC_HANDLE typeHandle,
unsigned* count, RFC_ERROR_INFO* info);
```

RfcGetFieldDescByIndex

Structure

```
RFC_RC SAP_API RfcGetFieldDescByIndex(RFC_TYPE_DESC_HANDLE
typeHandle, unsigned index, RFC_FIELD_DESC* fieldDescr,
RFC_ERROR_INFO* info);
```

RfcGetFieldDescByName

Structure

```
RFC_RC SAP_API RfcGetFieldDescByName(RFC_TYPE_DESC_HANDLE typeHandle,
SAP_UC const* name, RFC_FIELD_DESC* fieldDescr, RFC_ERROR_INFO*
info);
```

RfcGetTypeLength

Structure

```
RFC_RC SAP_API RfcGetTypeLength(RFC_TYPE_DESC_HANDLE typeHandle,
unsigned* nucByteLength, unsigned* ucByteLength, RFC_ERROR_INFO*
info);
```

RfcDestroyTypeDesc

Deletes the type description and release the allocated resources. Only description, that is not stored in a repository cache and not used by application can be deleted. Deleting of cached description will cause an error and deleting of description that is still in use will lead to crash.

Structure

```
RFC_RC SAP_API RfcDestroyTypeDesc(RFC_TYPE_DESC_HANDLE typeHandle,
RFC_ERROR_INFO *info);
```

RfcCreateFunctionDesc

Creates an empty function description with the given name. After invokation the `funcHandle` points to a new `RFC_FUNCTION_DESC_HANDLE`.

This handle has to be used for the `RfcFunctionDescrAddParameter` API.

Structure

```
RFC_FUNCTION_DESC_HANDLE SAP_API RfcCreateFunctionDesc(SAP_UC const*
name, RFC_ERROR_INFO* info);
```

RfcAddParameter

Use

Adds a new parameter to the type description. After the handle was used any modifications are forbidden.

Structure

```
RFC_RC SAP_API RfcAddParameter(RFC_FUNCTION_DESC_HANDLE funcHandle,
const RFC_PARAMETER_DESC* paramDescr, RFC_ERROR_INFO* info);
```

RfcEnableBASXML

Structure

```
RFC_RC SAP_API RfcEnableBASXML(RFC_FUNCTION_DESC_HANDLE funcHandle,
RFC_ERROR_INFO* info);
```

RfcGetFunctionName

Structure

```
RFC_RC SAP_API RfcGetFunctionName(RFC_FUNCTION_DESC_HANDLE
funcHandle, RFC_ABAP_NAME bufferForName, RFC_ERROR_INFO* info);
```

RfcGetParameterCount

Structure

```
RFC_RC SAP_API RfcGetParameterCount(RFC_FUNCTION_DESC_HANDLE
funcHandle, unsigned* count, RFC_ERROR_INFO* info);
```

RfcGetParameterDescByIndex

Structure

```
RFC_RC SAP_API RfcGetParameterDescByIndex(RFC_FUNCTION_DESC_HANDLE
funcHandle, unsigned index, RFC_PARAMETER_DESC* paramDesc,
RFC_ERROR_INFO* info);
```

RfcGetParameterDescByName

Structure

```
RFC_RC SAP_API RfcGetParameterDescByName(RFC_FUNCTION_DESC_HANDLE
funcHandle, SAP_UC const* name, RFC_PARAMETER_DESC* paramDesc,
RFC_ERROR_INFO* info);
```

RfcIsBASXMLSupported

Structure

```
RFC_RC SAP_API RfcIsBASXMLSupported(RFC_FUNCTION_DESC_HANDLE
funcHandle, int* isAble, RFC_ERROR_INFO* info);
```

RfcDestroyFunctionDesc

Deletes the function description and release the allocated resources. Only descriptions, which are not stored in a repository cache and not used by the application, can be deleted. Deleting a cached description will cause an error, and deleting a description that is still in use, will lead to a crash.

Structure

```
RFC_RC SAP_API RfcDestroyFunctionDesc(RFC_FUNCTION_DESC_HANDLE
funcHandle, RFC_ERROR_INFO *info);
```

RfcGetRcAsString

Structure

```
const SAP_UC* RfcGetRcAsString(RFC_RC rc);
```

RfcGetTypeAsString

Structure

```
const SAP_UC* RfcGetTypeAsString(RFCTYPE type);
```

RfcGetDirectionAsString

Structure

```
const SAP_UC* RfcGetDirectionAsString(RFC_DIRECTION direction);
```