

RFC Programming in ABAP



Release 4.6B



Copyright

© Copyright 2000 SAP AG. All rights reserved.

No part of this brochure may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft®, WINDOWS®, NT®, EXCEL®, Word® and SQL Server® are registered trademarks of Microsoft Corporation.

IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.

ORACLE® is a registered trademark of ORACLE Corporation, California, USA.

INFORMIX®-OnLine for SAP and Informix® Dynamic Server™ are registered trademarks of Informix Software Incorporated.

UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of The Open Group.







HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Laboratory for Computer Science NE43-358, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

JAVA® is a registered trademark of Sun Microsystems, Inc. , 901 San Antonio Road, Palo Alto, CA 94303 USA.

JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, SAP Logo, mySAP.com, mySAP.com Marketplace, mySAP.com Workplace, mySAP.com Business Scenarios, mySAP.com Application Hosting, WebFlow, R/2, R/3, RIVA, ABAP, SAP Business Workflow, SAP EarlyWatch, SAP ArchiveLink, BAPI, SAPPHIRE, Management Cockpit, SEM, are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax
	Tip

Contents

RFC Programming in ABAP	5
RFC Basics	6
The RFC Interface	7
RFC in SAP Systems	8
Technical Requirements	11
Calling Remote Function Modules in ABAP	13
Introduction	14
Parameter Handling in Remote Calls	16
Calling Remote Functions Locally	17
Calling Remote Functions BACK	18
Using Transactional Remote Function Calls	19
Transactional Integrity of tRFCs	22
qRFC With Send Queue	24
qRFC With Send Queue: Overview	26
Programming Serialization	28
Using	30
Transaction Sequence and Queue Assignment	32
Tools	35
Using Asynchronous Remote Function Calls	38
Calling Requirements for Asynchronous RFCs	40
Receiving Results from an Asynchronous RFC	41
Keeping the Remote Context	44
Parallel Processing with Asynchronous RFC	45
Checking Authorizations for RFC	49
Using Pre-Defined Exceptions for RFC	50
Writing Remote Function Modules in ABAP	51
Steps for Implementing Remote Function Modules	52
Programming Guidelines	53
Debugging Remote Function Modules	54
Maintaining Remote Destinations	55
Displaying, Maintaining and Testing Destinations	56
Entering Destination Parameters	58
Types of Destinations	60
Maintaining Group Destinations	63
Maintaining Trust Relationships Between R/3 Systems	64

RFC Programming in ABAP

RFC Basics

RFC Basics

This section gives a brief overview of the Remote Function Call (RFC) within an SAP System, that is

- how the RFC Interface works
- the functionality that is provided by the RFC and
- it explains the technical requirements for RFC on R/2, R/3 and external systems on all currently supported platforms.

The following background topics are available:

[The RFC Interface \[Page 7\]](#)

[RFC in SAP Systems \[Page 8\]](#)

[Technical Requirements \[Page 11\]](#)

The RFC Interface

A remote function call is a call to a function module running in a system different from the caller's. The remote function can also be called from within the same system (as a remote call), but usually caller and callee will be in different systems.

In the SAP System, the ability to call remote functions is provided by the Remote Function Call interface system (RFC). RFC allows for remote calls between two SAP Systems (R/3 or R/2), or between an SAP System and a non-SAP System.

RFC consists of the following interfaces:

- A calling interface for ABAP programs

Any ABAP program can call a remote function using the CALL FUNCTION...DESTINATION statement. The DESTINATION parameter tells the SAP System that the called function runs in a system other than the caller's. RFC communication with the remote system happens as part of the CALL FUNCTION statement.

RFC functions running in an SAP System must be actual function modules, and must be registered in the SAP System as "remote".

When both caller and called program are ABAP programs, the RFC interface provides both partners to the communication. The caller may be any ABAP program, while the *called* program must be a function module registered as remote.

- The topic [Calling Remote Function Modules in ABAP \[Page 13\]](#) provides details on calling function modules registered as remote.
- The topic [Writing Remote Function Modules in ABAP \[Page 51\]](#) provides information on writing function modules that you want to call remotely.

- Calling interfaces for non-SAP programs

When either the caller or the called partner is a non-ABAP program, it must be programmed to play the other partner in an RFC communication.

To help implement RFC partner programs in non-SAP Systems, SAP provides

- [External Interfaces \[Ext.\]](#)

RFC-based and GUI-based interfaces can be used by external programs to call function modules in SAP R/2 or R/3 systems and execute them in these systems. Vice versa, ABAP programs in R/2 or R/3 can use the functions provided by external programs via these interfaces.



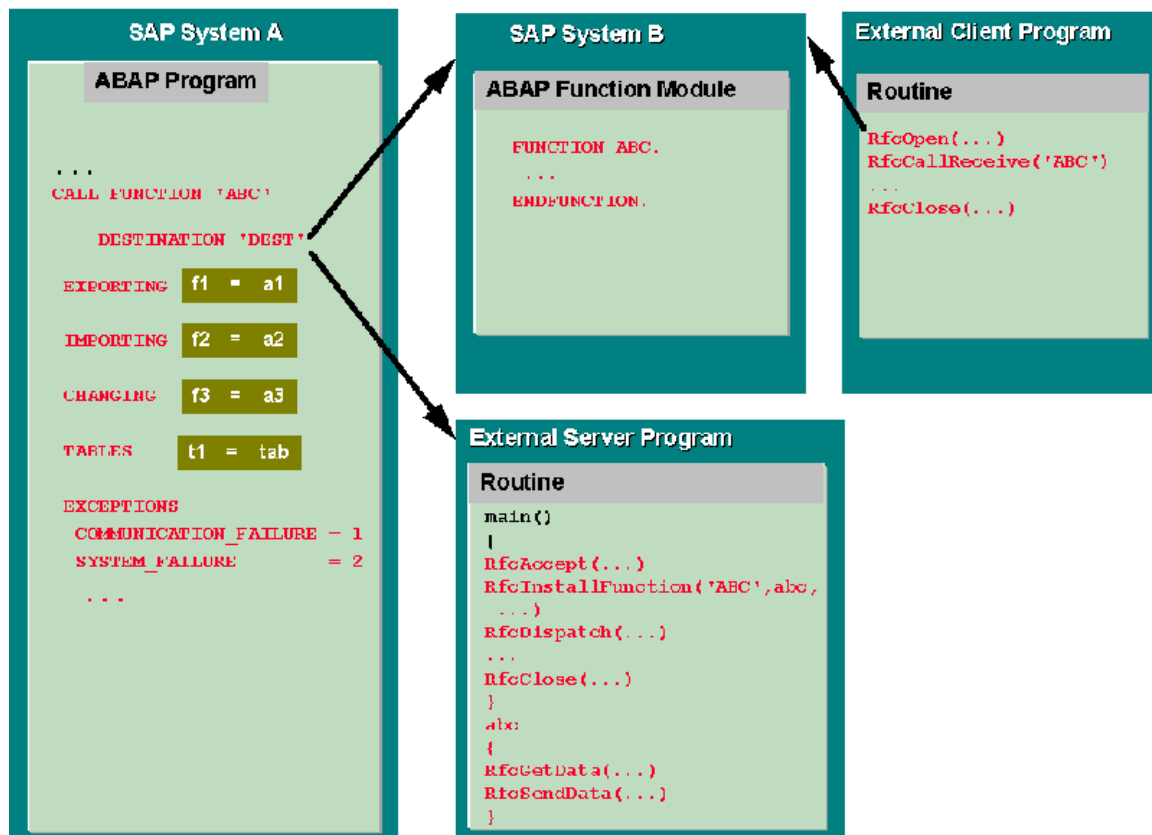
If you want to see some scenarios within a programming example, see the corresponding unit in the [Tutorial: Communication Interfaces \[Ext.\]](#).

RFC in SAP Systems

RFC in SAP Systems

In any R/3 System, **CALL FUNCTION** is an integral part of the ABAP language (in R/2 from Release 5.0 onwards). It is used to perform a function (function module) in the same system (R/2 or R/3).

REMOTE FUNCTION CALL (RFC) is an extension of CALL FUNCTION in a distributed environment. Existing function modules can be executed from within a remote system (R/2 or R/3) via an RFC call. This is done by adding a **DESTINATION** clause to the CALL FUNCTION statement:



The destination parameter displays an entry in the RFCDES table (which is defined with transaction sm59). This entry contains all necessary parameters to connect to and log in the destination system.

RFC can be used between SAP Systems (R/3 and R/2 Systems). With R/2 in an IBM environment, RFC is currently possible only with CICS as a DC system from Release 5.0D onwards. In IMS environment, RFC will probably not be available before IMS >= 4.1 and complete support of the LU6.2 protocol (via MVS/APPC).

The **RFC API** on OS/2, Windows, Windows NT and all R/3-based UNIX platforms makes it possible to use the RFC functionality between an SAP System (R/3 from Release 2.1 and R/2 from Release 5.0D onwards) and a C program on the above platforms. It is of no significance to the caller whether the remote function is provided in an SAP System or in a C program.

RFC frees the ABAP programmer from having to program his own communications routines. When you make an RFC call, the RFC interface takes care of:

- Converting all parameter data to the representation needed in the remote system. This includes character string conversions, and any hardware-dependent conversions needed (for example, integer, floating point). All ABAP data types are supported.



There is no support for Dictionary structures.

- Calling the communication routines needed to talk to the remote system.
- Handling communications errors, and notifying the caller, if desired. (The caller requests notification using the EXCEPTIONS parameter of the CALL FUNCTION statement.)

The RFC interface is effectively invisible to the ABAP programmer. Processing for *calling* remote programs is built into the CALL FUNCTION statement. Processing for *being called* is generated automatically (in the form of an RFC stub) for every function module registered as remote. This stub serves as an interface between the calling program and the function module.

A distinction is made between an **RFC client** and **RFC server**. **RFC client** is the instance that calls up the Remote Function Call to execute the function that is provided by an **RFC server**. In the following, the functions that can be executed remotely will be called **RFC functions** and the functions provided via RFC API will be called **RFC calls**.

All RFC functions available in a remote RFC server system, which are called by an RFC client, are processed **transactionally**. This means that after execution of the first RFC function in the RFC server system the complete context (all globally defined variables in the RFC server program or in the main program of a function module) is available for further RFC functions. The RFC connection is closed only

- when the context of the calling ABAP program has ended or
 - explicitly by *RfcAbort* or *RfcClose* in the external program.

Until Release 3.0 the connection to an R/3 System must be established to a previously defined application server. From Release 3.0 onwards, it is also possible to have an application server assigned by the message server on the basis of a load balancing procedure. This applies both for RFC between R/3 systems and between external systems and R/3 systems.

To make the execution of RFC functions reliable, safe and independent from the availability of the RFC server or RFC server system, the **transactional RFC (tRFC)** was introduced for R/3 systems from Release 3.0 onwards. This ensures that the called function module is executed **only once** in the RFC server system.

In transactional RFC calls, the data that belongs to an RFC function must first be stored temporarily on the SAP database in the RFC client system. When processing is completed, this must be reported back to the calling ABAP program. Everything else is handled by the tRFC component in the R/3 System.

Since a database is not always available on external systems, the link to the tRFC interfaces is implemented such that the client or server programs based on RFC API must take on some administrative functions to ensure that the respective function module is executed **“only once”**.

RFC in SAP Systems

In an R/3 System, other R/3 Systems can be defined as trusted systems. Trusted systems can access the called system (the trusting system) without having to provide a password.

For more information, see [Trusted System \[Page 64\]](#).

Technical Requirements

External Systems

systems must support TCP/IP.

• OS/2:	TCP/IP for OS/2 from IBM.
• Windows 3.1/3.11:	All TCP/IP products that support the socket interface.
• Windows NT/95:	Microsoft standard
• UNIX platforms:	Manufacturer's standard

The RFCSDK for the respective platforms contains the following libraries and include files:

• sa prf c. h	This include file contains all data types and structures required and the prototypes (declarations) of the RFC calls.	
• sa pit ab .h	This include file contains all the RFC calls required to manipulate internal tables	
• lib rfc	Depending on the platform, the following libraries are required:	
	OS/2:	librfc.dll and librfc.lib for Compile/Link
	Windows 3.1/3.11:	librfc16.dll , librfc2.dll , librfc3.dll , librfc4.dll
		and librfc5.dll and librfc16.lib for Compile/Link
	Windows NT/95:	librfc32.dll and librfc32.lib for Compile/Link
	UNIX-Platforms:	librfc.a

SAP R/3 Systems

For RFC between external systems and R/3, there are no specific requirements in the R/3 System, except that the R/3 System has to be Release >= 2.1.

Contrary to this, **RFC between SAP R/2 in an IBM environment and SAP R/3 or external systems** requires an SAP Gateway to run on a machine that supports the SNA LU6.2 protocol for the IBM host. The SNA product must also be installed on this machine, and the SAP gateway must be operable with this product. This is necessary, because some SNA products are not compatible on the same machines.

The following SNA products are currently supported:

- SNA services or SNA server on IBM-AIX systems

Technical Requirements

- SNAplusLink on HP-UX systems
- Communication Manager on OS/2
- SNA Server on WindowsNT systems
- SNALink SNA peer-to-peer 8.0 on SUN systems
- TRANSIT-SERVER and TRANSIT-CPIC on SNI-SINIX systems.

SAP R/2 Systems

- **IBM host (CICS):** Release 5.0D with the following components:
 - 082
Communication via Remote Function Call (RFC)
 - 153
SAP Intersystem Communication
 - 080
Host communication with DOS, OS/2
 - or 081
Host communication with other LU6.2 systems
- **IBM host (IMS):** Probably not before IMS >= 4.1 with MVS/APPC
- **SNI host:** Release 5.0D with the following components:
 - 082
Communication via Remote Function Call (RFC)
 - 153
SAP Intersystem Communication
 - 083
Host communication via TCP/IP (BS2000)

Calling Remote Function Modules in ABAP

This section contains the following topics:

[Introduction \[Page 14\]](#)

[Parameter Handling in Remote Calls \[Page 16\]](#)

[Calling Remote Functions Locally \[Page 17\]](#)

[Calling Remote Functions BACK \[Page 18\]](#)

[Using Transactional Remote Function Calls \[Page 19\]](#)

[Using Asynchronous Remote Function Calls \[Page 38\]](#)

[Checking Authorizations for RFC \[Page 49\]](#)

[Using Pre-Defined Exceptions for RFC \[Page 50\]](#)

Introduction

Introduction

You can use the CALL FUNCTION statement to call remote functions, just as you would call local function modules. However, you must include an additional DESTINATION clause to define where the function should run:

```
CALL FUNCTION RemoteFunction
```

```
DESTINATION Dest
```

EXPORTING	f1 =...
	f2 =...
IMPORTING	f3 =...
TABLES	t1 =...

```
EXCEPTIONS.....
```

The field *Dest* can be either a literal or a variable: its value is a logical destination (for example, "hw1071_53") known to the local SAP System. Logical destinations are defined in the RFCDES table (or the TRFCD table in R/2 Systems) via transaction sm59 or the following menu path:

Tools → *Administration, Administration* → *Network* → *RFC destinations*. You can also access logical destinations via the Implementation Guide (IMG) by choosing *Tools* → *Customizing* → *Enterprise IMG*. In the Implementation Guide, you can then choose *Cross-application components* → *ALE* → *Communication* → *Define RFC destination*.

The remote function call concept, for example, allows you to access a function module in an R/2 System from an ABAP program in an R/3 System. If you want to read a customer record from your R/2 System's database, create a remotely callable function module in the R/2 environment which retrieves customer records. Call this function from your R/3 System using a remote function call and listing the destination for the target R/2 System:

R/3 System: Client

```
CALL FUNCTION 'RFC_CUSTOMER_GET'
```

DESTINATION 'K50'	
EXPORTING	KUNNR = CUSTNO
TABLES	CUSTOMER_T = ITAB
EXCEPTIONS	NO_RECORD_FOUND = 01.

R/2 System: Server

```
FUNCTION RFC_CUSTOMER_GET.
```

```
.... (Read customer record)
```

```
ENDFUNCTION.
```

Programming guidelines are available in the following topics:

[Parameter Handling in Remote Calls \[Page 16\]](#)

[Calling Remote Functions Locally \[Page 17\]](#)

[Calling Remote Functions BACK \[Page 18\]](#)

[Using Transactional Remote Function Calls \[Page 19\]](#)

[Using Asynchronous Remote Function Calls \[Page 38\]](#)

[Using Pre-Defined Exceptions for RFC \[Page 50\]](#)

Parameter Handling in Remote Calls

When you make a remote function call, the system handles parameter transfer differently than it does with local calls.

TABLES parameters

The actual table is transferred, but not the table header. If a table parameter is not specified, an empty table is used in the called function.

The RFC uses a delta managing mechanism to minimize network load during parameter and result passing. Internal ABAP tables can be used as parameters for function module calls. In a local function module call, a parameter table is passed on by reference, and no new local copy has to be created. RFC does not support the "by reference" mechanism, so the whole table has to be exchanged between the RFC client and the RFC server. When the RFC server receives the table entries, it creates a local copy of the internal table. Then only delta information is returned to the RFC client. This information is not returned to the RFC client every time a table operation occurs, however; instead, all collected delta information is passed on at once when the function returns to the client.

The first time a table is passed, it is given an object-ID and registered as a "virtual global table" in the calling system. This registration is kept alive as long as call-backs are possible between calling and called systems. Thus, if multiple call-backs occur, the change-log can be passed back and forth to update the local copy, but the table itself need only be copied once (the first time).

Calling Remote Functions Locally

Sometimes you want to call a remote function (that is, a function registered as remote in the SAP System) from within the same system. This function can run either as a remote or a local call, depending on the CALL FUNCTION statement. Whether the call runs as remote or local affects parameter handling (as explained in [Parameter Handling in Remote Calls \[Page 16\]](#)).

The two options are:

- **CALL FUNCTION...DESTINATION = 'NONE'**

This is a remote call, even though **DESTINATION = 'NONE'** means that the remote function will run in the same system as the caller. As a remote call, the function module runs in its own roll area, and parameter values are handled as for other remote calls (described in [Parameter Handling in Remote Calls \[Page 16\]](#)).

CALL FUNCTION 'RFC_CUSTOMER_GET'

DESTINATION 'NONE'	
EXPORTING	KUNNR = CUSTNO
TABLES	CUSTOMER_T = ITAB
EXCEPTIONS	NO_RECORD_FOUND = 01.

- **CALL FUNCTION... [no DESTINATION used]**

This is a local call, even though the function module is registered as remote. The module does not run in a separate roll area, and is essentially like a normal function call. Parameter transfer is handled as for normal function modules. In particular, if the call leaves some EXPORTING parameters unspecified, it terminates abnormally.

CALL FUNCTION 'RFC_CUSTOMER_GET'

EXPORTING	KUNNR = CUSTNO
TABLES	CUSTOMER_T = ITAB
EXCEPTIONS	NO_RECORD_FOUND = 01.

You can also call a function for parallel processing within the same system. For details, see [Parallel Processing with Asynchronous RFC \[Page 45\]](#).

Calling Remote Functions BACK

Calling Remote Functions BACK

The client and the server are determined at the start of an RFC. While a function is being processed on the server, this server can call a function on the client. In other words, the remote function can invoke its own caller (if the caller is itself a function module), or any function module loaded with the caller. The called-back function then runs in the same program context as the original caller.

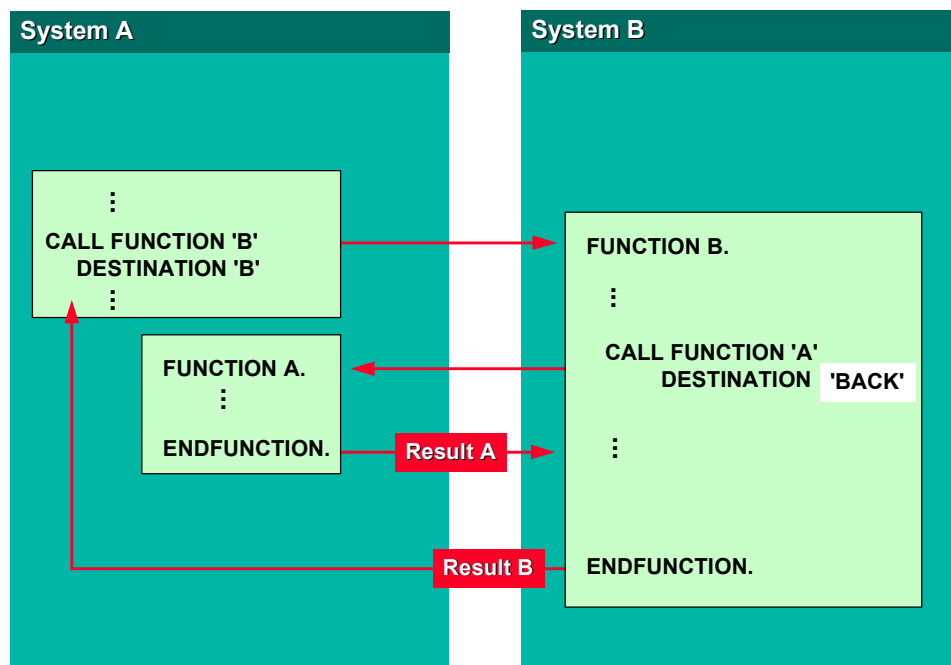
You can trigger this call-back mechanism by using the special destination name "BACK". If this name is specified in an RFC call on the system acting as the server, the system uses the same RFC connection that was established when the server received the first call. Once an RFC connection is established, it is maintained until it is either explicitly closed or until the calling program terminates. During a call-back, the system will always attempt to use existing RFC connections before establishing a new one.

To perform a call-back, the syntax is:

```
CALL FUNCTION... DESTINATION 'BACK'
```



In the diagram, remote function B of System B invokes remote function A in the calling System A.



Using Transactional Remote Function Calls

From Release 3.0 onwards, data can be transferred between two R/3 Systems **reliably** and **safely** via **transactional RFC (tRFC)**.



This type of RFC was renamed from **asynchronous** to **transactional** RFC, because asynchronous RFC has another meaning in R/3 Systems.

The called function module is executed **exactly once** in the RFC server system. The remote system need not be available at the time when the RFC client program is executing a tRFC. The tRFC component stores the called RFC function together with the corresponding data in the R/3 database, including a unique transaction identifier (TID).

If a call is sent, and the receiving system is down, the call remains in the local queue until a later time. The calling dialog program can proceed without waiting to see whether or not the remote call was successful. If the receiving system does not become active within a certain amount of time, the call is scheduled to run in batch.

Transactional RFCs use the suffix **IN BACKGROUND TASK**.

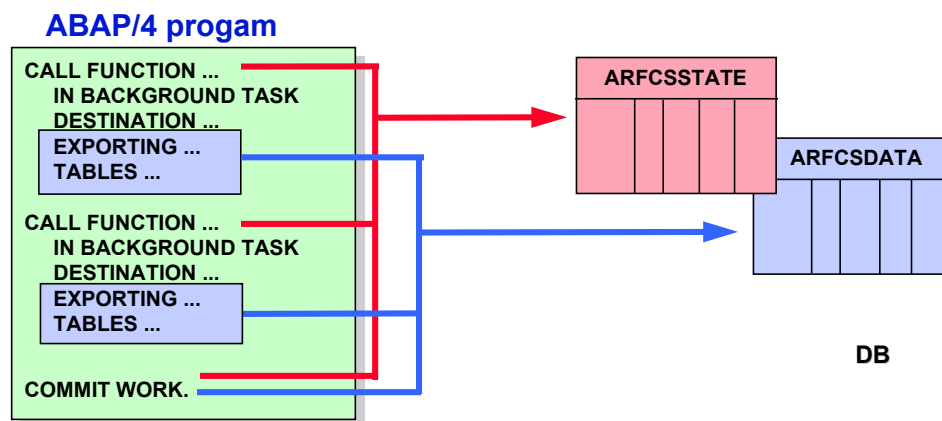
As with synchronous calls, the **DESTINATION** parameter defines a program context in the remote system. As a result, if you call a function repeatedly (or different functions once) at the same destination, the global data for the called functions may be accessed within the same context.

The system logs the remote call request in the database tables ARFCSSTATE and ARFCSDATA with all of its parameter values. You can display the log file using transaction SM58. When the calling program reaches a COMMIT WORK, the remote call is forwarded to the requested system for execution.

All tRFCs with a single destination that occur between one COMMIT WORK and the next belong to a single logical unit of work (LUW). For more information on LUWs, TIDs and on checking the status of transactional calls, see [Transactional Integrity of tRFCs \[Page 22\]](#).

Transactional RFC requests are transferred, with parameter data in byte-stream form, using TCP/IP or X400.

Using Transactional Remote Function Calls

Transaction
SM58

Transactional RFC		Log file example			
Caller	Function Module	Target system	Date	Time	Status text
SMITH	GET_DATA	P30	07.01.97	14:00	--

As an example, you can use transactional RFCs for specific types of update procedures. Some complex dialogs require that several related database tables be updated during different phases within a transaction. If the update functions needed are located on a remote machine, and if it is not essential that the table changes be carried out immediately before continuing the dialog, you can use transactional RFC calls. Instead of having to wait for each separate update procedure to be completed, the user can proceed to the end of the transaction without delay. Transactional RFC processing ensures that all the planned updates are carried out when the program reaches the COMMIT WORK statement.



As with synchronous remote calls, the EXPORTING and TABLES parameters need not be specified for transactional remote function calls.



You may **not** call remote functions with the transactional mechanism if EXPORT parameters are specified in the functions interface. An IMPORTING parameter in your CALL FUNCTION statement results in a compiler error.

Note also that you cannot make asynchronous calls to functions that perform call-backs.

When the Remote System is Unavailable

If the remote system is unavailable, the SAP System schedules the report RSARFCSE for background processing with the relevant transaction ID as variant. This report, which forwards

Using Transactional Remote Function Calls

asynchronous calls for execution, is called repeatedly until it succeeds in connecting with the desired system.

When scheduled in batch, RSARFCSE runs automatically at set intervals (the default is every fifteen minutes, for up to 30 attempts). You can customize this interval and the length of time the program should go on trying. To do this, use the extension programs SABP0000 and SABP0003 (or see the SAP Extension Concept and CALL CUSTOMER-FUNCTION).

In transaction SM59 (menu path: *Tools → Administration, Administration → Network → RFC destinations*) you can select *Destination→TRFC options* which enables you to configure each destination. Thus you can determine the number of connection attempts up to the task and the time between repeat attempts.

If the system is not reachable within the specified amount of time, the system stops calling RSARFCSE, and the status CPICERR is written to the ARFCSDATA table. Within another specified time (the default is eight days), the corresponding entry in the ARFCSSTATE table is deleted (this limit can also be customized). (It is still possible to start such entries in transaction SM59 manually.)

Transactional Integrity of tRFCs

Transactional Integrity of tRFCs

You can execute function modules in background tasks in another R/3 System or an external program. When you call function modules in this way, they are not executed at once, but wait until a **COMMIT WORK** is triggered.

Transactional RFCs receive their name from the fact that the associated remote function call mechanism guarantees transactional integrity for all calls made with the **IN BACKGROUND TASK** suffix. As with database updates, LUW's (logical units of work) are created for calls that are scheduled to run in background tasks. All tRFCs with a single destination that occur between one **COMMIT WORK** and the next belong to a single LUW. Within a given LUW, all calls:

- execute in the order they were called
- run in the same program context in the target system
- run as a single transaction: they are either committed or rolled back as a unit.

LUW's are identified by transaction ID's that are unique world-wide. The transaction ID can be determined from an ABAP program by calling function module **ID_OF_BACKGROUNDTASK**. (You must call this function after the first asynchronous **CALL**, and before the related **COMMIT WORK**.)

Because the RFC is like a transaction, database operations are either all executed or, if a function module terminates, all rolled back. If an LUW runs successfully, you cannot execute it again. In some cases, it may be necessary to program the roll back of an LUW, (for example, because a table is locked). To do this, you call the function module **RESTART_OF_BACKGROUNDTASK** which performs a rollback and ensures that the LUW is executed again later.

Normally, the LUW is executed immediately after **COMMIT WORK** in the specified target system. However, if you want it to start at a particular time, you can set a start time with the function module **START_OF_BACKGROUNDTASK** which must also be called within the LUW, i.e. after the first **CALL... IN BACKGROUND TASK** and before **COMMIT WORK**.



For more information on the transactional RFC, see the online help in the ABAP Editor.

Checking the Status of Transactional Calls

All transactional remote function calls are stored in the tables **ARFCSSTATE** and **ARFCSDATA** and each LUW is identified by a unique ID. When a **COMMIT WORK** occurs, the calls attached to this ID are executed in the relevant target system. The system function module **ARFC_DEST_SHIP** transports the data to the target system and the function module **ARFC_EXECUTE** executes the stored function calls. If an error or an exception occurs during one of the calls, all the database operations started by the preceding calls are rolled back and an appropriate error message is written to the file **ARFCSSTATE**.

There are two methods for checking on the status of a transaction ID:

- **From an ABAP program**

The function module **ID_OF_BACKGROUNDTASK** returns the ID of the LUW. You call this module after the first **CALL... IN BACKGROUND TASK** and before **COMMIT WORK**.

Transactional Integrity of tRFCs

```
CALL FUNCTION 'ID_OF_BACKGROUNDTASK' IMPORTING      TASK-ID = TID.
```

Once you have identified the ID of the LUW, you can use the function module `STATUS_OF_BACKGROUNDTASK` to determine the status of the transactional RFC.

```
CALL FUNCTION 'STATUS_OF_BACKGROUNDTASK'
```

EXPORTING TID	= TASK-ID
IMPORTING ERRORTAB	= ERTAB
EXCEPTIONS COMMUNICATION	= 01
(Connection not available: will try again later)	
RECORDED	= 02
(ARFC is scheduled)	
ROLLBACK	= 03
(Rollback triggered in target system)	

- **Online**

Call transaction SM58 (*Tools → Administration → Monitoring → Transactional RFC*). This tool lists only those transactional RFCs that could not be carried out successfully or that had to be planned as batch jobs. The list includes the LUW ID and an error message. Error messages displayed in SM58 are taken from the target system. To display the text of the message, double-click on the message.

Transaction SM58 also lets you control your transactional RFC at various stages. If the call ends abnormally during the sending process, you may need to use the *Rollback LUW* function to manually rollback the LUW before attempting a resend. If the target system was unavailable, you can use the *Backgr.job* function to display the batch job created for your call. *Execute funct. module* lets you restart the call after the occurrence of a temporary error (such as a syntax error).

If a LUW runs successfully in the target system, the function module `ARFC_DEST_CONFIRM` is triggered and confirms the successful execution in the target system. Finally, the entries in the Tables `ARFCSTATE` and `ARFCSDATA` are deleted.

RFC API

You can also execute programs asynchronously in 'C'-implemented function modules (connection type TCP/IP in transaction SM59, see [Types of Destinations \[Page 60\]](#)). Implementation of the function modules occurs as usual in connection with the RFC API. This contains the function modules `ARFC_DEST_SHIP` and `ARFC_DEST_CONFIRM` which call the appropriate functions.

For more information on this topic, refer to [The RFC API \[Ext.\]](#).

qRFC With Send Queue

qRFC With Send Queue

To guarantee an LUW sequence dictated by the application, the tRFC is serialized using queues. It is therefore called **queued RFC (qRFC)**. Due to the serialization, in R/3 a send queue for tRFC was created. This results in the general term **qRFC with send queue**.

In addition, there are applications that want to determine themselves the exact moment in which to process the LUW in the target system. These are usually applications on an external (non SAP) system that does not have a kind of send queue. **qRFC with recipient queue** including an appropriate enhancement of the RFC library is currently under development (scheduled for delivery in the next release).

Motivation

What Does tRFC Perform Right Now?

- tRFC guarantees that a called function module is executed in the target system **exactly once**.
- All tRFC calls terminated with the statement COMMIT WORK belong to one LUW (Logical Unit of Work). Each LUW automatically receives a transaction ID.
- Within an LUW, all function modules are executed in the target system in the sequence in which they are called in the send system.
- tRFC calls with the addition **AS SEPARATE UNIT** map a separate LUW each in the target system. Such a **sub LUW** contains exactly one tRFC call and is processed independent of the actual (superior) LUW. This sub LUW receives its own transaction ID.
- If within an LUW several destinations are used, all tRFC calls that belong to one destination also form a sub LUW. However, since this bundling happens only internally, there is **no** separate transaction ID assigned to the sub LUW.

Disadvantages of tRFC

tRFC processes all LUWs independent of one another. Due to the amount of activated tRFC processes, this procedure can reduce performance significantly in both the send and the target systems.

In addition, the **sequence of LUWs** defined in the application cannot be kept. Therefore, there is no guarantee that the transactions are executed in the sequence dictated by the application. The only guarantee is that all LUWs are transferred sooner or later.

Use and Availability

The typical applications for which the serialization of tRFC was implemented use distributed environments:

- **ALE** (Application Link Enabling)
- **APO** (Advanced Planner and Optimizer)
- **SFA** (Sales Force Automation)

qRFC with send queue is released with **Release 4.5B**.



qRFC With Send Queue

As a special feature for APO, the serialization is provided as a special release for APO systems of Release 3.1H or 4.5A, respectively.

Other Topics

[qRFC With Send Queue: Overview \[Page 26\]](#)

[Programming Serialization \[Page 28\]](#)

[Using "Mixed Mode" \[Page 30\]](#)

[Transaction Sequence and Queue Assignment \[Page 32\]](#)

[Tools \[Page 35\]](#)

qRFC With Send Queue: Overview

This topic assumes that the processing sequence of the transactions is defined in the application program and that more than one queue can be used for one transaction.

Characteristics

The characteristics of qRFC with send queue are:

- Queued RFC with send queue enforces a serialization on the side of the send system. The target system has no information about the serialization in the send system. This allows communication with any R/3 target system as of Release 3.0.
- qRFC with send queue is an **enhancement of tRFC**. It transfers an LUW (transaction) only if it has no predecessors (in reference to the sequence defined in different application programs) in the participating queues. In addition, after executing a qRFC transaction, the system tries to start all waiting qRFC transactions automatically according to the sequence.
- For queue administration, the system needs a queue name and a queue counter for each qRFC transaction. Each tRFC call to be serialized is assigned to a queue name that can be determined by the application. The application passes the queue name with the call of function module **TRFC_SET_QUEUE_NAME**. This function module is called immediately before each tRFC call to be serialized. See also [Programming Serialization \[Page 28\]](#). A queue name is a text of up to 24 bytes length. You can choose any text, but you must not use * (asterisk).



You should start queue names with your application initials to prevent different applications from using the same names unintentionally (for example, SFA_... or APO_...).

- There is no queue configuration, since this is a **logical queue**. There are no separate tables for the individual queues. For an R/3 system, there is only one table for the send queues. All entries for queues of all participating LUWs are stored in the queue table **TRFCQOUT**.
- If the tRFC calls to be serialized are distributed to **several destinations**, the system bundles them per destination and processes them independent of one another. This results in sub LUWs within one LUW, as known from the old tRFC. In this case, there is no guarantee that the sequence of the sub LUWs is kept; the only guarantee is that all sub LUWs are processed sooner or later. In contrast to the old tRFC, each sub LUW receives its own transaction ID for easier queue handling.
- The sequence of the tRFC calls to be serialized with or without the option **AS SEPARATE UNIT** within one LUW is guaranteed, even if the calls are assigned to different queues.
- When sending the **COMMIT WORK**, the system determines a counter that acts as criterion for the sequence of LUW processing in a queue. For two LUWs that are interdependent from the application's point of view, a COMMIT WORK for the second LUW can be executed only after the COMMIT WORK for the first LUW has been terminated.
- **"Mixed mode"** is supported: One LUW can contain normal tRFC calls as well as tRFC calls to be serialized, with or without the addition AS SEPARATE UNIT. The sequence of the tRFC calls is guaranteed.

qRFC With Send Queue: Overview

- The "normal" tRFC calls within an LUW in "mixed mode" and the tRFC calls to be serialized that share the same destination form a sub LUW in reference to their assignment. If within one LUW no qRFC call was defined for a certain destination, the "normal" tRFC calls form a sub LUW by themselves.



In the old tRFC, all calls with different destinations are assigned to a common transaction ID, but internally to different sub LUWs. For the new tRFC, however, each bundle of tRFC/qRFC calls with the same destination receives its own transaction ID for better queue handling. For compatibility reasons, an application must first of all call function module **TRFC_QUEUE_INITIALIZE** if it uses "mixed mode" within an LUW and if the first call is a tRFC call. See also [Using "Mixed Mode" \[Page 30\]](#).

- The "normal separate" tRFC calls (with addition AS SEPARATE UNIT) within an LUW in "mixed mode" are treated as before (individual transaction ID).
- The "separate" tRFC calls to be serialized are processed depending on the queue, but independent of the actual LUW.
- A side effect and advantage of qRFC with send queue compared to the old tRFC is the minimum load the serialization bears on send and target systems (due to low parallelizing). However, choosing queue names carelessly (using too many names) can neutralize this advantage.
This is a classical dilemma: Better performance due to more serialization versus faster processing due to more parallelizing. Therefore, each application must carefully weigh up how to define queue names usefully and how to assign them to the individual function calls.

Which Problems Can Occur?

Note the following general serialization problems:

If, for example, due to network/communication problems the first LUW in a queue cannot be executed, not only this queue keeps pending, but also all other LUWs that are interdependent with this queue. The resulting "jam" could cause a database problem. However, as soon as the communication problem is solved, all transactions and thus all queues can be processed automatically one after the other.

Programming Serialization

Programming Serialization

The procedure below describes how to program the queued RFC with send queue in its basic form (that is, without the addition AS SEPARATE UNIT and without "mixed mode").

Prerequisites

- The R/3 send system has Release 4.5B.
- All required destinations are maintained in table RFCDEST of the send system.
- To achieve a good performance of the application, determine the number and the assignment of the queue names carefully.

Procedure

To implement the qRFC with send queue, proceed as follows:

1. Assign queue names for the subsequent function call(s).
2. To pass the queue name, call function module TRFC_SET_QUEUE_NAME immediately before each function call.
3. Call the function module using CALL FUNCTION ... IN BACKGROUND TASK.
4. Repeat steps 2 and 3 (and 1 if required) for all other function calls.
5. Close the LUW using COMMIT WORK.
6. Repeat steps 1 to 5 for all other LUWs of the application.



```
REPORT ... .

...

DATA Q_NAME LIKE TRFCQOUT-QNAME.

Q_NAME = 'BASIS_TEST_Q1'.
CALL FUNCTION ' TRFC_SET_QUEUE_NAME '
      EXPORTING
          QNAME = Q_NAME.
CALL FUNCTION 'RFC_FUNCTION_1'
      IN BACKGROUND TASK
      DESTINATION 'DEST'
      EXPORTING ...
      TABLES ... .

...

Q_NAME = 'BASIS_TEST_QM'.
CALL FUNCTION ' TRFC_SET_QUEUE_NAME '

```

```

EXPORTING
    QNAME = Q_NAME.
CALL FUNCTION 'RFC_FUNCTION_N'
    IN BACKGROUND TASK
    DESTINATION 'DEST'
    EXPORTING ...
    TABLES ... .
COMMIT WORK.
* NEXT LUW
...

```

Result

The system stores the queue specifications in table TRFCQOUT.

With each COMMIT WORK, the system determines a counter that is used to keep the sequence of transaction processing.



As an introduction, execute the simple demo program **RSTRFCT0**.

Using "Mixed Mode"

Using "Mixed Mode"

The special feature of "mixed mode" is that qRFC calls with send queue and normal tRFC calls are processed together within one LUW. For more information characterizing "mixed mode", see [qRFC With Send Queue: Overview \[Page 26\]](#).

Prerequisites

See [Programming Serialization \[Page 28\]](#).

Procedure

When implementing "mixed mode" within an LUW, note the following:

If within an LUW in "mixed mode" the first call is a normal tRFC call, you must first of all call function module **TRFC_QUEUE_INITIALIZE**. This assures that the current LUW is processed via qRFC with send queue.

If within an LUW in "mixed mode" the first call is a qRFC call, this initialization is done by function module **TRFC_SET_QUEUE_NAME**.



This program sequence is an example of how to implement an LUW in "mixed mode":

```
REPORT ... .

...

DATA Q_NAME LIKE TRFCQOUT-QNAME.

CALL FUNCTION 'TRFC_QUEUE_INITIALIZE'.

CALL FUNCTION 'RFC_FUNCTION_1'           " tRFC call -> NO QUEUE
      IN BACKGROUND TASK
      DESTINATION 'DEST'
      EXPORTING ...
      TABLES ... .

...

Q_NAME = 'BASIS_TEST_Q1'.

CALL FUNCTION 'TRFC_SET_QUEUE_NAME'
      EXPORTING
          QNAME = Q_NAME.

CALL FUNCTION 'RFC_FUNCTION_2'           " qRFC call -> Q1
      IN BACKGROUND TASK
      DESTINATION 'DEST'
      EXPORTING ...
      TABLES ... .
```

```

...

CALL FUNCTION 'RFC_FUNCTION_N-1'          "trRFC call -> NO QUEUE
      IN BACKGROUND TASK
      DESTINATION 'DEST'
      EXPORTING ...
      TABLES ... .
Q_NAME = 'BASIS_TEST_QM'.
CALL FUNCTION 'TRFC_SET_QUEUE_NAME'
      EXPORTING
          QNAME = Q_NAME.
CALL FUNCTION 'RFC_FUNCTION_N'          " qRFC call -> QM
      IN BACKGROUND TASK
      DESTINATION 'DEST'
      EXPORTING ...
      TABLES ... .

...

COMMIT WORK.

```

Result

Because of the transaction characteristics, the normal tRFC calls and the qRFC calls with the same destination form a unit. If, however, within the current LUW there are no qRFC calls for a destination, the system bundles the normal tRFC calls into a sub LUW.

The normal tRFC calls with the addition AS SEPARATE UNIT map a separate LUW in the target system.

qRFC calls with the addition AS SEPARATE UNIT are processed depending on the queue, but independent of the actual LUW.



To start programming in "mixed mode", refer to the demo program **RSTRFCT1**.

Transaction Sequence and Queue Assignment

Transaction Sequence and Queue Assignment

The sample scenario below is designed to show the dependencies between the send sequence of the LUWs and the queue assignment of the individual calls. An example demonstrates which queue assignment exists after a certain transaction has been sent. Try demo program **RSTRFCT3** as well.

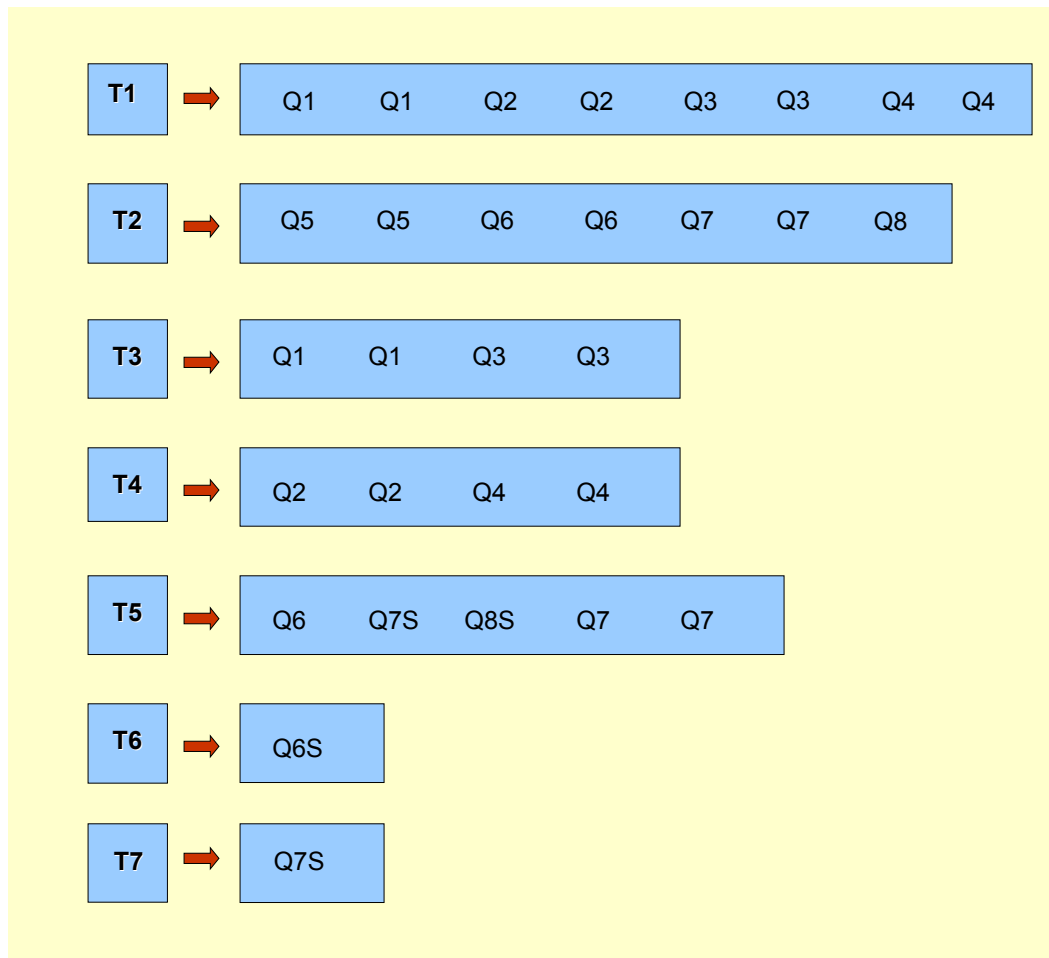
Sample Scenario

Queue Usage

The diagram shows the transaction sequence as dictated by the application as well as the respective queue usage of the individual calls:

For example, transaction **T3** contains two qRFC calls assigned to queue **Q1** and two calls assigned to queue **Q3**.

In contrast, transaction **T6** contains only the qRFC call with the addition **AS SEPARATE UNIT**, assigned to queue **Q6**.

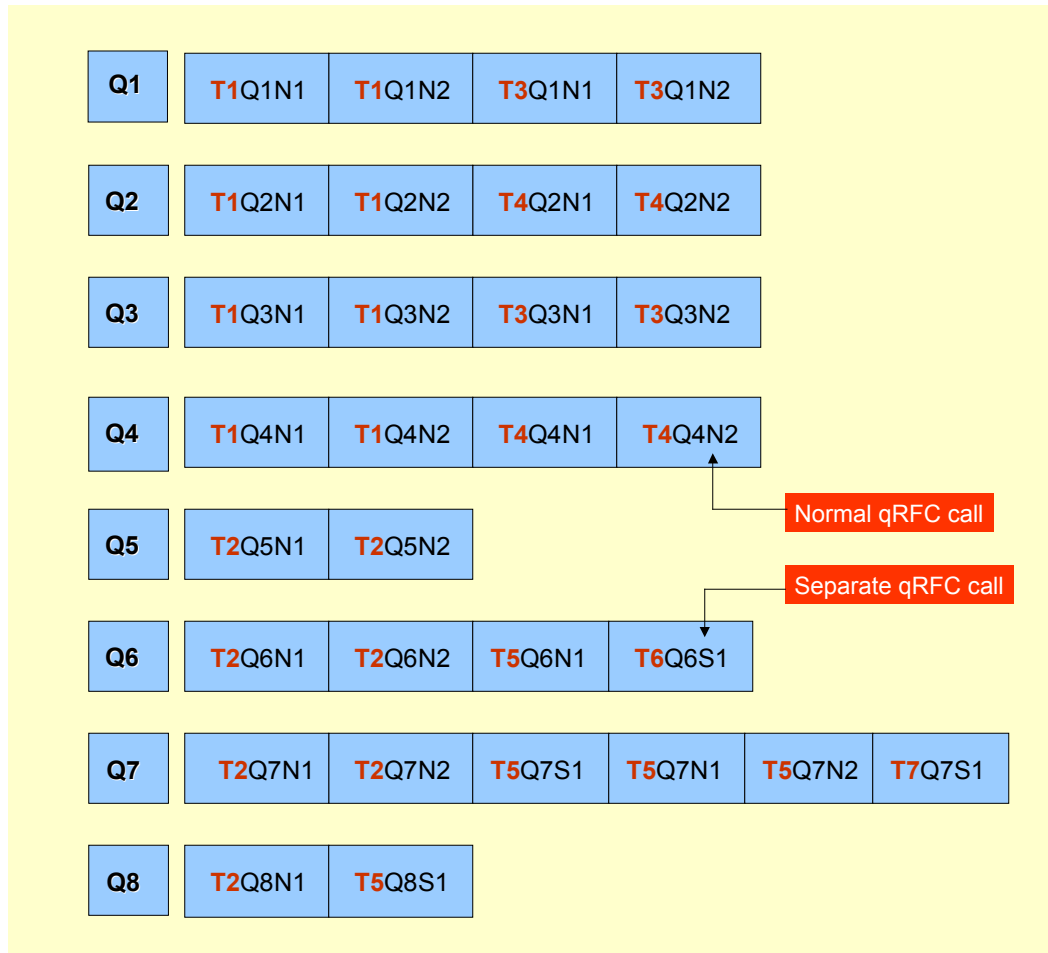


Transaction Sequence and Queue Assignment

Assignment of the Queues

Before Sending

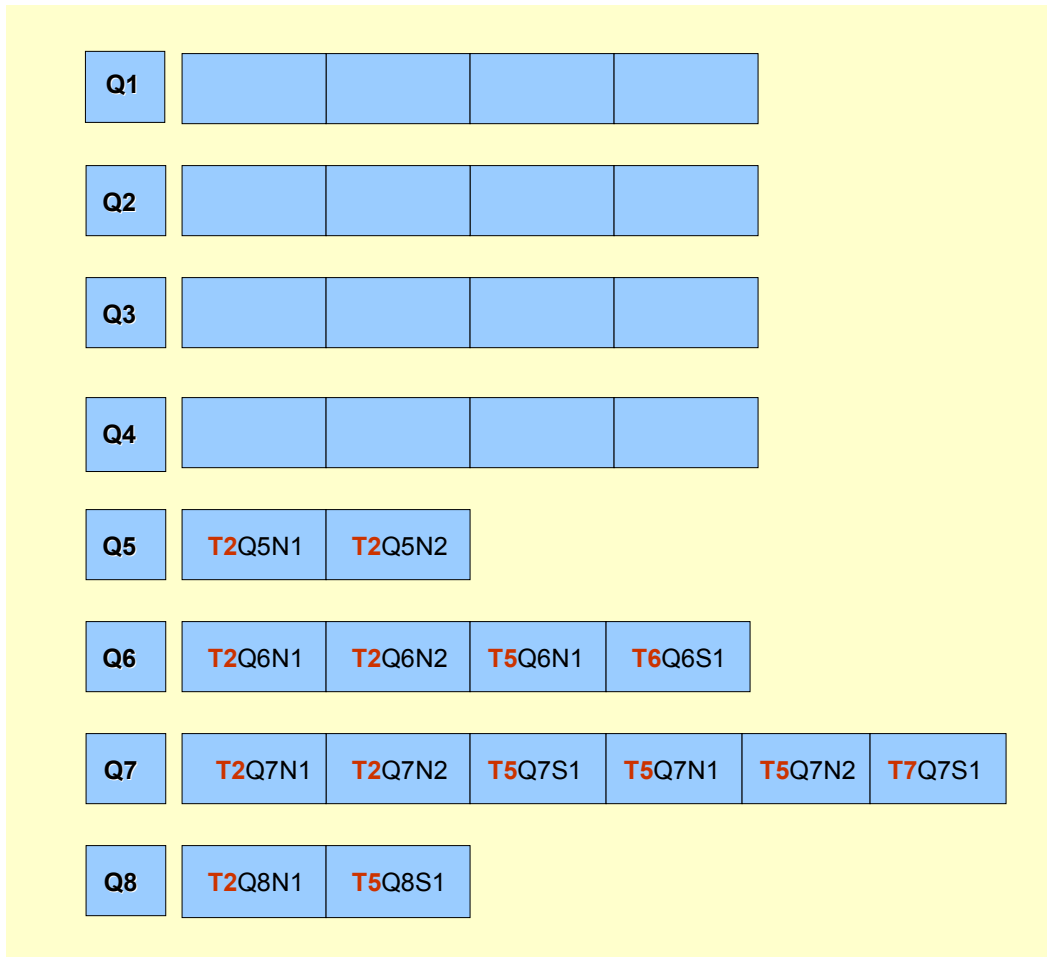
From the queue usage shown in the above example the following initial assignment for the individual queues results:



After Sending T1

After transaction T1 was sent successfully, the transactions T3 and T4 are prerared for sending. This results in the following queue assignment:

Transaction Sequence and Queue Assignment



First, the calls of T1 are sent in all involved queues (Q1, Q2, Q3, and Q4).

All other independent transactions that share queue participants with T1 are prepared for sending afterwards. In this case, these are transactions T3 and T4.

The transactions T2, T5, T6, and T7 remain in wait mode until T3 and T4 have been sent successfully.

Only after this the waiting transactions are started automatically one after another.

Tools

Tools

Passing Queue Names

TRFC_SET_QUEUE_NAME

This function module passes a queue name for only one function call immediately before a "call function ... in background task" (tRFC call) and indicates that the current LUW must be processed via tRFC with send queue. The queue name passed is valid only for the subsequent tRFC call.

Reading Queue Entries

TRFC_GET_QUEUE_INFO and TRFC_GET_QUEUE_INFO_DETAILS

Use these function modules to read the current contents of one or all queues of table TRFCQUOT in a more or less detailed form.

RSTRFCQR

This ABAP program uses the two function modules mentioned above to display the current contents of the send queue table TRFCQUOT.

RSTRFCQD

This ABAP program deletes all entries of single or all transactions. Use this program for test purposes or in emergencies only. If you delete the entries of a transaction, this transaction is no longer serialized. You may have to start it manually again.

Initializing in "Mixed Mode"

TRFC_QUEUE_INITIALIZE

Calling this parameter-free function module at the beginning of a qRFC LUW in "mixed mode" (using tRFC as well as qRFC) indicates that the current LUW must be processed via qRFC with send queue. However, this is necessary only if the first call is a normal tRFC call. If the first call is a qRFC call, the initialization is done by the function module TRFC_SET_QUEUE_NAME. See also demo program **RSTRFCT1**.

SM58

Transaction **SM58**, which you used as tRFC monitor for displaying and editing the tRFC transactions, now also allows processing qRFC transactions. If you delete a tRFC entry with this transaction, the system automatically deletes the corresponding entries in the send queue table, if they exist. If you start an LUW, the system does not immediately transfer this LUW but first checks whether this LUW needs serialization or whether it must wait due to predecessors in the queue.

Tools

Stopping Queues, Continuing Them, and Querying Their Statuses

You can stop and continue processing of one or more queues (using generic specification). In addition, you can query the status of a queue.

Use the following function modules:

TRFC_QOUT_STOP

Specify a queue name (single or generic, such as BASIS_TEST_*) and a destination to stop processing one or more queues. Depending on the FORCE parameter, you can stop processing at once (parameter FORCE = 'X') or process all requests that exist in the queue at the moment the STOP occurs and stop afterwards (FORCE = ' '). You can also stop an empty queue. In this case, all transactions assigned to this queue are not sent immediately. See also demo program **RSTRFCQ1**.

TRFC_QOUT_STATE

Specify the name of a queue and a destination to query the status of this queue. See also demo program **RSTRFCQ2**.

A queue can have one of these statuses:

- **RUNNING**
- **CPICERR**
- **SYSFAIL**
- **STOPPING**
- **STOP**
- **WAITSTOP**

Status **SYSFAIL** results from a serious error and is triggered by an exception in the R/3 kernel of the target system or in the called function module. In this status the queue remains. There is no automatic repetition. You can use **SM58** to send this LUW again or to delete it. If you delete it, the system automatically deletes the corresponding entries in the queue table TRFCQOUT. In status **CPICERR** an automatic repetition depends on the configuration of the destination in SM59 (by default set to "Yes").

TRFC_QOUT_RESTART

Specify a queue name (single or generic, such as BASIS_TEST_*) and a destination to continue processing one or more queues, regardless of whether the queues were stopped before by a queue stop function module (parameter FORCE) or by statuses CPICERR or SYSSFAIL. See also demo program **RSTRFCQ3**.

TRFC_QOUT_RESTART_COND

Specify a queue name (single or generic, such as BASIS_TEST_*) and a destination to continue processing one or more queues depending on whether processing of one or more other queues is terminated. Calling this function automatically stops the queues to be continued at this moment, if no other STOP call (depending on how parameter FORCE is set) was issued before. See also demo program **RSTRFCQ4**.

Sample Programs

- **RSTRFCT0**, **RSTRFCT1**, **RSTRFCT2**, and **RSTRFCT3** are sample or test programs for qRFC with send queue.
- With programs **RSTRFCQ1**, **RSTRFCQ2**, **RSTRFCQ3**, and **RSTRFCQ4** you can stop processing one or more queues, continue processing with or without condition, or query the status of a queue.

Using Asynchronous Remote Function Calls

Using Asynchronous Remote Function Calls

Asynchronous remote function calls (aRFCs) are similar to transactional RFCs, in that the user does not have to wait for their completion before continuing the calling dialog. There are three characteristics, however, that distinguish asynchronous RFCs from transactional RFCs:

- When the caller starts an asynchronous RFC, the called server must be available to accept the request.

The parameters of asynchronous RFCs are not logged to the database, but sent directly to the server.

- Asynchronous RFCs allow the user to carry on an interactive dialog with the remote system.
- The calling program can receive results from the asynchronous RFC.

You can use asynchronous remote function calls whenever you need to establish communication with a remote system, but do not want to wait for the function's result before continuing processing. Asynchronous RFCs can also be sent to the same system. In this case, the system opens a new session (or window) and allows you to switch back and forth between the calling dialog and the called session.

To start a remote function call asynchronously, use the following syntax:

```
CALL FUNCTION RemoteFunction STARTING NEW TASK taskname
```

```
Destination ...
```

```
EXPORTING...
```

```
TABLES ...
```

```
EXCEPTIONS...
```



The following calling parameters are available:

- TABLES
passes references to internal tables. All table parameters of the function module must contain values.
- EXPORTING
passes values of fields and field strings from the calling program to the function module. In the function module, the corresponding formal parameters are defined as import parameters.
- EXCEPTIONS
see [Using Pre-Defined Exceptions for RFC \[Page 50\]](#)

RECEIVE RESULTS FROM FUNCTION func is used **within** a FORM routine to receive the results of an asynchronous remote function call. The following receiving parameters are available:

- IMPORTING
- TABLES

Using Asynchronous Remote Function Calls

- EXCEPTIONS

The addition **KEEPING TASK** prevents an asynchronous connection from being closed after receiving the results of the processing. The relevant remote context (roll area) is kept for re-use until the caller terminates the connection.



Call a transaction asynchronously and display it in an amodal window:

```
DATA: MSG_TEXT(80) TYPE C.          "Message text
...
* Asynchronous call to transaction SM59 ->
* Create a new session
CALL FUNCTION 'ABAP4_CALL_TRANSACTION' STARTING NEW TASK 'TEST'
  DESTINATION 'NONE'
  EXPORTING
    TCODE = 'SM59'
  EXCEPTIONS
    COMMUNICATION_FAILURE = 1 MESSAGE MSG_TEXT
    SYSTEM_FAILURE        = 2 MESSAGE MSG_TEXT
  IF SY-SUBRC NE 0.
    WRITE: MSG_TEXT.
  ELSE.
    WRITE: 'O.K.'
  ENDIF.
```

Details are explained in the following topics:

[Calling Requirements for Asynchronous RFCs \[Page 40\]](#)

[Receiving Results from an Asynchronous RFC \[Page 41\]](#)

[Keeping the Remote Context \[Page 44\]](#)

[Parallel Processing with Asynchronous RFC \[Page 45\]](#)

Calling Requirements for Asynchronous RFCs

Calling Requirements for Asynchronous RFCs

When you call a remote function with the optional suffix **STARTING NEW TASK**, the system starts the function in a new session. Rather than waiting for the remote call to be completed, the user can resume processing as soon as the function module has been started in the target system.

The remotely called function module can, for example, display a new screen using **CALL SCREEN**, allowing the user to enter a dialog that connects him or her directly to the remote system:

Client System

```
CALL FUNCTION 'TRAVEL_CREATE_FLIGHT'  
  STARTING NEW TASK 'FLIGHT'  
  DESTINATION 'S11'.
```

Server System

```
FUNCTION TRAVEL_CREATE_FLIGHT.  
  CALL SCREEN 100.  
ENDFUNCTION.
```

If you do not specify a destination, the asynchronous RFC mechanism starts a new session within the calling system.



You must **not** use **IMPORTING** when calling aRFCs.

Receiving Results from an Asynchronous RFC

To receive results from an asynchronously called function, use the following syntax:

```
CALL FUNCTION RemoteFunction  
  
  STARTING NEW TASK Task  
  
  PERFORMING RETURN_FORM ON END OF TASK.
```

Once the called function is completed, the next dialog step in the calling program (such as **AT USER-COMMAND**) guides the system into the FORM routine that checks for results. This FORM routine consists of a special syntax and must be called with a using parameter that refers to the name of the task:

Client System

```
CALL FUNCTION 'TRAVEL_CREATE_FLIGHT'  
  
  STARTING NEW TASK 'FLIGHT'  
  
  DESTINATION 'S11'  
  
  PERFORMING RETURN_FLIGHT ON END OF TASK.  
  
...  
  
FORM RETURN_FLIGHT USING TASKNAME.  
  
  RECEIVE RESULTS FROM FUNCTION 'TRAVEL_CREATE_FLIGHT'  
  
    IMPORTING    FLIGHTID = SFLIGHT-ID  
  
    EXCEPTIONS   SYSTEM_FAILURE MESSAGE SYSTEM_MSG.  
  
    SET USER-COMMAND 'OKCD'.  
  
ENDFORM.
```



- If a function module returns no result, the addition **PERFORMING RETURN_FORM ON END OF TASK** can be omitted.
- If an asynchronous call calls several consecutive function modules with the same destination, you must assign a different task name to each.
- A calling program which starts an asynchronous RFC with **PERFORMING** cannot switch roll areas or change to an internal mode. This is because the asynchronous function module call reply cannot be passed on to the relevant program. You can perform a roll area switch with **SUBMIT** or **CALL TRANSACTION**.
- If the calling program which has executed the asynchronous call is terminated, despite the fact that it is expecting replies, these replies from the asynchronous call cannot be delivered.
- You can use the **WAIT** statement with **PERFORMING form ON END OF TASK** to wait for the reply to a previously started asynchronous call. In this case, **WAIT** must be in the same program context.

Receiving Results from an Asynchronous RFC

- The program processing continues after WAIT if either the condition of a logical expression was satisfied by the subroutine that performs the task in question, or a specified time period has been exceeded. For more information on the WAIT statement, see the online help in the ABAP editor.

The key word **RECEIVE** occurs only with the function module call **CALL FUNCTION func STARTING NEW TASK taskname**. If the function module returns no results, this part need not be defined.



The key word **RECEIVE** is new from R/3 Release 3.0 onwards. Therefore, both partner systems, client and server, must have Release 3.0 of the R/3 System.

For more information on **RECEIVE**, see the online help in the ABAP editor.

The effect of the **SET USER-COMMAND 'OKCD'** statement is exactly as if the user had entered the function in the command field and pressed **ENTER**. This means that the current positioning of the list and the cursor is taken into account.



No call-backs are supported.



The **SET USER-COMMAND 'OKCD'** statement replaces the **REFRESH SCREEN** command. **REFRESH SCREEN** is no longer maintained and should therefore **not** be used.



```
DATA: INFO LIKE RFCSI,
* Result of RFC_SYSTEM_INFO function
      MSG(80) VALUE SPACE.
* Exception handling
CALL FUNCTION 'RFC_SYSTEM_INFO'
      STARTING NEW TASK 'INFO'
      PERFORMING RETURN_INFO ON END OF TASK
      EXCEPTIONS
          COMMUNICATION_FAILURE = 1 MESSAGE MSG
          COMMUNICATION_FAILURE = 2 MESSAGE MSG.
IF SY-SUBRC = 0.
    WRITE: 'Wait for reply'.
ELSE.
    WRITE MSG
ENDIF.
...
AT USER-COMMAND.
```

Receiving Results from an Asynchronous RFC

```

* Return from FORM routine RETURN_INFO via SET USER-COMMAND
      IF SY-UCOMM = 'OKCD'.
        IF MSG = SPACE.
          WRITE: 'Destination =', INFO-RFCDEST.
        ELSE.
          WRITE MSG.
        ENDIF.
      ENDIF.

...

FORM RETURN_INFO USING TASKNAME.
  RECEIVE RESULTS FROM FUNCTION 'RFC_SYSTEM_INFO'
    IMPORTING RFCSI_EXPORT = INFO
  EXCEPTIONS
    COMMUNICATION_FAILURE   = 1 MESSAGE MSG
    SYSTEM_FAILURE          = 2 MESSAGE MSG.
  SET USER-COMMAND 'OKCD'.      "Set OK-code
ENDFORM.

```

Keeping the Remote Context

Keeping the Remote Context

In the FORM routine that checks for results of an asynchronously called function with RECEIVE RESULTS FROM FUNCTION, the addition **KEEPING TASK** prevents the connection from being closed after receiving the results of the processing.

```
FORM RETURN_INFO USING TASKNAME.  
    RECEIVE RESULTS FROM FUNCTION 'RFC_SYSTEM_INFO'  
    KEEPING TASK  
    ...  
ENDFORM.
```

The relevant remote context (roll area) is kept until the caller terminates the connection. If you specify the same taskname, you can re-use the remote context and roll area.

If the remote function module performs interactive tasks such as processing lists or dynpros, screens are displayed until the calling program terminates. If the remote call is made in debugging mode, this mode is visible until the caller dialog is terminated.



You should use the addition KEEPING TASK only if you want to re-use the current remote context for a subsequent asynchronous call.

Keeping a remote context increases storage load and decreases performance due to additional roll area management in the system.

Parallel Processing with Asynchronous RFC

To achieve a balanced distribution of the system load, you can use destination additions to execute function modules in parallel tasks in any application server or in a predefined application server group of an R/3 System.



Parallel-processing is implemented with a special variant of asynchronous RFC. It's important that you use only the correct variant for your own parallel processing applications: the CALL FUNCTION STARTING NEW TASK DESTINATION IN GROUP keyword. Using other variants of asynchronous RFC circumvents the built-in safeguards in the correct keyword, and can bring your system to its knees

Details are discussed in the following subsections:

- Prerequisites for Parallel Processing
- Function Modules and ABAP Keywords for Parallel Processing
- Managing Resources in Parallel Processing

Prerequisites for Parallel Processing

Before you implement parallel processing, make sure that your application and your R/3 System meet these requirements:

- Logically-independent units of work:

The data processing task that is to be carried out in parallel must be logically independent of other instances of the task. That is, the task can be carried out without reference to other records from the same data set that are also being processed in parallel, and the task is not dependent upon the results of others of the parallel operations. For example, parallel processing is not suitable for data that must be sequentially processed or in which the processing of one data item is dependent upon the processing of another item of the data.

By definition, there is no guarantee that data will be processed in a particular order in parallel processing or that a particular result will be available at a given point in processing.
- ABAP requirements:
 - The function module that you call must be marked as externally callable. This attribute is specified in the *Remote function call supported* field in the function module definition (transaction SE37).
 - The called function module may not include a function call to the destination “BACK.”
 - The calling program should not change to a new internal session after making an asynchronous RFC call. That is, you should not use SUBMIT or CALL TRANSACTION in such a report after using CALL FUNCTION STARTING NEW TASK.
 - You cannot use the CALL FUNCTION STARTING NEW TASK DESTINATION IN GROUP keyword to start external programs.
- R/3 System resources:

Parallel Processing with Asynchronous RFC

In order to process tasks from parallel jobs, a server in your R/3 System must have at least 3 dialog work processes. It must also meet the workload criteria of the parallel processing system: Dispatcher queue less than 10% full, at least one dialog work process free for processing tasks from the parallel job.

Function Modules and ABAP Keywords for Parallel Processing

You can implement parallel processing in your applications by using the following function modules and ABAP keywords:

- **SPBT_INITIALIZE**: Optional function module.
Use to determine the availability of resources for parallel processing.
You can do the following:
 - check that the parallel processing group that you have specified is correct.
 - find out how many work processes are available so that you can more efficiently size the packets of data that are to be processed in your data.
- **CALL FUNCTION <function> STARTING NEW TASK <taskname> DESTINATION IN GROUP**:
Use this ABAP keyword to have the R/3 System execute the function module call in parallel. Typically, you'll place this keyword in a loop in which you divide up the data that is to be processed into work packets. You can pass the data that is to be processed in the form of an internal table (EXPORT, TABLE arguments). The keyword implements parallel processing by dispatching asynchronous RFC calls to the servers that are available in the RFC server group specified for the processing.
Note that your RFC calls with CALL FUNCTION are processed in work processes of type DIALOG. The DIALOG limit on processing of one dialog step (by default 300 seconds, system profile parameter *rdisp/max_wprun_time*) applies to these RFC calls. Keep this limit in mind when you divide up data for parallel processing calls.
- **SPBT_GET_PP_DESTINATION**: Optional function module.
Call immediately after the CALL FUNCTION keyword to get the name of the server on which the parallel processing task will be run.
- **SPBT_DO_NOT_USE_SERVER**: Optional function module.
Excludes a particular server from further use for processing parallel processing tasks. Use in conjunction with SPBT_GET_PP_DESTINATION if you determine that a particular server is not available for parallel processing (for example, COMMUNICATION FAILURE exception if a server becomes unavailable).
- **WAIT**: ABAP keyword
WAIT UNTIL <logical expression>
Required if you wish to wait for all of the asynchronous parallel tasks created with CALL FUNCTION to return. This is normally a requirement for orderly background processing. May be used only if the CALL FUNCTION includes the PERFORMING ON RETURN addition.
- **RECEIVE**: ABAP keyword
RECEIVE RESULTS FROM FUNCTION function

Parallel Processing with Asynchronous RFC

Required if you wish to receive the results of the processing of an asynchronous RFC. RECEIVE retrieves IMPORT and TABLE parameters as well as messages and return codes.

Managing Resources in Parallel Processing

You use the following destination additions to perform parallel execution of function modules (asynchronous calls) in the R/3 System:

In a predefined group of application servers:

```
CALL FUNCTION RemoteFunction STARTING NEW TASK taskname  
Destination IN GROUP groupname
```

In all currently available and active application servers:

```
CALL FUNCTION RemoteFunction STARTING NEW TASK Task  
Destination IN GROUP DEFAULT
```

The addition first determines the amount of resources (work processes) currently available (i.e. in all servers or in a group of application servers, comparable with login servers). The resources available for executing asynchronous calls on each application server depends on the current system load.

The applications developer is responsible for the availability of RFC groups in the production system (i.e. the customer's system). For details on how to maintain the RFC groups, see [Maintaining Group Destinations For Load Distribution \[Page 63\]](#).

After determining the available resources, the asynchronous call is executed in an available application server. If no resources are available at that particular time, the system executes the exception routine RESOURCE_FAILURE (see the addition Exceptions). In the case of an asynchronous function module call, this exception **must** be handled by the application program.

The process for determining available resources in an RFC group is as follows:

First, the system determines the length of the dispatcher queue for the relevant application server. If it is greater than 10% of the overall length, the server makes no resources available. If it is smaller, the system makes available the current number of free dialog processes minus 2 (as a reserve instance for other purposes, e.g. for logon to the system or administration programs). Thus, one application server **must** have at least 3 dialog processes if RFC parallel processing is taken into account.



- At present, **only** one RFC group per program environment is supported for parallel execution of asynchronous calls. Using both additions (**DESTINATION IN GROUP <groupname>** and **DESTINATION IN GROUP DEFAULT**) in one program is not allowed.
- The exception routine **RESOURCE_FAILURE** is only triggered in connection with asynchronous RFCs with the additions **DESTINATION IN GROUP groupname** and **DESTINATION IN GROUP DEFAULT**.
- You are recommended (for performance and other reasons) to use an RFC group with sufficient resources for parallel processing of asynchronous calls

Parallel Processing with Asynchronous RFC

Checking Authorizations for RFC

If the system profile parameter *auth/rfc_authority_check* is set (value 1), then the System automatically checks at the CALL FUNCTION keyword whether the authorizations user has the required RFC authorization.

The RFC authorization object is *S_RFC Authorization check at RFC access*. The authorization checks access to function modules by function module group. That is, whether a user has the right to run function modules that belong to a particular group.

You can test a user's RFC authorization with the function module *AUTHORITY_CHECK_RFC*. This function module returns RC = 0 if the user is authorized for the group that you name. The function module does not check whether an authority check will actually take place.

Using Pre-Defined Exceptions for RFC

Using Pre-Defined Exceptions for RFC

While any exceptions arising in the called function module are handled by the addition...
`PERFORMING form ON END OF TASK`, the RFC interface defines two additional exception types. These are:

- `SYSTEM_FAILURE`

This exception reports all failures and system problems on the remote machine.

- `COMMUNICATION_FAILURE`

This exception is raised when a connection or communications failure occurs. It does not report system problems (for example, abnormal termination) that occur on the remote machine.

Requesting Error Messages

In the function modules that you call, you should use exceptions for any error reporting, and not the `MESSAGE` keyword.

```
CALL FUNCTION RemoteFunction
  DESTINATION 'hw1071_53'
  EXPORTING...
  IMPORTING...
  TABLES...
  EXCEPTIONS
    SYSTEM_FAILURE = 1 MESSAGE msg
    COMMUNICATION_FAILURE = 2 MESSAGE msg
```

The system sets the message variable (`msg`) to the system message. You can then display the message or log it in a file. You should not try to interpret message text in your program.



You can use `MESSAGE` **only** with the two system exceptions described here.

Writing Remote Function Modules in ABAP

This section provides specifics on writing function modules that can be called remotely. For information about writing function modules generally, see the system documentation on "Writing Function Modules".

This section contains the following topics:

[Steps for Implementing Remote Function Modules \[Page 52\]](#)

[Programming Guidelines \[Page 53\]](#)

[Debugging Remote Function Modules \[Page 54\]](#)

Steps for Implementing Remote Function Modules

Steps for Implementing Remote Function Modules

To implement a remote function module in ABAP, perform the following steps:

1. Register the module as remotely callable in the RFC server system.

In the function module *Administration* screen (transaction code SE37), set the field *Can be called via REMOTE CALL*. Registering a module as remote causes an RFC stub to be generated for it.

2. Write the code for the function module.

Guidelines for creating function modules in the R/3 Repository are given in the system documentation "Writing Function Modules".

3. Define the destination of the RFC server in the RFC client system that calls the remote function.

Either you or your system administrator can maintain the RFCDES table using transaction SM59 (*Tools → Administration, Administration → Network → RFC destinations*). Maintaining this table is described in the topic [Maintaining Remote Destinations \[Page 55\]](#).

Programming Guidelines

The following sections describe some points to remember when you write a remote function module.

Declaring Parameters

For normal (non-remote) function modules, if a parameter is not defined like an ABAP Dictionary field, it takes the data type of the actual parameter used at run-time.

A remote function module, however, does not have this information available. As a result, all parameter fields for a remote function module must be defined as reference fields, that is, like ABAP Dictionary fields. (This includes IMPORTING, EXPORTING, and TABLES parameters.)

For character structures or fields, the caller's parameters need not be as long as expected by the called program. When incoming parameters are shorter, RFC simply pads them with blanks. This means that the ABAP Dictionary definition of character parameters need not be exactly the same on the calling and called sides. (However, the caller's parameters may not be *longer* than expected on the called side).

Writing for Transactional Execution



There are two restrictions on writing remote functions that are to be called transactionally:

- Transactional calls cannot return parameter values. As a result, the interface for these functions should not specify any EXPORT parameters.
- Functions that run transactionally may not perform call-backs: the caller's context does not necessarily still exist when the call-back is relayed back to the original system.

Reporting on Exceptions

You can raise exceptions in a remote function just as you would in a locally called function.

Since the system raises COMMUNICATION_FAILURE and SYSTEM_FAILURE internally, there is no reason for you to raise them in your program. The MESSAGE keyword is only meaningful in relation to these system exceptions, so you need not program them into your function.

Calling Other Remote Functions

A remote function can call other remote functions, just like an ordinary function module.

In particular, it can use the call-back feature to call function modules running in the system of the original caller.

For details on call-backs between R/3 Systems, see [Calling Remote Functions BACK \[Page 18\]](#)

For details on call-backs between R/3 and external systems, see ['Call-Back' Feature with R/3 and External Systems \[Ext.\]](#).

Debugging Remote Function Modules

Debugging Remote Function Modules

When testing ABAP-to-ABAP RFC calls, you can use the ABAP debugger to monitor the execution of the RFC function in the remote system. Static breakpoints, single-stepping, variable-watching and source display are possible.

With remote calls, the ABAP debugger (including the debugging interface) runs on the local system. Data values and other run information for the remote function are passed in from the remote system.

Maintaining Remote Destinations

Choose *Tools* → *Administration* → *Administration* → *Network* → *RFC destinations*.

Details are explained in the following topics:

[Types of Destinations \[Page 60\]](#)

[Displaying, Maintaining and Testing Destinations \[Page 56\]](#)

[Entering Destination Parameters \[Page 58\]](#)

[Maintaining Group Destinations \[Page 63\]](#)

Displaying, Maintaining and Testing Destinations

Displaying, Maintaining and Testing Destinations

To display, create or modify destinations, choose *Tools* → *Administration* → *Administration* → *Network* → *RFC destinations* or enter transaction code SM59.

Remote Destinations are stored in table RFCDES. The RFCDES table describes logical destinations for remote function calls.

It is not possible to maintain the RFCDES table directly.



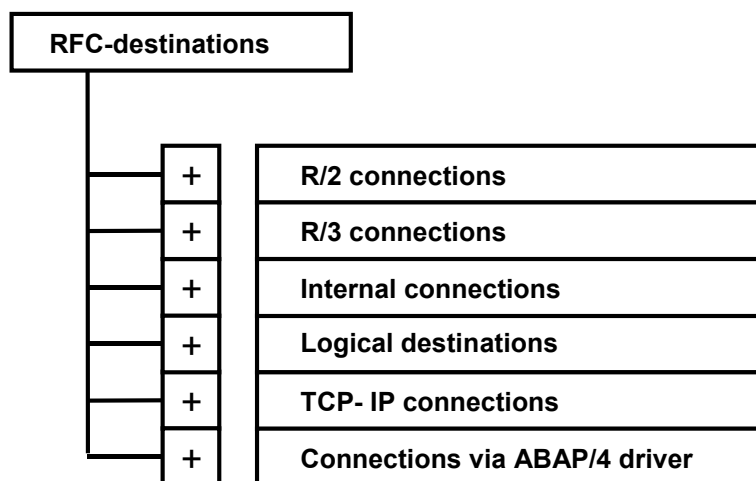
You can also access logical destinations via the Implementation Guide (IMG) by choosing *Tools* → *AcceleratedSAP* → *Customizing* → *Execute Project* → *SAP Reference IMG*.

In the Implementation Guide, expand the following hierarchy structure:

Basis
Application Link Enabling (ALE)
Sending and Receiving Systems
Systems in Network
Define Target Systems for RFC Calls

Displaying Destinations

The initial screen for this transaction displays a tree:



Different connection types (i.e. partner systems or programs) are possible. For further information, see [Types of Destinations \[Page 60\]](#).

To display all information for a given destination, double-click it, or place the cursor on it and press **F2**.

To search for a destination, press the *Find* button and specify your selection. You get a list of all entries matching your selection. Place the cursor on the one you want, and press **F2** or simply double-click the destination. All information for the given entry appears.

Creating Destinations

On the destinations overview screen (transaction code SM59), the connection types and all existing destinations are displayed in a tree structure.

All available connection types are explained in [Types of Destinations \[Page 60\]](#).

To create a new RFC destination, press the *Create* button. A new screen is displayed with empty fields for you to fill in.

If you want to create a new destination

As you create a remote destination, you can specify a particular application server or a group of servers for a balanced distribution of system load.

For details of the destination parameters, see [Entering Destination Parameters \[Page 58\]](#).

Changing Existing Destinations

On the destinations overview screen (transaction code SM59), the connection types and all existing destinations are displayed in a tree structure.

You can display all information for a given destination by double-clicking it or pressing **F2** on it.

To change an existing destination, double-click it, or place the cursor on it and press the *Change* button.

For details of the destination parameters, see [Entering Destination Parameters \[Page 58\]](#).

Testing Destinations

To test a destination, choose the appropriate function from the *Test* menu.

- *Connection* (also available via the *Test connection* pushbutton)
- *Authorization* (checks logon data)
- *Local network* (provides a list of application servers)

Entering Destination Parameters

Entering Destination Parameters

In addition to the *RFC destination*, you must enter the following information:

Technical settings

- *Connection type*

Enter an existing connection type or choose one via the field entry help.
All available connection types are explained in [Types of Destinations \[Page 60\]](#).
- *Trace*

Mark the *Trace* option to have the RFC communication logged and stored in a file. You can then display the file, both in the calling and receiving system, using report RSRFCTRC.
- *Load balance*

If you choose load balancing, you must specify the following information:

 - *Target system* (For a list of available servers, log on to the target system and choose *Tools* → *Administration*, *Monitor* → *System Monitor* → *Servers*.)
 - *Message server* (Log on to the target system and choose *Control* → *Control Panel* from the CCMS main menu. It is the server that offers the service M.)
 - *Group* (of servers) (see SAP Logon Group of Servers)

Otherwise, you must specify the following information:

 - *Target host*

The name of a server host of the target system that you want to use as a port to the system.
 - *System number*

Communications service used with the target system. To obtain it, choose *Tools* → *Administration* → *Monitor* → *System Monitor* → *Servers*.

Security Options

The following options are available only with some connection types:

- *Trusted system* (for type 3 only)

If the target system is a trusted system, choose *Yes*. For details on trusted systems, see [Maintaining Trust Relationships Between R/3 Systems \[Page 64\]](#).
- *SNC* (Secure Network Communications, available for types 3 and T only)

If you have an active SNC-supported security system, you can activate additional security options which you must set via *Destinations* → *SNC options*.

Description

Text description of the entry.

Logon

- *Language*

Entering Destination Parameters

System language to be used

- *Client*

Client code

- *User*

User name to be used for remote logon, if different from current user name

- *Password*

User password

- *Current user*

The current user name is to be used for remote logon.

- *Unencrypted password (2.0)*

If the target system is an R/3 System of Release 2.0, the password must not be encrypted.

The *Attributes* section contains creation and change information.

Types of Destinations

Types of Destinations

Each destination has a connection-type field (*Connection type*), which tells the kind of system connection:

- **R/2 connections (Type 2)**

Type 2 entries specify R/2 systems. No further specification is required, i.e. when you create a type 2 entry, you only need to give the host name; all communications information is already stored in the sideinfo table in the SAP Gateway host. You can, however, specify logon information if desired.

Example entry name: K50

- **R/3 connections (Type 3)**

Type 3 entries specify R/3 systems. When you create a type 3 entry, you must give the host name and communications service. You can also specify logon information if desired. From R/3 Release 3.0 onwards, you can also specify the load-balancing option if desired.



From R/3 Release 3.0 onwards, it is possible to specify an application server from the R/3 message server. The application server is then determined according to the load-balancing process. This applies both for RFCs between R/3 Systems and external calls to R/3 Systems.

Example entry name: K11

- **Internal connections (Type I)**

Type I entries specify R/3 systems connected to the same data base as the current system. These entries are pre-defined and cannot be modified. The entry names are the same as those used in the SAP Message Server (transaction SM51)

Example entry name: hs0010_K11_24

- hs0010=host name
- K11=system name (data base name)
- 24=TCP-service name

- **Logical destinations (Type L)**

Instead of specifying a system connection, type L (logical) entries refer to a physical destination. Type L destinations can also refer to other type L entries. A type L entry uses the information in the "referred-to" entry, and adds further information of its own. Typically, the "referred-to" entry gives the host information, and the type-L entry gives logon data. You can also set a user name, an explicit password, a logon language or an explicit client.

A type L entry can refer to other type L entries.

Example entry name: K11_SD or K11_01

- K11=name of RFCDES entry for R/3 system K11
- SD or 01: for the fields User='SD_INPUT' or Mandant='001'

Types of Destinations

- **Connections via ABAP driver (Type X)**

Type X entries specify systems where device drivers in ABAP have been specially installed. When you create a type X entry, you must give the name of the ABAP device driver.

- **TCP/IP Connections (Type T)**

Type T destinations are connections to external programs that use the RFC API to receive RFCs. The activation type can be either *Start* or *Registration*.

If it is *Start*, you must specify the host name and the pathname of the program to be started.

Activation Type *Start*

The communication method depends on how you select the program location:

- **Explicit host**

In this case, the program is started either by the default gateway for the system or by the explicitly specified gateway (gwr) via remote shell.

Ensure that the computer with the gateway process can access the specified computer by entering `/etc/ping <host name>`.

In order to be able to start a program on another computer using remote shell, the target system must fulfil certain conditions.

- The user ID of the gateway process must exist and a file called `.rhosts` must also be present in the user's home directory.
- The file `.rhosts` must contain the name of the calling computer.

To check this, logon to the computer containing the gateway process with the appropriate user ID and enter the command `remsh <host name> <program name>`. The `<host name>` and `<program name>` must be the same as in SM59. (If you call an RFC server program without any parameters, the `RfcAccept` call always returns an error code (RFC_HANDLE_NULL) and the program terminates at once.)

- **Application server**

On choosing *Application server* and specifying your program, you can start the program from the SAP application server.

First, ensure that the program can be accessed from the SAP application server and that the SAP application server has the authorization to start the program.

To check this, logon with the user ID of the SAP application server (e.g. c11adm). If possible, change to the working directory of the SAP application server (`/usr/sap/.../D.../work`) and try to start the RFC server program manually from there. (As in the above case, if you call an RFC server program without parameters, the `RfcAccept` call always returns an error code (RFC_HANDLE_NULL) and the program terminates at once.)

- **Front-end workstation**

On choosing *Front-end workstation* and specifying your program, you can start the program from the SAPGUI.

Ensure that you can access the program with SAPGUI.

Types of Destinations

Ensure that SAPGUI has the authorization to start the program.

To check this, simply call the RFC server program in your environment.

The function call can also be transactional (CALL FUNCTION... IN BACKGROUND TASK DESTINATION...).

Activation Type *Registration*

If the activation type is *Registration*, you have to identify a registered RFC program. With an SAP gateway, an RFC server program can be registered under this ID and then wait for RFC calls from different SAP Systems.

Example entry name: SERVER_EXEC

- **Type M**

Type M entries are asynchronous RFC connections to R/3 Systems via CMC (protocol X.400).

- **Type S**

Type S corresponds to type 2, except that the destination is SNA or APPC.

Maintaining Group Destinations

To achieve a balanced load distribution in the target system, you must define a group of application servers as an RFC destination. When processing tasks in parallel, you can use the group destination only in connection with the [asynchronous RFC \[Page 38\]](#).

The resources available on each application server depends on the current system load.

To display and maintain the RFC groups, proceed as follows:

1. From the RFC destination overview screen, choose *RFC → RFC groups*.

You'll see:

- the names of any RFC groups that have already been defined
- a list of the instances (host, system and instance number) in your R/3 System
- the current status (active or not) of each server.

2. To define an RFC group, choose *Edit → Create*. Enter a server group name and an instance in the dialog window.

To add instances to an existing group, double click the group name and enter a new instance in the dialog window.

By creating duplicate entries, you can assign a server to more than one group. In this case, jobs that use the group will compete for free work processes on the shared server(s).



Usage examples:

You could use groups to allow different parallel-processed jobs to run at the same time without competing for the same servers. In this case, the different groups used by the jobs would specify different servers.

You could also use groups to separate processing from servers on which dialog users are active. In this case, the group used for processing would name servers other than those in the logon groups for users.

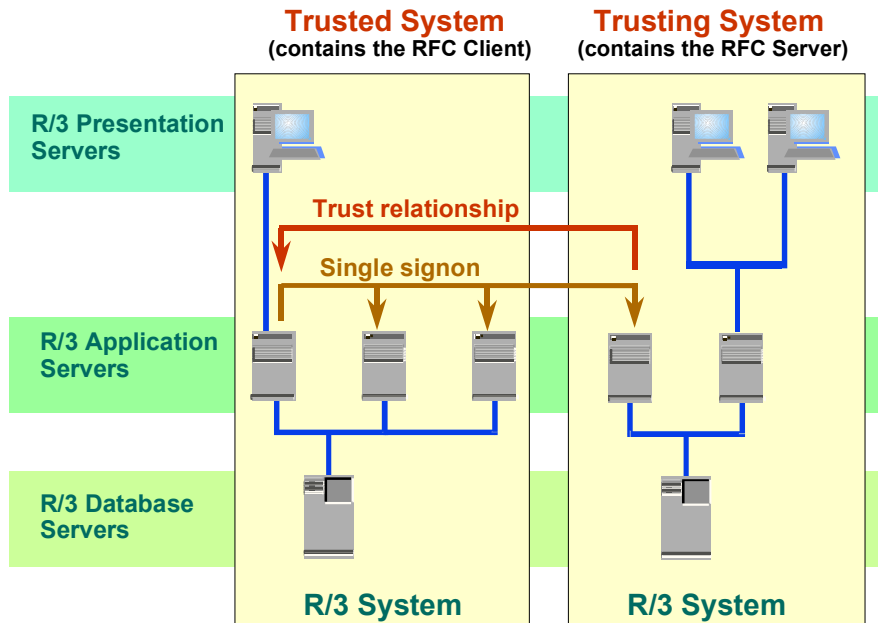
Maintaining Trust Relationships Between R/3 Systems

Maintaining Trust Relationships Between R/3 Systems

R/3 Systems may establish trusted relationships between each other.

If a calling R/3 System is known to the called system as a trusted system, no password must be supplied.

The calling R/3 System must be registered with the called R/3 System as a trusted system. The called system is called the trusting system.



Trust relationships between R/3 Systems have the following advantages:

- Single signon is possible beyond system boundaries.
- No passwords are transmitted in the network.
- Timeout mechanism protects against replay attacks.
- User-specific logon data are checked in the trusting system.

Using this feature, you can create a virtual R/3 System consisting of various R/3 Systems that are called remotely. Remote logon data are checked in the trusting system.

The trust relationship is not mutual, which means, it applies to one direction only. To establish a mutual trust relationship between two partner systems, you must define each of the two as trusted systems in its respective partner system.

For additional security, you can make use of SAP's SNC interface (Secure Network Communications) for third-party security systems such as Kerberos and SECUDE.

Displaying, Maintaining and Testing Trusted Systems

To display or maintain a trusted system in the trusting system, proceed as follows:

Maintaining Trust Relationships Between R/3 Systems

1. If you want to define an R/3 System as a trusted system, you must first create a logical destination that allows a trusted system relationship. For details, see [Maintaining Remote Destinations \[Page 55\]](#).

2. From the RFC destination overview screen (transaction SM59), choose *RFC → Trusted systems* or enter transaction code SMT1.

If trusted systems have already been defined, they are displayed in a hierarchy tree. To display existing trusted systems, expand the nodes in the hierarchy tree.

For details, double click a trusted system.

3. To create a trusted system, click the *Create* icon.

In the dialog window, enter the destination for the remote system. To change a destination, see *Changing Trusted Destinations* below.

All the necessary information such as application server name and security key is supplied automatically.

4. If you want to restrict the validity period of the logon data, enter an end date in the *Validity period* field.
5. If you want take over the transaction code of the calling program into the called system, mark the appropriate checkbox.

Only then will an authorization check be performed in the called system for the transaction code (field RFC_TCODE of the S_RFCACL authorization object, see *Logon Authorization Checks in the Trusting System* below).

6. To delete a trusted system relationship, display the trusted system details and click the *Delete* pushbutton.



As you delete a trusted system relationship, the logon screen of the relevant system is displayed, if no valid logon data are provided. You must log on to that system to complete the deletion.

Changing Trusted Destinations

You can change existing destinations for each system from the trusted system maintenance screen (*RFC → Trusted systems*, transaction code SMT1) by clicking on the *Maintain destination* pushbutton.

In trusted systems, destinations for trusting systems are automatically created. These destinations are used when you display trusting systems via *RFC → Trusting systems* (transaction code SMT2).

To prevent others from making changes to your trusted destination, mark the checkbox *Destination not changeable* in the *Attributes* section. To make the destination changeable again, doubleclick the checkbox.

For more details on fields, invoke field help.

Please note that destinations must remain consistent, which means you must neither change the target system ID, the system number nor the destination name.

Maintaining Trust Relationships Between R/3 Systems

Displaying Trusting Systems

In a trusted system, you can obtain a list of all trusting systems.

Choose *RFC* → *Trusting systems* to display the list of trusting systems.

Click on the name of a trusting system to display the application servers of that system. The application server names contain the suffix `_TRUSTED`.

Double-clicking an application server name provides a dialog window with an entry field for a transaction code to be performed in the trusting system, and whether the transaction is to be carried out in the same mode or in a new mode.

Logon Authorization Checks in the Trusting System

The logon data used for logging on to a trusting system undergo an authorization check.

The data provided by the trusted system is checked for system name, client, user name, and other optional data. These data must match the field values of authorization object `S_RFCACL`.

The system administrator can check a user's logon data using the function module `AUTHORITY_CHECK_TRUSTED_SYSTEM`.

Error return codes are explained in the *Troubleshooting* section below.

Testing Trusting Systems

To test a trusted system, you can perform authorization checks for the current server and the trusting system via the *Entry* menu. If no valid logon data are supplied, the logon screen of the trusted systems appears. You should log on to the system. If your test is not successful, read the section *Troubleshooting in Trusted/Trusting Systems* below.

Troubleshooting in Trusted/Trusting Systems

After you have created a trusted system, you must test the destination by logging in to the trusted system via the *Remote logon* pushbutton.

Alternatively, you can perform an authorization check for the trusted server using the appropriate function from the *Test* menu.

If your login attempt fails, you will receive the following message: *No authorization to log in as trusted system* (error code = <0|1|2|3>). Note that the special users DDIC and SAP* must not be used.

The error code explanation is as follows:

- 0 Invalid login data (user ID and client) for the trusting system
Solution: Create the user ID for the client in the trusting system.
- 1 No trusted system entry exists for the calling system, or the security key for the system is invalid.
Solution: Create the trusted system entry again.
- 2 The user does not have a trusted system authorization (object `S_RFCACL`).
Solution: Provide the user with the necessary authorization.
- 3 The time stamp of the login data is invalid.

Maintaining Trust Relationships Between R/3 Systems

Solution: Check the clock settings on both the client and server host and the expiration date of the login data. (Note that the default expiration period 00:00:00 means no limit.)

You can check whether correct login information has been provided for the trusted system in the trusting system by means of the function module `AUTHORITY_CHECK_TRUSTED_SYSTEM`.

If all your tests are successful and you still don't get access to the trusting system, refresh the relevant database buffers by choosing *Utilities* → *Mass changes* → *Reset all buffers* from the user maintenance screen.



To find out the cause of an error, activate the trace flag on the destination details screen, reproduce the error and read the information provided with the error ID `CALL_FUNCTION_SINGLE_LOGIN_REJ` in the short dump created in the called system (the trusting system).