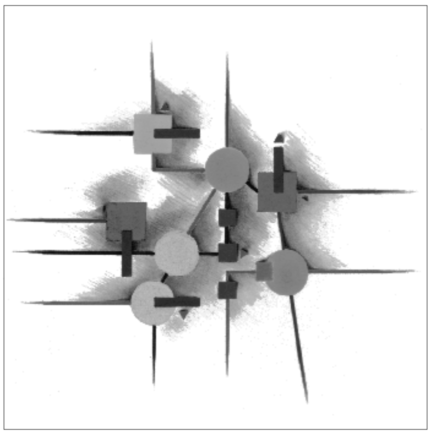
RFCSDK Guide

SAP Release 6.40



SAP SYSTEM

SAP® AG • Neurottstr. 16 • D-69190 Walldorf

Copyright

Contents

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`.

Introduction

The RFC Library offers an interface to a R/3 system. The RFC library is the most commonly used and installed component of existing SAP software. This interface provides the opportunity to call any RFC function in a SAP system from an external application. Moreover, the RFC Library offers the possibility to write a RFC server program, which is accessible from any R/3 system or an external application. Most R/3 connectors use the RFC library as communication platform to SAP systems.

SAP provides the RFC API in the form of C-routines, incorporated in the RFC-Library. RFCSDK is available on all SAP supported platforms. RFCSDK contains:

- a platform specific RFC library,
- four platform independent header files (saprfc.h, sapitab.h, sapucx.h, sapuc2.h),
- some sample RFC programs as srfctest.c, srfcserv.c etc.

The RFC library is **forward and backward compatible,** i.e., an older release of the RFC Library can communicate with an R/3 system at a higher version and vice versa.

The most important design features of the RFC Library are:

- Working with the native RFC protocol.
- Maximum functionality, i.e., almost all features of RFC in SAP R/3 systems should be supported by the RFC Library too.
- Maximum performance.
- Maximum flexibility.
- Full compatibility to other RFC releases.

Naturally, this is sometimes at the expense of the usability of RFC API's. But every attempt to simplify the interface, will be at the expense of all the design goals described above.

The RFC Library exists in a Unicode and non-Unicode version. Thus, the Unicode RFC Library is **forward, backward and sideward** compatible, i.e., a current Unicode RFC application can communicate with any non-Unicode RFC application independently of its release and vice versa.

As expected by modern software, the RFC Library could be used in a multithreaded environment. However, the RFC Library itself **thread safe**. This enables the user to write powerful multithreaded RFC applications.

This Guide describes the current RFC Library Release 7.0.

Basic knowledge

The RFC client is the instance who calls the remote function module, which is provided by **a RFC server**.

The RFC calls are accomplished via a RFC connection. Up to 100 (default value) active RFC connections at a time are possible. This value can be changed by using environment variable CPIC_MAX_CONV.

There are two types of RFC connections:

- **Client connection**.

- **Server connection**. Sometimes the server connection is called **accepted** connection.

A RFC handle represents a RFC connection. Technically a handle is an index of an internal memory structure, which contains necessary information about given RFC connections. A RFC handle is an instance of type RFC_HANDLE. There are two kinds of RFC handles:

- **Invalid RFC handle**. This is either an RFC handle with value RFC_HANDLE_NULL or an already closed or aborted RFC connection. It is impossible to make any API call via this handle.
- **Valid RFC handle**. API calls have to be done by using a valid RFC handle.

A try to call any API-function with an invalid handle causes an error with return code RFC_INVALID_HANDLE.

`RfcGetAttributes` API delivers the user relevant data describing a given RFC connection.

During execution of an RFC function at the server side, it is sometimes useful to call another RFC function in the original (client) system. For example, if one needs data from the caller before continuing with the current RFC function. This behavior is called **call back feature** in SAP jargon. The RFC server uses the same RFC connection established by the client and calls a remote function at the client side. The call back hit the same context in R/3 system. RFC Library supports a call back mechanism for client and server. It is not necessary to open a new connection, i.e., the same RFC handle will be used.

RFC Client Program

An external RFC client program calls a remote RFC function. The following steps have to be conducted by an external RFC client:

1. Open an RFC-Connection. In this phase a valid RFC client handle will be created.
2. Call one or more remote RFC functions.
3. Close or terminate an RFC connection. The valid RFC handle will invalidate. All connection data is lost.

Open a RFC-Connection

To create a valid RFC client handle, it is necessary to establish a RFC connection to a RFC partner. The following connection types are possible:

- Connection to a R/2-System
- Connection to a R/3 System
- Connection to an external RFC server

Every RFC connection will be established in two steps:

- Connection from a RFC program to a SAP gateway
- Connection from the SAP gateway to a RFC server

A connection to a remote RFC server has to be established via `RfcOpenEx` API-call. This API takes as input parameter a zero terminated string with connection data and returns a RFC handle. If the connection was successfully established, the handle is valid. If the handle is returned, it is invalid and has the value RFC_HANDLE_NULL.

It is possible to precede the logon check during opening of a RFC connection.

To create a RFC connection the following data is needed:

- **Connection data**. This data is necessary to establish the connection to the RFC server.
- **Logon data** (user identity, client and logon language). This data is used for logon to the given R/3 system.
- **Additional data.** By using this data, some connection features can be customized. For example, it is possible to turn the RFC trace on, start SAPGUI to open time etc.

Abbreviator BAER2 (B - load Balancing, A - explicit Application server, E - External RFC program, R - Registered RFC server, 2 - R/2 connection) could describe the connection kinds.

The connection string has to have the following format: "`ID=value ID=value ... ID=value`".

```
RFC_HANDLE          handle = RFC_HANDLE_NULL;
RFC_ERROR_INFO_EX   error_info_ex;
rfc_char_t          connect_string[] = " … ";

handle = RfcOpenEx (connect_string, &error_info_ex);
```

```
if (handle == RFC_HANDLE_NULL)
{
  /* Connection could not be established
  ...
}
…
```

It is possible to only declare the data in the connection string. In this case all necessary connection data is defined as input parameter of `RfcOpenEx` API.

Another way is to declare the data in the saprfc.ini file. To use the saprfc.ini file, the connection parameter string must have the following entry: "DEST=<your destination in saprfc.ini file>". In this case the corresponding entries from the sapRFC.ini file will be used to establish the connection.

A third possibility is to use the SAPLOGON data (this is available for Windows platforms only). To use the connection data from sapmsg.ini file, the connection string must have the following entry: "SAPLOGON_ID=<entry from saplogon> ".

Connection to a R/3 System

There are two types of connection to a R/3-System:

- Connection to an explicit application host
- Connection to a R/3 via Load Balancing

Connection to an explicit application server (Connection type A)

You remember that the connection to an external system is established in two steps (connection to a SAP gateway and connection from a SAP gateway to a R/3 system). To establish a connection to an explicit application server the following information is needed:

- Application host name and system number. This information is mandatory.
- Gateway host name and gateway services. This information is optional. The default is the local gateway on the application server itself.

The gateway options are useful if the gateway is running on a different host. Another host could be located in another network with a different network card and could have a differing IP-address.

The data can be assigned:

- Either via a connection string using the following entries: `"… TYPE=3 ASHOST=<host name of application server> SYSNR=<R/3 system number> GWHOST=<optional: gateway host; default: local gateway on application server> GWSERV=<optional: gateway service; default: local gateway on application server> …".`
- Or via following entries in saprfc.ini file for corresponding destination:

```
DEST = ...
TYPE = A
ASHOST=<host name of application server>
SYSNR=<R/3 system number>
GWHOST=<optional: gateway host; default: local
gateway on application server> GWSERV=<optional:
gateway service; default: local gateway on application server>.
```

The host name and service name of the specific application server have to be defined in 'host' and 'service' files. The service name for the application server has to be defined as follows:

<service name>=**sapdp**<R/3 system number>.

The host name and service name of the SAP gateway have to be defined in the 'host' and 'service' files. The service name for the SAP gateway has to be defined as follows:

<service name>=**sapgw**<R/3 system number>.

Please notice: If GWHOST and GWSERV are not used (this is optional), the service name of the SAP gateway has to be defined in the 'service' files anymore.

Load balancing feature (Connection type B)

It is possible to connect to a R/3 system via "Load Balancing" feature. This functionality enables the user to connect to an application server with the minimum load within a group of predefined application servers. This feature has the following advantages:

- The load in a R/3 system is distributed among different application servers
- The application server will be determined at run time. In this case, the RFC connection is independent of a specific RFC server. This improves the flexibility of your RFC application
- Only host name and port number of the R/3 message server will be made known in your system.

To establish a connection to a R/3 system via Load Balancing the following information is necessary:

- **Message Server host name**. This information is optional. If no information about host name of message server is available, RFC library tries to retrieve this information from `sapmsg.ini` file according given system name. The sapmsg.ini file is usually used by SAP-GUI. If the SAPGUI is not installed on your system, you may copy the sapmsg.ini into your working directory or into Windows-Directory (MS-Windows platforms only) manually.
- **Message Server services**. This information is optional. Default is **sapms<R/3 system name>**.
- **System name.** This information is mandatory.
- **Application servers group name.** This information is optional. The default is the group 'PUBLIC'.

Please notice: The data could be assigned:

- Either via connection string with following entries: " … TYPE=3 MSHOST=<message server host name> R3NAME=<system name> MSSERV=<message server service> GROUP=<group of application servers> …".

  Or via the following entries in the saprfc.ini file for the corresponding destination:

```
DEST= …
TYPE=B
MSHOST=<message server host name>
R3NAME=<system name>
GROUP=<group of application servers; optional, default 'PUBLIC'>
```

The host name and service name of the message server have to be defined in the 'hosts' and 'services' files. The service name has to be defined as follows:

<service name>=**sapms**<R/3 system name>.

The connection string has to have the following entries:

"TYPE=3 MSHOST=<message server host name> R3NAME=<system name> GROUP=<group of application servers> …"

Logon data

To log on a SAP system the user has to be initialized. The following logon data is needed:

- Client. This entry describes the client in the connect string: CLIENT=000
- Language. Entry describes the logon language has the format: LANG=<ISO-Language>.
- User identity information

The user can be identified via:

- Username and password. Format is: USER=<user name> PASSWD=<password>
- Mysapsso2 certificate. In this case password is not required and user name is optional.
- X509 certificate.
- EXTID data.

Connection to an external RFC program

Sometimes it is useful to write both, RFC client and server program, as external programs and let them to communicate with each other via a SAP gateway. The RFC Library supports this option too. The RFC server program can be started by a SAP gateway or it can use a registering feature

Connection to an external program started by a SAP gateway

The RFC client has to open the connection of type 'E'. For successful build up of the connection the following information is needed:

- Gateway data: gateway host name and service names,
- Host name of the rfc server program,
- Name of the rfc server program.

This data could be delivered to RfcOpenEx:

- Either directly via entries in connection string: "… TYPE=E GWHOST=<gateway host> GWSERV=<gateway service name> TPHOST=<host name of rfc server> TPNAME=<name of the rfc server program> …"
- Or via entries in saprfc.ini file for corresponding destination:
  DEST= …
  TYPE=E
  GWHOST=<gateway host>
  GWSERV=<gateway service name>
  TPHOST=<host name of rfc server>
  TPNAME=<name of the rfc server program>

Connection to a registered RFC server

To establish a RFC connection to an external registered RFC server, the RFC client has to open a connection of type 'R'. The following data is needed:

- Gateway data: gateway host name and gateway service name
- Program Id of the external RFC server.

This data could be assigned:

- Either directly via entries in the connection string: "… TYPE=E GWHOST=<gateway host> GWSERV=<gateway service name> TPNAME=<program Id of RFC server program> …"
- Or via entries in the saprfc.ini file for the corresponding destination:
  DEST= …
  TYPE=R
  GWHOST=<gateway host>
  GWSERV=<gateway service name>
  PROGID=<program Id of RFC server program>

Please notice: If an external RFC client communicates with an external server, which runs in registered mode, the RFC client has to close the connection and reconnect the RFC server after each RFC function call. If this is not done, the behavior is undefined. This behavior is based on the following background:

- The registering feature of an external rfc server has been designed to allow the multi-client functionality. This means that an external server may be able to serve several requests from different clients simultaneously.
- To enable this, closes external rfc server after each rfc call the connection to the client. The connection will be closed independent of the client's kind (external or R/3-system). In case of an ABAP-Program the connection at the client side will be reestablished automatically by the ABAP/RFC-Runtime. In contrast external RFC-Client program has to reestablish the connection explicitly via RfcOpenEx-API.

It is possible to create a special rfc connection to an external registered server. This connection kind is called **explicitly bound server**. In this case the external server is able to server only request from a once client and the connection is **not closed** after each rfc request. Of course the external rfc server is not able to server request from another clients until the connection is explicitly bound. How to do it, is closely described in the Chapter 0.

Call a RFC function

There are four kinds of results of an RFC-call:

1. **Successful.** In this case exporting data and tables were transported to remote system and the local system received the importing data and tables.
2. **Exception**. In this case the callee (remote system) raised an application exception. This means that:

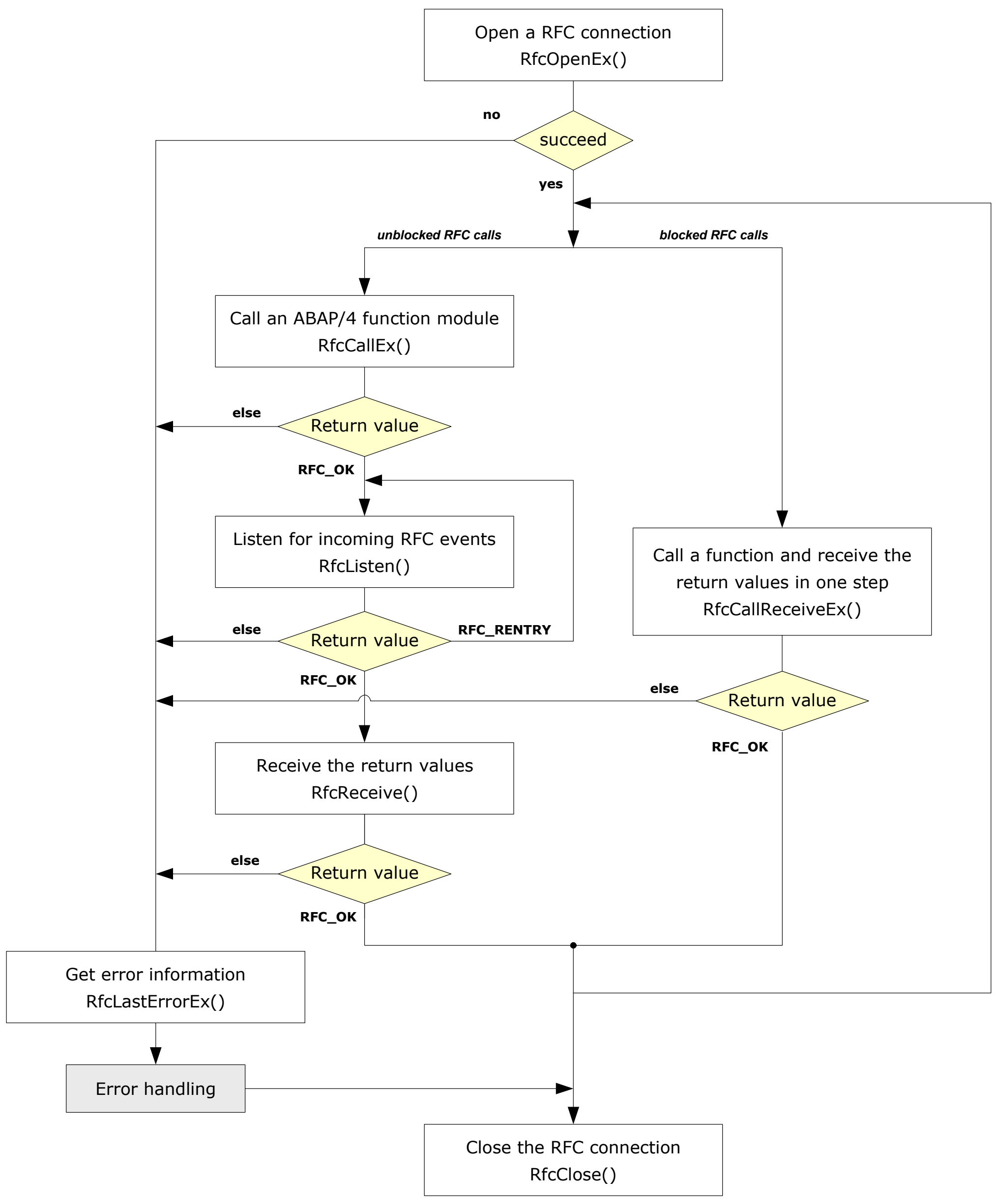
- The exporting data has been transferred to the remote system.
- Remote function raised an application exception.
- Exception has been transferred to the caller.
- The importing data was not transferred to caller (local system).
- RFC connection is not closed.

3. **System Exception**. A system exception was raised by runtime. Connection is closed.
4. **Failure.** Call unsuccessful. Connection is closed. No warranty for data transport neither to remote system nor from remote system.

Important: Every call of any RFC function has to be conducted via a valid RFC handle. It is recommended to call RfcClose after each failure of an RFC call.

A RFC call can be performed as blocked or unblocked call. The blocked call waits until the called ABAP/4 function module is finished on the server side before it returns to the program. By an unblocked call one has the possibility to execute some code at client side at the same time the ABAP/4 function module is execute on the server.

It follows a flow diagram showing the basic structure of a RFC client program.



Flow diagram for a RFC client program without call-back.

Blocking RFC Calls

The API RfcCallReceiveEx implements the whole part of an RFC call, i.e., the exporting and changing data are already sent to the RFC server and importing and changing data are already received if the API returns.

Example:
```
int main (itn argc, rfc_char_t ** argv)
{
     RFC_HANDLE          handle;
     RFC_RC                   rfc_rc;
     RFC_ERROR_INFO_EX        error_info_ex;
     rfc_char_t        * exception = NULL;
```

```
          rfc_char_t          function_name[] = “ABC”;
          RFC_PARAMETER       importing[2],
                              exporting[2],
                              changing[2];
          RFC_TABLES          tables[2];
          RFC_INT                 value1;

          handle = RfcOpenEx (« « , &error_info_ex);

          if (handle == RFC_HANDLE_NULL)
          {
            /* Could not establish an rfc connection */
            ...
            return 1;
          }

          importing[0].name = “VALUE1”
          importing[0].nlen = strlen (importing[0]. name);
          importing[0].type = RFCTYPE_INT;
          importing[0].addr = &value1;
          importing[0].size  = sizeof (value1);

          importing[1].name = NULL;

          rfc_rc = RfcCallReceiveEx (handle, function_name,
                                     importing, exporting,
                                     changing, tables,
      &exception);

        if (rfc_rc == RFC_EXCEPTION)
        {
          /* Catch an application exception.
                   No data were transported.  Rfc connection
      is not closed. */
          printf (“Exception raised: %s\n”, exception);

        }
        else if(rfc_rc == RFC_SYS_EXCEPTION)
        {
          /* System exception raised. No data were transported.
                   Connection is closed by rfc runtime. */
          printf (“System Exception raised: %s\n”, exception);
          printf (“RFC connection closed by runtime\n”);
        }
        else if(rfc_rc != RFC_OK)
        {
          /* An error occurred */
          RfcClose (handle);
          ...
        return 1;
      }

      ...
      RfcClose (handle);
    }
```

Unblocking RFC Calls

Sometimes a RFC call is very long. In this case, the waiting for the answer takes a lot of time. In the meantime the caller could do some work. To do this, it is necessary to split the RFC call into three components:

- `RfcCallEx` – sends exporting and changing parameters to RFC *buffer*. The data is not really sent to the RFC server with the RfcCallEx call. The data is only marshalled (serialized) and put into the RFC buffer. The success of RfcCallEx means, that the marshalling (serialization) of data was successful. Only after RfcListen or RfcReceiveEx are called, the data will be sent to the server.
- `RfcListen` – sends the data to a server from the RFC buffer (remember RfcCallEx does not send data to a server over a network) and checks whether the imported data is received from the server. This call returns immediately and delivers the information, whether an event has occurred. An event can be an error, or data is received, or nothing happened. This call enables useful waiting. It is possible to do some work until the data to be imported is received from the server. If the event "data received" has occurred, RfcReceiveEx has to be used for receiving RFC data.
- `RfcReceiveEx` – sends data from the RFC buffer to the RFC server (remember RfcCallEx does not send data to a server over a network) and receives importing and changing parameters from the server. This API blocks until a result is received from the RFC server.

The use of RfcCallEx and RfcReceiveEx without RfcListen is technically possible but rather useless!

Example:

```
        ...
        rfc_rc = RfcCallEx (handle, function_name,
                            exporting, changing, tables);

        do {
        /* While waiting for next incoming RFC call,
             do some useful things. */

        rfc_rc = RfcListen (handle);
        } while (rfc_rc == RFC_RETRY);

        if (RFC_RC != RFC_OK)
      {
        /* An error occurred */
        RfcClose (handle);
        return 1;
      }
```

```
    rfc_rc = RfcReceiveEx (handle, importing,
                          changing, tables,
                          &exception);
    if (rfc_rc != RFC_OK)
    {
      /* An error or an exception occurred  */
      RfcClose (handle);
      return 1;
    }
      ...
```

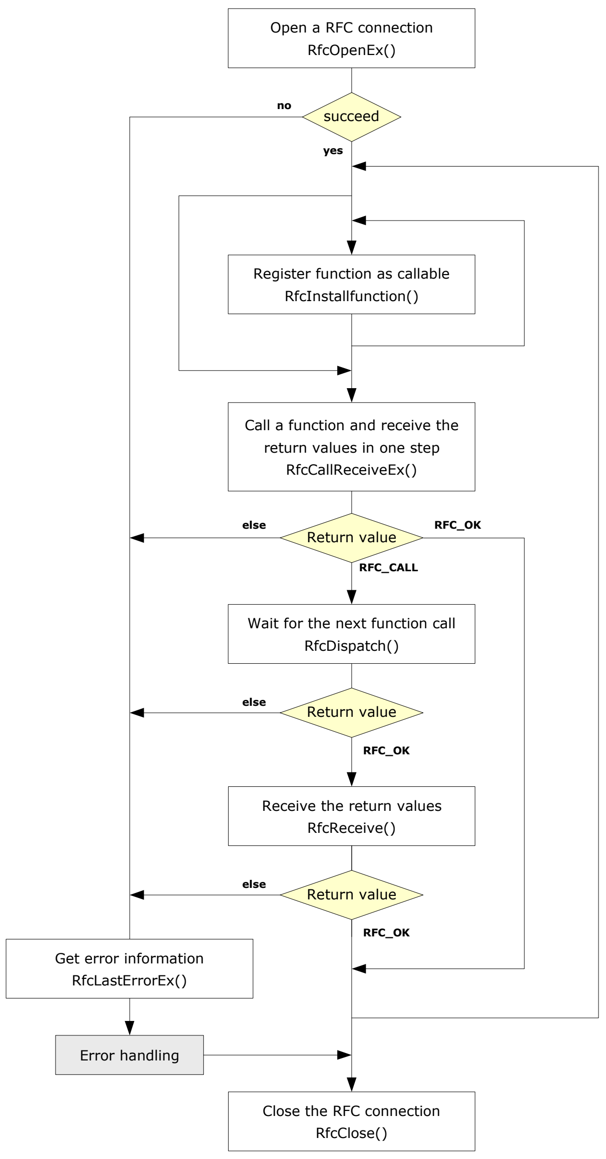
The RFC call-back mechanism for the client

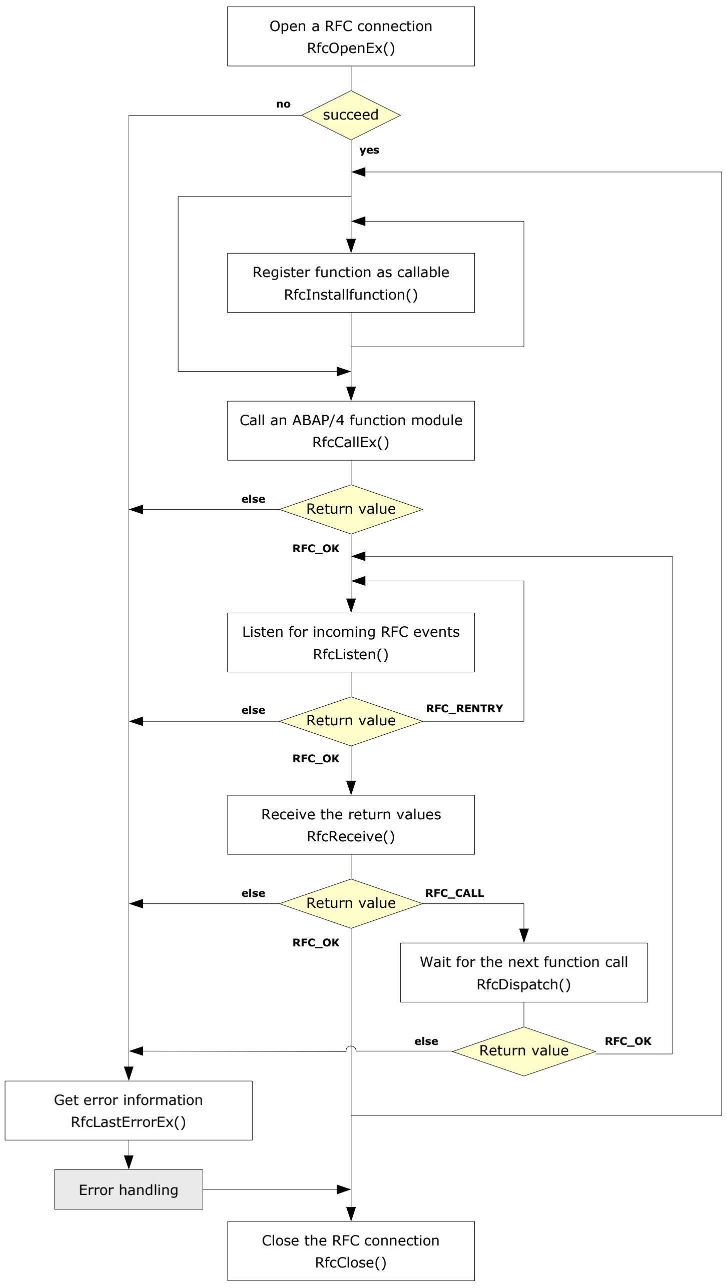
The following example demonstrates the call back handling by the RFC client:

The RFC client calls a remote function ‘ABC’. The ‘ABC’ function makes a call (back) and calls ‘XYZ’ function implemented by the client.

```
int main ()
{
      ...
   /* install call back function  */
   RfcInstallFunction (“XYZ”, xyz_function);

   handle = RfcOpenEx (...);

   rfc_rc = RfcCallReceiveEx (handle, ABC, ...);

   if (rfc_rc == RFC_CALL)
   {
     rfc_rc = RfcDispatch ();

     if (rfc_rc != RFC_OK)
     {
           /* An error occurred */
           RfcClose (handle);
           return 1;
     }

     rfc_rc = RfcReceiveEx (...);
   }

   ...
   RfcClose (handle);
   return 0;
}

static RFC_RC SAP_API xyz_function (RFC_HANDLE    handle)
{
   RFC_RC   rfc_rc;
   rfc_rc = RfcGetData (handle, ...);

   ...
   rfc_rc = RfcSendData (handle, ...);
   return rfc_rc;
}
```

It is also possible to perform a call-back in an unblocked RFC call. The two possibilities are shown in the next diagrams.


Open a RFC connection
RfcOpenEx()
succeed
no
yes
Register function as callable
RfcInstallfunction()
Call a function and receive the return values in one step
RfcCallReceiveEx()
else
Return value
RFC_OK
RFC_CALL
Wait for the next function call
RfcDispatch()
else
Return value
RFC_OK
Receive the return values
RfcReceive()
else
Return value
RFC_OK
Get error information
RfcLastErrorEx()
Error handling
Close the RFC connection
RfcClose()



Detailed flow diagrams for blocked and unblocked RFC calls with call-back mechanism.

RFC with SAPGUI

It is possible for an external RFC client program to call ABAP/4 function modules, which are using '**DYNPROS**' or **SAP Graphics.** An external RFC client is able to call complete SAP transactions. For using this functionality, it is necessary, that SAPGUI has already been started before the function module with DYNPROS is called by the external client. SAPGUI has to be started by an external program.

An external RFC client program can start the SAPGUI with one of the three following options:

- **By using the `RfcOpenEx` API call. When the `RfcOpenEx` API returns the** SAPGUI will already be active. To do this, please insert the following ID value pair: USE_SAPGUI=1 or USE_SAPGUI=2 to connect_string. The content USE_SAPGUI=0 means that SAPGUI should not be started at open time.

- **Call of function module 'SYSTEM_ATTACH_GUI' in R/3.** This feature provides the function to start the SAPGUI at any time. To start the SAPGUI, the external RFC client has to call function module 'SYSTEM_ATTACH_GUI' in R/3 without any parameters before using the SAPGUI functionality.
- **Using sapRFC.ini file**. In this case the SAPGUI will be started at open time too. To do this please define an entry in 'saprfc.ini' file for a corresponding destination as follows: USE_SAPGUI=1 or USE_SAPGUI=2. The entry USE_SAPGUI=0 suppresses the start of SAPGUI at open time.

RFC using ABAP debugger

The full functionality of the ABAP/4 debugger can be used while developing or testing an external RFC client application. For using this functionality, it is necessary to have a SAPGUI installed at the client side.

An external RFC client can use this helpful feature with one of the following options:

- **Using** `RfcOpenEx` **API**. Insert the following ID=value pair: ABAP_DEBUG=1 to connection description. The entry ABAP_DEBUG=0 suppresses the entering into the ABAP debugging mode for its connection (default).
- **Using saprfc.ini file.** It is possible to define an entry in 'saprfc.ini' file for a corresponding destination as follows: ABAP_DEBUG=1. With the entry ABAP_DEBUG=0 the ABAP debugger will not be used (default).
- **Setting environment variable RFC_DEBUG**. If the environment variable RFC_DEBUG is set to value unless 0, then all calls to R/3 systems will enter the debugging mode.

Close a RFC connection

The used connection has to be closed as soon as possible to free internal memory. Normally, a RFC client connection is closed via `RfcClose` API. The RfcClose API should be called after each failed RFC call too. This is necessary in order that all internally used control areas (except internal tables) are released. After the RfcClose order the RFC handle is invalid, i.e., it is impossible to make any API call using this handle.

Please notice:

- RfcClose is only able to destroy the connection .
- RfcClose with RFC_HANDLE_NULL as parameter closes all Client-RFC connections. RfcClose (RFC_HANDLE_NULL) only closes the handles, which were last active in the same thread of the calling program.
- In case of extern-extern communication with a registered RFC server, the client connection has to be closed after every RFC call.

Example using `RfcClose` API:

```
...
RFC_HANDLE      handle;
handle = RfcOpenEx (...);
...
```

```
/* Close a specific RFC connection */
RfcClose (handle);
/ * Close all by this thread established connections
  * Connections created by another thread are not closed
*/
RfcClose(RFC_HANDLE_NULL);
...
```

RFC Server Program

A RFC server program implements functions, which can be called by a RFC client. The following steps have to be accomplished by the RFC server:

- Establish a server connection. In this phase a valid server handle will be created. Furthermore, an RFC server handle will be called *accepted handle* and the RFC server connection will be called *accepted connection*.
- Wait for incoming RFC calls and perform RFC functions, which were called by an RFC client. The same connection can handle several RFC requests.
- Close or abort the RFC connection. The valid RFC handle will be invalidate. All connection data is lost.

RFCSDK conations program srfcserv.c. This program can be used as an example for a RFC server.

Accept a server connection

To establish a server connection, the RFC server should call `RfcAccept` or `RfcAcceptExt` API. The delivered handle is a valid accepted handle. The both API's RfcAccept and RfcAcceptExt differ only in the input parameters format. The functionality of the both API's is the same.

Example

```
int main (int argc, rfc_char_t ** argv)
{
     RFC_HANDLE handle   = RFC_HANDLE_NULL;
   RFC_HANDLE   handle2 = RFC_HANDLE_NULL;

   handle = RfcAccept (argv);
   if (handle == RFC_HANDLE_NULL)
   {
     RFC_ERROR_INFO_EX  error_info;
     RfcLastErrorEx (&error_info);
     ...
   }
   ...

   handle2 = RfcAcceptExt ("-aedi -gbinmain -xsapgw53");
   if (handle2 == RFC_HANDLE_NULL)
   {
     RFC_ERROR_INFO_EX  error_info;
     RfcLastErrorEx (&error_info);
     ...
    }
   ...

    RfcClose (handle);
    RfcClose (handle2);
}
```

Important: If `RfcAccept` fails (returns RFC_HANDLE_NULL), it is necessary to call `RfcLastErrorEx` API. If the RfcLastErrorEx is not called after each failure of RfcAccept, a memory leak of ca. 16000 bytes will be caused for each unsuccessful RfcAccept.

A RFC server program can run in two modes:

- **Registered server.** This server will be started before a client makes the first call and registers itself by an SAP gateway. A registered RFC server could be considered as a demon process (UNIX) or a service (MS-Windows). After registering at the SAP gateway the server is waiting for incoming calls from any RFC client. This behavior enables the server to serve rfc request from different clients simultaneously i.e. registered server is a multi-client server.
- **Started RFC server**. This server will be started during opening the RFC connection by a client and processes incoming RFC calls until the client closes this connection. The server connection stays alive until:
  - the user context on the ABAP-side exists i.e. the ABAP-report made the remote function call ends or an ABAP-transaction which made the remote function call ends (for example with PF3)
  - or external client closes the rfc connection. The started rfc server is able to serve the requests only from rfc client, which opened the connection to this server.

  It is impossible to access to started rfc server from different clients simultaneously i.e. started rfc server is a single-client server.

Registered RFC server

The RFC server program can be registered at the **SAP gateway** with **Program ID** and then wait for incoming RFC requests. This feature has the following properties:

- A RFC server program registers itself at a SAP gateway under a Program-ID and not for a specific SAP system.
- A RFC server program can be registered several times under the same ID at the same Gateway. The incoming RFC request will be propagated to one of the registered program. This allows the writing of powerful multithreaded or multi-processed RFC server programs. Of course the synchronization problem has to be solved by the server program.
- After each executed RFC function the accepted connection will be closed.
- It is possible to bind an rfc client to an explicit registered server. Until this connection kind is active, the server is not available for another rfc clients.

The following connection data is needed to establish an accepted RFC connection:

- Program Id
- SAP gateway name and gateway services.

As already described above, an accepted connection will be established via API RfcAccept. There are two ways to define the input parameter '`argv`' of `RfcAccept` or connection string for `RfcAcceptExt` to register a RFC connection at a SAP gateway:

- Indirect working with the 'saprfc.ini' file and the corresponding destination. In this case the argv argument of RfcAccept contains only a reference to a destination in the saprfc.ini file. All connection data will be read from the saprfc.ini file.

- Direct without the 'saprfc.ini' file. In this case all connection data will be transferred via the argv argument of RfcAccept.

To declare a connection to a registered server as explicitly bound, the client (extern or R/3) should call the RFC-function RFC_SET_REG_SERVER_PROPERTY with export parameter EXCLUSIVE = ‘Y’. The effect of this call is that external registered server is explicitly available for the caller until:

- the RFC-client calls RFC_SET_REG_SERVER_PROPERTY with export parameter:
- either with EXCLUSIVE = ‘N’ i.e. stop exclusivity
- or with EXCLUSIVE = ‘E’ i.e. disconnect
- rfc connection is closed by the client.

After an inactive phase, a registered RFC Server can no longer be accessed. After an RFC Server has registered on an SAP gateway, RFC functions without any problems. Nevertheless, after a lengthy inactive period, this RFC Server may no longer be accessible. For example, the connection test from SM59 terminates with an error. In transaction SMGW, however, this server is still displayed as registered with the status WAITING.

Depending on the TCP/IP implementation and configuration, a TCP/IP connection can be closed after an inactive period without the related TCP/IP application (NI/CPIC/RFC shift) being informed. This phenomenon mostly occurs when a firewall is set between the registered server and the SAP Gateway.

To get around the problem mentioned above, as of an internal communication step is implemented every five minutes between the RFC library and the SAP Gateway, in case the RFC Server does not receive an RFC request within this period the RfcWaitForRequest or RfcDispatch in the RFC Server is being used.

By setting environment variable RFC_MAX_REG_IDLE=<sec>, the default value of 300 seconds can be changed. Please do not set this value too low. If the value of this environment variable is -1, no communication will take place between the registered server and the gateway during RfcDispatch.

Started RFC server

A RFC server can be started by:

- A SAP gateway.
- By the currently used SAPGUI.
- By a R/3 application server.

The starter of the server will generate the connection parameters and transfer them to the program via the *argv* argument of *main*. This data should be transferred without any changes to RfcAccept.

Performing a RFC function

The performing of a RFC function is carried out in two steps:

- **Dispatching** of RFC calls to corresponding function implementations. In this step the function name will be determined and a corresponding function body will be called.

- **Performing** of a RFC call. This step receives importing and changing parameters, processes the function body and eventually sends the exporting and changing parameter to the client. A call back can be performed during the execution of the functions body. In this case the existing accepted connection will be used for the call back.

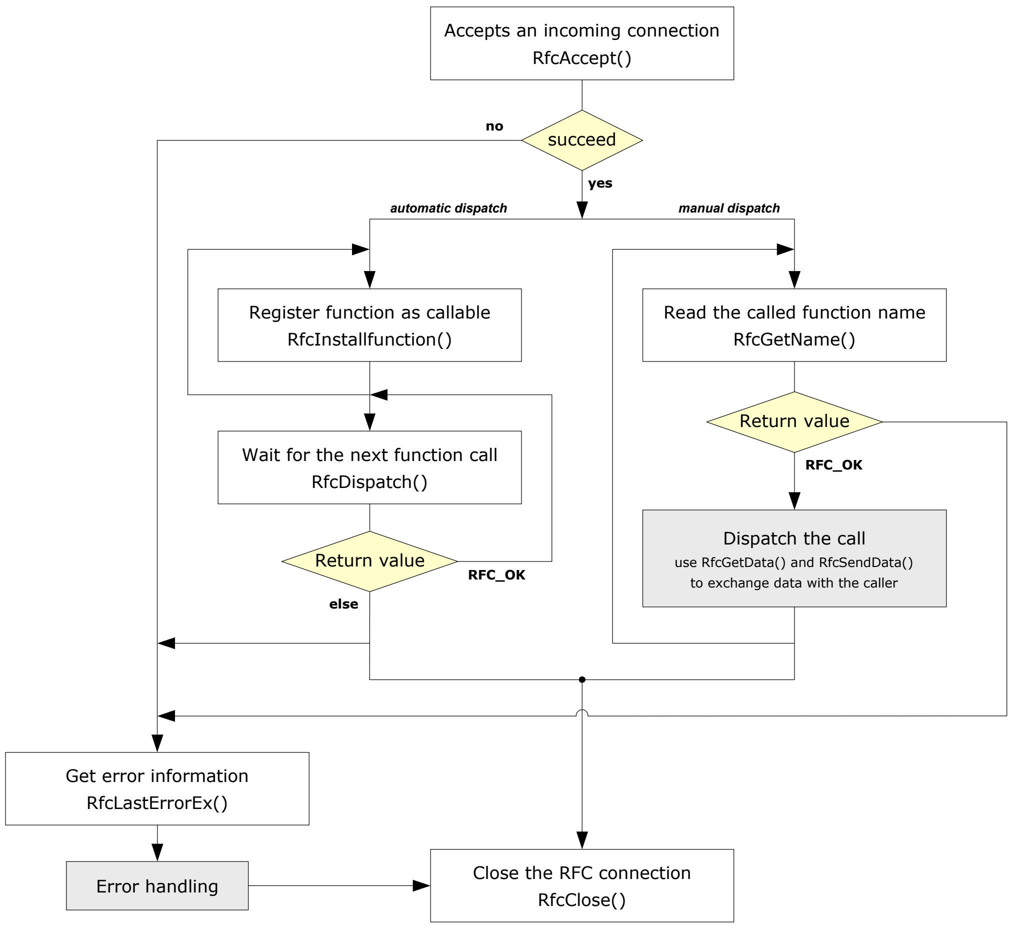
An incoming RFC request can be dispatched:

- Either by the RFC Library, using `RfcDispatch` API;
- Or by the RFC application itself, using `RfcGetName` API.

The dispatching of RFC requests *should be done in a loop*:

- Connection test can be performed only for RFC servers implemented with execution of dispatcher function (RfcDispatch or RfcGetName) in a loop.
- As explained above, a registered RFC connection will be closed automatically after each executed RFC call. If an RFC server works with an RfcDispatch or RfcGetName in a loop, this server will be automatically registered again at the same gateway under the same program ID. This will be done invisible for the API caller.

The basic structure of a server program is shown in next diagram.



Flow diagram for a RFC server program.

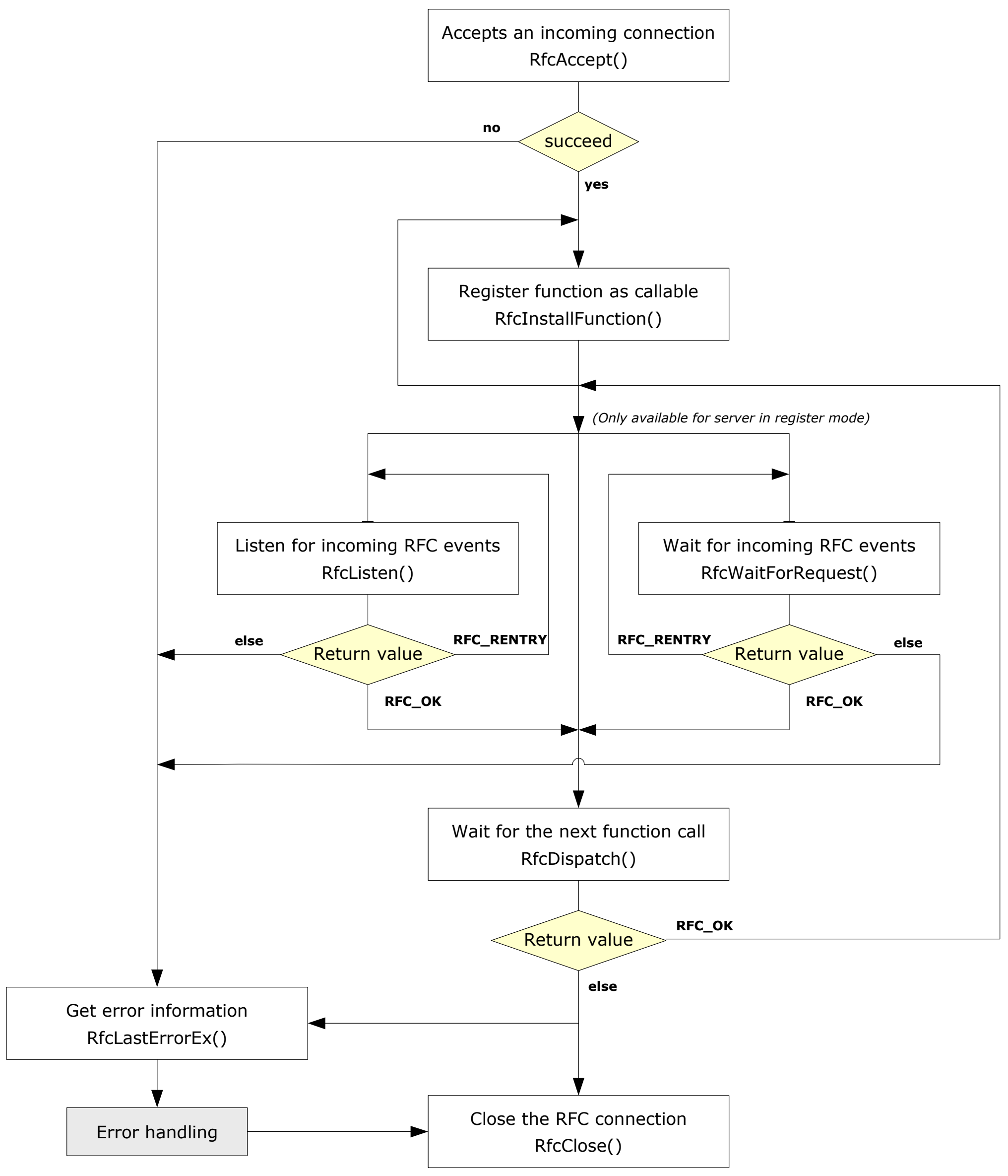
Dispatching of RFC calls by the RFC Library

To enable the dispatching of incoming RFC calls, the RFC server should be informed about all available RFC functions implemented by this RFC server. This should be done via `RfcInstallFunction` API.

After installing of the supported RFC functions, the RFC server can wait for incoming requests and dispatch them to installed functions. This is done using `RfcDispatch` API. As rule this API should be called in a loop in order that the same or even other installed functions can be called by an RFC client application (independent of the fact if it is an R/3 system or an another external application).

Instead of waiting for the next RFC requests blocking in RfcDispatch, the RFC server program can inform itself, whether an RFC requests was incoming via RfcListen API. RfcListen returns immediately and informs the caller whether an RFC request has been received or not. In this case, the RFC server does something useful, until RfcListen tells that there is an RFC request incoming. If RfcListen has detected an RFC requests, this request should be dispatched and performed by RfcDispatch API as usual.

The RFC Library can only dispatch installed functions. A request for a function not installed will be aborted showing an error message. To avoid automatic abort of RFC requests for non-installed functions, it is possible to install a global handler for this error situation. The RFC library will call this function if an RFC request is received and there is no function installed for this. In this specific function, the RFC server can use RfcGetName to get the requested function name and then dispatch for it. This functionality must be used if you are working with transactional RFC and you don't want to work with RfcDispatch but RfcGetName. The handler will be installed via RfcInstallFunction API as a usual RFC function with name "%%USER_GLOBAL_SERVER".



Flow diagram for a RFC server program dispatching explicit installed functions.

Example RFC server working only with RfcDispatch

The following example shows a RFC server program working with RfcDispatch:

```
static abc_function (RFC_HANDLE);

int main (int argc, rfc_char_t ** argv)
{
    RFC_HANDLE handle;
    RFC_RC     rfc_rc;

    /* Establish an accepted connection */
    handle = RfcAccept (argv);
    if (handle == RFC_HANDLE_NULL)
```

```
        {
          /* Connection could not be established */
          RFC_ERROR_INFO_EXerror_info;
          RfcLastErrorEx (&error_info);
          return 1;
        }

        /* Install RFC function ABC and its implementation */
        rfc_rc = RfcInstallFunction (“ABC”, abc_function);
        if (rfc_rc != RFC_OK)
        {
          /* Could not install a server function */
          return 1;
        }

        /* Wait for an RFC call and execute one of installed
    rfc functions
           until the connection is terminated */
        do {
          rfc_rc = RfcDispatch (handle);
        } while (rfc_rc == RFC_OK);

        /* Close connection */
        RfcClose (handle);
        return 0;
    }
```

Example RFC server working with RfcListen and RfcDispatch

The following example shows a RFC server program working with RfcListen and RfcDispatch:

```
static abc_function (RFC_HANDLE);

int main (int argc, rfc_char_t ** argv)
{
     RFC_HANDLE handle;
     RFC_RC          rfc_rc;

     /* establish an accepted connection */
     handle = RfcAccept (argv);
     if (handle == RFC_HANDLE_NULL)
     {
     /* Connection could not be established */
     RFC_ERROR_INFO_EX   error_info;
     RfcLastErrorEx (&error_info);
     return 1;

     }
```

```
  /* Install RFC function ABC and its implementation */
  rfc_rc = RfcInstallFunction (“ABC”, abc_function);

  if (rfc_rc != RFC_OK)
  {
    /* Could not install a server function */
    RfcClose (handle);
    return 1;
  }

  /* Wait for an RFC call. While waiting, do something
useful.
     If an RFC call is incoming, execute one of installed
RFC functions.
     Perform above steps, until the connection is
terminated. */
  do {
    do {
      /* While waiting for next incoming RFC call,
         do some useful things. */
      rfc_rc = RfcListen (handle);
    } while (rfc_rc == RFC_RETRY);

    rfc_rc = RfcDispatch (handle);
  } while (rfc_rc == RFC_OK);

  /* Close connection */
  RfcClose (handle);
  return 0;
}
```

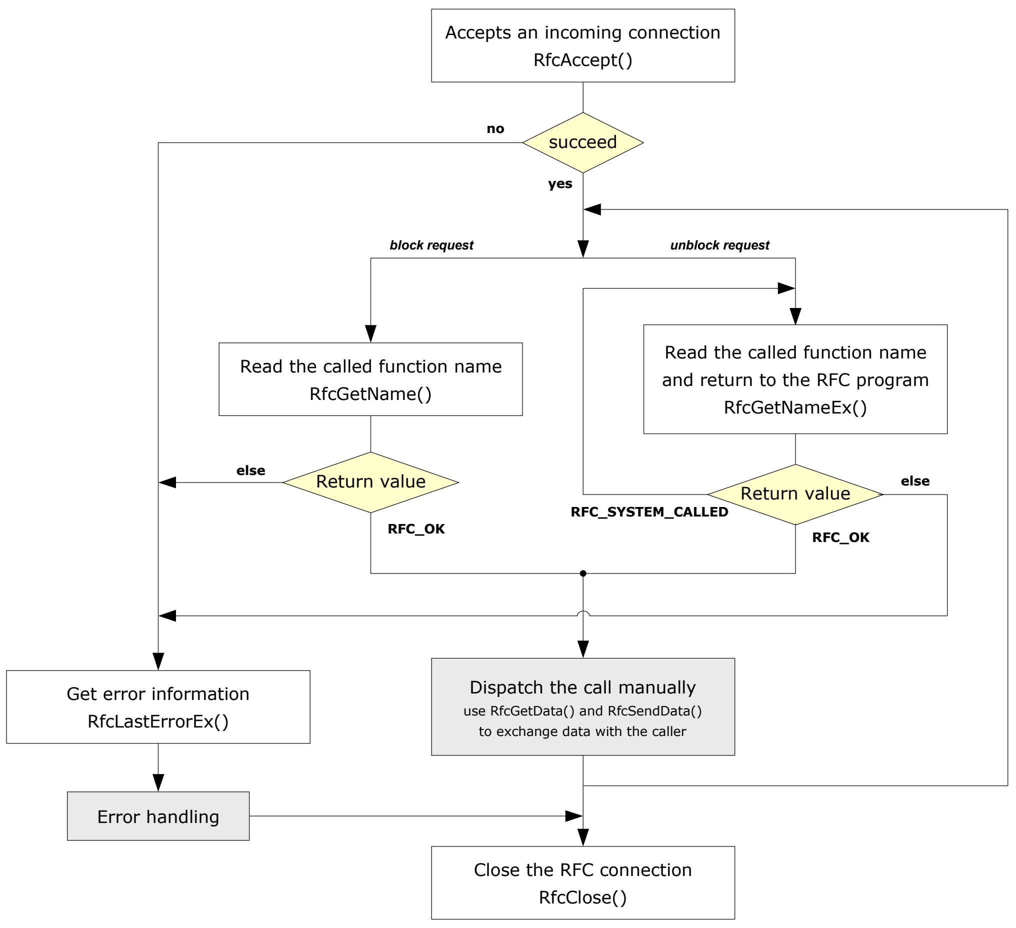
Dispatching of RFC calls by the RFC application

Alternatively a RFC server program can use `RfcGetName` API to identify the name of the called RFC function and then dispatch the call. The following example demonstrates the use of RfcGetName API:

```
  ...
  handle = RfcAccept (argv);
  ...

  /* Get function name directly and start it */
  do {
    rfc_rc = RfcGetName (handle, function_name);

    if (rfc_rc == RFC_OK)
    {
      if (strcmp (“ABC”, function_name))
```

```
        {
            rfc_rc = abc_function (handle);
        }
        else
        {
            /* RfcAbort closes the connection */
            RfcAbort (handle, "RFC function %s is not\
                          implemented in this server program");
        }
      }
      else /* error occurred */
      {
        return 1;
      }
   }   while   (rfc_rc    ==    RFC_OK    ||    rfc_rc    ==
RFC_SYSTEM_CALLED);

   RfcClose (handle);
   ...
```

It follows a detailed flow diagram showing the possibilities by manual dispatching.



Flow diagram for a RFC sever program dispatching the RFC call manually.

Performing a RFC function, data exchange and RFC exceptions

The performing of a RFC server function can be divided into the following parts:

- Receiving of importing, changing parameters and tables
- Logical part.
- A call back to the client using the accepted connection can be done if necessary.
- Sending of exporting, changing parameters and tables to the caller.
- Instead of sending data to a client an application exception can be raised by an RFC server on behalf of the RFC user.

The receiving of data should be done using RfcGetData API. RfcSendData API should be used for sending the data.

Please notice:

- In RFC functions, offered by a RFC server program, all importing, exporting parameters and also all **internal tables** must be defined, before RfcGetData is issued. Nevertheless, it is not necessary to use `ItCreate` API for internal tables. The RFC library will do this automatically. After an RFC function is ended the RFC library will free the storage for all used internal tables in this function.
- RFC server with **changing** parameter should use the same name in RfcGetData and RfcSendData for corresponding importing and exporting parameter values.

Instead of RfcSendData an external RFC server can call `RfcRaiseTables` API to raise an RFC exception. In this case the RFC client should catch this exception as required. This API throws an exception and does not close the existing RFC connection.

Important: An application exception is **not an error**. Your RFC-Function **should not** return RFC_FAILURE after RfcRaiseTables but return code delivered by RfcRaiseTables (in most cases RFC_OK).

RFC call back mechanism for RFC server

The following example demonstrates the call back handling by the RFC server:

The RFC server provides the remote function ‘ABC’. The ‘ABC’ function makes a call back and calls ‘XYZ’ function implemented by the client.

```
int main (int argc, rfc_char_t ** argv)
{
      RFC_HANDLE handle   = RFC_HANDLE_NULL;
      RFC_RC     rfc_rc   = RFC_OK;
      handle = RfcAccept (argv);

      if (handle == RFC_HANDLE_NULL)
      {
      RFC_ERROR_INFO_EX  error_info;
      RfcLastErrorEx (error_info);
      return 1;
```

```
        }

        rfc_rc = RfcInstallFunction ("ABC", abc_function);
        if (rfc_rc != RFC_OK)
        {
          RfcClose (handle);
          return 1;
        }

        do {
          rfc_rc = RfcDispatch (handle);
        }(rfc_rc == RFC_OK)

           RfcClose (handle);
           return 0;
           }

           static RFC_RC abc_function (RFC_HANDLE handle)
           {
            RFC_RC   rfc_rc = RFC_OK;

            rfc_rc = RfcGetData (handle, ...);

            /* Call back 'XYZ' */
            rfc_rc = RfcCallReceiveEx (handle, ...);

            if (noExceptionRequired)
              {
                rfc_rc = RfcSendData (handle, ...);
            }
            else
            {
                rfc_rc = RfcRaiseTables (handle, ...);
          }

            return rfc_rc;
           }
```

Closing and aborting of accepted connections

The accepted connection can be regularly closed with `RfcClose` API. If RfcClose is called with RFC_HANDLE_NULL as parameter, all existing client connections will close. The RfcClose API has to be called after each failed RFC call, in order to release all internally used control areas (except internal tables). The use of RfcClose is as described in the client section of this paper.

Additionally a RFC server can use API `RfcAbort` to terminate an accepted connection at any time. RfcAbort is done by RFC runtime only if the thread that created the connection calls RfcAbort. If RfcAbort is called from another thread, the call is ignored.

An error text can be passed to the RFC partner via this API. The error text is then available for the partner as a ABAP Error Message. RfcAbort closes the existing connection.

Example using RfcAbort API:

```
static RFC_RC abc_function (RFC_HANDLE handle)
{
     RFC_RC rfc_rc;
     rfc_rc = RfcGetData (handle,...);

     if (rfc_rc != RFC_OK)
     {
       return RFC_RC;
     }

     ...
     if (<some condition>)
     {
       RfcAbort (handle, "Could not proceed message");
       return RFC_FAILURE;
     }

     ...
     rfc_rc = RfcSendData (handle, ...);

     return rfc_rc;
}
```

RFC Data Model

The RFC library supports the following data types:

- **Scalar data types**. Elementary data types like characters, arrays of characters, date, time, integer, float etc. On top of that, the RFC Library supports flat structures, based on these elementary data types. This kind of data type is called a scalar data type.
- **Tables of scalar data.** The RFC library supports tables of scalar elements too.

There is no support for complex structures and tables of them.

To ship the data to a RFC partner the data description is needed. The metadata will be defined via an instance of RFC_PARAMETER structures for scalar data and RFC_TABLE structure for tables. Following information is needed for every scalar parameter:

- Parameter name and the length of the name in characters.
- Type of parameter (native or installed).
- Reference (pointer) to the corresponding memory area.
- The maximal memory size reserved for the corresponding memory area.

The metadata will never be transferred to any RFC partner. The RFC data will be sent as described by the sender. The received data will be handled by the receiver as described. If the data description is different for server and client, the data will handle different (according the given description) at every side.

Elementary data types

Arithmetical data types

These are the least complex data types. The RFC library automatically converts the arithmetical data from the internal sender format into the internal receiver format. For example, during a RFC conversation between an NT-System and a RS6000 system, a Big-Endian integer will be converted into Little-Endian format and vice versa.
The arithmetical data types are:

- RFCTYPE_INT
- RFCTYPE_FLOAT
- RFCTYPE_BCD.

Character-like data types

This data is either a character or an array of characters. The RFC Library supports the following character-like data types:

- RFCTYPE_CHAR
- Array of characters
- RFCTYPE_DATE
- RFCTYPE_NUM
- RFCTYPE_TIME

The RFC Library automatically converts every received character from the sender codepage into the receiver codepage. For every RFC connection, independent or accepted client, the RFC Library determines the codepage. If no codepage has been defined by the RFC application, the default codepage will be used for every RFC connection. The default code pages are as follows:

- For the 1 byte character size the default codepage is 1100 on ASCII systems (windows, Unix) and 0110 on the EBCDIC platforms (AS400).
- For the 2 byte character size (Unicode) the default is either 4103 or 4102.

The default codepage can be (re-) set by:

- Either by the environment variable SAP_CODEPAGE,
- Or using RFC API `RfcSetSystemCodepage.`

It is possible to set codepage for a client connection. The codepage for a client connection could be set

- Either at open time using the entry CODEPAGE=<your codepage> in the connection string for the RfcOpenEx API. Example:

```
RfcOpenEx ("… CODEPAGE=8000 …",…) ;
```

- Or at run time for a valid RFC handle using RFC API `RfcSetCodepage.` Example:

```
RfcSetCodepage (handle, "8000");
```

The path to the conversion table should be defined by the environment variable PATH_TO_CODEPAGE.

Strings

Almost all types of data have a constant length, i.e., it is possible to reserve enough memory to store the received data. Strings are data which length is variable, i.e., it is impossible to determine at receiver side, which length the received string will have. The RFC Library automatically allocates enough memory for the received string. The user of the RFC Library should allocate only the memory for sending strings.

Please note: The RFC Library allocates the memory for received strings but does not free them. To avoid memory leaks, the user should free the memory allocated for strings.

The RFC Library supports two types of strings:

- Character String. These are zero terminated Utf8 encoded character strings.
- XString. These are raw data strings.

Character Strings

This data type has a variable length and is similar to the ABAP/4 data type string. The difference is that the RFC_STRING is a zero terminated, but the ABAP/4 string is blank padded. The conversion between zero terminated Utf8 (extern RFC program) and blank padded (ABAP) is done automatically by the RFC engine. All strings are to be handled by the respective sending and receiving external programs as UTF8 zero terminated strings.

Please note: The RFC Library does not convert RFC_STRINGS into Utf8 format, but expects, that strings already have Utf8 format.

The following example demonstrates the string handling:

```
     RFC_STRING          question,
                         answer;
     RFC_RC              rfc_rc;
     RFC_PARAMETER       exporting[2],
                         importing[2];
     RFC_TABLE           tables[1];
     rfc_char_t          exception[512];
     rfc_char_t        * except_ptr = exception;

     /* Allocate only export parameter
      *  The  rfc  library  will  do  Allocation  of  import
parameter */
     question = RfcAllocString (11);
    /* allocates 12 bytes and fills them with Null's */

    strncpy (question, "How are you", 11); // we are UTF8

     exporting[0].name = "QUESTION";
     exporting[0].nlen = strlen (exporting[0].name);
     exporting[0].type = RFCTYPE_STRING;
     exporting[0].addr = &question;
     exporting[0].leng = strlen  (question);
     /*  does  not  matter  the  value,  because  RFC  lib
restores this
      * value from  .addr field using strlen () */
     exporting[1].name = NULL;

     importing[0].name = "MYANSWER";
     importing[0].nlen = strlen (importing[0].name);
     importing[0].type = RFCTYPE_STRING;
     importing[0].addr = &answer;
     importing[0].leng = 0;
     /* you do not know how long is the received string
   *  The  value  of  the  field  will  be  set  by  the  RFC
library */

     importing[1].name = NULL;
     tables[0].name = NULL

     rfc_rc = RfcCallReceiveEx (handle, "STFC_STRING",
                              exporting,        importing,
NULL,
          tables,  &except_ptr);

     ...

     //  Free  both  import  and  export  strings  to  avoid
memory leaks
```

```
     RfcFreeString (question);
     RfcFreeString (answer);
```

Raw strings or XStrings

The following Example demonstrates the handling of XStrings in a RFC program:

```
  RFC_XSTRING        question,
                  answer;
  RFC_RC             rfc_rc;
  RFC_PARAMETER      exporting[2],
 importing[2];
  RFC_TABLE          tables[1];
  rfc_char_t         exception[512];
  rfc_char_t       * except_ptr = exception;


      // initialise xstrings
     question.content = NULL;
     question.length = 0;

     answer.content = NULL,
     answer.length = 0;

      /* Allocate only export parameter
       * The rfc library will do Allocation of import
parameter */
      RfcResizeXString (&question, 2000);

     exporting[0].name = "QUESTION";
      exporting[0].nlen = strlen (exporting[0].name);
      exporting[0].type = RFCTYPE_XSTRING;
      exporting[0].addr = &question;
      exporting[0].leng = question.length;
      /* does not matter the value, because rfc lib
restores
    * this value from RFC_XSTRING.length field of the
data type. */

      exporting[1].name = NULL;

      importing[0].name = "MYANSWER";
     importing[0].nlen = strlen (importing[0].name);
     importing[0].type = RFCTYPE_XSTRING;
     importing[0].addr = &answer;
     importing[0].leng = 0;
     /* does not matter the value, because rfc lib
restores
```

```
      * this value from RFC_XSTRING.length field of the
data type. */

     importing[1].name = NULL;
     tables[0].name = NULL

     rfc_rc = RfcCallReceiveEx (handle, "STFC_XSTRING",
                               exporting, importing,
NULL,
                              tables, &except_ptr);
     ...
     /* Free both import and export xstrings to avoid
memory leaks */
     RfcResizeXString (&question, 0);
     RfcResizeXString (&answer, 0);
```

Raw data

Raw data is never converted. Supported are bytes and array of bytes.

```
     RFC_PARAMETER   param[3];
     RFC_BYTE        byte,
                 array_of_bytes[10];

     param[0].name = ”...”;
     param[0].nlen = strlen(param[0].name);
     param[0].type = RFCTYPE_BYTE;
     param[0].addr = &byte;
     param[0].leng = sizeof(byte);

     param[1].name = ”...”;
     param[1].nlen = strlen(param[1].name);
     param[1].type = RFCTYPE_BYTE;
     param[1].addr = &array_of_bytes;
     param[1].leng = sizeof(array_of_bytes);

     param[2].name = NULL;
```

Structured data

A structured data type has to be installed via the `RfcInstallStructure` API. This API returns a new data type handle. The use of structured data types handle is the same as for the elementary data types. They are used in the fields type from the structures `RFC_PARAMETER` and `RFC_TABLE`. `RfcCallReceiveEx`, `RfcCallEx`, `RfcReceiveEx`, `RfcGetData`, and `RfcSendData` uses this type handles to marshal/unmarshal the data. The layout of the data, (the data pointed by `RFC_PARAMETER.addr`, or passed by the `It*()` functions is the *exact* memory layout of the corresponding structure in the ABAP VM. The offsets are the same on all platforms, the endianess is always the native endianess of the local machine.

There are two scenarios:

- **Early** bound scenarios that use generated structure descriptions at compile time,
- **Late** bound scenarios that dynamically retrieve the structure layout from the attached Web Application Server.

Early bound scenario

Please us the tool `genh` that can be found in the …./rfcsdk/bin or …\rfcsdk\bin\ directory. This tools generate on standard out an header that contains a) a `typedef` for a 'C' description from the structure, and b) an array of `RFC_TYPE_ELEMENT` for use with `RfcInstallStructure`.

Sample call:

```
genh   ashost=appserv   sysnr=00   client=001   user=myuser   passwd=mypass
SOME_STRUC

genh   ashost=appserv   sysnr=00   client=001   user=myuser   passwd=mypass
SOME_STRUC > some_struc.h
```

Late bound scenario

The raw list of fields that contributes to a structure is not sufficient to compute the memory layout that the Web Application Server uses. If some fields have been included via `.INCLUDE` statement in the Data Dictionary, or if one of the field is itself a structure then those substructure may introduce an unexpected layout that cannot be computed out of the raw list of fields an there type. For this reason, the `RFC_TYPE_ELEMENT` array generated by the tool `genh` sometime contains dummy fields of type `RFCTYPE_PADDING` that are used to enforce the correct layout. Since most late bound programs uses the function module `RFC_GET_STRUCTURE_DEFINITION` and/or `DDIF_FIELDINFO_GET` this programs would have to compute this dummy fields by themselves. The easier solution is to use RfcInstallStructure2 that conveniently takes an `RFC_TYPE_ELEMENT2` array shaped like the return table of `RFC_GET_STRUCTURE_DEFINITION`.

UNICODE Programs

When an ASCII program communicates with an UNICODE partner, the UNICODE partner is responsible for both; the UNICODE <-> ASCII/MBCS conversion and the transformation between UNICODE systems and ASCII/MBCS memory layout. So that ASCII programs are not affected by the fact that the peer is an UNICODE system. They can continue to use `RfcInstallStructure` or `RfcInstallStructure2`.

The user of RfcInstallStructure2 however must be careful that the function module `RFC_GET_STRUCTURE_DEFINITION` of a UNICODE Web Application Server returns the UNICODE memory layout. The recommended solutions are: either call the function module `DDIF_FIELDINFO_GET` passing the value "`01`" to the parameter `UCLEN` an then call `RfcInstallStructure2`. or call the function module `RFC_GET_UNICODE_STRUCTURE` and call `RfcInstallUnicodeStructure()`.

When a UNICODE program communicates with a UNICODE partner, it must use the UNICODE memory layout for marshalling/unmarshalling. But it must use the ASCII/MBCS memory layout when communicating with an ASCII/MBCS peer. Thus, UNICODE programs must inform the RFC-Library about both memory layouts. The UNICODE program uses `RfcInstallUnicodeStructure` for this. `RfcInstallStructure` and `RfcInstallStructure2` cannot be used in UNICODE programs. Late bounds program may use one call to the function module `RFC_GET_UNICODE_STRUCTURE` or two calls to `DDIF_FIELDINFO_GET` to retrieves the information needed to call `RfcInstallUnicodeStructure`. Early bound programs should use `genh-Tool` to generate an array of `RFC_UNICODE_TYPE_ELEMENT`.

Tables

RFCSDK contains functions, which allow the processing of ABAP/4 internal tables in a C environment. ABAP/4 internal tables follow the model of a relational database table.

ABAP/4 internal tables consist of type identical rows. If created, a table is yet empty. In ABAP/4 you can fill rows into a table by the statements 'Insert' or 'Append'. You can access a row by 'Read Table' and you can delete a row by 'Delete'. You can free a Table by 'Free Table' and you can retrieve information about tables by 'Describe'. These language constructs correspond to the following C routines:

- `ItCreate` – creates a new internal table.
- `ItAppLine` – appends a line (row) to an internal table.
- `ItInsLine` – inserts a line into the given position.
- `ItDelLine` – deletes a line.
- `ItGetLine` – reads a line.
- `ItGupLine` – reads a line for update.
- `ItFree` – resets an internal table to initial state.
- `ItDelete` – deletes (frees) a complete table.
- `ItFill` – returns the number of lines in a table.
- `ItLeng` – returns the width of a table, i.e., the size of a row of the table.

The following example demonstrates table handling in a RFC Client:

```
static void display_table(ITAB_H itab_h);
static void fill_table(ITAB_H itab_h, int table_leng);

int main (int argc, rfc_char_t ** argv)
{
    RFC_TABLE      tables[3];   /* Working with one
internal table */

    RfcEnvironment(...);        /* Install error handling
function   */
```

```
     /* Open RFC connection      */
     rfc_handle = RfcOpenEx(...);

     if (rfc_handle == RFC_HANDLE_NULL)       /* Check
return code*/
     {
     ...
     return 1;
     }

     tables[0].name = "ITAB1000";    /* Must fit with
definition
                                              in SAP-FM    */
     tables[0].nlen = 8;         /* Length of name
     */
     tables[0].type = TYPEC;     /* Only character
     */
     tables[0].leng = 1000;      /* Length of a table line
     */
     tables[0].itmode   = RFC_ITMODE_BYREFERENCE; /*
Recommended*/
     /* Allocate storage for internal table   */
     tables[0].ithandle = ItCreate(…);


     tables[1].name = "ETAB1000";    /* Must fit with
definition
                                              in SAP-FM    */
     tables[1].nlen = 8;         /* Length of name
     */
     tables[1].type = TYPEC;     /* Only character
     */
     tables[1].leng = 1000;      /* Length of a table line
     */
     tables[1].itmode   = RFC_ITMODE_BYREFERENCE; /*
Recommended*/
     /* Allocate storage for internal table   */
     tables[0].ithandle = ItCreate(…);

     tables[2].name = NULL;

     if (tables[0].ithandle = ITAB_NULL)/* Check success
     */
     {
     ...
     return 1;
     }
     /* Fill internal table with 10 lines text     */
     fill_table (table[0].ithandle, 10);

      /* Call up function module with the filled table */
      rfc_rc = RfcCallReceiveEx(handle, ...);

      if (rfc_rc != RFC_OK)  /* Check return code*/
      {
      ...
      return 1;
     }

     RfcClose (...);     /* Close RFC connection */
     display_table (table[1].ithandle);
     ItDelete (…);  /* Free storage of intern tables */

     return 0;
}


/*  Fill  internal  table  with  text  "This  is  a  test"  as
requested */
static void fill_table(ITAB_H itab_h, int table_leng)
{
     int            linenr;      /*  Actual  line  number  in
table  */
     int            lineleng;    /* Length of a table line
       */
     rfc_char_t  * ptr;      /* Pointer to a table line
     */
     rfc_char_t table_data[] = "This is a test";

   if (table_leng == 0)
     {
    return; /* Table with no entry    */
     }

     /* Determine length of a table line  */
     lineleng = ItLeng(itab_h);

     /* Fill table as requested           */
     for (linenr = 1; linenr <= table_leng; linenr++)
     {
     /* Get address of next line      */
       ptr = (char *) ItAppLine(itab_h);

     if (ptr == NULL) /* Check return code    */
       {
       /* Output error message and die hard */
```

```
           printf(“\nMemory insufficient\n”);
         exit(1);
         }
         /* Transfer data to internal table     */
       memcpy (ptr, table_data, lineleng);
     }

     return;
   }

   /* Output received internal table on screen */
   static void display_table(ITAB_H itab_h)
   {
        int            linenr;  /* Actual line number in
   table */
        int            lineleng;    /* Length of a table line
        */
        rfc_char_t  * ptr;  /* Pointer to a table line     */
        rfc_char_t table_data[8193];/* Max. length of a int.
   table */

        /* Get length of a table line*/
        lineleng = ItLeng(itab_h);

        /* Loop at internal table   */
        for (linenr = 1; ; linenr++)
        {
        /* Get address of next line */
          ptr = (char *) ItGetLine(itab_h);
        if (ptr == NULL)
        {
          /* End of table reached */
          break;
        }

        /* Read a table line into buffer     */
        memcpy(table_data, ptr, lineleng);

        /* Set string end in buffer for output        */
          table_data[lineleng] = ‘\0’;

        /*  Ouput on screen          */
        printf(“’%s’\n”, table_data);
        }

        return;
   }
```

Transactional RFC

A RFC client program (ABAP or external program) will normally answer the call of a RFC function if a network error (CPI-C error) is returned by this call. If the network error occurred through calling a RFC function, it is necessary to repeat this action. But if this network error occurred during, or at the end of the execution of an RFC function, the RFC function may already be successfully executed.

In both cases the RFC-component in SAP systems, or the RFC library will have the same CPI-C error code from the CPI-C layer.

To run this RFC call again is not always correct because it will execute the required RFC function once more!

By using a R/3 system, the RFC data can be transferred between two R/3 systems reliably and safely via transactional Remote Function Call (It was renamed from asynchronous RFC to transactional RFC because asynchronous RFC has a different meaning in R/3 systems).

This ensures that the called function module(s) will be executed exactly once in the RFC server system. Moreover, the RFC server system or the RFC server program needn't be available at the time when the RFC client program is doing tRFC.

Transactional RFC between R/3 Systems

A transaction, also known as Logical Unit of Work (LUW), begins with the first 'Call Function … In Background Task' and ends with 'Commit Work' in an ABAP program. A transaction can include one or more RFC calls as follows (ABAP coding):

```
CALL FUNCTION 'F1'      …..  IN BACKGROUND TASK.
…..
CALL FUNCTION 'Fn'      …..  IN BACKGROUND TASK.
…..
COMMIT WORK.
```

With each 'Call Function … In Background Task' the tRFC-component will store the being called RFC function together with the according data in the SAP database.

Only after 'Commit Work' the tRFC-component tries to pass on these data together with a Transaction Identifier (TID), which is unique worldwide (also on different R/3 systems) to the R/3 server system.

The tRFC-components in both systems communicate with each other in two phases:

- **Function shipping**. All RFC functions together with the according data and the TID will be transferred to the RFC server system. If CPI-C error occurred during this phase the transfer will be repeated by the tRFC-component in the client system (The number of tries and the time between two tries can be defined with sm59). The R/3 server system uses the TID to check whether the transaction with this TID has been transferred and executed. The tRFC-component in server system informs the client system about the successful execution of this transaction.
- **Confirmation.** The tRFC-component in the client system sends a confirmation to the client system and both systems can delete the entry for this TID in their own TID-management. The confirmation will not be repeated even if a CPIC error occurs in this phase. Therefore, the TID-management can grow up if working within a bad network.

The transaction is completely ended after this second phase.

Transactional RFC between R/3 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 the event of an error such as a network error.

Therefore, the transactional RFC interfaces from external systems to an R/3 is currently implemented as follows:

- **RFC library.** The RFC library provides some special RFC calls such as RfcCreateTransID, RfcIndirectCall, RfcIndirectCallEx, RfcConfirmTransID and RfcInstallTransactionControl for working with tRFC and will convert data between the RFC and the tRFC format. For a R/3 system, there is no difference, whether these calls are requested from another R/3 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 such as it is called via 'normal' RFC (with RfcGetData and RfcSendData). RFC client programs and RFC server programs Both programs have to manage the TID's themselves in order to check and execute the requested RFC functions exactly once, just as the tRFC-component in a R/3 system does.
- **R/3 systems.** In R/3 systems, there are no additional changes necessary to ABAP programs working with external RFC programs using the tRFC-Interface. For ABAP programs as the RFC client program, the destination defined in CALL FUNCTION …. must have 'T' as connection type.

Transactional RFC Client Program

In contrast to tRFC between R/3 systems, a transaction from an RFC client program contains only one RFC function.

There are two different ways to make a tRFC call.

Obsolete tRFC API

After connecting to a R/3 system (via RfcOpenEx), a RFC client program must use the two following RFC calls in order to be able to work with the tRFC-Interface:

- RfcCreateTransID. With this call, the RFC-Library tries to get a TID created by the R/3 system. In the case of an error, 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 RFC call is unsuccessful, it can be repeated later.
- RfcIndirectCall. With this call, the RFC library will pack all RFC data belonging to a RFC function together with the TID and sends them to the R/3 system using the tRFC protocol. If an error occurs, the RFC client program has to reconnect later and must try to repeat the call. In this case, it has to use the old TID and must not get a new TID with RfcCreateTransID. Otherwise, it is not guaranteed that this RFC function will be executed exactly once in R/3 system. After a successful execution of this call, the transaction is completely terminated. The RFC client program can then update its own TID-management (e.g. delete the TID-entry).

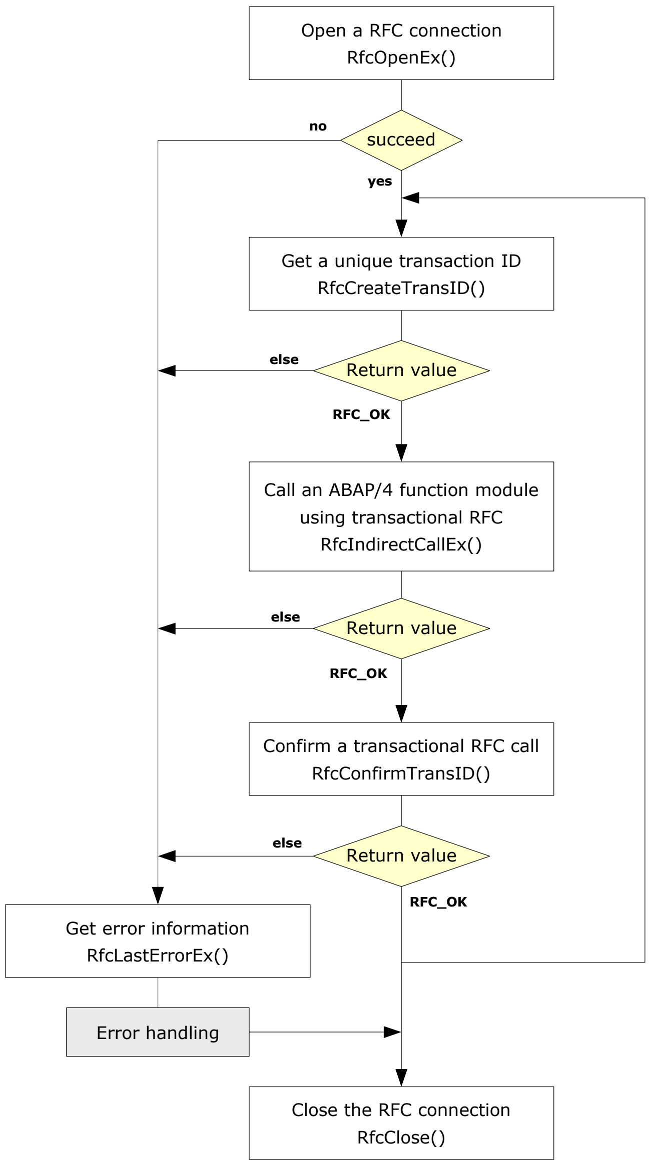
Current tRFC API

RfcIndirectCallEx and RfcConfirmTransID represents the new interface for tRFC client.

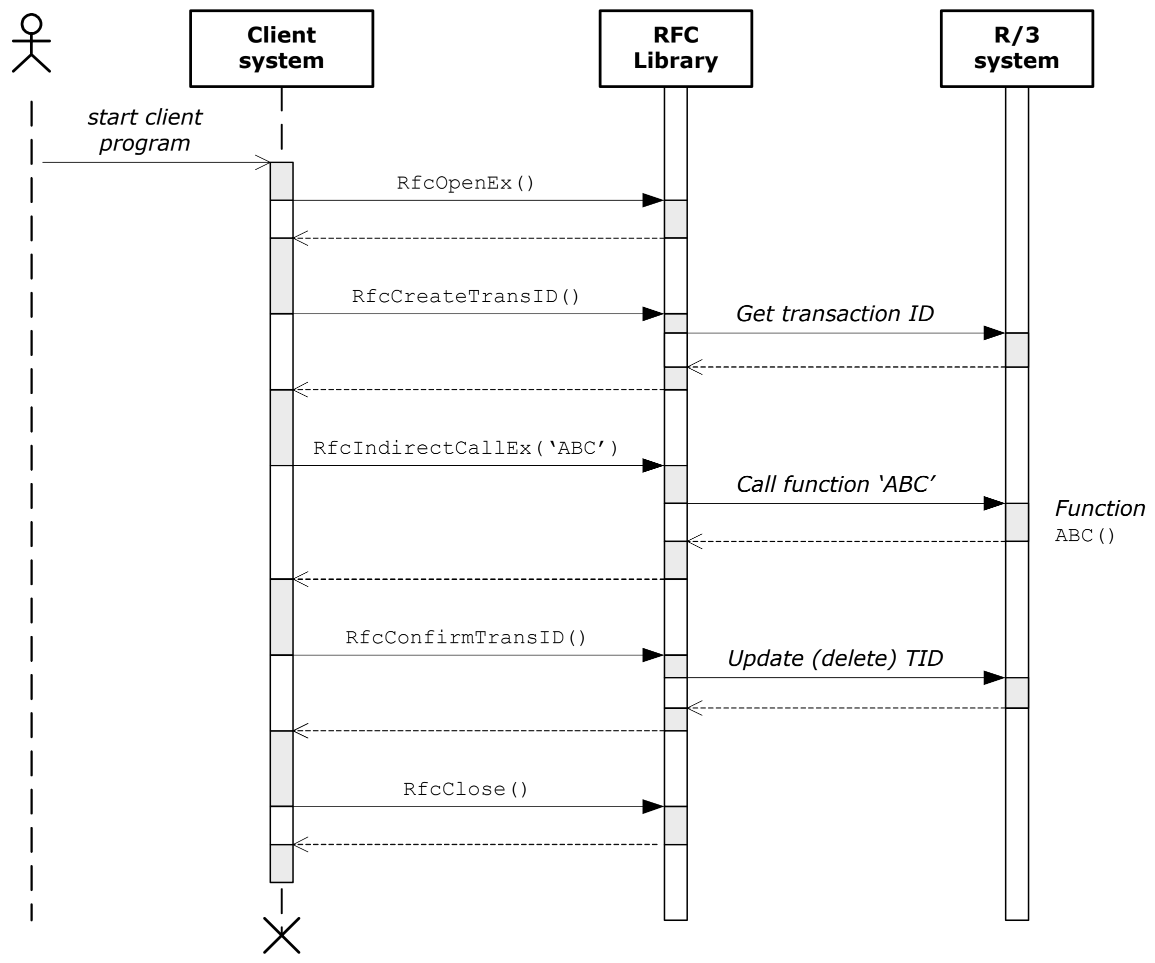
With RfcIndirectCall, the RFC client can have a problem with the 'exactly once'-functionality because if the RFC client has broken down before it can update its own TID management, it will try to call the according RFC function once more. To avoid this error, the RFC library from 4.0C onwards supports two new calls: RfcIndirectCallEx and RfcConfirmTransID.

The RFC client must update the TID as 'executed' before it issues RfcConfirmTransID. Because the tRFC server will delete the TID in its TID management after the RfcConfirmTransID is executed, the RFC client can recall the RFC function with RfcIndirectCallEx at any time before RfcConfirmTransID. The tRFC server will or will not call the RFC function depending on the state of this TID.

The basic structure of a tRFC client program is shown in the following diagrams. See the delivered program trfctest.c for more details.



Flow diagram for a tRFC client program.



Sequence diagram for a tRFC connection between a non R/3 client system and a R/3 server.

The Sample Test Program 'trfctest.c'

The C-program trfctest.c, delivered with the RFC SDK (executable and source code) is a tRFC client program as an example. To get connected to a R/3 system, a 'saprfc.ini' file is needed. Data to transfer to R/3 via tRFC must be in a file. The file name must be defined by starting this program. Each line in this file is one line in an internal table. Only one internal table with 72 B line length is used. Received data in R/3 will be written in the TCPIC-table in an R/3 system (only the first 40 bytes) and the function module STFC_WRITE_TO_TCPIC will be activated. It uses the file I/O on the running platform for management the TID's. For each TID there is an entry in the TID-management which contains the date and time, the TID itself, the state of this transaction (CREATED, CONFIRMED, …), and the name of the data file. It is possible to interrupt the execution of the program (for example with ^C) to simulate error cases. Anytime, when this program is started it will look at the TID-management for aborted transactions. If any exists, it will first try to re-run these transactions. Since this program can run on different platforms an according flag (SAPonUNIX, SAPonNT, …) must be set if you want to compile and link this program to your environment.

See source code for more details.

Transactional RFC Server Program

Implementation rules

After being connected to a R/3 system (via RfcAccept) and having installed the supported RFC functions, the RFC server program has to use the RFC call RfcInstallTransactionControl to work with the TID´s to check and execute the real RFC functions it supports before entering in the loop with RfcDispatch.

This function installs the following 4 functions (e.g. C-routines) to control the transactional behavior:

- **onCheckTid.** This function will be activated if a transactional RFC is called from an R/3 system. The current TID has been handed over to the function. The function has to store this TID in permanent storage and return 0. If the same TID will be called later again, it must return a value <> 0. If the same TID has already been started by another process, but is not completed, the function has to wait until the transaction is completed, or the user can stop the RFC connection with RfcAbort.
- **onCommit.** This function will be called, if all RFC functions, which belong to this transaction, are done and the local transaction can be completed. It should be used to locally commit the transaction, if working with database.
- **onRollback.** This function is called instead of the second function (onCommit), if there occurs an error in the RFC library while processing the local transaction. This function can be used to roll back the local transaction (working with database).
- **onConfirmTid.** This function will be called if the local transaction is completed. All information about this TID can be deleted.

Pay Attention:

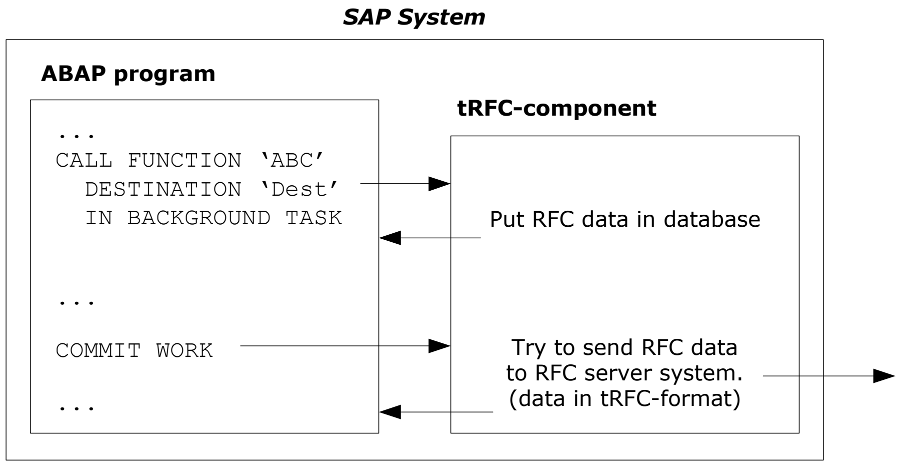
- These four functions must be realized in any tRFC server program and are independent to the real RFC function offered in a RFC server program. A server program can offer more than 1 RFC function but only 4 functions above and not 4 functions for each RFC function.
- To guarantee that each tRFC function will only be processed once, install all 4 functions.

Error handling. RFC differs between application and system failures. TRFC calls are asynchronous calls to the back end and the application exception could never be provided back to the caller. This is the reason to throw only system failures in all tRFC-function modules. The client may repeat the call because there was only a system error and the repetition of the call may be successful. This approach does not make any sense if the application throws any application failure (exception). Accordingly the repetition of the call with the same parameter will never succeed.

Important:

- Throw a system exception using RfcAbort. In this case the onRollback function will be called. And the caller will be informed, that the calls failed. The message will be provided to the caller. The tRFC-Server connection will be closed and RfcDispatch returns RFC_CLOSED.
- Never throw application exception i.e. you should not call RfcRaiseTables in your tRFC-Server.

Technical description (Within R/3 System)



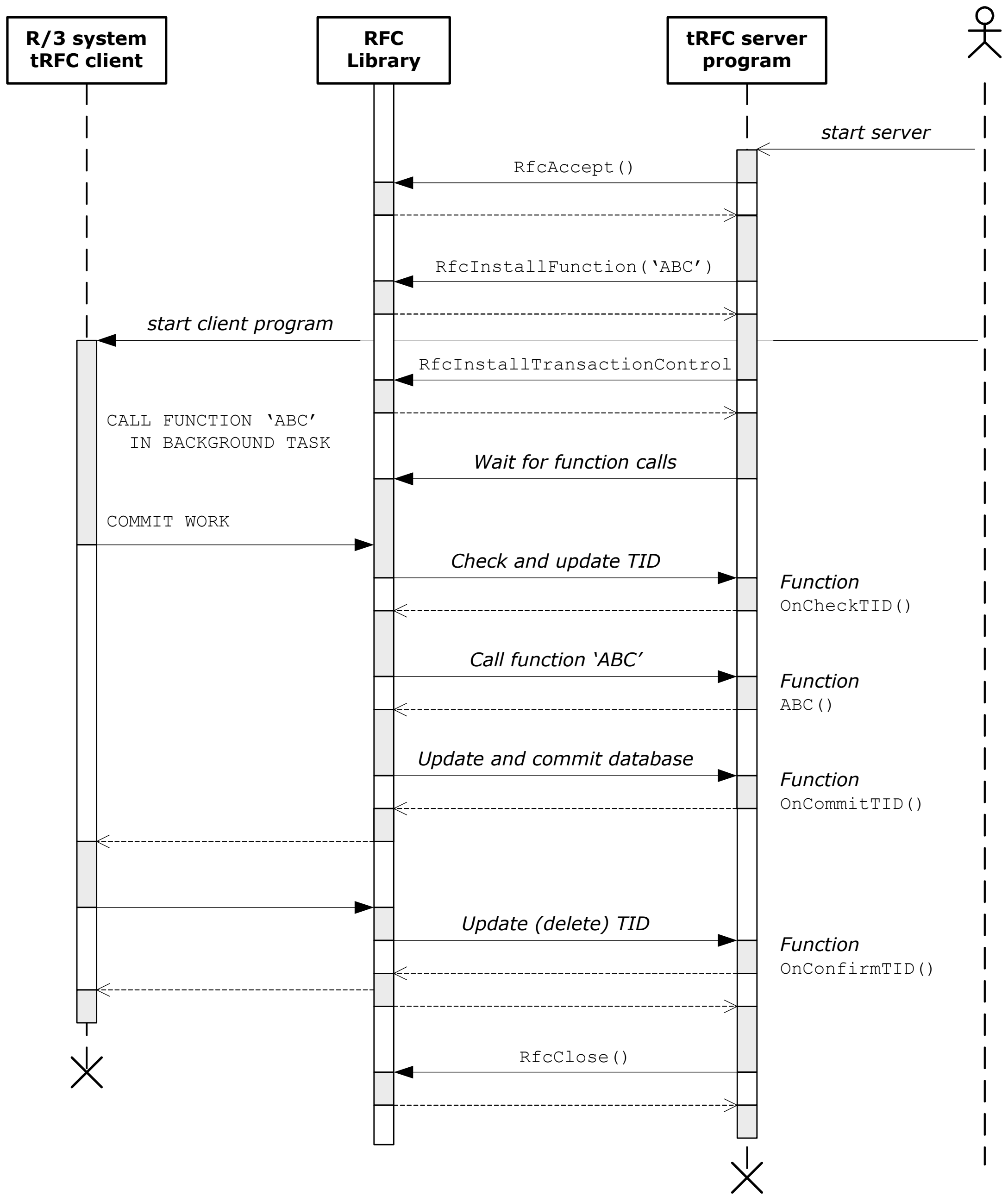
The following parameters can be configured using transaction sm59:

- try or do not try to connect to a RFC server program in the case of an error
- how many times consumed for trying
- the time between two tries.

Program location 'User' defined in sm59 (start RFC server program via currently using SAPGUI) is not available because the tRFC-component is not assigned to any SAPGUI while running.

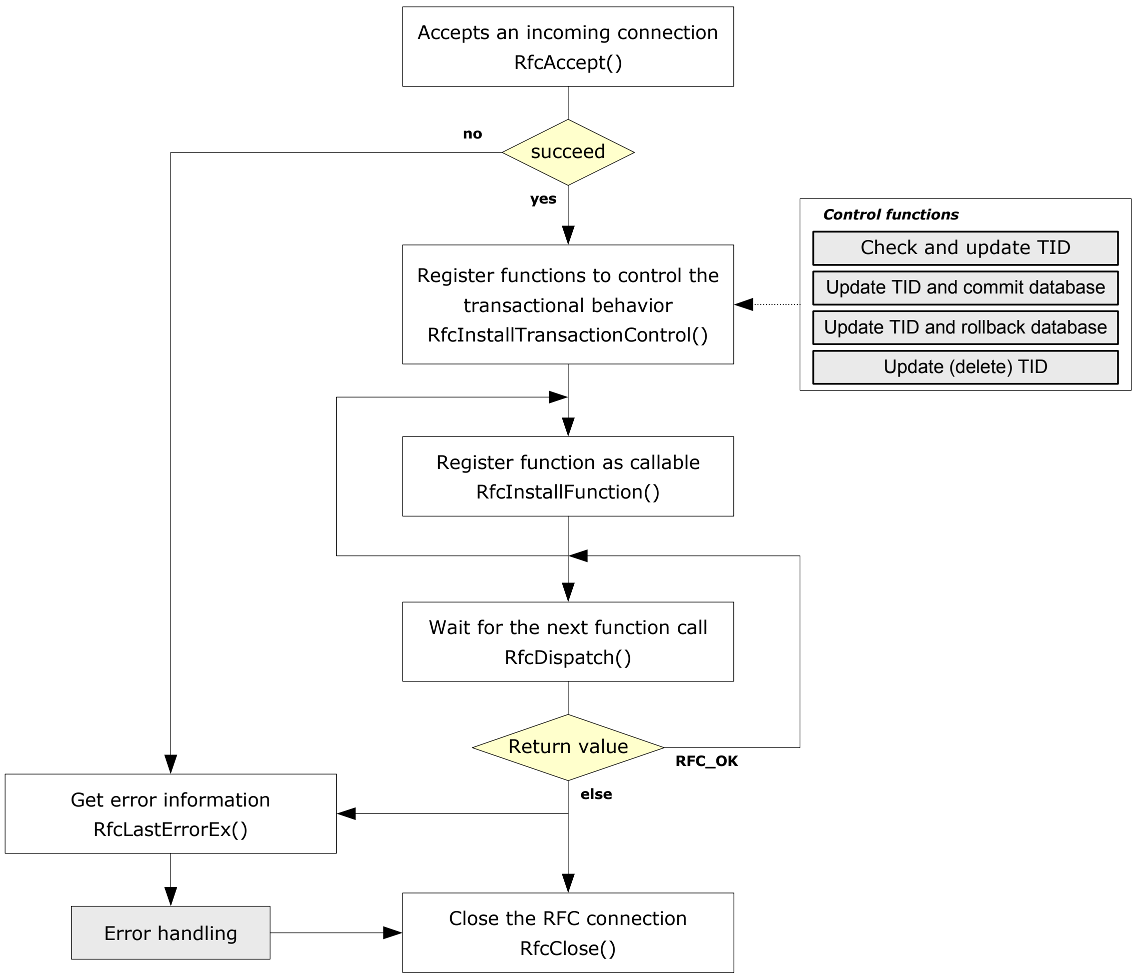
The transaction sm58 shows the running state of a transaction (if the transaction has not already been successfully executed).

The next two diagrams show the basic structure of a tRFC connection between a R/3 client and a external server.



Sequence diagram

Please have a look to the following sequence diagram for a tRFC connection between a R/3 client and an external server.


Accepts an incoming connection
RfcAccept()
succeed
no
yes
Register functions to control the transactional behavior
RfcInstallTransactionControl()
Control functions
Check and update TID
Update TID and commit database
Update TID and rollback database
Update (delete) TID
Register function as callable
RfcInstallFunction()
Wait for the next function call
RfcDispatch()
Return value
RFC_OK
else
Get error information
RfcLastErrorEx()
Error handling
Close the RFC connection
RfcClose()

Flow diagram for a tRFC server program.

The Sample Test Program ‘trfcserv.c’

The C-programs trfcserv.c, delivered in form of executable and source code with the RFC SDK, is a tRFC server program serving as an example. The ABAP program SRFCTEST can be used to test with this server program. Received data from R/3 will be written in the ‘trnn…n.dat’ on the running platform. Each line in this file is one line in an internal table. Only one internal table with 72 B line length is used. It uses the file I/O on the running platform for management the TID’s. For each TID there is an entry in the TID-management, which contains the date and time, the TID itself, the state of this transaction (CREATED, CONFIRMED, …) and the name of the data file. Since this program can run on different platforms, an according flag (SAPonUNIX, SAPonNT, etc…) must be set if you want to compile and link this program to your environment.

See source code for more details.

RFC Library and UNICODE

The Unicode Library is able to communicate with any RFC partner, regardless if the partner is Unicode or Non Unicode. The fact is, that a Unicode system can communicate with any system Unicode or ASCII and vice versa.

The simplest situation is, if **both RFC server and RFC client are homogenous**, i.e., both are either using Unicode, or non-Unicode systems. In this case the data will be sent to the RFC server as is, i.e., without any conversion at sender side for character-like data. The receiver converts the data into its own internal format. This was the common approach of the RFC library of previous releases.

Another possible situation is, if **only one RFC partner is using a Unicode system**. In this case the Unicode system must convert the data into a suitable ASCII data format (an 1 byte code page) before sending it. The matching codepage will be determined according to the logon language. For example, if the logon language is Japanese, the Unicode partner will convert the character-like data into a 8000 codepage before sending it. This codepage is called **communication codepage**.

This technique makes it necessary, that the data from a RFC client will be sent per default in ASCII format (1 byte char size) to the RFC server. If both partners are using Unicode Systems, the data exchange format will be switched to the homogeneous data exchange method. The handshake is done during the first call. Sure, this procedure causes useless overhead in case of Unicode-Unicode communication. To avoid this, it is possible for a **Unicode RFC client**, to declare the connection as a connection to a Unicode system:

- Either via the following entry in the connection string of `RfcOpenEx`: 'PCS=2'
- Or via the entry 'PCS=2' in the saprfc.ini file for the corresponding destination.

PCS is an abbreviation for Partners Character Size. This value is optional (default is '1').

The restriction of declaring the connection as Unicode connection cause, if the RFC server is a traditional system (1 byte character size system), the call to fail at server side with:

- An short dump '`FORMAT_NOT_SUPPORTED`' in R/3 or
- Return code `RFC_INVALID_PROTOCOL` in the external RFC program.

A RFC Unicode server determines automatically the partner character format and sends the answers in the correct format:

- without any conversion if the client is an Unicode program
- or, converted into the client's codepage if the client is a non-Unicode system.

There is not any activity required. If a Unicode client opens a connection to a Unicode server program using the PCS=1 option, the data exchange format will be switched to a normal (homogeneous) case at first call time. The RFC answer will already be sent in Unicode format without any conversion. The conversion is than done by the receiver.

The RFC Library exists in two variations:

- Non-Unicode – a traditional ASCII library. This Library is binary compatible to further releases. The names for the libraries did not change.
- Unicode – a RFC Library, which uses a SAP-Unicode data type for cha like data. This is an independent Library and has the postfix 'u' in its name. SAP offers the following Unicode RFC Libraries: NT (librfc32u.dll), UNIX (librfcu.a and librfccmu.<platform depending extension>.

Every application that uses static libraries must decide, when compiling, which library it will use. An application using shared libraries can use both libraries simultaneously. The Unicode and Non Unicode applications are interoperable, i.e., an Unicode RFC program can communicate not only with another Unicode RFC application but also with a non-Unicode RFC application and vice versa.

Special Features for EBCDIC-Based Systems

Customers who are using an OS/400 system and write their own programs in RPG, COBOL and so on, can include the RFC library in their programs to create an RFC client. However, they must note the following:

OS/400 programs are usually based on EBCDIC and not on ASCII. After the EBCDIC character set is made known to the RFC library, the RFC library can automatically convert the data, but not the metadata of interface functions since the RFC library is only shipped in ASCII or Unicode. However, there are conversion functions for the metadata that can convert the different country-specific EBCDIC character sets to ASCII and vice versa if the SAP code page to be used is specified. Therefore, before an RFC API can be called, all metadata must be converted from EBCDIC to ASCII (that is from one corresponding SAP code page to the other one). After the execution of an RFC function, all returned metadata must be converted from ASCII to EBCDIC. For this conversion, the ASCII version of the RFC library must be used.

For the conversion from EBCDIC to ASCII, the API RfcConvertE2A is provided; for the conversion from ASCII to EBCDIC, the API RfcConvertA2E is provided. (These functions are declared in SAPRFC.H.) The code page you specify should be the code page of the SAP system (an ASCII code page) and must match the IBM EBCDIC code page used on the system. For example, the SAP code page 0120 corresponds to the IBM EBCDIC code page 500 and the corresponding ASCII SAP code page is 1100. In such an environment, you specify 1100 as the code page. Alternatively, you can specify 0 as the code page; the IBM EBCDIC code page of the job will then be determined automatically and the corresponding ASCII SAP code page will be used.

If you call RfcOpen and then evaluate the errors, the RPG might look as follows:

```
*****************************************************************
** Data structure similar to C structure RFC_OPTIONS in saprfc.h
*****************************************************************
D RFCSID          S               4A
D RFCCLIENT       S               4A
D RFCUSER         S               9A
D RFCPASSWORD     S               9A
D RFCLANG         S               3A

DRFC_OPTIONS      DS                   ALIGN
D DESTINATION                     *    INZ(%ADDR(RFCSID))
D MODE                          10I 0 INZ(0)
D CONNOPT                         *    INZ(%ADDR(RFC_CONNOPT))
D CLIENT                          *    INZ(%ADDR(RFCCLIENT))
D USER                            *    INZ(%ADDR(RFCUSER))
D PASSWORD                        *    INZ(%ADDR(RFCPASSWORD))
D LANGUAGE                        *    INZ(%ADDR(RFCLANG))
D TRACE                         10I 0 INZ(3)

*************************************************************************
** Data structure similar to C structure RFC_ERROR_INFO in saprfc.h
*************************************************************************
D MSG52           S              52A

DRFC_ERROR_INFO   DS                   ALIGN
D KEY                            33A
D STATUS                        128A
D MESSAGE                       256A
```

```
D INTSTAT                     128A

C      'A46'         CAT       NULL          EBCDICTEXT
C                    EVAL      RC = RfcConvertE2A(RFCSID        :
C                                                EBCDICTEXT     :
C                                                %SIZE(RFCSID)  :
C                                                3              :
C                                                0              :           Null
terminate
C                                                1100)
C      '000'         CAT       NULL          EBCDICTEXT
Client
C                    EVAL      RC = RfcConvertE2A(RFCCLIENT     :
C                                                EBCDICTEXT     :
C                                                %SIZE(RFCCLIENT) :
C                                                3              :
C                                                0              :           Null
terminate
C                                                1100)
C      'XXXX'        CAT       NULL          EBCDICTEXT                     User
C                    EVAL      RC = RfcConvertE2A(RFCUSER       :
C                                                EBCDICTEXT     :
C                                                %SIZE(RFCUSER) :
C                                                4              :
C                                                0              :           Null
terminate
C                                                1100)
C      'YYYYYYYY'    CAT       NULL          EBCDICTEXT
Password
C                    EVAL      RC = RfcConvertE2A(RFCPASSWORD   :
C                                                EBCDICTEXT     :
C                                                %SIZE(RFCPASSWORD) :
C                                                8              :
C                                                0              :           Null
terminate
C                                                1100)
C      'E'           CAT       NULL          EBCDICTEXT
Language
C                    EVAL      RC = RfcConvertE2A(RFCLANG       :
C                                                EBCDICTEXT     :
C                                                %SIZE(RFCLANG) :
C                                                1              :
C                                                0              :           Null
terminate
C                                                1100)
C      'as0030'      CAT       NULL          EBCDICTEXT
Hostname
C                    EVAL      RC = RfcConvertE2A(RFCHOSTNAME   :
C                                                EBCDICTEXT     :
C                                                %SIZE(RFCHOSTNAME) :
C                                                8              :
C                                                0              :           Null
terminate
C                                                1100)
C                    EVAL      SYSNR = 48
C      'as0030'      CAT       NULL          EBCDICTEXT
Gateway host
C                    EVAL      RC = RfcConvertE2A(RFCGATEWAYHOST:
C                                                EBCDICTEXT     :
C                                                %SIZE(RFCGATEWAYHOST) :
C                                                8              :
C                                                0              :           Null
terminate
C                                                1100)
C      'sapgw48'     CAT       NULL          EBCDICTEXT
Gateway service
C                    EVAL      RC = RfcConvertE2A(RFCGATEWAYSRV :
```

```
C                                                   EBCDICTEXT    :
C                                                   %SIZE(RFCGATEWAYSRV) :
C                                                   7             :
C                                                   0             :          Null
terminate
C                                                   1100)
********************************************************************************
**********
** Call C function to open the RFC connection to R/3 and do the error handling
**
********************************************************************************
**********
C                   EVAL      RFC_Handle = 999
C                   EVAL      RFC_HANDLE = RfcOpen(RFC_OPTIONS)
C     RFC_HANDLE    IFEQ      0
 * ERROR HANDLING
C                   EVAL      RC = RfcLastError(RFC_ERROR_INFO)
C     RC            IFEQ      0
C                   EVAL      RC = RfcConvertA2E(MSG52      :
C                                                   MESSAGE    :
C                                                   %SIZE(MSG52) :
C                                                   %SIZE(MESSAGE)    :
C                                                   64            :          fill
space char
C                                                   1100)
C                   DSPLY                    MSG52
 * Variable MESSAGE contains error text
C                   ENDIF
C                   ENDIF
```

After the connection has been successfully established, you must make the SAP code page of the client data known to the RFC library. For example, if a client works with data in the IBM EBCDIC code page 500, the corresponding SAP code page is '0120'. The data will then be directly accepted in EBCDIC code page 500 and the results of a query in the SAP system will be passed on to the client in this code page. The corresponding code might look as follows:

```
C                   EVAL      EBCDICTEXT = '0120' + NULL
C                   EVAL      RC = RfcConvertE2A(RFC_CODEPAGE  :
C                                                   EBCDICTEXT    :
C                                                   %SIZE(RFC_CODEPAGE) :
C                                                   %SIZE(EBCDICTEXT)   :
C                                                   0             :          Null
terminate
C                                                   1100)
C                   EVAL      RC = RfcSetCodepage (RFC_Handle  :
C                                                     RFC_CODEPAGE)
```

Writing multithreaded RFC applications

A RFC application can have more than one thread working parallel with RFC. This enables to create powerful RFC applications. The following issues have to be kept in mind when developing multithreaded RFC programs:

- **Flexibility. The thread safe RFC Library can be used in a single or multithread environment. No additional API calls are needed for using in a multithreaded environment.**
- **Trace mechanism. The trace files will be written per thread.**
- **Synchronisation.** The RFC library implements a model which is quite similar to the COM+ Thread Neutral Synchronized Components: A RFC handle can be used in several threads, but can only be active in one thread at a time. A RFC handle of a RFC connection, created by one thread can be used in another thread, but these threads have to synchronize the access to this handle. The RFC library does not protect itself against simultaneous calls from various threads for the same handle. Therefore the application must be aware that this simultaneous use of RFC handles should be avoided. The RFC library does not check whether an RFC call with this handle is still working. The behaviour is undefined in case of unsynchronised concurrency access from different threads to the same RFC handle.
- **Access synchronisation to internal tables.** A handle of an internal table can also be used in different threads. The access on this handle has to by synchronized. The behaviour is undefined in case of unsynchronised concurrency access from different threads to the same table too.

Our suggestions for writing of:

- **Multithreaded RFC client program.** The main thread or a service thread waits for work orders. Each work order is then delegated to a worker thread, which does the respective work. In doing so, a worker thread retains its private RFC handles, which are opened when necessary (RfcOpenEx) and then closed again after having been used. In this, each handle remains assigned to a thread for its entire lifetime. A pool of open RFC handles is kept. When necessary, the worker threads remove a handle from this pool, use it to do their work and place it then back in the pool when the work has been completed. While the worker thread is working with the handle, any other thread does not use the handle. RFCSDK contains the sample RFC multithreaded client program rfcthcli.c. This program can be used as basis for new multithreaded RFC applications.
- **Multithreaded RFC server program.** Multithread RFC server programs are normally registered servers. Each thread implements the normal technology of a simple RFC server program by waiting for the incoming calls in an `RfcWaitForRequest` loop and by passing these on to the respective server function via the `RfcDispatch`.. RFCSDK contains the sample RFC multithreaded server program rfcthsrv.c. This program can be used as basis for new multithreaded RFC applications.

Technical Details

This chapter describes important technical details.

Thread safe library on UNIX platforms

SAP offers a shared RFC library on UNIX platforms, which is thread safe. Depending on the Unix derivative, this library is called in non-Unicode case:

- `librfccm.so (Alphaosf, Linux, SUN),`
- `librfccm.sl (HP),`
- librfccm.o (AIX)

The Unicode version of the shared RFC library is called:

- `librfcum.so (Alphaosf, Linux, SUN),`
- `librfcum.sl (HP),`
- `librfcum.o (AIX)`

Compiling the SAP-Interface

To compile your applications you must use the same compiler, or a compatible one, that compiled the SAP-Interface. The `sapinfo` example application (part of SDK), called with option `-v`, gives the compiler version used to compile the SDK example applications.

The following sections list the commands and options used to compile, link, and run the example RFC applications in `RFCSDK/bin`.
Note: Please modify these values as appropriate for your compilation environment; e.g. the location of the `RFCSDK` in the listed commands must be adjusted to your environment; likewise the compiler/linker will probably be installed elsewhere in your environment.

Non Unicode Make

• **Alphaofs** DigitalUnix/Tru64, COMPAQ TRU64 4.0F, COMPAQ TRU64 4.0G, COMPAQ TRU64 5.0A

```
/bin/cc -std1 -warnprotos -ieee_with_no_inexact -unsigned -
 pthread -readonly_strings -DSAPwithTHREADS -DSAPonALPHA -
 DSAPonUNIX -c sapinfo.c

cxx -std1 -pthread -rpath .:RFCSDK/lib -o sapinfo sapinfo.o
 librfccm.so
```

• **AIX 64**

```
/bin/cc_r -z -q64 -qlanglvl=ansi -qinfo -qarch=com -
 spill=1024
-DSAPwithTHREADS -qmaxmem=4096 -O -DNDEBUG -DSAPonRS6000 -c
 sapinfo.c

/bin/cc_r -z -q64 -LRFCSDK/lib -L/usr/lib
-o sapinfo sapinfo.o librfccm.o
```

• **AS390 und AS400**

SAP provides only the library for these platforms. The RFC SDK does not be assisted.

• **HP-Itanium 64**

```
cc -DSAPonUNIX -z +DD64 +DSitanium2 -Ae +Olibmerrno
 +Oinitcheck +w1 +We281 +W392,829,818,887 +uc -
 DSAPwithTHREADS -mt -DSAPonHPPA -DSAPonHPIA64 -c sapinfo.c

/opt/aCC/bin/aCC +DD64 -Wl,+n -lnsl -lpthread -LRFCSDK/lib
-l:librfccm.so -Wl,-a,default -ldld -lsec -o sapinfo
 sapinfo.o
```

- **HP-UX 64** (On HP, use the command 'chattr' to adjust the path from which the shared library is to be loaded):

```
cc -DSAPonUNIX -z +DD64 -Ae +w1 -DSAPwithTHREADS -
 D_REENTRANT
-D_POSIX_C_SOURCE=199506L -DSAPonHPPA -c sapinfo.c

/opt/aCC/bin/aCC +DD64 -Wl,+n -lnsl -Wl,-a,default -
 lpthread -ldld -lsec -LRFCSDK/lib -o sapinfo sapinfo.o
 librfccm.sl
```

To compile on a HP 32 bit platform please suppress the compiler option `+DD64`.

- **Linux 32** Linux Intel:

```
gcc -funsigned-char -Wcast-align -pthread -fPIC -
 DSAPwithTHREADS -DSAPonLIN -c sapinfo.c

gcc -funsigned-char -pthread -fPIC –Wl,-rpath,FCSDK/lib
-LRFCSDK/lib -o sapinfo sapinfo.o  librfccm.so
```

- **SUN 64** (SOLARIS/SPARC 2.6, SOLARIS/SPARC 7, SOLARIS/SPARC 8):

```
/opt/SUNWspro/bin/cc -v –xarchh=v9 -xchar=unsigned  -
 D__XPG4_CHAR_CLASS__
-KPIC -xs -mt -DSAPwithTHREADS -DNDEBUG –DSAPonSUN -O -c
 sapinfo.c

/opt/SUNWspro/bin/cc –xarch=v9 -R.:RFCSDK/lib:/opt/SUNWlu62
 -KPIC -mt
-ldl -lnsl -lsocket -o sapinfo sapinfo.o librfccm.so
```

To compile on a SUN 32 bit platform please suppress the compiler option `–xarch=v9`.

- **NT Intel 32**

```
cl  -DBCDASM -Od -Ob1 -Op -Gy -GX -W3 -D_X86_ -DWIN32 -
 D_AFXDLL -D_DLL -MD -FR -J -DSAPonNT -c sapinfo.c


link -STACK:0x800000  ole32.lib oleaut32.lib oledb.lib
 uuid.lib kernel32.lib advapi32.lib user32.lib gdi32.lib
 winspool.lib ws2_32.lib netapi32.lib setargv.obj
 comdlg32.lib shell32.lib dbghelp.lib version.lib mpr.lib -
 OPT:REF -LARGEADDRESSAWARE  -out:sapinfo.exe -
 subsystem:console sapinfo.obj librfc32.lib
```

- **NT Itanium 64**

```
cl  -Op -GX -O2 -G2 -Gy -Zi -W3 -Wp64 -DWIN32 -DWIN64 -
 D_IA64_  -D_MT -MT -J -DNDEBUG -DPRAGMA_WARNING -c
 sapinfo.c

link -STACK:0x200000  ole32.lib oleaut32.lib oledb.lib
 uuid.lib kernel32.lib advapi32.lib user32.lib gdi32.lib
 winspool.lib ws2_32.lib netapi32.lib setargv.obj
 comdlg32.lib shell32.lib dbghelp.lib version.lib mpr.lib -
 OPT:REF
-out:sapinfo.exe -subsystem:console sapinfo.obj
 librfc32.lib
```

- **NT AMD 64**

```
cl -GX -O2 -GF- -Gy -Zi -W3 -Wp64 -DWIN32 -DWIN64 -D_AMD64_
-D_CRT_NON_CONFORMING_SWPRINTFS  -D_MT -MT -J -DNDEBUG -
 DPRAGMA_WARNING -c sapinfo.c


link -STACK:0x200000  ole32.lib oleaut32.lib oledb.lib
 uuid.lib kernel32.lib advapi32.lib user32.lib gdi32.lib
 winspool.lib ws2_32.lib netapi32.lib setargv.obj
 comdlg32.lib shell32.lib dbghelp.lib version.lib mpr.lib -
 OPT:REF
-out:sapinfo.exe -subsystem:console sapinfo.obj
 librfc32.lib
```

Unicode Make

For compilers that do not support UTF-16 encoded string literals, please use a modified compile procedure:

1. preprocess your source;
2. run the preprocessed text through an SAP supplied Perl script;
3. compile the resulting text.

See Note 763741 for details on this procedure, the script, and its usage. The note gives a reference to the UTF-16 string literal standard, which the compiler manufacturers are implementing.

Currently (2004-10) these compilers, among possibly others, support UTF-16:

- the HP compiler *HP C/HP-UX Version B.11.11.06*
- Microsoft Windows C/C++ Compilers.

On some platforms the compiler call is prefixed by a call to a procedure setting up the compile environment. You should remove or modify the latter, depending on your compilation environment.
Note: when `-DSAPwithUNICODE` is absent from this command, this description pertains to the non-Unicode `RFCSDK`.

When the pre-compilation script is set you must use the modified compile procedure described above. When this script name is not set the compiler understands UTF-16 literals, or this description pertains to the non-Unicode RFC SDK, and you can use standard compile procedures.

- **Alphaosf**

pre-compilation script = `u16lit.pl`

Compile and link command:
```
/bin/cc -DSAP_RFC_TIME -c -DSAPwithUNICODE  sapinfo.c

cxx -LRFCSDK/lib -lrfcum -lsapu16_mt -lsapucum -o sapinfo
 sapinfo.o
```

Run (environment)
```
LD_LIBRARY_PATH=RFCSDK/lib
```

- **AIX 64** `rs6000_64, as400_pase_64`

pre-compilation script = `u16lit.pl`

Compile and link command:
```
cc_r -q64 -c -DSAPwithUNICODE -IRFCSDK/include sapinfo.c
xlC_r -q64 -brtl -bnortllib -LRFCSDK/lib librfcum.o
 libsapu16_mt.so libsapucum.so –o sapinfo sapinfo.o
```

Run (environment)
```
LIBPATH = RFCSDK/lib
```

- **AS390 und AS400**

SAP provides only the library on these platforms. The RFC SDK does not be assisted.

- **HP-UX 64**
- **HP-UX Itanium 64**

Compile and link command:
```
cc +DD64 -c -IRFCSDK/include -DSAPwithUNICODE  sapinfo.c

/opt/aCC/bin/aCC +DD64 -LRFCSDK/lib -l:librfcum.sl -
 l:libsapucum.sl -o sapinfo sapinfo.o
```

Run (environment)
```
LD_LIBRARY_PATH = RFCSDK/lib
```

- **Linux 32** `linuxintel,`
- **Linux 64** `linuxx86_64, linuxia64, linuxppc64,`

pre-compilation script = `u16lit.pl`

Compile and link command:
```
gcc –IRFCSDK/include -DSAP_RFC_TIME -c –DSAPwithUNICODE
 sapinfo.c

gcc -Wl,-rpath,RFCSDK/lib,-LRFCSDK/lib, -lrfcum, -lsapucum
 –o sapinfo sapinfo.o
```

```
Add option –m64 to compile and/or link on 64 bit platforms.
```

Run (environment)
```
LD_LIBRARY_PATH=RFCSDK/lib
```

- **Sun 64**

pre-compilation script = `u16lit.pl`

Compile and link command:
```
/opt/SUNWspro/bin/cc -xarch=v9 -c -DSAPwithUNICODE  sapinfo.c

/opt/SUNWspro/bin/CC -xarch=v9 -ldl -lc -LRFCSDK/lib -lrfcum -
 lsapu16_mt -lsapucum -o sapinfo sapinfo.o
```

Run (environment)
```
LD_LIBRARY_PATH = RFCSDK/lib
```

- **NT Intel 32**
- **NT AMD 64**
- **NT Itaninum 64**

Compile and link command:
```
cl -c -MD -IRFCSDK\include -DSAPwithUNICODE   sapinfo.c

link ole32.lib oleaut32.lib oledb.lib uuid.lib kernel32.lib
 advapi32.lib user32.lib gdi32.lib winspool.lib ws2_32.lib
 netapi32.lib setargv.obj comdlg32.lib shell32.lib dbghelp.lib
 version.lib mpr.lib secur32.lib /LIBPATH:RFCSDK\lib
 librfc32u.lib libsapucum.lib /OUT sapinfo.exe sapinfo.obj

Add library atl21asm.lib to link on NT Itanium 64.
Add library  bufferoverflowU.lib to link on NT AMD 64.
```

Run (environment)
```
set PATH=%PATH%;RFCSDK\lib
```

Stack size

The RFC requires at least a 300KB stack.

The following example demonstrates the stack handling when using RFC Library:

```
    #define RFC_MIN_STACK     (300 * 1024)
    #define OWN_MIN_STACK     (64  * 1024)
    #define TOTAL_MIN_STACK   (RFC_MIN_STACK +
OWN_MIN_STACK)
    #ifndef PTHREAD_STACKSIZE_MIN  // sometimes missing
    #define PTHREAD_STACKSIZE_MIN RFC_MIN_STACK
    #endif

    ...
    pthread_attr_init(&thr_attr);
    //set threads attributes such as PTHREAD
    //PTHREAD_CREATE_JOINABLE or other
```

```
        {
        /* and now make sure that we have enough stack space
         * some defaults are too small */
                size_t stack_size, min_size;

                rc = pthread_attr_getstacksize(&thr_attr,
&stack_size);
                if (0 != rc)
                {
                printf("cannot get threads stacksize\n");
                 exit(1);
                }

   min_size = PTHREAD_STACKSIZE_MIN;

   if (min_size <= TOTAL_MIN_STACK)
                {
    min_size = TOTAL_MIN_STACK;
                }

   if (stack_size < min_size)
              {
       int page_size;

       page_size = (int) sysconf(_SC_PAGESIZE);
       min_size = ((min_size / page_size) + 1 ) *
page_size;

                   printf("adjusting stack size from %d to
%d\n",
                          stack_size, min_size);

                rc = pthread_attr_setstacksize(&thr_attr,
min_size);
       if (0 != rc)
                   {
                   printf ("cannot set threads stacksize\n");
                exit(1);
                   }
   }
    }
```

Initialization

You must initialise the library using RfcInit API. RfcInit must return before the application continues with any other API call.

The following example demonstrates the correct initialisation:

```
static pthread_once_t initialize_rfc_once =
PTHREAD_ONCE_INIT;

int main(int argc, char** argv)

    {

        (void) pthread_once (&initialize_rfc_once,
RfcInit);

     ...

    }
```

Closing RFC connection in multithreaded environment

Because RFC handles can only be active in one thread at a time, the RFC handles have to be closed individually. You should **NEVER** call RfcClose (RFC_HANDLE_NULL) in an multi thread application.

Restrictions

Due to the development of multithreaded rfc applications, please notice the following restrictions:

- **No support for load balancing via SNC** at this time.
- **RFC is not cancel safe**. If a thread, which executes an RFC call, is cancelled, a memory leak or deadlock may occur. After returning from an RFC-API call, the worker threads can check whether they must be cancelled. This may be the case after an RfcWaitForRequest or RfcCallReceiveEx or RfcReceiveEx. With RfcClose or RfcAbort, you can release the RFC handle and cancel the thread.
- **RFC is not "fork safe".**

The saprfc.ini file

The RFC application can read the saprfc.ini file to determine the RFC parameters needed to establish an RFC connection.

The saprfc.ini file has to be in the working directory of the external RFC program or its location could be defined by the environment variable RFC_INI. The content of RFC_INI could be either the full path or only the directory path of the sapRFC.ini file. Both entries for the same sapRFC.ini are correct:

RFC_INI = d:\RFCenvironment\sapRFC.ini

RFC_INI = d:\RFCenvironment

The advantage of this file is, that all specific RFC parameters known at this time (load balancing, ABAP-Debug, RFC with SAPGUI) ,and in the future, can be used without changes to RFC programs

To use this file, **RFC client** programs must call the RfcOpenEx with the option DEST=<destination> in their connection string and the destination must point to the corresponding entry in the saprfc.ini file

To use this file, **RFC server** programs must call the RfcAccept with -D<destination> as parameter and the destination must point to the corresponding entry of type ‘R’ in the saprfc.ini file.

If working with Load Balancing, and there is no info about MSHOST, the RFC library will try to get this host name from the sapmsg.ini customized for SAPLOGON on Windows platforms. Usually, these files are installed in the Windows directory. You can also copy these files in a directory, which is specified by the environment 'RFC_LOGON_INI_PATH'. On NON-Windows platforms, you can work with this environment variable or copy these files in your working directory.

The SAP-Router string must be defined in the R/3 system name:

- R3NAME = /H/sapgate1/S/3297/LOI, where LOI is the R/3 name
- R3NAME = "/SAP America/LOI", where 'SAP America' is the router name defined in 'saproute.ini' for SAPLOGON to Windows.

There are 5 connection types available:

- Type **B** is recommended to connect to a R/3 system (using Load Balancing).
- Type **A** is only to be used if you want to connect to a specific application server.
- Type **E** is for RFC client program working with another external program as RFC server program.
- Type **R** is for RFC server programs or for a client program working with another external program as RFC server program, which is already registered at a SAP gateway.
- Type **2** is only for connecting to a R/2 system.

Type 'R' entries

Register **a RFC server** at a SAP gateway and wait for RFC calls by a R/2 or R/3 system or establish a connection from an **RFC client** to an external program, which is already registered at a SAP gateway.

The following parameters can be used:

```
DEST=<destination in RfcAccept or RfcOpenEx>
TYPE=<R: Register at SAP-GW or connect to reg. program>
PROGID=<Program ID, optional, default: destination>
GWHOST=<Host name of the SAP gateway>
GWSERV=<Service name of the SAP gateway>
RFC_TRACE=<0/1: OFF/ON, optional, default: 0 (OFF)>
SNC_MODE=<0/1: OFF/ON, optional, default: 0 (OFF)>
SNC_QOP=<1/2/3/8/9, optional, default: 8>
SNC_MYNAME=<Own SNC name, optional>
SNC_PARTNERNAME=<Partner SNC name>
SNC_LIB=<Path and file name of the SNC library>
```

Please note:

- The host name and service name of the SAP gateway must be defined in the 'hosts' and 'service' files.
- If SNC_MODE is ON, the SNC_LIB must be defined. SNC_MYNAME and SNC_QOP are optional.
- SNC_PARTNERNAME is only needed for an RFC client.

Type 'B' entries

Connect to a **R/3 system via the Load Balancin**g feature of a R/3 system.

The following parameters can be used:

DEST=<destination in RfcOpenEx>

TYPE=<B: use the load balancing feature

R3NAME=<Name of R/3 system, optional default: destination>

MSHOST=<Host name of the message server>

GROUP=<Application servers group name, optional, default 'PUBLIC'>

RFC_TRACE=<0/1: OFF/ON, optional, default: 0 (OFF

ABAP_DEBUG=<0/1: OFF/ON, optional, default: 0 (OFF)>

USE_SAPGUI=<0/1/2: OFF/ON/INVISIBLE_SAPGUI after each RFC-Function, def.: 0 (OFF)>

SNC_MODE=<0/1: OFF/ON, optional, default: 0 (OFF

SNC_QOP=<1/2/3/8/9, optional, default: 8>

SNC_MYNAME=<Own SNC name, optional>

SNC_PARTNERNAME=<SNC name of the message server>

SNC_LIB=<Path and file name of the SNC library>

Please note:

- The host name and service name of the message must be defined in the 'hosts' and 'service' files: (<service name> = sapms<R/3 system name>).
- If SNC_MODE is ON the SNC_LIB must be defined.

Type 'A' entries

Connect to a **specific R/3 application server**.

Following parameters can be used:

DEST=<destination in RfcOpenEx>

TYPE=<A: RFC server is a specific R/3 application Server>

ASHOST=<Host name of a specific R/3 application Server>

SYSNR=<R/3 system number>

GWHOST=<optional, default: gateway on application Server>

GWSERV=<optional, default: gateway on application Server>

RFC_TRACE=<0/1: OFF/ON, optional, default: 0 (OFF)>

ABAP_DEBUG=<0/1: OFF/ON, optional, default: 0 (OFF)>

USE_SAPGUI=<0/1/2: OFF/ON/INVISIBLE_SAPGUI each RFC-Function, def.: 0 (OFF)>

SNC_MODE=<0/1: OFF/ON, optional, default: 0 (OFF

SNC_QOP=<1/2/3/8/9, optional, default: 8

SNC_MYNAME=<Own SNC name, optional

SNC_PARTNERNAME=<Partner SNC name>

SNC_LIB=<Path and file name of the SNC library>

Please notice:

- The host name and service name of the application server must be defined under 'hosts' and 'service' (<service name> = sapdp<R/3 system number>).
- The host name and service name of the SAP gateway must be defined in the 'hosts' and 'service' files.
- If you don't define GWHOST and GWSERV in this entry, the service name of the SAP gateway still needs to be defined in the 'service' file: (<service name> = sapgw<R/3 system number.
- If SNC_MODE is ON the SNC_LIB must be defined.

Type '2' entries

Connect to a R/2 system.

The following parameters can be used:

DEST=<destination in RfcOpenEx and in sideinfo file for gateway

TYPE=<2: RFC server is a R/2 system

GWHOST=<Host name of the SAP gateway>

GWSERV=<Service name of the SAP gateway

RFC_TRACE=<0/1: OFF/ON, optional, default: 0 (OFF

SNC_MODE=<0/1: OFF/ON, optional, default: 0 (OFF

SNC_QOP=<1/2/3/8/9, optional, default: 8>

SNC_MYNAME=<Own SNC name, optional>

SNC_PARTNERNAME=<SNC Partner Name>

SNC_LIB=<Path and file name of the SNC library>

Please notice:

- The host name and service name of the SAP gateway must be defined in the 'hosts' and 'service' files.
- SNC for RFC connections to R/2 is not supported.

Type 'E' entries

Connect to an **external RFC server program,** which will be started by the SAP gateway.

The following parameters can be used:

DEST=<destination in RfcOpenEx>

TYPE=<E: Server program will be started by SAP gateway>

GWHOST=<Host name of the SAP gateway>

GWSERV=<Service name of the SAP gateway>

TPHOST=<Host name of the server program>

```
TPNAME=<Path name and server program name>
RFC_TRACE=<0/1: OFF/ON, optional, default: 0 (OFF)>
SNC_MODE=<0/1: OFF/ON, optional, default: 0 (OFF)>
SNC_QOP=<1/2/3/8/9, optional, default: 8>
SNC_MYNAME=<Own SNC name, optional>
SNC_PARTNERNAME=<Partner SNC name>
SNC_LIB=<Path and file name of the SNC library>
```

Please notice:

- The host name and service name of the SAP gateway must be defined in the 'hosts' and 'service' files.
- If SNC_MODE is ON, the SNC_LIB must be defined.
- After being started by the SAP gateway, the program will run with the SNC library defined for the respective SAP gateway.

RFC tracing mechanism

The RFC tracing mechanism provides the opportunity to analyze RFC error situations and helps to find a few bugs, which have not been explored yet. The trace file(s) will be written

- either to the RFC working directory,
- or to the directory defined by the environment variable RFC_TRACE_DIR.

If one RFC partner turns trace on, every RFC partner for this RFC tree turns on the trace.

The simplest way to turn RFC trace on, is to set the environment variable RFC_TRACE=1. In this case all RFC connections will be traced.

To turn the RFC trace on for only one client, the connection string of the RFC connection should contain the following entry: "… TRACE=1 …". Another possibility with the same effect is to insert the following entry into the saprfc.ini file for the corresponding destination: RFC_TRACE=1.

RFC and SAP-Router

SAP-Router is a SAP-Software product acts like a firewall by regulating access to/from your network. See SAP Note 30289 for more detail about SAP-Router.

General one or more SAP-Routers could be involved into communication chain.

RFC-Client Program and SAP-Router

Any RFC client program can connect to a SAP system via SAP-Router. The feature RFC with SAPGUI is also possible via Sap-Router. One or more SAP-Routers could be involved to communication chain. The RFC library has to be informed about all used SAP-Routers via parameter about hostname in RfcOpenEx-API.

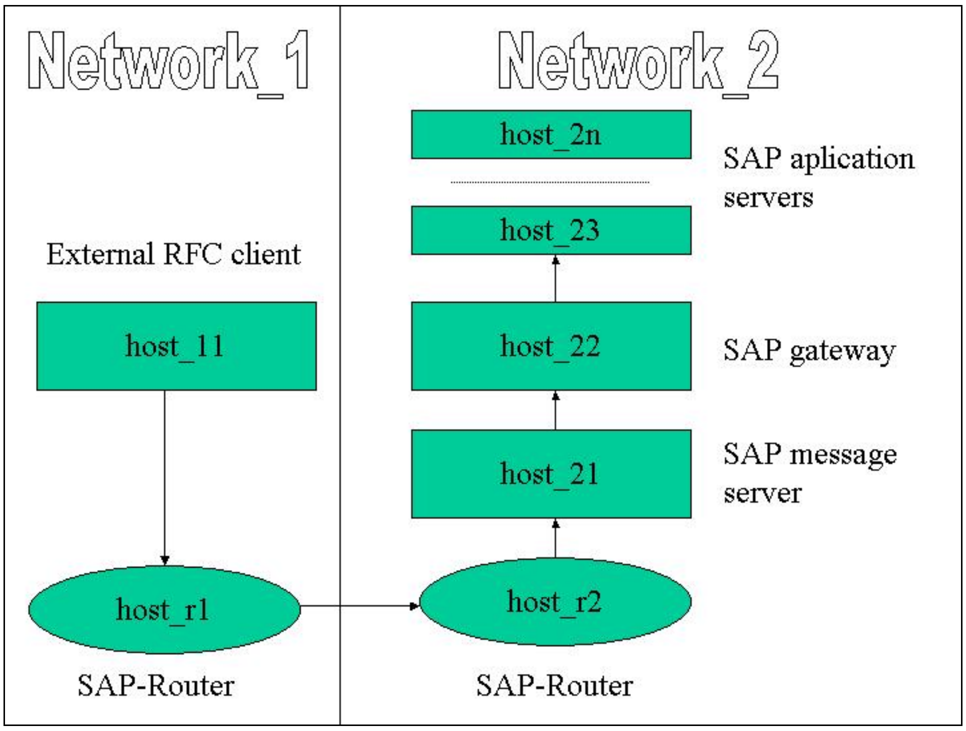


Figure 0-1

Using Load Balancing

The host name of the message server must contain the route string. For the above example (see Figure 0-1), the RFC client must set the name of the message server as following: **/H/host_r1/H/host_r2/H/host_21**.

The permission table of saprouter on host_r2 in Neework_2 has to contain following entries:

P	host_r1	host_21	sapms <R3name>
P	host_r1	<R/3-ashost 1>	sapgw <R/3 system number>
…			
P	host_r1	<R/3-ashost n>	sapgw <R/3 system number>

Connection to an explicit application host using default gateway

The hostname of specified application host must contain the router string. For the example above, the RFC client should set the host name of the application server as following: **/H/host_r1/H/host_r2/H/host_23.**

The permission table of saprouter on host_r2 in Network_2 has to contain following entry:

P host_r1 host_22 sapgw <R/3 system number>

Connection to an explicit application server using explicit gateway

If working with an explicit SAP gateway the host name of SAP gateway must contain the route string. The hostname may not contain the route string. For the example above, the RFC client must set the host name of application sever and host name of SAP gateway for connection data as following:

GW-Host: **/H/host_r1/H/host_r2/H/host_22**

AS-HOST: **host23**

The route permissions table must contain following entry:

P host_r1 host_22 sapgw<GW service number>

RFC Server and SAP-Router

SAP-Router feature is available for following kinds of external RFC servers:

- Registered RFC server
- Server started by Sap gateway
- Server started by SAPGUI

An RFC server started directly by application server runs on the same machine as the application server. In this case SAP-Router is not involved in the communication chain.

Due to the kind of RFC server there are different ways how an external server works in an SAP-Router environment.

Registered RFC server and SAP-Router

The hostname of SAP gateway must contain the router string. For the example shown in the Figure 0-1 the gateway host should be defined as following: **/H/host_r1/H/host_r2/H/host_21**.

The RFC server program should be registered at the SAP gateway with following options:

```
srfcserv -a<Prog.ID> -g/H/host_r1/H/host_r2/H/host_21 -x
sapgw<GW service number>.
```

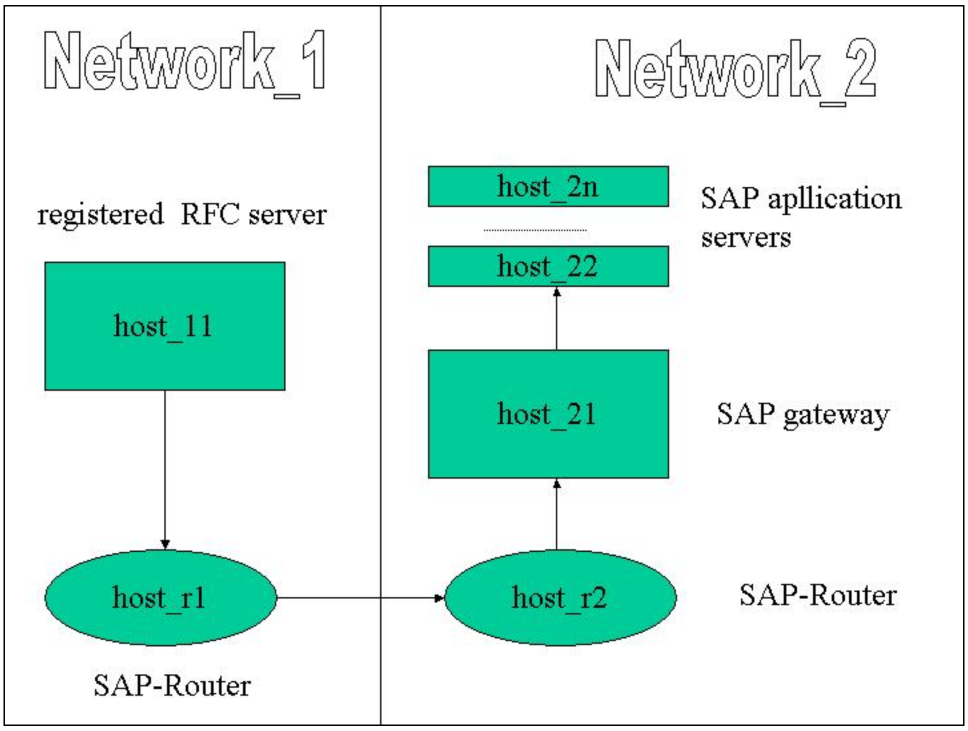


Figure 0-1

The route permission table of SAP-Router on host_r1 in the Network_1 must contain following entry:

```
P   host_11      host_r2       3299
```

The route permission table of SAP-Router on host_r2 in the Network_2 must contain following entry:

```
P   host_r1      host_21       sapgw<GW service number>
```

The corresponding destination in SM59 should be defined as following:

Connection type:	T
Activate type:	Registering
Program ID:	<Prog.ID>
Gateway host:	host_21
Gateway Service:	sapgw<GW service number>

RFC server started by a SAP gateway

Due an SAP gateway cannot start an RFC server program with remote shell on another machine over SAP-Router it is necessary to install an SAP gateway on a machine in the network where the external RFC server program will run. For better performance both should run on the same computer.

Following example (see Figure 0-2) demonstrates two different networks with two SAP-Routers and how an RFC program can be started by an SAP gateway.

Destination for RFC server in SM59 should be defined as following:

Connection Type:	T
Activate Type:	Start on explicit host
Program:	../rfcsdk/bin/srfcserv
Target host:	host_11

Gateway host:	/H/host_r2/H/host_r1/H/host_11
Gateway service:	sapgw<GW service number>

The route permission table of SAP-Router on host_r1 in Network_1 should contain following entry:

P host_r2 host_11 sapgw<GW service number>

The route permission table of SAP-Router on host_r2 in Network_2 should contain following entry:

P host_21 host_r1 3299

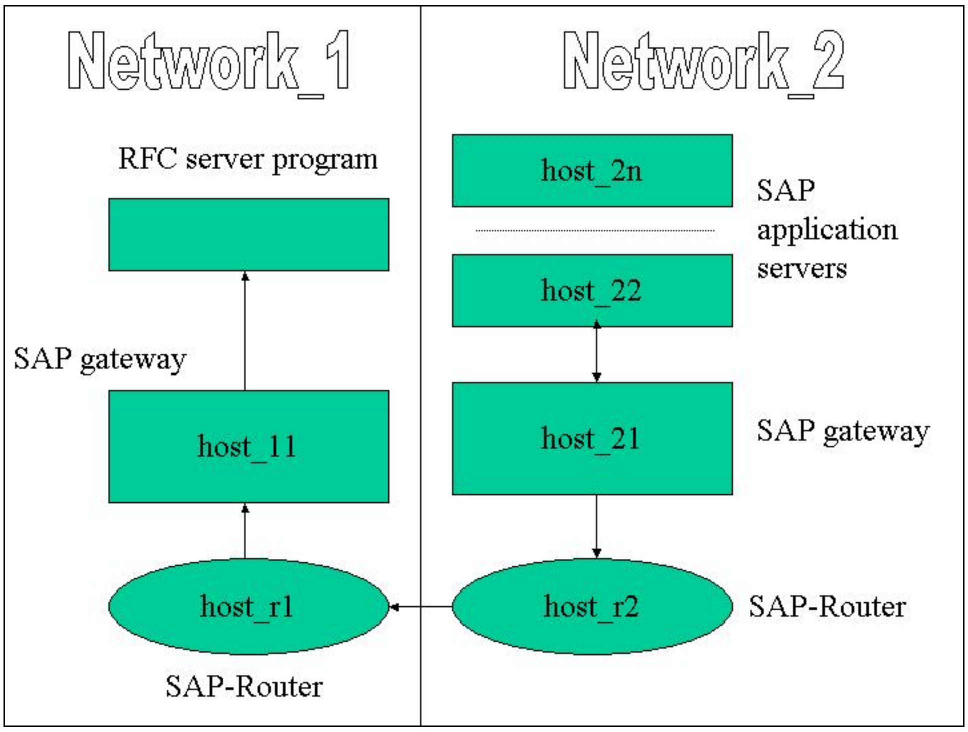


Figure 0-2

RFC server started by SAPGUI

Following example (see Figure 0-3) shows two different networks with two SAP-Routers and how current running SAPGUI can start an RFC server program.

A destination in SM59 should be defined as following:

Connection Type:	T
Activate Type:	Started by front end
Program:	../rfcsdk/srfcserv

The route Permission table on host_r1 in Network_1 must contain following entry:

P host_11 host_r2 3299

The route Permission table on host_r2 in Network_2 must contain following entries:

```
#entry for SAPGUI
P       host_r1   host_21       sapdp<R/3 system number>
#entry for RFC server program
P       host_r1   host_21       sapgw<R/3 system number>
```

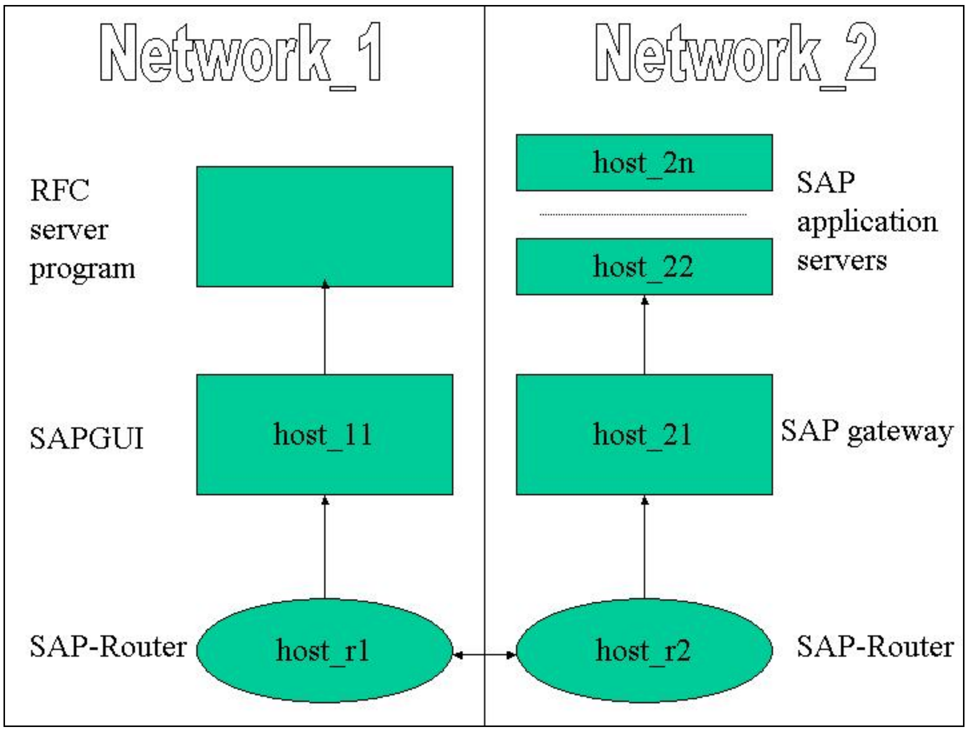


Figure 0-3

Used environment variables

The environment will be read at process start only. If you change any environment variable, you have to restart your RFC application.

Table 1: Most used environment variables

Name	Description	Default value
RFC_TRACE	Trace all RFC connections	0
RFC_TRACE_DIR	Defines location for RFC trace files.	Working directory
RFC_TRACE_DUMP	Full hex dump of the data at the API level (only active if RFC_TRACE is also set).	0
RFC_TRACE_ITAB	Trace the ITAB functions	0
RFC_MAX_TRACE	The maximal size (in megabytes) of trace files.	8 MB
RFC_DEBUG	Enter debugging mode for every R/3 connection	0
RFC_INI	Location of the **saprfc.ini** file.	Working directory.
RFC_NO_COMPRESS	The data compression for tables is off.	0
RFC_WAN_THRESHOLD	Threshold for wan connections. If an connection is established over WAN (flag in RfcOpenEx is set) all tables bigger that 251 bytes will be compressed for sending. This value increases the threshold.	251
RFC_MAX_REG_IDLE	The connection between an registered server, which waits for incoming calls in RfcDispatch or RfcWaitForRequest API, and the SAP Gateway will be checked in every 300 seconds by the RFC Library automatically. This variable enables to change this timeout.	300 seconds (5 Minutes)
RFC_SAFE_MODE	Lets RFC library call RfcHealthCheck API automatically. This will be done by almost every RFC call. This call enables to check memory consistence at runtime.	0
PATH_TO_CODEPAGE	Defines the location of mapping tables needed for codepage conversion.	

CPIC_MAX_CONV	Defines the maximum number of active RFC connections at a time	100
CPIC_TRACE	Activates CPIC trace. Possible values: 1-minimal information 2-medium 3-full information content	0
CPIC_TRACE_DIR	Defines the location of CPIC trace files	Working directory
RFC_TRACE_NO_HEX_DUMP	Defines maximal number of bytes, which will be dumped into rfc trace file. This may help to reduce the volume of the rfc trace file.	Full hex dump
RFC_LG_TIMEOUT	Defines the timeout for the connection to the message server in case of load balancing connections.	10 Seconds

Following environment variables are active by the RFC Library only on Microsoft Windows platforms:

RFC_LOGON_INI_PATH: Location of sapmsg.ini and saproute.ini files. Default is the Windows directory.