

Enhancements, Modifications, ... (CA-BFA)



HELP.CABFABAPIREF

Release 4.6C



Copyright

© Copyright 2001 SAP AG. All rights reserved.

No part of this publication 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[®], PowerPoint[®] 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.

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, Massachusetts Institute of Technology.

JAVA[®] is a registered trademark of Sun Microsystems, Inc.

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

SAP, SAP Logo, R/2, RIVA, R/3, ABAP, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com 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

Enhancements, Modifications, ... (CA-BFA)	6
Customer Enhancement and Modification of BAPIs (CA-BFA)	7
Customer Enhancement of BAPIs	10
Enhancing the BAPIs Based on Existing SAP Database Tables.....	13
Appending Customer Fields.....	14
Using Additional Existing Database Fields.....	19
Combination of Appending Customer Fields and Adding Existing Table Fields.....	21
Enhancing the BAPI by Including Additional Customer Database Tables.....	22
Enhancing the BAPI with Import Data that Does Not Affect the Database Level.....	26
Actions by the BAPI Developer.....	28
Actions by the Customer.....	36
Actions for an Enhancement Based on Existing SAP Database Tables.....	37
Actions when Including Additional Customer Database Tables.....	44
Actions when Including Additional Import Data that Does Not Affect the Database Level.....	45
Use with the Standardized BAPIs.....	46
BAPI Modifications	50
Example.....	54
Creating Subtypes.....	55
Redefining a BAPI.....	56
Defining Delegation Relationships.....	57
Examples	58
Example for Developing the BAPI Function Module.....	59
Example for Filling the ExtensionIn Parameter.....	63
SAP Enhancements to Released BAPIs	65
Compatible Enhancements	67
Incompatible Enhancements	69
BAPIs for Mass Data Transfer (CA-BFA)	72
Basics of Mass Data Transfer	74
Process Flow of the Mass Data Transfer via BAPI	80
Developing BAPIs for Mass Data Transfer	83
Implementing a BAPI.....	85
Programming Create() BAPIs.....	86
Example of a Create() BAPI.....	90
Programming Change() BAPIs.....	91
Example of a Change () BAPI.....	96
Programming Delete()/Undelete() BAPIs.....	97
Example of a Delete() BAPI.....	101
Programming Cancel() BAPIs.....	102
Example of a Cancel() BAPI.....	105
Programming Replicate()/SaveReplica() BAPIs.....	106
Example of a SaveReplica() BAPI.....	110
Programming Methods for Sub-Objects.....	111
Example of an Add<Name of Sub-Object> BAPI.....	115

Generating the BAPI-ALE Interface	116
Writing a report.....	117
Registering the BAPI.....	118
Details	119
Using ALE Services (CA-BFA)	121
Basic Concepts of ALE Technology	123
Implementing Narrow Coupling with BAPIs	127
Querying the Distribution Model.....	128
Calling BAPIs	130
Implementing Loose Coupling with BAPIs	131
Querying the Distribution Model.....	133
ALE Outbound Processing.....	135
Dispatching IDocs	137
ALE Inbound Processing.....	138
Processing BAPIs	139
Developing an ALE Business Process Based on BAPIs	141
Implementing the BAPI	142
Defining Hierarchies Between BAPI Parameters.....	144
Maintaining the BAPI-ALE Interface	145
Notes.....	159
Maintaining the Distribution Model.....	161

Enhancements, Modifications, ... (CA-BFA)

The documents located at this level cover topics that start with the standard case for new development described in the *BAPI Programming Guide*, and expand it with practical requirements.

Customer Enhancement and Modification of BAPIs (CA-BFA)

Purpose

Customers often have to change the standard system supplied by SAP in order to meet their own special requirements. This fact is reflected in the implementation of BAPIs. A number of concepts have been developed to allow IBUs, partners, and customers to change existing BAPIs.

Target Audience

The target audience of this document consists of:

- *Customers* who want to enhance or modify the BAPIs supplied by SAP
The modification concept can also be used by IBUs and partners. This group is simply referred to as 'customers' below.
- *SAP developers* who have to design the BAPIs to allow enhancement by the customers

Implementation Considerations

To enhance or modify a BAPI, you need:

- Basic knowledge of BAPIs, as described in [BAPIs – Introduction and Overview \[Ext.\]](#)
- Detailed knowledge of BAPI development, as described in the [BAPI Programming Guide \[Ext.\]](#)
- Knowledge of the ABAP programming language and ABAP Workbench
- Basic knowledge of the R/3 System

Features

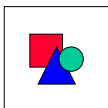
Types of Changes to Existing BAPIs

Customer changes to existing BAPIs can be implemented as either a [customer enhancement \[Page 10\]](#) or a [customer modification \[Page 50\]](#):

Customer Enhancement

The customer enhancement concept should be used (instead of the modification concept) whenever possible, since changes to the standard implemented by SAP (in a new release, for example) will be activated automatically in an enhanced BAPI. In contrast, if the BAPI has been modified, these changes will not be activated automatically.

Customer enhancements are transferred to the BAPI via **containers** that are located near the interface, so they are not directly visible in the interface. This concept is useful, for example,



when customers want to enhance the underlying database tables for a BAPI, or when they want to use an existing BAPI to manipulate their own tables as well.

Customer Enhancement and Modification of BAPIs (CA-BFA)

The customer concept described was new in Release 4.5B. As a result, not all of the existing BAPIs have the containers required for enhancement in their interfaces. In these cases, either modify the BAPIs or enhance them in accordance with the old concept. Customer exits that are implemented using the enhancement concept do not have to be reprogrammed.



The customer enhancement concept is not suitable for developing individual industry solutions within SAP, nor is it intended for partner developments, because it does not support multi-level changes. For this reason, IBUs and partners always have to use the modification concept.

Customer Modification

Customers have to modify existing BAPIs when they want to extend their functionality – for example, adding new parameters or parameter fields to the BAPIs, or changing the function module coding. When a BAPI is modified, any changes to the version supplied by SAP in a new release will not automatically be activated in the modified version.

BAPI modifications cannot be made to the business object types delivered in the standard system. Instead, you create a **subtype** of the business object type which inherits all the characteristics and methods of this object type. The original object type becomes the **supertype**. The necessary changes are then performed on the derived subtype.

In this way, the SAP standard delivered to customers is not changed, and the modified version is retained when the release is upgraded.

This procedure also enables BAPIs to be modified in steps because further subordinate subtypes of a business object type can be created for an existing subtype.



Multi-level change: A core BAPI is modified by an IBU. In turn, the modified IBU version can be modified by a customer.

For general information about object types and subtypes, please see the documentation for the [SAP Business Object Builder \[Ext.\]](#).

The following table contrasts the relative advantages and disadvantages of customer enhancements and modifications:

	Customer Enhancement	Customer Modification
SAP changes to the BAPI activated automatically	Yes	No
SAP changes have no effect on own enhancement	Yes	No
Own changes to BAPI interface remain explicitly visible	No	Yes
Changes to SAP coding are possible	No	Yes

See also:

[Customer Enhancement of BAPIs \[Page 10\]](#)

[Customer Modification of BAPIs \[Page 50\]](#)

[Examples \[Page 58\]](#)

Customer Enhancement of BAPIs

Customer Enhancement of BAPIs

Purpose

You should use the customer enhancement concept whenever it is important for you to automatically activate subsequent SAP changes to the BAPI, but these SAP changes should not affect the customer enhancements.

Implementation Considerations

To meet these two requirements, it is essential that you do not change the BAPI parameters. Instead, customer enhancements are passed to the BAPI in a container, and can be processed within the BAPI function module. Accordingly, the following minimum precautions must be taken for every BAPI that can potentially be enhanced:

- **Extension parameter in the BAPI interface**
Depending on the requirements of the BAPI in question, the BAPI developer creates one extension parameter for the data import (*ExtensionIn*) and/or one extension parameter for the data export (*ExtensionOut*) in the interface of the BAPI function module. These extension parameters serve as a container in which all the customer enhancements are passed to the BAPI.
- **Customer exits**
Each BAPI function module must contain a maximum of two extra exits in addition to the customer exits provided by the application. The first customer exit should enable the customer to check all the data passed to the BAPI. In particular, the content of the *ExtensionIn* parameter should be checked before its contents are processed further in the BAPI. The second customer exit should be provided whenever the customer is allowed to implement further processing (such as writing to additional tables, reading additional values, or selecting data by additional criteria). If the customer has enhanced the export parameter of the BAPI, the second customer exit should be used to fill the *ExtensionOut* parameter.

Features

In detail, the customer enhancement concept for BAPIs supports the following types of enhancement:

1. [Enhancement to the BAPI based on existing SAP database tables \[Page 13\]](#)
The customer can consider values in the BAPI that are contained in the underlying SAP database tables, but are not included in the supplied BAPI interface. We differentiate between two different applications:
 - a. [Appending customer fields to SAP tables \[Page 14\]](#)
In this case, the customer has appended additional customer fields to the SAP tables used by the BAPI. As a result, the enhancement concept must allow these additional values to be passed on to the BAPI, processed in the BAPI, and/or returned by the BAPI.



New fields are appended to a standard SAP table, and they are subject to the detail parameter of a *GetDetail()* BAPI. When this *GetDetail()* BAPI is called, the export container (*ExtensionOut*) can be used to return the additional detailed information. The customer can fill the *ExtensionOut* in a customer exit.

Customer Enhancement of BAPIs

This type of BAPI enhancement necessitates a separate concept, since, although the customer can make enhancements to the database table using APPEND or INCLUDE, the APPEND technique is prohibited with the associated BAPI structures for reasons of compatibility.

b. [Using existing database fields \[Page 19\]](#)

If the SAP developer did not add all the table fields of the underlying database tables to the BAPI interface, customers can subsequently add these fields to the BAPI.



In a *GetDetail()* BAPI, the SAP table with the detailed values contains more fields than the corresponding detail parameter in the BAPI. If customers want to return some of the values contained in these fields, they can use the export container (*ExtensionOut*), which they filled previously in the customer exit.



You can also combine both methods, appending customer fields to a database table and add existing fields to the BAPI. This option is described in detail under [Combination of Appending Customer Fields and Adding Existing Database Fields \[Page 21\]](#).

2. [Enhancing the BAPIs by including additional customer database tables \[Page 22\]](#)

The enhancement concept also allows customers to process their own tables in a BAPI.



The customer uses separate tables to hold additional values for an object created by an *Create()* BAPI. These values can be passed on to the BAPI using the import container (*ExtensionIn*) and written to the customer tables within a customer exit in the BAPI.

3. [Enhancing the BAPI with import data that does not affect the database level \[Page 26\]](#)

In addition to the enhancement of BAPIs with values that affect database tables, customers can also pass values on to the BAPI that are only required at runtime.



The customer wants to add an additional selection criterion to a *GetList()* BAPI. To do this, the customer can use the *ExtensionIn* parameter to pass on the selection criteria and use the customer exit to perform the selection.

Important : Because the extension parameters are defined in the BAPI and supported in the customer exits, customers can automatically use all three types of enhancement for the BAPI. If the customer requires additional support for implementing certain types of enhancement, then additional precautions, which are described in the next chapter, are required.

See also:

[Enhancing the BAPI Based on Existing SAP Database Tables \[Page 13\]](#)

[Enhancing the BAPI by Including Additional Customer Database Tables \[Page 22\]](#)

[Enhancing the BAPI with Import Data that Does Not Affect the Database Level \[Page 26\]](#)

Customer Enhancement of BAPIs

[Actions by the BAPI Developer \[Page 28\]](#)

[Actions by the Customer \[Page 36\]](#)

[Use with the Standardized BAPIs \[Page 46\]](#)

Enhancing the BAPIs Based on Existing SAP Database Tables

Purpose

Customers can be allowed to modify BAPIs:

- They can take fields into account which have been subsequently added to the underlying SAP tables
- They can include fields that belong to the supplied SAP table, but are not yet taken into account in the BAPI interface

Implementation Considerations

In this case, the following precautions must be applied to the BAPI in question:

1. **Extension parameter in the BAPI interface**
Depending on the requirements of the BAPI in question, the BAPI developer creates one extension parameter for the data import (*ExtensionIn*) and/or one extension parameter for the data export (*ExtensionOut*) in the interface of the BAPI function module.
2. **Creating BAPI table extensions**
A **BAPI table extension** must exist for each underlying table for a BAPI that contains at least one field that is to be taken into account in that BAPI. These BAPI table extensions are help structures that are used during the data import to copy the customer enhancements from the container (*ExtensionIn*), and to fill the additional fields of the database table. During data export, the BAPI table extensions are used to copy data from a customer table extension and to place it in the *ExtensionOut* parameter.
The BAPI developer creates the corresponding BAPI table extensions for those tables for which enhancement is considered a useful option. If the customer wishes to use other BAPI tables for extensions, he/she must create the BAPI table extensions him-/herself.
3. **Customer exits**
As mentioned above, each BAPI function module must contain a maximum of two **extra exits** in addition to the customer exits provided by the application, in which the customer can perform data verification or other processing.

The interaction between the involved components is illustrated in the examples below:

- a) [Adding Additional Fields to a Database Table \[Page 14\]](#)
- b) [Taking Existing Database Fields into Account \[Page 19\]](#)
- c) [Adding Own Fields and Taking Existing Fields into Account \[Page 21\]](#)

Appending Customer Fields

Appending Customer Fields

In this example, a customer wants to use BAPI *TravelAgency.CreateFromData*. This BAPI is based on database table STRAVELAG, to which the customer wants to add three fields compared to the version supplied by SAP.

The Extended Database Table

To extend the database table supplied by SAP, the customer first creates a data structure that contains the customer-specific fields. In our example, this is structure TRAVELAG, with the three fields PLANETYPE, COMPANY, and SEATSMAX.

Structure	TRAVELAG				
Short description	Include for sample extension in travel agency				
Component	Component type	DType	Length	DecPl.	Short description
PLANETYPE	S PLANETYE	CHAR	10	0	Aircraft type
COMPANY	S CARRNAME	CHAR	20	0	Name of airline
SEATSMAX	S SEATSMAX	INT4	10	0	Maximum occupancy

This structure is included in both the database table and the **BAPI table extension**. This ensures that the enhancements to the database table and in the table extension are always identical.

The database table is extended by adding an APPEND structure that contains an INCLUDE with the data structure of the customer-specific fields. We use an APPEND structure to ensure that the enhancements always appear at the end of the table.

The diagram below shows the extended table.

Appending Customer Fields

Transparent table	STRAVELAG							
Short description	Travel agency							
Fields	Key	Init.	Field type	Date	Length	DecPl.	Check table	Short description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_MANDT	CLNT	3	0	T000	Client for WB training data model
AGENCYNUM	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_AGCYNUM	NUMC	8	0	SBUSPART	Number of travel agency
NAME		<input checked="" type="checkbox"/>	S_AGCYNAM	CHAR	25	0		Name of travel agency
STREET			S_STREET	CHAR	30	0		Street for WB training data model
POSTBOX			S_POSTBOX	CHAR	10	0		PO box for WB training data mode
POSTCODE		<input checked="" type="checkbox"/>	POSTCODE	CHAR	10	0		ZIP code for WB training data mo
CITY		<input checked="" type="checkbox"/>	CITY	CHAR	25	0		City for WB training data model
COUNTRY			S_COUNTRY	CHAR	3	0		Country code for WB training data
REGION			S_REGION	CHAR	3	0		Region of a country
TELEPHONE		<input checked="" type="checkbox"/>	S_PHONED	CHAR	30	0		Phone number of a customer
URL			S_URL	CHAR	50	0		URL of travel agency's home page
LANGU			SPRAS	LANG	1	0	T002	Language key
.APPEND			APPENDSA		0	0		Append for STRAVELAG
.INCLUDE			TRAVELAG		0	0		Include for sample extension of ta
PLANETYPE			S_PLANETYPE	CHAR	10	0		Aircraft type
COMPANY			S_CARRNAME	CHAR	20	0		Name of airline
SEATSMAX			S_SEATSMAX	INT4	10	0		Maximum occupancy

The BAPI Table Extension

As mentioned above, BAPI table extensions are help structures that allow the customer enhancement of BAPIs. They must exist for **every** SAP database table that is to be used in the customer enhancement concept. These BAPI table extensions are used during the data import to copy the customer enhancements from the *container (ExtensionIn)* to the BAPI interface, and to fill the additional fields of the database table. During data export, the BAPI table extensions are used to copy data from a customer table extension and to place it in the *ExtensionOut* parameter. Places where BAPI table extensions are used and how their use must be implemented in detail are described in [Actions by the BAPI Developer \[Page 28\]](#).

BAPI table extensions are created as data structures in the ABAP Dictionary with transaction **SE11**. They generally consist of the following components:

- A key part predefined **by SAP**. This is/are the key field(s) of the database table to which the BAPI table extension refers. Accordingly, SAP supplies BAPI table extensions only with a key part, without a data part.
- A data part that the **customer determines** through the APPEND technique and which contains the additional fields. If customers want to add additional table fields to the SAP

Appending Customer Fields

table, these table fields are inserted in the APPEND as an INCLUDE. Therefore, the customer subsequently adds the data part to the "initial" table extension supplied by SAP.



When a customer creates a BAPI table extension, he/she must define both the key part and the data part.

The following guidelines have to be observed when working with BAPI table extensions:

- The naming convention for BAPI table extensions is BAPI_TE_<table_name>. If you cannot follow this naming convention due to the length restriction for structure names, then you must specify this explicitly in the documentation of the table extension and the BAPI.
- A separate BAPI table extension must be created for every database table that the BAPI developer thinks could be enhanced. Please note, however, that each table can only have one table extension. Therefore, if several BAPIs use the same table, they will have to share the table extension.

Exception If a BAPI has two or more parameters that refer to the same database table, then a separate BAPI table extension is created for each of these BAPI parameters. The naming convention in this case is BAPI_TE<table_name><consecutive_number>. In our example, the BAPI table extensions would be called BAPI_TE_STRAVELAG1, BAPI_TE_STRAVELAG2, and so on.



If customers want to extend a table that does not have a table extension defined by SAP, they will have to define their own table extensions.

- The structure of the BAPI table extension must contain all the identified fields (key fields) of the table to which the BAPI table extension relates. This is the only way to determine which line of the extended database table will save the extensions in the BAPI.
- The data structure that is included in the APPEND must be the same one used for the extension of the database. This is the only way to make sure that the additional table fields are filled automatically.
- Customers can use only fields of data type CHAR and similar data types in BAPI table extensions. This restriction is due to the reference structure *BAPIPAREX* of the extension parameters. Customers cannot use fields from the standard table in the APPEND of the BAPI table extension because a 'move corresponding' would overwrite the SAP field.
- The data part of a BAPI table extension can have a maximum length of 960 characters.

The following diagram shows how the BAPI table extension has to look in our example. The important thing is that the INCLUDE structure contained in the APPEND is the same one used to extend the database table. In this case, this is structure TRAVELAG with the fields PLANETYPE, COMPANY, and SEATSMAX.

Appending Customer Fields

Structure

Short description

Component	Component type	DType	Length	DecPI	Short description
AGENCYNUM	S_AGENCYNUM	NUMC	0	0	Number of travel agency
.APPEND	APPDSA		0	0	Append for BAPI_TE_SA
.INCLUDE	TRAVELAG		0	0	Include for example extension
PLANETYPE	S_PLANETYPE	CHAR	10	0	Aircraft type
COMPANY	S_CARRNAME	CHAR	20	0	Name of airline
SEATSMAX	S_SEATSMAX	INT4	10	0	Maximum occupancy

Extension Parameters and BAPIs

The two extension parameters (*ExtensionIn* and *ExtensionOut*) in the BAPI interface are used to pass on the enhancements to the BAPI function module or forward them from the function module to the calling program in container format.

The extension parameters are always based on data structure BAPIPAREX. It determines the format of the data records used to pass the enhancements back and forth between the BAPI and the calling program.

The diagram below shows the BAPIPAREX structure.

Structure

Short description

Component	Component type	DType	Length	DecPI	Short description
STRUCTURE	TE_STRUC	CHAR	30	0	Structure name of BAPI table ext
VALUEPART1	VALUEPART	CHAR	240	0	Data part for BAPI extension
VALUEPART2	VALUEPART	CHAR	240	0	Data part for BAPI extension
VALUEPART3	VALUEPART	CHAR	240	0	Data part for BAPI extension
VALUEPART4	VALUEPART	CHAR	240	0	Data part for BAPI extension

The individual fields have the following purpose:

- STRUCTURE**

The STRUCTURE field contains the name of the BAPI table extension to which the respective data record refers. In turn, because a table extension is allocated to exactly one database table, the program can determine the extended table in which to save the data record.

Appending Customer Fields

- **VALUEPART1 through VALUEPART4**

Each data record of the extension container contains – in addition to the name of the table extension – the key values and the values that will be inserted in the additional table fields. The key values have to be passed on in order to determine the line in the database table where the data in the data record will be written.

Exception It is not possible to pass on the key value(s) in *Create()* BAPIs with internal number assignment, as these values are generated in the BAPI itself. For information on how to proceed in the case of *Create()* BAPIs, refer to [Actions for Enhancements Based on Existing Database Tables \[Page 37\]](#).

Therefore, the VALUEPART1 through VALUEPART4 fields contain both the key value(s) that identify the table line and the data fields to be inserted into the table. The assignment of key values and data to VALUEPART1 through VALUEPART4 is performed by consecutively filling the VALUEPART fields. VALUEPART1 first saves all the key fields of a data record. If VALUEPART1 still has capacity, then the value of the first customer-specific table field is also saved in VALUEPART1. This is continued with the further table fields until the capacity of VALUEPART1 is exhausted. The process is continued with VALUEPART2 through VALUEPART4, until all the values in the data record have been included in the container.



In situations where the remaining capacity of VALUEPART1 (for example) is less than the maximum length of the next table field to add, then the value of this table field is split between VALUEPART1 and VALUEPART2. As much of the value as fits is saved in VALUEPART 1, and the rest is allocated to VALUEPART2.

The advantage of this type of container structure is that the container data record can be converted to the table extension format with a single MOVE command in the BAPI.

Continuing the above example, the extension container could look like this:

STRUCTURE	VALUEPART1		VALUEPART2		...
BAPI_TE_STRAVELAG	4711	BOEING747 LUFTH	ANSA	456 SEATS	...
BAPI_TE_STRAVELAG	8732	CONCORDE AIR FR	ANCE	745 SEATS	...
...

The values “BOEING747”, “LUFTHANSA”, and “456 SEATS” in the first data record would be written to the additional fields of database table STRAVELAG specified in BAPI_TE_STRAVELAG . The table line in which the values will be inserted is identified by key “4711”.

The check and further processing of the data from the import container and the filling of the export container take place subsequently, in customer exits supplied by the BAPI developer.

Using Additional Existing Database Fields

If customers want to include existing database fields in the BAPI, all they have to do is fill the BAPI table extension for this table with these fields (or create a BAPI table extension if none exists yet). This procedure is illustrated using BAPI *TravelAgency.GetDetail*. This BAPI is based on database table STRAVELAG, which contains the detail data for a travel agency.

The Existing Database Table

Table STRAVELAG contains the SAP-defined fields shown in the following diagram.

Transparent table	STRAVELAG							
Short description	Travel agency							
Fields	Key	Init.	Field type	Date	Length	DecPl.	Check table	Short description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_MANDT	CLNT	3	0	T000	Client for WB training data model
AGENCYNUM	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_AGNCYNUM	NUMC	8	0	SBUSPART	Number of travel agency
NAME		<input checked="" type="checkbox"/>	S_AGNCYNAM	CHAR	25	0		Name of travel agency
STREET			S_STREET	CHAR	30	0		Street for WB training data model
POSTBOX			S_POSTBOX	CHAR	10	0		PO box for WB training data mod
POSTCODE		<input checked="" type="checkbox"/>	POSTCODE	CHAR	10	0		ZIP code for WB training data mo
CITY		<input checked="" type="checkbox"/>	CITY	CHAR	25	0		City for WB training data model
COUNTRY			S_COUNTRY	CHAR	3	0		Country code for WB training data
REGION			S_REGION	CHAR	3	0		Region of a country
TELEPHONE		<input checked="" type="checkbox"/>	S_PHONED	CHAR	30	0		Phone number of a customer
URL			S_URL	CHAR	50	0		URL of travel agency's home page
LANGU			SPRAS	LANG	1	0	T002	Language key

This table is used to return all the fields (except URL and LANGU) through the corresponding detail parameters of the BAPI supplied by SAP. To use the BAPI sensibly, however, the customer wants the BAPI to return these two additional fields as well.

To do this, the customer must first define the fields to include in the corresponding BAPI table extension.

The BAPI Table Extension

BAPI table extensions also perform their function for this type of enhancement, and the guidelines described above should be followed. Their general structure is also equivalent to that described under [Appending Customer Fields \[Page 14\]](#) – that is, it consists of a key part (usually

Using Additional Existing Database Fields

defined by SAP) and a data part determined by the customer using the APPEND technique. The only difference is the way the data part is defined:

- The fields to add are defined **directly** in the APPEND structure of the BAPI table extension.
- Make sure that the names and data types of the fields included in the APPEND are identical to the corresponding fields in the database table.

The diagram below illustrates this situation.

Structure	BAPI_TE_SA2				
Short desc.	Customer enhancement of table STRAVELAG: Add addt'l SAP fields				
Component	Component type	DType	Length	DecPl.	Short description
AGCYNUM	S AGCYNUM	NUMC	8	0	Number of travel agency
.APPEND	APPDSA2		0	0	Append for BAPI_TE_SA2
URL	S_URL	CHAR	50	0	URL of travel agency's home pa
LANGU	SPRAS	LANG	1	0	Language key

Extension Parameters and BAPIs

The structure of the extension parameters in the BAPI interface and the way these containers are filled with data is equivalent to an enhancement that appends customer fields.

Combination of Appending Customer Fields and Adding Existing Table Fields

Combination of Appending Customer Fields and Adding Existing Table Fields

In addition to the separate application of the two enhancement types described above, you can also add customer fields and include existing fields in the BAPI for a single SAP table.

The same components described above are involved, and the same rules and guidelines apply. In addition, make sure to observe the following in the structure of the BAPI table extensions:

- First take all the existing fields from the corresponding database table that you want to include in the BAPI and add them to the APPEND structure that contains the data part of the BAPI table extension.
- Then add the INCLUDE that contains the new customer fields to the same APPEND structure.
Please note that you have to use the same INCLUDE in both the database table and in the corresponding BAPI table extension.

The following diagram illustrates how the BAPI table extension has to be defined both to use the existing fields URL and LANGU in the BAPI and to extend the table with the three customer fields PLANETYPE, COMPANY, and SEATSMAX.

Structure

Short desc.

Component	Component type	DType	Length	DecPl	Short description
AGNCYNUM	S_AGNCYNUM	NUMC	8	0	Number of travel agency
.APPEND	APPDSA2		0	0	Append for BAPI TE SA2
URL	S_URL	CHAR	50	0	URL of travel agency's home pa
LANGU	SPRAS	LANG	1	0	Language key
.INCLUDE	TRAVELAG		0	0	Include for example extension
PLANETYPE	S_PLANETYE	CHAR	10	0	Aircraft type
COMPANY	S_CARRNAME	CHAR	20	0	Name of airline
SEATSMAX	S_SEATSMAX	INT4	10	0	Maximum occupancy

Enhancing the BAPI by Including Additional Customer Database Tables

Purpose

The customer enhancement concept makes it possible for customers to process tables they have defined themselves in a BAPI.

Implementation Considerations

To enable the customers to implement this type of enhancement, at least the following precautions must be implemented in the BAPI:

1. **Extension parameter in the BAPI interface**

Depending on the requirements of the BAPI in question, the interface of the BAPI function module must contain one extension parameter for the data import (*ExtensionIn*) and/or one extension parameter for the data export (*ExtensionOut*).

2. **Customer exits**

For this case, as well, each BAPI function module must contain a maximum of two **extra exits** in addition to the customer exits provided by the application, in which the customer can perform data verification or other processing.

The extension parameters must be defined in the BAPI interface, and the customer exits must be provided. In contrast, the order in which the data from the new customer tables is passed to or taken from the extension containers is entirely up to the customer.

As a result, the use of BAPI table extensions is not absolutely necessary for the types of enhancement described above. We do recommend, however, defining help structures for the new tables, as they allow the extension containers to be filled and evaluated with more certainty.

Example

In the example below, BAPI *TravelAgency.CreateFromData* is used to illustrate which components are involved in this type of enhancement. In this example, the customer has created her own table, YSTRAVELAG, in addition to the underlying SAP tables for the BAPI, and uses this table to store additional data for object *TravelAgency*.

The Defined Customer Table

The customer table YSTRAVELAG has the following structure:

Enhancing the BAPI by Including Additional Customer Database Tables

Transparent table

Short description

Fields	Key	Init.	Field type	Date	Length	DecPl.	Check table	Short description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_MANDT	CLNT	3	0	T000	Client for WB training data model
AGENCYNUM	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_AGNCYNUM	NUMC	8	0	SBUSPART	Number of travel agency
NAME		<input checked="" type="checkbox"/>	S_AGNCYNAM	CHAR	25	0		Name of travel agency
STREET			S_STREET	CHAR	30	0		Street for WB training data model
POSTBOX			S_POSTBOX	CHAR	10	0		PO box for WB training data mod
POSTCODE		<input checked="" type="checkbox"/>	POSTCODE	CHAR	10	0		Zip code for WB training data mo
CITY		<input checked="" type="checkbox"/>	CITY	CHAR	25	0		City for WB training data model
COUNTRY			S_COUNTRY	CHAR	3	0		Country code for WB training data
REGION			S_REGION	CHAR	3	0		Region of a country
TELEPHONE		<input checked="" type="checkbox"/>	S_PHONED	CHAR	30	0		Phone number of a customer
URL			S_URL	CHAR	50	0		URL of travel agency's home page
LANGU			SPRAS	LANG	1	0	T002	Language key

The Corresponding Help Structure

To simplify the filling and analysis of the extension parameters, each customer table should have a **data structure** containing the fields from the corresponding customer table that will be taken into account in the BAPI. In the process, make sure to observe the following:

- The individual fields of this data structure should have the same names and data types as the corresponding fields in the customer table.
- In particular, the data structure must contain all the key fields of the customer table, to enable unique identification of the data records.
- The name of the data structure should follow the naming convention <namespace>BAPI_<table_name> to allow a unique allocation to the database table.

The following diagram shows how help structure YBAPI_YSTRAVELAG should appear for this example:

Enhancing the BAPI by Including Additional Customer Database Tables

Structure

Short desc.

Component	Component type	DType	Length	DecPl	Short description
AGENCYNUM	S_AGENCYNUM	NUMC	8	0	Number of travel agency
NAME	S_AGENCYNAM	CHAR	25	0	Name of travel agency
STREET	S_STREET	CHAR	30	0	Street for WB training data model
POSTBOX	S_POSTBOX	CHAR	10	0	PO box for WB training data model
POSTCODE	POSTCODE	CHAR	10	0	ZIP code for WB training data model
CITY	CITY	CHAR	25	0	City for WB training data model
COUNTRY	S_COUNTRY	CHAR	3	0	Country code for WB training data model
REGION	S_REGION	CHAR	3	0	Region (of a country)
TELEPHONE	S_PHONED	CHAR	30	0	Phone number of a customer
URL	S_URL	CHAR	50	0	URL to home page of travel agency
LANGU	SPRAS	LANG	1	0	Language key

Important : Creating the help structure is optional, and only makes it easier to fill and evaluate the extension container. As an alternative, you can use other help structures or none at all.

Extension Parameters and BAPIs

As mentioned above, the two extension parameters (*ExtensionIn* and *ExtensionOut*) in the BAPI interface are used to pass on the enhancements to the BAPI function module or forward them from the function module to the calling program in container format.

In contrast to an enhancement using BAPI table extensions, there are no guidelines in this case for arranging the data in the extension containers.

If the help structure is used in the form described above, however, then the data structure in these containers is equivalent to the structure described under [Appending Customer Fields \[Page 14\]](#), with one exception: The only difference is that field STRUCTURE of structure BAPIPAREX passes on the name of the corresponding help structure instead of the name of the BAPI table extension.

Continuing the above example, the extension container could look like this if help structure YBAPI_YSTRAVELAG was used:

STRUCTURE	VALUEPART1		VALUEPART2		...
YBAPI_YSTRAVELAG	4711	BOEING747 LUFTH	ANSA	456 SEATS	...
YBAPI_YSTRAVELAG	8732	CONCORDE AIR FR	ANCE	745 SEATS	...
...

Enhancing the BAPI by Including Additional Customer Database Tables

The values "BOEING747", "LUFTHANSA", and "456 SEATS" in the first data record would be written to the fields of customer table YSTRAVELAG specified in YBAPI_YSTRAVELAG. The table line in which the values will be inserted is identified by key "4711".

The check and further processing of the data from the import container and the filling of the export container also take place in customer exits supplied by the BAPI developer.

Enhancing the BAPI with Import Data that Does Not Affect the Database Level

Purpose

In another feasible enhancement scenario, the customer wants to pass on additional data to the BAPI that is only required at runtime. The difference to the enhancement types described above is that these values are not assigned to any fields in database tables. As a result, the data has a different structure in the extension container, since no key values have to be passed on to determine the corresponding line in the database table.

Examples of this type of enhancement include:

- Adding additional selection criteria to a *GetList()* BAPI that limit the number of data records in the returned list
- Adding additional filter criteria to a *GetDetail()* BAPI that determine which detailed information is displayed
- Expanding the checkbox bar in a *Change()* BAPI that has been implemented according to the "Change after selection" concept

Implementation Considerations

The BAPI must contain the following items to support this type of enhancement:

1. **Extension parameters in the BAPI interface**
To pass on the additional data to the BAPI, the interface of the BAPI function module must have the *ExtensionIn* parameter.
2. **Customer exits**
For this case, as well, each BAPI function module must contain a maximum of two **extra exits** in addition to the customer exits provided by the application, in which the customer can perform data verification or other processing.

Therefore, while the BAPI interface must contain the *ExtensionIn* parameter and the customer exits, the order in which the transferred data is arranged in the extension container is entirely up to the customer.

We recommend creating help structures, however, whose fields represent the additional values. These data structures make evaluating the *ExtensionIn* parameter much easier and more reliable.

Example

A *GetList()* BAPI is used below to illustrate the steps required to add an additional selection criteria to a BAPI. Because BAPI selection parameters always consist of the four fields SIGN, OPTION, HIGH, and LOW, these four additional values must be passed on to the BAPI within the *ExtensionIn* container.

The Corresponding Help Structure

To simplify the filling and evaluation of the *ExtensionIn* parameter, you should create **data structures** that contain related fields that are to be passed on to the BAPI.

Enhancing the BAPI with Import Data that Does Not Affect the Database Level

In the example, you would create a data structure that contains the four fields SIGN, OPTION, HIGH, and LOW for the additional selection criteria.

Structure

Short desc.

Component	Component type	DType	Length	DecPl.	Short description
SIGN	BABISIGN	CHAR	1	0	Include/Exclude criterion SIGN
OPTION	BABIOPTION	CHAR	2	0	Selection operator OPTION for
HIGH	S_AGNCYNUM	NUM	8	0	Number of travel agency
LOW	S_AGNCYNUM	NUM	8	0	Number of travel agency

Important : Creating the help structure is optional, and only makes it easier to fill and evaluate the *ExtensionIn* container. As an alternative, you can use other help structures or none at all.

The ExtensionIn Parameter and BAPIs

As mentioned above, the *ExtensionIn* parameter in the BAPI interface is used to pass on the enhancements to the BAPI function module in container format.

In contrast to an enhancement using BAPI table extensions, there are no guidelines in this case for arranging the data in the *ExtensionIn* container.

If a help structure like the one described above were used, the data would have the following structure in the container:

STRUCTURE	VALUEPART1	VALUEPART2	...
YBAPI_RANGE_AGENCY	ICP0000001000000001		...
...

Field STRUCTURE of structure BAPIPAREX contains the name of the help structure that contains the fields required for selection. In this case, field SIGN would be set to value "I", OPTION would be set to "CP", HIGH would be set to "00000010", and LOW would be set to "00000001".

The check and further processing of the data from the import container also take place in customer exits supplied by the BAPI developer.

Actions by the BAPI Developer

Actions by the BAPI Developer

Purpose

BAPI developers must first supply the *ExtensionIn* and/or *ExtensionOut* parameters in the BAPI interface, as well as up to two customer exits in the BAPI function module, according to the requirements of the specific BAPI. This allows the customers to implement all three types of enhancement with this BAPI.

Developers can also support customers further by anticipating which SAP database tables might sensibly be appended with customer fields, and which existing fields from other tables might be included in the BAPI.

Process Flow

The example below explains which actions a BAPI developer would have to perform for a specific case. In this example, we will design an extendable BAPI *TravelAgency.CreateFromData* and provide for specific enhancements to database table STRAVELAG.

We can divide our activities into the following five steps:

1. Creating the extension parameter in the BAPI interface

Depending on the requirements of the BAPI in question, the BAPI developer creates one extension parameter for the data import (*ExtensionIn*) and/or one extension parameter for the data export (*ExtensionOut*) in the interface of the BAPI function module.

You must observe the following guidelines:

- The naming convention for the extension parameter is *ExtensionIn* for import parameter enhancements and *ExtensionOut* for export parameter enhancements.
- The extension parameters must be defined as table parameters.
- The extension parameters are always based on reference structure *BAPIPAREX*.
- All the BAPI table extensions used in the BAPI and their assignment to the respective BAPI parameters must be described in the documentation of the extension parameter.



The section [Use with the Standardized BAPIs \[Page 46\]](#) contains recommendations on which extension parameters to use for the individual standardized BAPIs.

2. Defining the BAPI table extensions

This step is only necessary if you plan to think about underlying SAP tables for the BAPI that the customers can sensibly enhance with their own fields and existing fields. If so, use transaction SE11 to create a BAPI table extension as a data structure for every SAP table that can be sensibly extended. The following guidelines have to be observed when creating BAPI table extensions:

- The naming convention for BAPI table extensions is BAPI_TE_<table_name>. If you cannot follow this naming convention due to the length restriction for structure names, then you must specify this explicitly in the documentation of the table extension and the BAPI.

Actions by the BAPI Developer

- Create one BAPI table extension for each table to be extended. Please note, however, that each table can only have one table extension. Therefore, if several BAPIs use the same table, they will have to share the table extension.

Exception: If a BAPI has two or more parameters that refer to the same database table, then a separate BAPI table extension is created for each of these BAPI parameters. The naming convention in this case is BAPI_TE<table_name><consecutive_number>. In our example, the table extensions would be called BAPI_TE_STRAVELAG1, BAPI_TE_STRAVELAG2, and so on.

- The structure of the BAPI table extension must contain all the identified key fields of the table to which the BAPI table extension relates. The table extension does not contain any other fields except the key fields.

In the above example, the BAPI table extension must be called BAPI_TE_STRAVELAG and possess key AGENCYNUM.

3. Modifying the BAPI function module

At the most, each BAPI function module can have the following structure:

```
FUNCTION BAPI_ERWEITERUNG_TEST
  <program text>
  <CUSTOMER_EXIT_1>
  <program text>
  <read ExtensionIn>
  <program text>
  <CUSTOMER_EXIT_2>
  <program text>
END FUNCTION
```

Therefore, in the most complex case, BAPI developers must allow customer enhancement to their function modules in three places:

Actions by the BAPI Developer– **Customer exit 1**

This customer exit is only required when the BAPI has an *ExtensionIn* parameter. This exit allows the customer to check all the data that is passed on to the BAPI, including the *ExtensionIn* parameter. The exit should have the following parameters:

- All BAPI import parameters and tables are passed on as an import. In particular, the *ExtensionIn* parameter must be available in the exit.
- The exit should return the *Return* parameter as an export, to allow the BAPI to report any errors that occurred in the exit.
- If required by the application, additional parameters can also be passed on to the exit as import or export.

– **Reading *ExtensionIn***

This step is only required when the customer has added their own fields to an SAP table or included existing fields in the BAPI, and the developer wants to make sure that the enhancements passed on in the *ExtensionIn* are automatically written to the additional table fields. This means no customer programming is required to write the customer enhancements directly to the corresponding database tables.

In particular, you should program this automatic filling of the extended fields in the case of *Create()*, *Add<sub-object>()*, and *SaveReplica()* BAPIs.

The extraction of a data record from the container is performed in two steps:

- a) The container format is first converted to the required temporary structure by the corresponding BAPI table extension. Because the values have the same sequence in both the container and the BAPI table extension, this can be implemented with a single MOVE command.
- b) This structure is then used to fill the additional database fields with MOVE-CORRESPONDING. To do this, the corresponding key value must be used to determine the line in the database where the values will be inserted.

The entire process can be described in pseudo-coding as follows:

```
L
O
O
P
a
t
E
x
t
e
n
s
i
o
n
I
n
```

```
IF required table extension exists in ExtensionIn
```

Actions by the BAPI Developer

```
THEN MOVE values from ExtensionIn TO table extension END.

Find correct line in database table.
Fill database table with table extension
using MOVE-CORRESPONDING.
```

E
N
D
L
O
O
P

Therefore the table enhancements can only be filled *automatically* when the fields the customer added to the BAPI table extension correspond exactly with the fields of the extended table, as the MOVE-CORRESPONDING command only works in this case.

Our concrete example results in the following coding:

```
l
o
o
p
a
t
e
x
t
e
n
s
i
o
n
i
n
.

case extensionin-structure.
  when 'BAPI_TE_STRAVELAG'.

    move extensionin+c_lenstruc to wa_bapi_te_stravelag.
    * Constant c_lenstruct possesses the length of field
    * STRUCTURE of structure BAPIPAREX. This constant
    * ensures that only the values from VALUEPART1 through
    * VALUEPART4 are assigned to the table extension .
    read table t_stravelag with key agencynum = agencynumber.
    catch system-exceptions conversion_errors = 1.

      move-corresponding wa_bapi_te_stravelag to t_stravelag.
    endcatch.
```

Actions by the BAPI Developer

Exception: In the case of *Create()* BAPIs with internal number assignment, the value of the key is not yet known when the BAPI is called, and therefore cannot be included in the container. As a consequence, the key value generated in the BAPI must be explicitly assigned to the table extension after the container has been extracted. In this case, we would modify the above example as follows:

```
l
o
o
p
a
t
e
x
t
e
n
s
i
o
n
i
n
.
```

```
case extensionin-structure.
  when 'BAPI_TE_STRAVELAG'.
    move extensionin+c_lenstruc to wa_bapi_te_stravelag.
    * We now pass the generated key on to the
    * table extension :
    move agencynumber to wa_bapi_te_stravelag-agencynum.
    read table t_stravelag with key agencynum = agencynumber.
    catch system-exceptions conversion_errors = 1.
      move-corresponding wa_bapi_te_stravelag to t_stravelag.
    endcatch.
```

– **Customer exit 2**

This customer exit is required when the customer is allowed to further process the data passed on with *ExtensionIn* or otherwise manipulate the database. If the customer has enhanced the export parameter of the BAPI, the second customer exit should be used to fill the *ExtensionOut* parameter. The exit should have the following parameters:

- The exit is given all BAPI import parameters, including *ExtensionIn*, as an import. Table parameters in the BAPI function module remain as tables in the customer exit.

Actions by the BAPI Developer

- The exit should return *ExtensionOut* and the *Return* parameter as an export.
- If required by the application, additional parameters can also be passed on to the exit as import or export.



The section [Use with the Standardized BAPIs \[Page 46\]](#) contains recommendations on which precautions should be taken for the respective underlying function modules.

A detailed programming example for implementing a BAPI module is available under [Examples \[Page 58\]](#).

4. **Filling *ExtensionIn* before a BAPI call**

If the coding supplied by SAP calls a BAPI, the developer has to decide whether or not to automatically fill the *ExtensionIn* parameter with the enhancements before the call. In this case, the filling of *ExtensionIn* has to be programmed explicitly. The concrete data flow for filling *ExtensionIn* is described under [Actions by the Customer \[Page 36\]](#).

5. **Evaluating *ExtensionOut* after a BAPI call**

If the coding supplied by SAP calls a BAPI, the developer can make sure that the *ExtensionOut* parameter is evaluated automatically. Please note, however, that evaluations of this type can only be anticipated in rare cases.

Actions by the Customer

Actions by the Customer

Purpose

The actions required by the customer depend on the type of enhancement to the BAPI. The following sections describe the actions required for each type of enhancement.

See also:

[Actions for an Enhancement Based on Existing SAP Database Tables \[Page 37\]](#)

[Actions when Including Additional Customer Database Tables \[Page 44\]](#)

[Actions when Including Additional Import Data that Does Not Affect the Database Level \[Page 45\]](#)

Actions for an Enhancement Based on Existing SAP Database Tables

Purpose

We assume that the customer wants to append his own fields to SAP tables involved in a BAPI and/or include additional existing fields from these tables in the BAPI after the fact.

Process Flow

In this case, the following steps are required at most:

1. **Extending the database table**

The customer must perform the following actions for each SAP table to append that is involved in the BAPI:

- a. Create a data structure that contains the customer fields that will be added to the table.
- b. Then add an APPEND to the database table that includes the data structure created in a) above.

An example is available under [Appending Customer Fields \[Page 14\]](#).

2. **Identifying and/or creating the BAPI table extensions**

The customer has to make sure that each SAP table that is used for this type of enhancement has its corresponding BAPI table extension. This is done by reading the documentation on the BAPI extension parameters. If a required table extension has not been supplied by SAP, then the customer is responsible for creating it. Make sure to observe the following guidelines:

Actions for an Enhancement Based on Existing SAP Database Tables

- The naming convention for BAPI table extensions is <name>_TE<table name>. If you can not

Actions for an Enhancement Based on Existing SAP Database Tables

- Each table should only have one table extension. Therefore, if several BAPIs use the same table, they will

Actions for an Enhancement Based on Existing SAP Database Tables

E If a BAPI has two or more parameters that refer to the same database table, then a
x separate BAPI table extension is created for each of these BAPI parameters. The
c naming convention in this case is
e <namespace>BAPI_TE<table_name><consecutive_number>. In our example, our
pt table extensions would be named YBAPI_TE_STRAVELAG1,
io YBAPI_TE_STRAVELAG2, and so on.
n:

Actions for an Enhancement Based on Existing SAP Database Tables

- The structure of the BAPI table extension must contain all the identified key fields of the table to which the BAPI

Actions for an Enhancement Based on Existing SAP Database Tables**3. Enhancing the BAPI table extensions**

The APPEND technique is now used to enhance the BAPI table extensions that you identified and/or created above. As mentioned above, we can differentiate between the following cases:

- a. If the customer wants to add additional fields from the SAP table for the BAPI table extension, he adds these fields directly to the APPEND structure.
- b. If the customer has appended his own fields to the corresponding SAP table, then he should include the same data structure – with the customer fields – in the APPEND as the one used to extend the database table.
- c. If you want to both add existing fields from the SAP table to the BAPI and append customer fields, proceed as follows:
 - First add all existing fields from the corresponding table to the APPEND structure.
 - Then add the INCLUDE that contains the new customer fields to the same APPEND structure. Please note that you have to use the same INCLUDE in both the database table and in the corresponding BAPI table extension.

4. Programming the customer exits

The first customer exit gives customers the option of performing data checks, while the second exit allows the further processing of the data passed on to the BAPI (also see [Actions by the BAPI Developer \[Page 28\]](#)). The standard rules for customer exits apply and transactions **SMOD (SAP internal)** and **CMOD (customers)** are used for maintaining them.

5. Filling *ExtensionIn* before a BAPI call

Before a BAPI that involves extended tables can be called, you have to make sure that the enhancements are written to parameter *ExtensionIn*. *ExtensionIn* is filled as follows for each data record that is written to the container:

- Field VALUEPART1 is first filled with the keys that identify the table lines.
- Fields VALUEPART1 through VALUEPART4 are then filled with the values of the customer fields in the proper sequence. By structuring the container in this way, its contents can be converted with a single MOVE command in the structure of the BAPI table extension.

The part for implementation can be described in pseudo-coding as follows:

Actions for an Enhancement Based on Existing SAP Database Tables

```
F
O
R
e
v
e
r
y
e
x
t
e
n
d
a
b
l
e
t
a
b
l
e
X
```

```
REPEAT
```

```
    Fill ExtensionIn-VALUEPART1 with key values.
```

```
    Fill VALUEPART1 through VALUEPART4 with data in sequence.
```

```
    APPEND ExtensionIn.
```

```
UNTIL table X is processed
```

```
E
N
D
F
O
R
```

Exception: It is not possible to give the container a key value in Create() BAPIs with internal number assignment, as these values are generated in the BAPI itself. In this case, you should give the container an initial key value. The BAPI developer has already anticipated the assignment of the key generated in the BAPI to the BAPI table extension.

A detailed **programming example** on filling the ExtensionIn parameter and calling the enhanced BAPI is available in the [Examples \[Page 58\]](#).

5. Evaluating *ExtensionOut* after a BAPI call

The conversion of the *ExtensionOut* container to a structure may have to be implemented, depending on your specific situation.

Actions when Including Additional Customer Database Tables

Purpose

We assume that the customer wants to process his own tables within the BAPI.

Process Flow

In this case, the following steps are required at most:

- 1. Creating the customer tables**
If you have not already done so, create the customer tables in the customer namespace.
- 2. Creating help structures**
Every customer table should have a help structure that contains all the fields of the customer table that will be used in the BAPI. As mentioned above, this step is optional and is only used to ensure that the extension container has a clear, uniform structure.
More information on help structures is available in [Enhancing the BAPI by Including Additional Customer Database Tables \[Page 22\]](#).
- 3. Programming the customer exits**
In this case, as well, the first customer exit gives customers the option of performing data checks, while the second exit allows the further processing of the data passed on to the BAPI (also see [Actions by the BAPI Developer \[Page 28\]](#)). The standard rules for customer exits apply and transactions **SMOD** (**SAP internal**) and **CMOD** (**customers**) are used for maintaining them.
- 4. Filling *ExtensionIn* before a BAPI call**
The customer can generally use any structure for the data to transfer in the *ExtensionIn* container. If help structures are used, the structuring described in [Actions for an Enhancement Based on Existing SAP Database Tables \[Page 37\]](#) is available.
- 5. Evaluating *ExtensionOut* after a BAPI call**
The conversion of the *ExtensionOut* container to a structure may have to be implemented, depending on your specific situation.

Actions when Including Additional Import Data that Does Not Affect the Database Level

Purpose

We assume that the customer wants to use additional import data that is only required at runtime.

Process Flow

This type of enhancement requires the following steps at most:

- 1. Creating help structures**
In the first step, create data structures that contain the additional, related fields that are to be passed on to the BAPI. As mentioned above, this step is optional and is only used to ensure that the extension container has a clear, uniform structure.
More information on the help structures is available under [Enhancing the BAPI with Import Data that Does Not Affect the Database Level \[Page 26\]](#).
- 2. Programming the customer exits**
In this case, as well, the first customer exit gives customers the option of performing data checks, while the second exit allows the further processing of the data passed on to the BAPI (also see [Actions by the BAPI Developer \[Page 28\]](#)). The standard rules for customer exits apply and transactions **SMOD** (**SAP internal**) and **CMOD** (**customers**) are used for maintaining them.
- 3. Filling *ExtensionIn* before a BAPI call**
The customer can generally use any structure for the data to transfer in the *ExtensionIn* container. If help structures are used, the type of structuring described in Section 2.3.2.1 is available.

Use with the Standardized BAPIs

Use with the Standardized BAPIs

This section describes the precautions that BAPI developers have to take to ensure that every standardized BAPI can be used for customer enhancements. Please note: this information is a **recommendation**, which can be altered in justified cases.



If the BAPI to enhance is not a standardized BAPI, you should follow the recommendations for the BAPI whose character is most similar to the one at hand.

GetList()

- A *GetList()* BAPI should contain an *ExtensionIn* parameter to allow customers to extend the selection parameters.
- An *ExtensionOut* parameter is required to allow additional values to be returned.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.
- Supply both customer exits, because data checks may be necessary and the *ExtensionOut* parameter can be filled.
- Do not use MOVE CORRESPONDING to read the *ExtensionIn* parameter, because no data is written to extended tables.

GetDetail()

- Create an *ExtensionIn* parameter to allow the definition of additional filter criteria that determine the scope of the details.
- An *ExtensionOut* parameter is required to allow additional detail values to be returned.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.
- Supply both customer exits, because data checks may be necessary and the *ExtensionOut* parameter can be filled.
- Do not use MOVE CORRESPONDING to read the *ExtensionIn* parameter, because no data is written to extended tables.

Create() and CreateFromData()

- A *Create()* BAPI should have an *ExtensionIn* parameter to allow customers to define additional fields in the object to create.
- An *ExtensionOut* parameter is only required if additional export parameters exist besides the generated keys and the *return* parameter. This is primarily the case when new tables are generated generically in the BAPI.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.

Use with the Standardized BAPIs

- Both customer exits must be supplied. On one hand, the customer must be capable of performing data checks; on the other hand, the customer might have defined or extended tables for the sub-object to create that were not supplied by SAP. These would have to be filled in the second exit.
- You must use a MOVE CORRESPONDING to read the *ExtensionIn* parameter for each table that has been assigned to the object to create and for which a table extension has been defined.

Change()

- A *Change()* BAPI should have an *ExtensionIn* parameter, since the customer might have defined additional fields in the object to create.
- An *ExtensionOut* parameter is only required if additional export parameters exist besides the *return* parameter. Please note, however, that this should be the exception.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.
- Both customer exits must be supplied. On one hand, the customer must be capable of performing data checks; on the other hand, the customer might have defined or extended tables for the sub-object to create that were not supplied by SAP. These would have to be filled in the second exit.
- Do not use MOVE CORRESPONDING to read the *ExtensionIn* parameter, because it makes no sense to overwrite the extended fields without further checks.

Cancel()

- No *ExtensionIn* parameter is required, since usually only key fields of the objects are passed on to the BAPI.
- If you want to return additional detail values, an *ExtensionOut* parameter will also be required. Please note, however, that this should be the exception.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.
- Only provide the second customer exit (if any), to allow data to be deleted from customer tables and fill the *ExtensionOut* parameter.
- Do not use MOVE CORRESPONDING to read the *ExtensionIn* parameter, because no data is written to extended tables.

Delete() and Undelete()

- No *ExtensionIn* parameter is required, since usually only key fields of the objects are passed on to the BAPI.
- If you want to return additional detail values, an *ExtensionOut* parameter will also be required. Please note, however, that this should be the exception.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.

Use with the Standardized BAPIs

- Only provide the second customer exit (if any), to allow data to be deleted from customer tables and fill the *ExtensionOut* parameter.
- Do not use MOVE CORRESPONDING to read the *ExtensionIn* parameter, because no data is written to extended tables.

Add<sub-object>()

- An *Add<sub-object>()* BAPI should have an *ExtensionIn* parameter to allow customers to define additional fields in the sub-object to create.
- An *ExtensionOut* parameter is only required if additional export parameters exist besides the generated keys and the *return* parameter. This is primarily the case when new tables are generated generically in the BAPI.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.
- Both customer exits must be supplied. On one hand, the customer must be capable of performing data checks; on the other hand, the customer might have defined or extended tables for the sub-object to create that were not supplied by SAP. These would have to be filled in the second exit.
- You must use a MOVE CORRESPONDING to read the *ExtensionIn* parameter for each table that has been assigned to the sub-object to create and for which a table extension has been defined.

Remove<sub-object>()

- No *ExtensionIn* parameter is required, since only the values that identify the sub-object are passed on to the BAPI.
- If you want to return additional detail values, an *ExtensionOut* parameter will also be required. Please note, however, that this should be the exception.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.
- Only provide the second customer exit (if any), to allow data to be deleted from customer tables and fill the *ExtensionOut* parameter.
- Do not use MOVE CORRESPONDING to read the *ExtensionIn* parameter, because no data is written to extended tables.

Replicate()

- A *Replicate()* BAPI should contain an *ExtensionIn* parameter to allow customers to extend the selection parameters.
- An *ExtensionOut* parameter is required to allow additional values to be returned.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.
- Supply both customer exits, because data checks may be necessary and the *ExtensionOut* parameter can be filled.

Use with the Standardized BAPIs

- Do not use MOVE CORRESPONDING to read the *ExtensionIn* parameter, because no data is written to extended tables.

SaveReplica()

- A *SaveReplica()* BAPI should contain an *ExtensionIn* parameter to allow customers to replicate objects with additional fields.
- An *ExtensionOut* parameter is only required if additional export parameters exist besides the generated keys and the *return* parameter. This is primarily the case when new tables are generated generically in the BAPI.
- A BAPI table extension should be created for every database table that the BAPI developer thinks could be enhanced. Make sure to list all the table extensions used in the BAPI in the documentation of the extension parameters.
- Both customer exits must be supplied. On one hand, the customer must be capable of performing data checks; on the other hand, the customer might have defined or extended tables for the sub-object to create that were not supplied by SAP. These would have to be filled in the second exit.
- Whether or not a MOVE CORRESPONDING is required to read the *ExtensionIn* parameter or not depends on the type of the involved *SaveReplica()* BAPI:
 - If you want to save the objects to replicate completely, without considering any existing data, then program the read for each table that has been assigned a BAPI table extension.
 - If you only want to transfer individual fields of the object to replicate, depending on specific conditions, then do not program the read.

BAPI Modifications

BAPI Modifications

Purpose

If customers want to functionally enhance a BAPI supplied by SAP by adding new parameters, changing existing parameters, or changing the coding of the function module upon which the BAPI is based, then this is only possible through modifications.

BAPI modifications cannot be made to business object types delivered in the standard system. Instead, you have to create a subtype of the business object type and then redefine the BAPI method inherited from the superior business object for this subtype.

Only the redefined BAPI is then available for the subtype; the original inherited BAPI can only be called from the supertype.

IBUs, customers and partners who want to create new development objects required in the context of a modification should follow the general guidelines in the documents “Changing the SAP Standard” and “Guidelines for Modifying the SAP Standard”, which are available in SAPNet.

You will also have to make sure that the involved development objects are assigned names from the customer namespace.

Process Flow

The process flow for customer modification of BAPIs:

1. Create a subtype for the involved business object type
The procedure for creating a subtype is described under [Creating Subtypes \[Page 55\]](#).
2. Change the definition of the inherited BAPI for the subtype.
 - You can add additional parameters, for example, or modify the implementation of the BAPI. The following implementation options are available for modifications:
 - Creating a new function module
 - Compatible modifications to the superior function module
 - Creating the implementing function module as a copy of the superior function module
3. If modifications have been made, you must **redefine the changed BAPI for the subtype in the BOR**. The steps required to redefine a BAPI are described in [Redefining BAPIs \[Page 56\]](#)
4. From the customer perspective, it makes sense when the superior BAPI can be called as the supertype as well as the subtype. To achieve this, you have to define a **delegation relationship** between the supertype and the subtype. To define a delegation relationship, perform the steps described under [Defining Delegation Relationships \[Page 57\]](#).

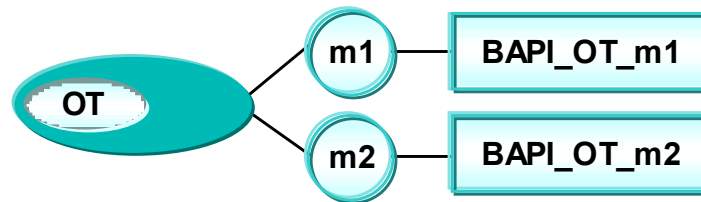
A delegation relationship is only significant for object-oriented BAPI method calls and not if the implementing function module is called directly.

Overview of Modification Options

The various ways to implement a modification are illustrated in the example below:

BAPI Modifications

Object type OT has two BAPIs m1 and m2, which are implemented by the function modules BAPI_OT_m1 or BAPI_OT_m2:

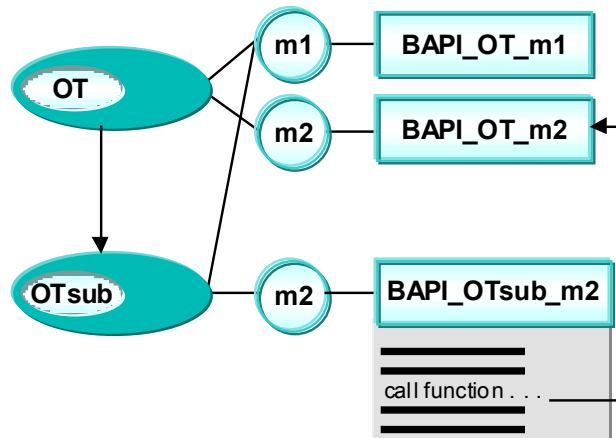


If enhancements to BAPI m2 are required, you can use the following implementation options:

Creating a new function module

In this case, a new function module (BAPI_OTsub_m2) is created to call the function module of the superior BAPI method (of the supertype). The original functionality is retained and can be enhanced for customer needs.

The diagram below illustrates the implementation:



If you use this method, when releases are upgraded, the system checks which interface enhancements have been carried out on the original BAPI. For example, if the interface of the original BAPI BAPI_OT_m2 is enhanced, the call in BAPI BAPI_OTsub_m2 must also be enhanced if the enhanced functionality of the original BAPI is to be used.

BAPI Modifications

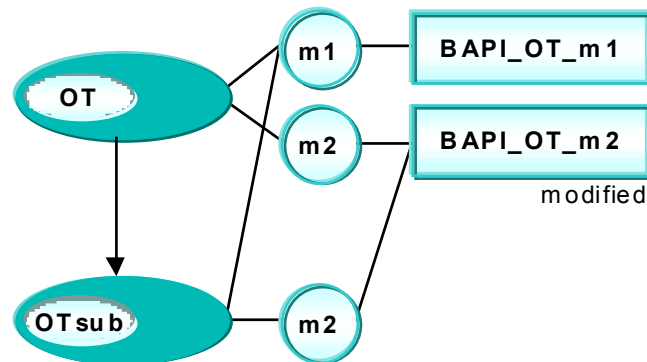
The advantage of using this method is that the original superior BAPI (BAPI_OT_m2), whose stability has been guaranteed, is cleanly encapsulated. Any changes carried out in the future to the original superior BAPI which do not affect the interface or semantics (for example, corrections and optimizations), are automatically copied to the new BAPI without requiring the subtype to be changed in any way.

This method is not a good choice, however, if the encapsulation has a negative effect on system performance. This could be the case, for example, if both BAPI_OT_m2 and BAPI_OTsub_m2 contain UPDATE operations for different fields of the same data record, and the developer wants to implement this in one single update record (that is, with one UPDATE).

Compatible modifications to the superior function module

In principle, the BAPI method of a subtype can be implemented by the same function module as the method of the supertype (BAPI_OT_m2 in this example). This is possible if the modifications of the interface are compatible, for example, if only new optional parameters have been added.

The diagram below illustrates the implementation:



With this implementation method, remember that the modifications must be merged with the original BAPIs every time a release is upgraded. You can use the Modification Assistant for this starting in Release 4.5A.

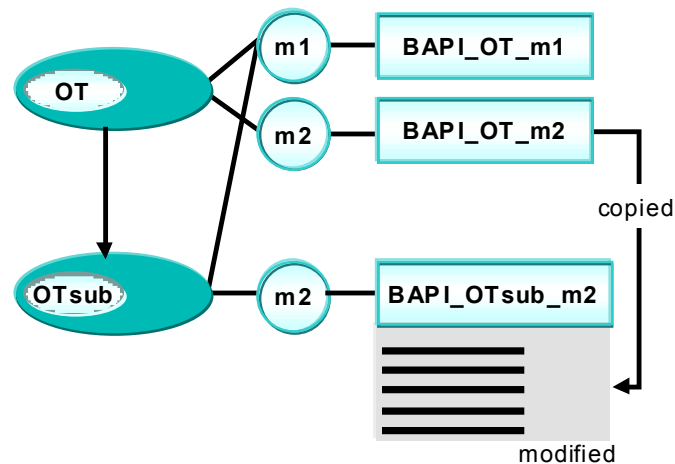
The advantage of using this option is that the original code and enhancement can be easily compared with one another, to achieve, for example, optimal performance of database UPDATE operations.

The main disadvantage is that the encapsulation that was achieved by separating the supertype and subtype for the BAPI definition is revoked. When changes have been made, you must ensure that the function module continues to behave the same when the original method of the supertype **OT** is called. This option can cause problems, for example, if several subtypes of an object type exist in the same system – in the same development system or for several industry solutions in one system – even though they have been created independently of each other (for different industry solutions, for example).

Creating the Implementing Function Module as a Copy of the Superior Function Module

In this case, a copy of the function module of the superior BAPI method is created (in this example, BAPI_OT_m2).

The diagram below illustrates the implementation:



You should only use this option if the enhancement cannot be implemented by calling the function module of the superior BAPI method. If global data is defined or form routines are called in the function group associated with a BAPI function module, you may have to also copy these data declarations and form routines to ensure the copy of the BAPI can be executed.

Although it is more flexible to modify a copy of the function module, you still have to check after every release upgrade whether changes have been made to the implementation of the superior BAPI method. If changes have been made, you have to copy them to the modified copy. As the origin of the copy is not always supported, we recommend that you do not use this option.

Once you have made the required modification to the copy of the function module, assign it as a method to the relevant business object type in the Business Object Repository using the BOR/BAPI Wizard.

See also:

- [Example \[Page 54\]](#)
- [Creating Sub-Types \[Page 55\]](#)
- [Redefining a BAPI \[Page 56\]](#)
- [Defining Delegation Relationships \[Page 57\]](#)

Example

Example

The BAPI *Material.GetList()* for the business object type *Material* should contain additional parameters. The function module associated with this BAPI is BAPI_MATERIAL_GETLIST.

Customers should first create a sub-type *YMaterial* of the existing object type *Material* in the BOR. They will also note that method calls of the business object type *YMaterial* for the original object type *Material* are to be processed, provided that the associated method is implemented there. This process is referred to as “delegation”. (For all methods that are not implemented for the subtype *YMaterial*, the method call is executed from the superior business object type, *Material*.)

Customers make the necessary changes in the source code of the function module BAPI_MATERIAL_GETLIST, or make changes to the interface by creating new, **optional** parameters.

Next, customers create the method *GetList()* for the sub-type *YMaterial* using the BOR/BAPI Wizard. The method is linked to the function module BAPI_MATERIAL_GETLIST, where the new parameters are included as part of the method definition.

If, on the other hand, the interface is enhanced with new, **mandatory** parameters, a new function module must be created and assigned to the method *YMaterial.GetList()*. (Customers can also copy the module BAPI_MATERIAL_GETLIST, for example, to Y_BAPI_MATERIAL_GETLIST, and then edit this copy.)

At runtime the following process takes place:

- When the BAPI *Material.GetList()* is called, the BAPI *YMaterial.GetList()* is the one actually executed.
- In the case of all other method calls, the methods of the superior business object type *Material* are executed, because these are not implemented for the sub-type *YMaterial*.

Creating Subtypes

To create a subtype:

1. From the SAP menu, choose *Tools* → *Business Framework* → *BAPI Development* → *Business Object Builder*, or enter the transaction code `SWO1`.
2. In the field *Object type* enter the name of the subtype you want to create, and choose *Create*.
3. In the next dialog box, enter the required details:
 - In the field *Supertype*, enter the name of the object type for which you want to create a subtype.
 - In the field *Object type*, enter the name of the subtype you want to create.
 - Enter appropriate values in the remaining fields.

Redefining a BAPI

Redefining a BAPI

To redefine a BAPI of a subtype, perform the following steps::

1. Choose *Tools* → *Business Framework* → *BAPI Development* → *Business Object Builder* or enter transaction code **SWO1**.
2. Display the subtype just created in the change mode.
3. Position the cursor on the BAPI you want to modify and choose *Edit* → *Redefine*.
4. Double-click on the BAPI and select the *ABAP* register.
5. In the *Name* field of the modified function module enter and save your information.

Defining Delegation Relationships

To define the delegation relationship between the supertype and subtype:

1. From the SAP menu, choose *Tools* → *Business Framework* → *BAPI Development* → *Business Object Builder*, or enter the transaction code `SWO1`.
2. Choose *Settings* → *Delegate* → *System-wide*.
3. Switch to the change mode and select *New entries*.
4. Enter the name of the original object type (supertype) in the field *Object type* and the name of the subtype in the field *Delegation type*.
5. Deactivate the check box *GUI-specific*.
6. Save your entries.

Examples

Examples

[Example for Developing the BAPI Function Module \[Page 59\]](#)

[Example for Filling the ExtensionIn Parameter \[Page 63\]](#)

Example for Developing the BAPI Function Module

```

FUNCTION BAPI_TRAVELAGENCY_CREATE .
*# -----
*#*# Local interface:
*# IMPORTING
*#     VALUE(AGENCYDATA_IN) LIKE BAPISADTIN
*#     STRUCTURE BAPISADTIN
*# EXPORTING
*#     VALUE(AGENCYNUMBER) LIKE BAPISADETA-AGENCYNUM
*# TABLES
*#     EXTENSIONIN STRUCTURE BAPIPAREX OPTIONAL
*#     RETURN STRUCTURE BAPIRET2 OPTIONAL
*# -----

*   global buffer is t_sbuspart, t_stravelag
*****

* length of name field for extension table in extension structure
CONSTANTS:
  C_LENSTRUC  TYPE I VALUE 30.

* work areas for the table extension structures
DATA:
  WA_BAPI_TE_SA LIKE BAPI_TE_SA,
  WA_BAPI_TE_SP LIKE BAPI_TE_SP.

* clear workareas for database tables
CLEAR T_SBUSPART.
CLEAR T_STRAVELAG.
*****

*   perform authority checks
*****

*   check the incoming data of SAP parameter AGENCYDATA_IN
PERFORM CHECK_AGENCYDATA_IN TABLES RETURN
      USING AGENCYDATA_IN.
CLEAR RETURN.
LOOP AT RETURN WHERE TYPE = 'E' OR TYPE = 'A'.
  EXIT.
ENDLOOP.
IF RETURN-TYPE = 'E' OR RETURN-TYPE = 'A'.
  EXIT.
ENDIF.
*****

*   Here a customer exit should be provided to check all data
*   including the EXTENSIONIN parameter.
*   This exit should be realized with the new exit technology,
*   however this technology is still under construction.
*   This example will be extended as soon as this technology is
*   available.
*   The Exit should have the following parameters:
*   -> All BAPI-parameters: AGENCYDATA_IN, EXTENSIONIN
*   <- RETURN
*   The function module should pass the RETURN parameter from
*   the exit to the client. 'E' means that the client can

```

Example for Developing the BAPI Function Module

```

*   'commit' the BAPI call, if there is an 'A' message in the
*   RETURN parameter, the client should 'rollback work'.
*****
*   get the missing id's STRAVELAG-ID & SBUSPART-BUSPARTNUM
CALL FUNCTION 'NUMBER_GET_NEXT'
  EXPORTING
    NR_RANGE_NR      = '01'
    OBJECT           = 'SBUSPID'
    QUANTITY         = '1'
  IMPORTING
    NUMBER           = AGENCYNUMBER
  EXCEPTIONS
    INTERVAL_NOT_FOUND = 1
    NUMBER_RANGE_NOT_INTERN = 2
    OBJECT_NOT_FOUND   = 3
    QUANTITY_IS_0      = 4
    QUANTITY_IS_NOT_1  = 5
    INTERVAL_OVERFLOW  = 6
    OTHERS              = 7.
IF SY-SUBRC <> 0.
  RETURN-TYPE = 'E'.
  RETURN-ID = 'BCTRAIN'.
  RETURN-NUMBER = '601'.
  RETURN-MESSAGE_V1 = TEXT-500.
  APPEND RETURN.
  EXIT.
ENDIF.
*****
*   Before adding data to the buffer, subscribe the function module
*   to delete this buffer in case of 'rollback work' by the client.
CALL FUNCTION 'BUFFER_SUBSCRIBE_FOR_REFRESH'
  EXPORTING
    NAME_OF_DELETEFUNC = 'SAGENCY_REFRESH_BUFFER'.
*****
*   Fill global buffer for table STRAVELAG.
MOVE-CORRESPONDING AGENCYDATA_IN TO T_STRAVELAG.
T_STRAVELAG-AGENCYNUM = AGENCYNUMBER.
APPEND T_STRAVELAG.
*   Fill global buffer for table SBUSPART.
T_SBUSPART-BUSPARTNUM = AGENCYNUMBER.
T_SBUSPART-CONTACT = T_STRAVELAG-NAME.
T_SBUSPART-CONTPHONO = T_STRAVELAG-TELEPHONE.
T_SBUSPART-BUSPATYP = 'TA'.
APPEND T_SBUSPART.
*   Collect the appended fields of tables STRAVELAG & SBUSPART.
LOOP AT EXTENSIONIN.
  CASE EXTENSIONIN-STRUCTURE.
    WHEN 'BAPI_TE_SA'.
      MOVE EXTENSIONIN+C_LENSTRUC TO WA_BAPI_TE_SA.
*   check if table entry to extension entry exist. for that
*   we fill the internal given number into the extension table.
      MOVE AGENCYNUMBER TO WA_BAPI_TE_SA-AGENCYNUM.
      READ TABLE T_STRAVELAG
        WITH KEY AGENCYNUM = AGENCYNUMBER.
      CATCH SYSTEM-EXCEPTIONS CONVERSION_ERRORS = 1.
      MOVE-CORRESPONDING WA_BAPI_TE_SA TO T_STRAVELAG.

```

Example for Developing the BAPI Function Module

```

ENDCATCH.
IF SY-SUBRC <> 0.
  RETURN-TYPE = 'E'.
  RETURN-ID = 'BCTRAIN'.
  RETURN-NUMBER = '888'.
  RETURN-MESSAGE_V1 = TEXT-800.
  RETURN-MESSAGE_V2 = SY-TABIX.
  RETURN-MESSAGE_V3 = TEXT-900.
  APPEND RETURN.
  EXIT.
ENDIF.
MODIFY T_STRAVELAG INDEX SY-TABIX.
WHEN 'BAPI_TE_SP'.
  MOVE EXTENSIONIN+C_LENSTRUC TO WA_BAPI_TE_SP.
  MOVE AGENCYNUMBER TO WA_BAPI_TE_SP-BUSPARTNUM.
  READ TABLE T_SBUSPART
    WITH KEY BUSPARTNUM = AGENCYNUMBER.
  CATCH SYSTEM-EXCEPTIONS CONVERSION_ERRORS = 1.
  MOVE-CORRESPONDING WA_BAPI_TE_SP TO T_SBUSPART.
ENDCATCH.
IF SY-SUBRC <> 0.
  RETURN-TYPE = 'E'.
  RETURN-ID = 'BCTRAIN'.
  RETURN-NUMBER = '888'.
  RETURN-MESSAGE_V1 = TEXT-700.
  RETURN-MESSAGE_V2 = SY-TABIX.
  RETURN-MESSAGE_V3 = TEXT-900.
  APPEND RETURN.
  EXIT.
ENDIF.
MODIFY T_SBUSPART INDEX SY-TABIX.
ENDCASE.
ENDLOOP.

IF RETURN-TYPE = 'E'.
*****
*   If an error of type 'E' occurred, we will have to make the
*   buffer consistent again by deleting the data from this call.
*   Errors of type 'A' can not occur at this point.
  DELETE T_SBUSPART WHERE BUSPARTNUM = T_SBUSPART-BUSPARTNUM.
  DELETE T_STRAVELAG WHERE AGENCYNUM = T_STRAVELAG-AGENCYNUM.
  EXIT.
ENDIF.
*****
*   Create agencies from global buffer in update task. This form will
*   be performed only once during the 'commit' statement of the
*   client, thus the data from all BAPI calls can be written to
*   the database with only one array insert for each table
  PERFORM CREATE_AGENCYCS ON COMMIT.
*****
*   Here a customer exit should be provided in some way
*   to insert the data from the ExtensionIn parameter.
*   This exit should be realized with the new exit technology,
*   however this technology is still under construction.
*   This example will be extended as soon as this technology is
*   available.

```

Example for Developing the BAPI Function Module

- * Within this exit, customer should modify db IN UPDATE TASK
- * to guarantee consistency of the data !
- * Be aware that nothing has happened on database yet so the
- * customer cannot read data from this or any previous call in
- * this LUW.
- * The customer could use buffers too, but has to be aware of
- * the refreshing concept!
- * -> All BAPI-parameters: AGENCYDATA_IN, EXTENSIONIN and the id
- * AGENCYNUMBER.
- * Nothing comes back!

```
* assume that everything is O.K., fill return parameters
RETURN-TYPE = 'S'.
RETURN-ID = 'BCTRAIN'.
RETURN-NUMBER = '603'.
RETURN-MESSAGE_V1 = TEXT-600.
RETURN-MESSAGE_V2 = T_STRAVELAG-AGENCYNUM.
RETURN-MESSAGE_V3 = TEXT-400.
APPEND RETURN.
ENDFUNCTION.
```

Example for Filling the ExtensionIn Parameter

REPORT REPORT_TRAVELAGENCY .

```

*
*
* Example to show how to fill the EXTENSION_IN Structure in a BAPI *
* which uses table extension as possibility for a customer to extend *
* SAP functionality without modification.
*
*
* In this example we use the BAPI
* BAPI_TRAVELAGENCY_CREATE
*

```

data x_agencynumber type bapisadeta-agencynum.

data x_agencydata_in type bapisadtin.

data x_cus_data_in type travelag.

data begin of x_extensionin occurs 0.

include structure bapiparex.

data end of x_extensionin.

data begin of x_return occurs 0.

include structure bapiret2.

data end of x_return.

data x_bapi_te_sa type bapi_te_sa.

* only for this example we hardcode the data, normally you would

* write an own transaction to fill this fields.

x_agencydata_in-name = 'Galactic Travel Agency'.

x_agencydata_in-street = 'Lunatic'.

x_agencydata_in-postbox = '123456'.

x_agencydata_in-postcode = '984735'.

x_agencydata_in-city = 'Luna 1'.

x_agencydata_in-country = 'Moon'.

x_agencydata_in-region = 'Moon'.

x_agencydata_in-telephone = '30457584375374957'.

x_agencydata_in-url = 'http://lunatic.com'.

x_agencydata_in-langu = 'E'.

Example for Filling the ExtensionIn Parameter

```
x_cus_data_in-planetype = 'Space Shuttle'.
x_cus_data_in-company   = 'Moon Shuttle Service'.
x_cus_data_in-seatsmax  = 24.
```

* now fill the key of the extension parameter

```
move 'BAPI_TE_SA' to x_extensionin-structure.
```

* now normally you fill the object key into the BAPI_TE_SA structure

* but in this case the create will give internal the key for the new

* travel agency, so here only a clear

```
clear x_bapi_te_sa-agencynum.
```

* but the other fields we have, so fill

```
move-corresponding x_cus_data_in to x_bapi_te_sa.
```

* fill the fields in the data part of x_extension in, take care that

* you have 960 bytes in pieces of 240 byte. Luckily we have less than

* 240 byte so we need only one move.

```
move x_bapi_te_sa to x_extensionin-valuepart1.
```

```
append x_extensionin.
```

```
CALL FUNCTION 'BAPI_TRAVELAGENCY_CREATE'
```

```
  EXPORTING
```

```
    AGENCYDATA_IN = x_agencydata_in
```

```
  IMPORTING
```

```
    AGENCYNUMBER = x_agencynumber
```

```
  TABLES
```

```
    EXTENSIONIN = x_extensionin
```

```
    RETURN      = x_return.
```

```
  .
```

```
  *
```

```
write : / 'Return messages in handling the bapi.'
```

```
loop at x_return.
```

```
  write : / 'return...:', x_return.
```

```
endloop.
```

```
write : / 'The new travel agency has the number...:', x_agencynumber.
```

SAP Enhancements to Released BAPIs

Purpose

Application developers who use BAPIs in their programs must be able to rely on the BAPI interface remaining the same. As a result of this, once a BAPI is released, it must fulfill certain requirements regarding the stability of its interface.

Whenever SAP enhances a BAPI, downward compatibility of syntax and contents must be guaranteed whenever possible. Downward compatibility means that applications that were programmed with BAPIs from a specific R/3 Release will not be affected in later R/3 Releases if the syntax or the content of this BAPI changes.

Examples of *syntax changes* are changes to parameter names, or changes to the type or length of a domain. The ABAP Dictionary can automatically test whether syntax changes are compatible.

Content changes are involved, for example, when existing coding of the BAPI function module is changed or new coding is added. Only the developer can ensure that content changes are downwardly compatible.

Accordingly, when you enhance a BAPI, you can differentiate between [a compatible enhancement \[Page 67\]](#) and [an incompatible enhancement \[Page 69\]](#), depending on whether the downward compatibility of the BAPI can be guaranteed.

To protect the stability of the interface, compatible enhancements are always preferred to incompatible enhancements.

For SAP internal development, each enhancement to a BAPI must be created in a project in the BAPI Explorer.

Integration

Tool Support for Enhancements

Tool support covers the following aspects:

- Changes and version management in the BOR
Changes made to a BAPI only take effect when the changes are defined in the Business Object Repository (BOR), that is, they have been saved and generated.
Version management of BAPIs is also carried out in the BOR.
- Checking in the ABAP Dictionary
Changes to the syntax of BAPIs are automatically checked by the ABAP Dictionary, thereby preventing the BAPI data structure being changed by mistake.
 - The ABAP Dictionary rejects incompatible changes to data elements, domains or structures that are being used by a BAPI that has been released.
 - Compatible changes or changes to data elements, domains or structures of a BAPI that has not been released are accepted by the ABAP Dictionary.

See also:

[Compatible Enhancements \[Page 67\]](#)

[Incompatible Enhancements \[Page 69\]](#)

SAP Enhancements to Released BAPIs

Compatible Enhancements

Use

Compatible enhancements are interface enhancements that change the BAPI without effecting the downward compatibility of the BAPI. Applications which access the BAPI are not affected by compatible enhancements.

Integration

Compatible enhancements are:

- *New optional parameters*
A parameter is considered to be optional if it does not have to be included in a BAPI call. A new optional parameter can be added in any place in the interface.



A new parameter is added to the BAPI `SalesOrder.GetList()`, which can be used as an additional selection criteria for selecting purchase orders.

- *New optional fields in structures*
A field is considered to be optional if it can be left out completely in a BAPI call. The fields must be added to the end of a structure. This is because the function module upon which the BAPI is based is called via RFC. It does not matter how the fields are arranged in a structure or table because the whole structure is always forwarded. It is not first broken up into fields.



An additional input field for the applicant's educational background is added to the BAPI `Applicant.CreateFromData()`.

The table below lists the compatible changes in the function module. We cannot guarantee that this list is exhaustive.

Compatible Changes to Function Modules	
In interface	New optional parameter as a field
	New optional parameter as a structure
	New optional parameter as a table
	Adding new optional field to the structure
	Adding new optional field to the table
	Compatible changes to field types (in the ABAP Dictionary)
In program code	Converting mandatory fields to optional fields
	New additional coding that does not involve changes to the interpretation/processing logic

Compatible Enhancements

	Changes to the existing code which do not involve changing the interpretation or processing logic
	Using <u>customer exits</u>

When making changes, be sure to follow the guidelines in the [BAPI Programming Guide \[Ext.\]](#).

Incompatible Enhancements

Purpose

Changes to the contents or functionality of a BAPI often result in the introduction of new parameters without which the interface can no longer function. Often, these changes also cause existing parameters to lose their original meaning. Such modifications are considered to be incompatible enhancements, because they no longer enable the BAPI to be downward compatible.

Syntactically incompatible enhancements are:

- Changes to the field length
- Changes to the field type
- Renaming parameters in the function module or in the method
- Inserting a field within a structure
- Deleting parameters and fields
- Inserting new mandatory parameters and fields
Parameters can be flagged as mandatory in the BOR. However, this is not the case with fields. Fields can only be categorized as mandatory at a semantic level and not at a technical level. This is why the documentation for each parameter must specify which fields can be filled.

The table below lists the incompatible changes to function modules. We cannot guarantee that this list is exhaustive.

Incompatible Changes to Function Modules	
In interface	New mandatory parameter
	Adding new fields between existing fields in the structure
	Adding new fields between existing fields in the table
	Adding new mandatory fields to the structure
	Adding new mandatory fields to the table
	Incompatible changes to field types (in the ABAP Dictionary)
	Changing optional fields to mandatory fields
	Renaming parameters
In program code	New additional source code that involves changes to the interpretation/processing logic
	Changes to the existing source code that involve changing to the interpretation/processing logic
	Adding or removing COMMIT WORK commands in the program

Incompatible Enhancements

Process Flow

In cases of incompatible changes to a BAPI, you should work through the following three steps: **Create an additional BAPI, Support and label the expiring BAPI** and **Delete the replaced BAPI**.

Create an Additional BAPI

To ensure that the interface stability of an existing BAPI is not impaired, you must not make any incompatible changes to the existing BAPI. Instead, create one or, if necessary, several additional BAPIs to replace the existing one.

The new BAPI must retain the same name as the BAPI to be replaced. A numeric suffix is simply added to it. This suffix changes if further incompatible changes are made.



A number of incompatible changes must be made to the BAPI *FixedAsset.Create()*. To implement these changes, a new BAPI, *FixedAsset.Create1()*, is created, in order to maintain the interface integrity of the BAPI *FixedAsset.Create()*.

If further incompatible modifications must be made to the BAPI at a later date, yet another BAPI, *FixedAsset.Create2()*, must be created.

When creating the additional BAPIs, you must follow the guidelines in the [BAPI Programming Guide \[Ext.\]](#).

Support and Label the Expiring BAPI

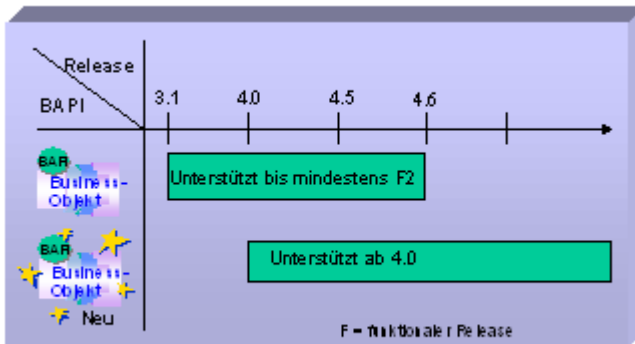
After the BAPI has been superseded by a new BAPI, you should not remove the replaced BAPI from the Business Object Repository (BOR). Instead, you first label this BAPI as expired, and continue its support in the release in which you have introduced the new BAPI as well as in the next functional release. During this time the original BAPI must remain fully functional and executable.

The following tasks are required when a BAPI has expired (become obsolete):

- Select the relevant SAP business object type in the Business Object Builder, and open the node *Methods*. Position the cursor on the BAPI, and choose *Edit* → *Change status to* → *Obsolete*.
- In the BOR, document which method(s) are to replace the expiring BAPI.
- Record the BAPIs that have been set to “obsolete” in a particular release in note number 0107644, “Collective Note for Obsolete BAPIs from Release 4.5A”, in SAPNet – R/3 Frontend.
- Inform your users about the relevant successor BAPIs in Release Notes.

The diagram below illustrates the expiry phase of BAPIs: In this example, the successor BAPI was implemented in Release 4.0. The replaced BAPI will therefore be supported in Release 4.0 (that is, in the correction release in which the successor BAPI was implemented) and in the next functional release, “F1”. In the following functional release “F2”, this BAPI will no longer be available.

BAPI Expiry Phase



Delete the Replaced BAPI

If the expiry phase of a BAPI set to obsolete has come to an end, you can remove the BAPI from the BOR. You should delete an obsolete BAPI as close to the beginning of a new release as possible, so that developers have time to adjust to its successor.

To delete a BAPI, follow the steps below:

- Delete the method from the BOR
To do this, display the relevant SAP business object type in the Business Object Builder in change mode. Expand the node *Methods*. Place the cursor on the appropriate BAPI, and delete it by choosing *Edit* → *Delete*.
- Delete the function module that implements the BAPI
In the Function Builder, enter the name of the function module in the *Function module* field, and choose *Function module* → *Other functions* → *Delete*.
- Record the release in which the BAPI was deleted from the BOR in the note number 0107644, "Collective Note for Obsolete BAPIs from Release 4.5A", in SAPNet – R/3 Frontend.

BAPIs for Mass Data Transfer (CA-BFA)

BAPIs for Mass Data Transfer (CA-BFA)

Purpose

When a customer implements a new R/3 System, an important part of the migration process is copying objects from the legacy system. To transfer this mass data, each business object type in the R/3 System currently has its own program for performing this initial data load with batch input, direct input, or other SAP developments (such as IS-B).

SAP developed the Data Transfer (DX) Workbench in Release 3.1 to give users a uniform startup screen for loading the different business object types, as well as simplified handling of standard files.

Starting in Release 4.6A, the DX Workbench allows BAPIs to be used for the data load in the R/3 System. The use of BAPIs is increasingly important, because the previous techniques are only of limited or no use for data transfer from Release 4.6 onwards:

- The batch input procedure cannot be used for the new Enjoy transactions because the batch input recorder does not support the controls used in these transactions. Until SAP implements a data transfer BAPI for these new transactions, SAP will continue to support the old transactions that do not yet use these controls. The disadvantage with this is that users have to deal with an additional transaction.
- The administration transaction associated with the direct input method will no longer be supported from Release 5.0 onwards (at the latest). This means that existing direct input programs can be used, but the data transfer should be converted to BAPIs in the medium term.

Target Audience

The target audience of this document consists of:

- *BAPI developers* who want to implement BAPIs capable of supporting mass data
The document describes which steps are required to use these BAPIs with the DX Workbench. In addition, the programming models of all standardized BAPIs that are relevant for mass data transfer are discussed in detail.
- *Potential users of mass data-capable BAPIs* who would like a conceptual overview of the process flow of the mass data transfer using BAPIs
Please note, however, that this document does not contain a description of any specific tool (for example the DX Workbench).

Implementation Considerations

To develop or use mass data-capable BAPIs, you need:

- Basic knowledge of BAPIs, as described in [BAPIs – Introduction and Overview \[Ext.\]](#)
- Detailed knowledge of BAPI development, as described in the [BAPI Programming Guide \[Ext.\]](#)
- Knowledge of the ABAP programming language and ABAP Workbench
- Basic knowledge of the R/3 System

See also:

[Basics of Mass Data Transfer \[Page 74\]](#)

[Process Flow of the Mass Data Transfer via BAPI \[Page 80\]](#)

[Developing BAPIs for Mass Data Transfer \[Page 83\]](#)

[Details \[Page 119\]](#)

Basics of Mass Data Transfer

Basics of Mass Data Transfer

Prerequisites

Before you begin the initial data load from a non-SAP system to an SAP system, you have to answer the following questions:

- Which SAP business object types are involved in the target system?

You have to determine which business object type can be allocated the imported data in the destination system. This is important because the business dependencies are mapped across an object model in the R/3 System. For example, if you want to transfer data involving sales processing, then business objects *Material*, *Customer*, and *Sales document* are involved.

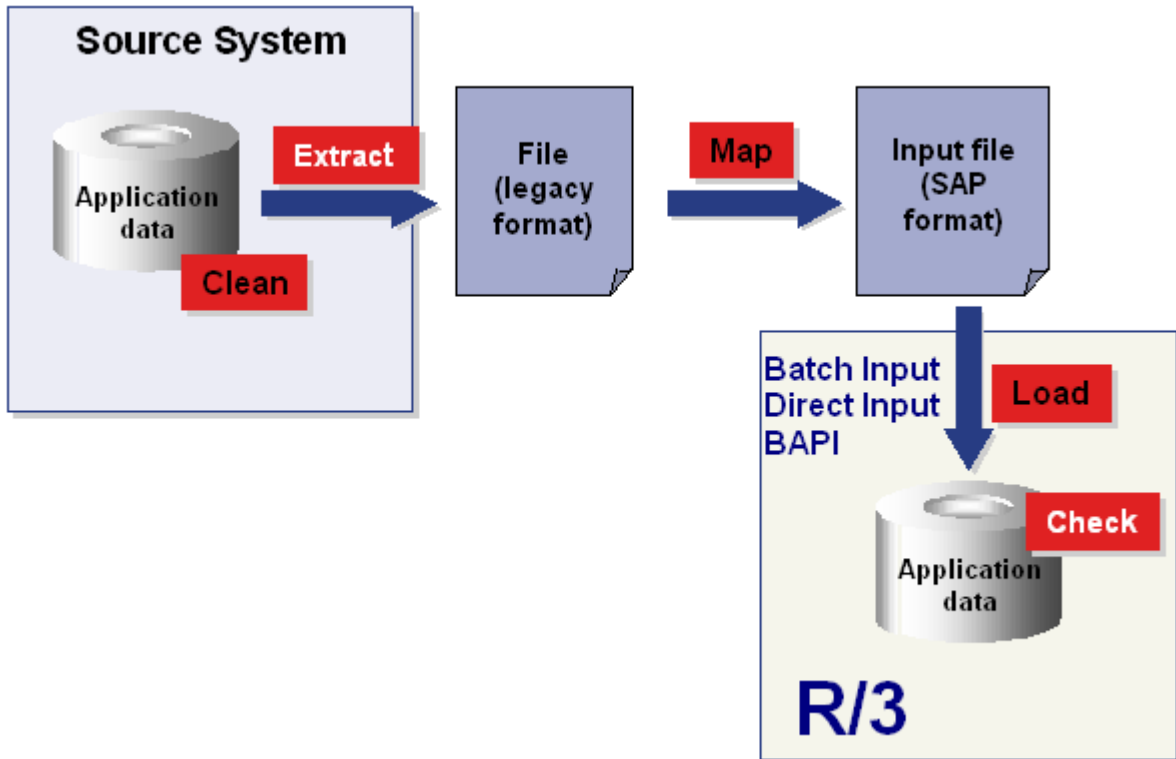
- How should the data be transferred to the R/3 System?

You have to determine the data transfer technique to use for each involved business object type. The techniques **BAPI**, **batch input**, and **direct input** are available for loading the data into the target system. For details of the individual techniques, please refer to the description of phase *Transferring the Data into the R/3 System*.

Process Flow

The key feature of the data transfer is that data can only be transferred from a non-SAP system to an R/3 System when it is available in SAP format. For this reason, the data from the non-SAP system initially has to be extracted into a file, and then copied into a transfer file in SAP format. In turn, this transfer file is the foundation for the actual data import into the R/3 System.

The entire data transfer usually takes place in five technical phases, which are illustrated in the following diagram.



These phases are described in detail below:

1. Analysis and cleanup of data in the non-SAP system

This phase involves answering three related questions:

- Which data exists in the non-SAP system?
- How is this data structured (length, sequence)?

This step is required to assign the transferred data to the correct SAP structures later.

- Which data can be transferred unchanged, which data has to be supplemented, and which data cannot be transferred at all?



A power company manages its customers for electricity, gas, and water in three separate systems. These three systems are to be merged in a single R/3 System. If customers exist who purchase both gas and water from this company, for example, then the data on these customers may have a different status in the different systems. For this reason, the customer data from the individual systems first has to be cleaned up and unified before it can be transferred to the R/3 System.

2. Extraction of data from the non-SAP system

In this step, the determined data is extracted from the non-SAP system to a file. The data in this file still has the format of the non-SAP system. The following options are available for extracting the data:

Basics of Mass Data Transfer

- Using an extraction tool supplied by the non-SAP system
- Using the LSM Workbench
- Using a database tool
- Using the extraction function of a mapping tool

3. Mapping the data in SAP format

Data from a non-SAP system can only be transferred to an R/3 System when it is available in SAP format. For this reason, the file containing the data in the non-SAP system format has to be transformed into a transfer file in SAP format.

Because the format of the data extracted from the non-SAP system usually differs from the SAP format of the transfer file, mapping between the different structures is required. To perform this mapping, you will have to analyze both the fields and structures of the data to transfer and the involved SAP structures, in order to determine how the conversion should be performed.

The following options are available for performing the actual mapping:

- Writing an ABAP program
- Using a commercial mapping tool
- Using SAP's mapping tool (available from Release 4.6C onwards)

4. Transferring the data to the R/3 System

Before you can import the data from the transfer file to the R/3 System, you have to answer the following questions:

- In which sequence will the source data be copied to the business object types, and in which sequence will the involved business object types be filled with data?

This question must be answered because dependencies may exist between business object types that require the data to be transferred in a specific order.

- Which data transfer technique will be used for the individual business object types?

The following three standard techniques are available for the data transfer:

Basics of Mass Data Transfer

- **BAPIs**

Calling a BAPI in the appropriate application transfers the data to the R/3 System. If BAPIs are used as interfaces to the SAP system, the same technique is used as for the continuous data transfer between R/3 Systems or between non-SAP systems and R/3 Systems via ALE.

For more information on data transfer using BAPIs, please see [Process Flow of the Mass Data Transfer using BAPIs \[Page 80\]](#).

- **Batch input**

Batch input is a standard technique used to transfer large volumes of data into the R/3 System. In the process, the transaction flow is simulated, and the data is transferred as if it were entered online. The advantage of this procedure is that all the transaction checks are performed, which guarantees data consistency.

For detailed information on batch input, please refer to the Basis documentation on batch input sessions under [Data Transfer: An Overview of Batch Input Folders \[Ext.\]](#).

Basics of Mass Data Transfer

- **Direct input**

During direct input, the data in the data transfer file is first subjected to various checks, and is then imported directly into the R/3 System. The R/3 database is updated directly with the transferred data.

Important: You should use BAPIs exclusively to transfer data in Release 4.6A and later.

5. Checking the data

After the data has been transported into the R/3-System, it must be checked for completeness. It must be ensured, for example, that the sum of the data transferred into the R/3 system is the same as the sum of data from the external system. These checks are often implemented through separate check programs that are run in the R/3 System. These programs can be supplied either by SAP or by an external provider.

Tool Support

SAP created the DX Workbench for Release 3.1 and later to support the transfer of mass data. Starting in Release 4.6A, this tool provides integrated project management for all the steps involved in transferring data to the R/3 System.

In particular, the Data Transfer Workbench offers the following functions:

- Managing and organizing data load projects
- Tools for analyzing the required SAP structures

Basics of Mass Data Transfer

- Integration of the standard data transfer programs
- Registration and integration of separate data transfer programs
- Support of various techniques for loading data into the R/3 System

For more information, please refer to the documentation for the DX Workbench under [Data Transfer Workbench \[Ext.\]](#).

Process Flow of the Mass Data Transfer via BAPI

Process Flow of the Mass Data Transfer via BAPI

Purpose

To perform the mass data transfer using BAPIs, SAP supplies the DX Workbench, which integrates all the required steps in a single user interface. This section describes at a conceptual level how the process flow of the mass data transfer via BAPI takes place in the DX Workbench. For a detailed description of the process flow, please refer to the documentation for the DX Workbench under [Data Transfer Workbench \[Ext.\]](#).

The mass data transfer via BAPI utilizes the same ALE technique that is used for continuous data transfer. This means the data to copy from the source system is imported into the target system in IDoc format. In the target system, the IDocs are used to generate BAPI calls that make sure the data is written to the database.

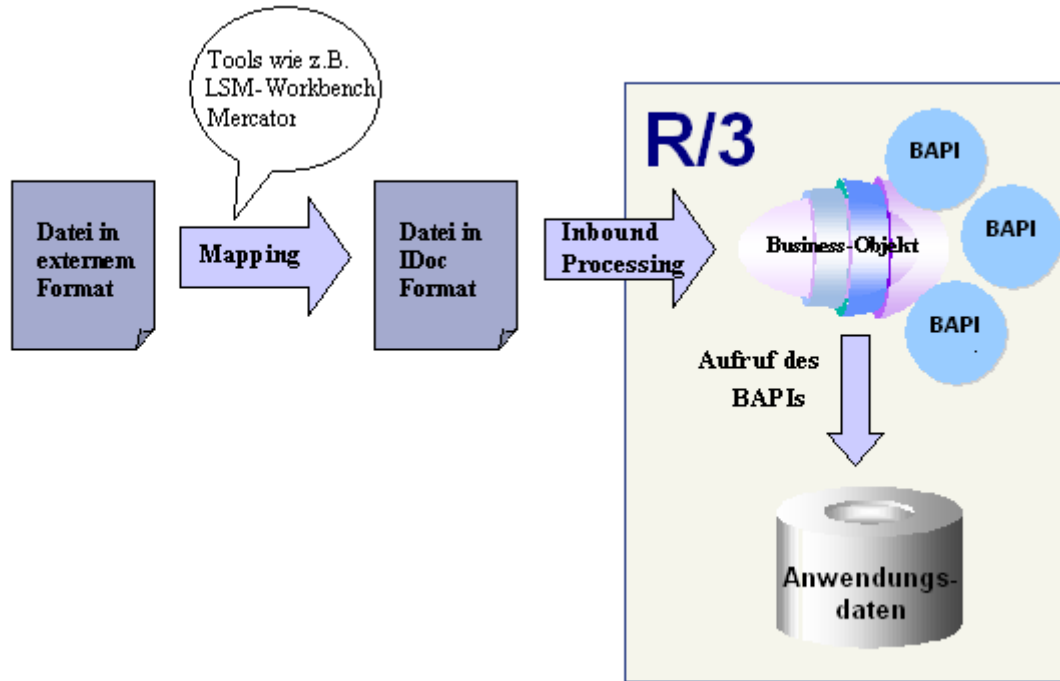
Before the mass data transfer can be performed, you have to identify the involved business objects in the target system and determine which BAPIs will be used to import the data. To determine which objects are involved, you have to establish a connection between the online transactions used during production operation and the underlying business objects.

For a BAPI to be suitable for mass data transfer, it has to have certain attributes, which are described under [Developing BAPIs for Mass Data Transfer \[Page 83\]](#).

In the description below, we assume that the data to transfer is already available in a file in the external format. The steps that resulted in the creation of this file are similar to those described in the [Basics of Master Data Transfer \[Page 74\]](#), and are therefore not repeated here.

The resulting process flow is illustrated in the diagram below and is described in more detail in the following.

Process Flow of the Mass Data Transfer via BAPI



Process Flow

If the data to transfer is available in a file with an external format, then we differentiate between **analyzing the underlying structures**, **mapping the data in IDoc format**, and **performing the actual data import**.

1. Analyzing the underlying structures

The transfer file into which the data is imported from the external file and which forms the foundation for importing the data into the R/3 System must be available in IDoc format. For this reason, an IDoc must be generated during the mapping process in the transfer file for each object to generate in the R/3 System.

To implement the object mapping in an IDoc, you have to analyze how the structure appears to the corresponding BAPI call that will generate the object in the R/3 System. Therefore, you have to determine how the data used for the BAPI call has to be represented in the IDoc, to allow the BAPI call to be generated automatically from the IDoc during ALE inbound processing.

The DX Workbench supports this analysis by integrating reports that

- a. Generate default IDocs for a BAPI without application data
- b. Generate IDocs for a BAPI whose parameters have been filled with data from an existing R/3 object

Therefore, you could perform the analysis as follows, for example:

- Use the corresponding online transactions to create test objects of the business object types in the R/3 System that you want to transfer from the non-SAP system.

Process Flow of the Mass Data Transfer via BAPI

- Call the corresponding reports, which generate IDocs for these objects.

This allows you to determine how the values that are involved in the online transaction are represented in the corresponding IDoc. In turn, this information can be used during the mapping process to correctly structure the IDocs for the objects to transfer.

2. Mapping the non-SAP system data in SAP format

Once you have identified how the individual IDocs are structured in the transfer file, you can transfer the data from the external-format file to this transfer file. Mapping is usually required to transform the external-format data into the IDoc format of the transfer file.

You can use SAP's own mapping tool (LSM Workbench) or an external mapping tool for the actual mapping. These mapping tools require the information you determined during the above structural analysis.



We plan to integrate an SAP mapping tool in the DX Workbench starting in Release 4.6C.

Once mapping is complete, the transfer file is imported into the target system.

3. Performing the actual data import

Once the transfer file is received in the target system, the following steps are performed for each IDoc contained in the file:

- The IDoc is read and its contents are mapped to the parameters of the corresponding BAPI.
- The BAPI is then called with the data from the IDoc, and fills the database tables with this application data. The BAPI performs all the required business checks of the data before writing it to the database.

If a BAPI call fails, an appropriate error message is returned in the BAPI *Return* parameter.

Once the data transfer is complete, more comprehensive consistency checks can be performed.

Developing BAPIs for Mass Data Transfer

Process Flow

The following aspects are relevant for developing a BAPI that can be used for mass data transfer in the DX Workbench:

1. Implementing a BAPI

BAPIs for mass data transfer must be developed as described in the [BAPI Programming Guide \[Ext.\]](#). All write BAPIs can potentially be used for mass data processing. In detail, we differentiate between the following standardized write BAPIs:

- **Create() or CreateFromData()**

The BAPIs *Create()* and *CreateFromData()* create an instance of an SAP business object type, such as a purchase order.

- **Change()**

The BAPI *Change()* changes an existing instance of an SAP business object type.

- **Delete() and Undelete()**

The BAPI *Delete()* deletes a complete instance of an SAP business object type from the database or sets a deletion flag. The BAPI *Undelete()* removes a deletion flag.

- **Cancel()**

Unlike the BAPI *Delete()*, the BAPI *Cancel()* cancels an instance of a business object, that is the instance to be cancelled remains in the database and an additional instance is created that is canceled.

- **Replicate() and SaveReplica()**

The BAPIs *Replicate()* and *SaveReplica()* are implemented as methods of replicable business object types. They enable specific instances of an object type to be copied to one or more different systems. The replicated instances are usually created under the same object key as the original object. These BAPIs are used mainly to transfer data between distributed systems within the context of Application Link Enabling (ALE).

- **Add<sub-object>() and Remove<sub-object>()**

The BAPI *Add<sub-object>()* adds a sub-object to an existing object instance and the BAPI *Remove<sub-object>()* removes a sub-object from an object instance.

2. Generating the BAPI-ALE interface

Because the data for the BAPI call is received in the R/3 System in IDoc format, the BAPI-ALE interface has to be generated to automate the mapping of the IDoc to the parameters of the BAPI.

3. Writing a report

The report is responsible for writing existing R/3 objects in IDoc format to a file. Because users are normally only familiar with the online transactions, but not the BAPIs or IDocs, the reports can be used to find out how the objects that are created in the online transaction have to be represented in the IDoc. This information is required during the mapping process to correctly structure the IDocs for the objects to transfer.

Developing BAPIs for Mass Data Transfer

This report is **required**, since it is the **only** way for users to establish the connection between the online transaction, the BAPI, and the IDoc.

4. Registering the BAPI

This step is required if the BAPI will be used in the DX Workbench. To register the BAPI, including its corresponding report, use transaction **BDLR**.

The individual aspects are described in detail below:

See also:

[Implementing a BAPI \[Page 85\]](#)

[Generating the BAPI-ALE Interface \[Page 116\]](#)

[Writing a Report \[Page 117\]](#)

[Registering the BAPI \[Page 118\]](#)

Implementing a BAPI

Purpose

To implement a BAPI, you should generally follow the development process described in the [BAPI Programming Guide \[Ext.\]](#) and observe the guidelines specified therein.

Because every write BAPI can potentially be used to transfer mass data (in addition to “normal” processing), every BAPI should be implemented to meet the needs of both usage types. This requirement is reflected in the guidelines for standard write BAPIs.

Detailed information on the individual programming models of the relevant standardized BAPIs is available below.

See also:

[Programming Create\(\) BAPIs \[Page 86\]](#)

[Programming Change\(\) BAPIs \[Page 91\]](#)

[Programming Delete\(\) BAPIs \[Page 97\]](#)

[Programming Cancel\(\) BAPIs \[Page 102\]](#)

[Programming Replicate\(\)/SaveReplica\(\) BAPIs \[Page 106\]](#)

[Programming Methods for Sub-Objects \[Page 111\]](#)

Programming Create() BAPIs

Programming Create() BAPIs

Use

The BAPI *Create()* creates one instance of an SAP business object type. Likewise, the BAPI *CreateMultiple()* creates several instances of a business object type simultaneously.

If a workflow method called *Create()* already exists for the business object type in question, you can use the name *CreateFromData()* for your BAPI. *Create()* is the preferred name for this BAPI.

The BAPIs *Create()* and *CreateMultiple()* are class methods (instance-independent).

For each *Create()* BAPI, a method must be provided with which the created business object instance can be deleted or canceled. To do this, depending on the business application practice, you should implement one of the BAPIs below:

- *Delete()*, which deletes a business object instance from the database
For more information, see [Programming Delete\(\) BAPIs \[Page 97\]](#).
- *Cancel()*, which cancels a business object instance
For more information, see [Programming Cancel\(\) BAPIs \[Page 102\]](#).

Features

BAPI Interface

Import Parameters

You should keep the following points in mind when defining the import parameters:

- The BAPI's import parameters in the function module contain the data required to uniquely identify an instance. All business key information may be provided, it must, however, be able to be derived. Since these are class methods, none of the BOR key fields may be a parameter in the function module or of the BAPI.
- The BAPI must also have a *TestRun* parameter that enables it to execute test runs. If this parameter is filled with the value "X", the BAPI executes normally, but does not write the results to the update task. In this way, after the BAPI *BapiService.TransactionCommit* executes, all results, such as the application log, for example, can be evaluated, but the BAPI has not modified the database.

For further information see, [Test Run Parameters \[Ext.\]](#).

- If customers are to be enabled to enhance the import parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionIn*.

For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Create()*-BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).

- You can create a change parameter to identify the values to create the instance with and to differentiate these values from the initial values when the BAPI is called. We recommend that you identify the change-relevant fields by flagging them. For more information, see [Change Parameters \[Ext.\]](#).

Export Parameters

You should keep the following points in mind when defining the export parameters:

- To make the object handle available to the calling program, the entire key fields must be returned in the export parameters of the function module. To this end, for each key field of the object type, you create an export parameter with the same name in the function module. This parameter is then filled with the appropriate values. These key parameters must have the same name as the corresponding key fields of the object type in the BOR.
- You can also create further export parameters in which, for example, generically created information is returned.
- If customers are to be allowed to extend the export parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionOut*. For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Create()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).
- To report messages from the method call back to the calling program, you should create the export parameter *Return*. To ensure the results of a *Create()* BAPI, called either synchronously or asynchronously, are monitored extensively, the following conventions for filling the *Return* parameter must be observed:
 - If the *Create()* BAPI executes successfully, in other words, an instance is created, the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	S
ID	BAPI
NUMBER	000
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
MESSAGE_V4	<corresponding_id>
SYSTEM	<system>

You should keep the following points in mind:

- The field MESSAGE_V1 contains the external name of the associated business object type, such as *SalesOrder*.
- The key under which the created object can be accessed at subsequent calls is returned in the field MESSAGE_V2. This is the external key included in the data passed when external number assignment is used, or the newly created internal key when internal number assignment is used. If the object has several key fields, the values of these key fields must be concatenated in MESSAGE_V2. The sequence in which the key fields are defined in the BOR must be adhered to for the concatenation. Moreover, the maximum length must be used for each key field. This may require the use of padding.



If the complete key exceeds the maximum capacity of MESSAGE_V2 (that is, 50 characters), it is divided up between the fields MESSAGE_V2 and MESSAGE_V3. The first 50 characters are stored in MESSAGE_V2 and the remaining characters in MESSAGE_V3.

Programming Create() BAPIs

- If the corresponding key exists, it is returned in MESSAGE_V4. This can be the key under which the object is stored in a non-SAP system, for example an external material number. The filling of this field serves exclusively to identify and not to access the object. If the corresponding key also consists of several fields, these must likewise be concatenated.
- If an error occurs during the execution of the *Create()* BAPI, in addition to the application-specific error messages the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	E
ID	BAPI
NUMBER	001
MESSAGE_V1	<objectname>
MESSAGE_V2	
MESSAGE_V3	
MESSAGE_V4	<corresponding_id>
SYSTEM	<system>

The sole contrast to successful execution is that the fields MESSAGE_V2 and MESSAGE_V3 must be passed blank. The external key is returned in MESSAGE_V4 to facilitate identification.

- In addition to the standardized message, further messages can be written in the *Return* parameter.
- It is of particular importance when errors occur that messages describing the errors in detail are returned. You should, therefore, use all of the fields of the structure BAPIRET2. You should, in particular, fill the fields *Parameter*, *Row* and *Field*.

For more information about this parameter, see [Return Parameters \(Error Handling\) \[Ext.\]](#).

BAPI Coding

Several Instances in one LUW

A *Create()* BAPI must be implemented in such a way as to enable several calls of the same *Create()* method to be initiated within one LUW and, consequently, to create several instances of the same object type in one LUW.

Using Temporary Storage

Both to enable several instances of the same object type to be created within one LUW and to achieve high performance, the results of a *Create()* BAPI should be bundled in the update task until they are actually written to the database.

This is achieved by the BAPI calling the update module "IN UPDATE TASK" after the application logic has executed successfully. These update modules do not initiate individual database accesses. Instead, each operation to be performed is stored temporarily in the function group memory, and is then called indirectly in the FORM routine "ON COMMIT". This FORM routine, therefore, runs once only and, using techniques such as ARRAY insert, performs the update in only one database access.

For more information about bundling calls, see [Details \[Page 119\]](#).

Avoiding Incompatibilities with Customizing Settings

It could be possible that certain values set by Customizing appear differently in the BAPI interface. These values cannot be overwritten by a *Create()* BAPI.

To avoid any incompatibility with Customizing settings, you should implement a *Create()* BAPI in the following way:

- The BAPI should transfer all the fields and check them against the Customizing settings. If the Customizing settings write-protect a field, the data in this field cannot be overwritten by data in the BAPI interface.
- For every incidence of incompatibility an error message must be returned in the *Return* parameter to the calling program. For example, “*Customizing settings do not allow the field 'MyAsset-AssetName' to be modified*”.
- All the fields that are assigned default values by Customizing and are therefore write-protected, must be documented.

External Key Assignment

For *Create()* BAPIs with an external key assignment, the caller transfers the key (ID) to the object instance to be created, for example, to a document number.

Important: These parameters for the external keys must not have the same name as the keys of the object type in the BOR. If they were to, the BAPI would be incorrectly identified as an instance method. Moreover, the keys created in the BAPI must be defined as export parameters



Keep in mind that you have to convert the specified keys explicitly in upper case letters in the source code of this type of *Create()* BAPIs. Otherwise keys are created that cannot be used in dialog applications. This is because with dialog applications external keys are always converted implicitly in upper case letters.

See also:

[Example of a Create\(\) BAPI \[Page 90\]](#)

Programming Change() BAPIs

Use

The BAPI *Change()* changes an existing instance of an SAP business object type, for example, a sales order.

The BAPI *Change()* is an instance method, whereas the BAPI *ChangeMultiple()* is a class method (instance-independent).

Features

BAPI interface

Import Parameters

You should keep the following points in mind when defining the import parameters:

- Since the *Change()* BAPI is an instance method, the key fields of the corresponding business object type must be created as import parameters in the function module. The names of these parameters must be identical to the names of the object keys in the BOR and must have the same data elements.



For the associated method definition in the BOR, the key fields must not also be specified as parameters. For this reason, the BOR/BAPI Wizard does not include the function module parameters for the key fields in the method definition, when it creates a BAPI.

- The BAPI must also have a *TestRun* parameter that enables it to execute test runs. If this parameter is filled with the value "X", the BAPI executes normally, but does not write the results to the update task. In this way, after the BAPI *BapiService.TransactionCommit* executes, all results, such as the application log, for example, can be evaluated, but the BAPI has not modified the database. For further information see, [Test Run Parameters \[Ext.\]](#).
- If customers are to be enabled to enhance the import parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionIn*. For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Change()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).
- Implement the *Change()* method according to the "*Change by flagging*" concept. Here you must create a change parameter for every parameter that has fields containing modified values. These change parameters serve to distinguish parameter fields containing modified values from parameter fields that have not been modified.

By contrast, if the *Change by comparison* concept is implemented, you must create an additional equivalent parameter for every parameter with fields containing modified values. For more information, see [Change Parameters \[Ext.\]](#).

Export Parameters

You should keep the following points in mind when defining the export parameters:

Programming Change() BAPIs

- If customers are to be allowed to extend the export parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionOut*. For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Change()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).
- To report messages from the method call back to the calling program, you should create the export parameter *Return*. To ensure the results of a *Change()* BAPI, called either synchronously or asynchronously, are monitored extensively, the following conventions for filling the *Return* parameter must be observed:
 - If the *Change()* BAPI executes successfully, the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	S
ID	BAPI
NUMBER	002
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

You should keep the following points in mind:

- The field MESSAGE_V1 contains the external name of the business object type, such as *SalesOrder*.
- The key under which the object to be changed can be accessed at subsequent calls is returned in the field MESSAGE_V2. This is the external key when external number assignment is used, or the internal key when internal number assignment is used. If the object has several key fields, the values of these key fields must be concatenated in MESSAGE_V2. The sequence in which the key fields are defined in the BOR must be adhered to for the concatenation. Moreover, the maximum length must be used for each key field. This may require the use of padding.



If the complete key exceeds the maximum capacity of MESSAGE_V2 (that is, 50 characters), it is divided up between the fields MESSAGE_V2 and MESSAGE_V3. The first 50 characters are stored in MESSAGE_V2 and the remaining characters in MESSAGE_V3.

- If an error occurs during the execution of the *Change()* BAPI, in addition to the application-specific error messages the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	E
ID	BAPI
NUMBER	003
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

Programming Change() BAPIs

The meaning of the various fields is the same as for successful execution.

- In addition to the standardized message, further messages can be written in the *Return* parameter. It is of particular importance when errors occur that messages describing the errors in detail are returned. You should, therefore, use all of the fields of the structure BAPIRET2. You should, in particular, fill the fields *Parameter*, *Row* and *Field*. For more information about this parameter, see [Return Parameters \(Error Handling\) \[Ext.\]](#).

BAPI Coding

Possible Strategies for Implementing a Change() BAPI

In BAPIs that cause database changes (for example, *Change()* and *Create()* BAPIs), you must be able to distinguish between parameter fields that are to be modified (change-relevant fields) and parameter fields that are to remain unmodified.

Using an initial value is no solution because an initial value could also represent a valid new value. Also, in the ABAP programming language as well as on other development platforms, the value "NULL" should not be assigned to a data type to indicate an empty field.

For this reason, the change-relevant fields must be identified using a "workaround". This can take two forms:

1. Flagging Fields to Identify Fields Containing Modified Values

In this approach, parameter fields containing modified values are identified by including a flag in an additional "change parameter".

- An additional change parameter must be created with the same number of fields and the same field names for every parameter in the BAPI containing modified field values.
- When the BAPI is called, the fields in the additional change parameter whose counterparts in the corresponding parameter contain modifications, must be marked with an update flag.

This way the BAPI can identify both modified and unmodified fields in the parameter.

Follow the conventions below when you create change parameters to identify modified fields:

- The name of the additional change parameter consists of the parameter name with the suffix "X". For instance, if the parameter is called *EquiSales*, the name of the additional change parameter is *EquiSalesX*.
- The additional change parameter must contain exactly the same number of fields and the same field names as the parameter. You must use the data element *BAPIUPDATE* (CHAR 1) as the data element for the update fields. This can have the following values:

Programming Change() BAPIs

'X':
 This value means that the corresponding parameter field contains a modified value.

' ' (no value):
 This means that the corresponding parameter field does not have to be updated.

- If the parameter is a table, the additional change parameter must also be a table.

For more information about this concept, see [Change Parameters \[Ext.\]](#).

2. Comparing Fields to Identify Fields Containing Modified Values

In this approach, fields containing modified values are identified by comparing two parameters, one containing the current valid data and the other the new, modified data.

When the *Change()* BAPI is called, the current data in the database and the new, modified data must be entered in the corresponding parameters. The current data set can be transferred, for instance, from a *GetDetail()* BAPI that has been called.

The following comparisons can be made:

- The current data can first be checked against the database contents to ascertain whether it has changed in the meantime. This way any database changes made between the time the data was read by the *GetDetail()* BAPI and the time the database is updated can be identified.
- The data in both the parameters can be compared field by field. If the data in two corresponding fields is different, the relevant values must be transferred from the parameter with the new data.

Note the following when you are comparing fields containing modified data.

- A calling program must be able to provide data in all the parameter fields of the *Change()* BAPI. For this reason, the parameter fields of the *Change()* BAPI must be exactly the same as the fields in the *GetDetail()* BAPI of the same business object type.
- The names of the parameters to be compared must be the same, but the parameter containing the modified data must also have the suffix '*New*'.



For example, if the parameter with the current valid data is called *EquiSales*, the parameter with the modified data is called *EquiSalesNew*.

- Both parameters must have exactly the same fields and the same structure.

Avoiding Incompatibilities with Customizing Settings

It could be possible that certain values set by Customizing appear differently in the BAPI interface. These values cannot be overwritten by a *Change()* BAPI.

To avoid incompatibility with Customizing settings, you should implement a *Change()* BAPI in the following way:

- The BAPI should transfer all the fields and check them against the Customizing settings. If the Customizing settings write-protect a field, the data in this field cannot be overwritten by data in the BAPI interface.
- For every incidence of incompatibility an error message must be returned in the *Return* parameter to the calling program. For example, “*Customizing settings do not allow the field 'MyAsset-AssetName' to be modified*”.
- All the fields that are assigned default values by Customizing and are therefore write-protected, must be documented.

See also:

[Example of a Change \(\) BAPI \[Page 96\]](#)

Example of a Change () BAPI

Example of a Change () BAPI

The BAPI *Change()* of the business object type *SettlementReqstList* (BUS2100001) is used here as an example. This BAPI has been implemented according to the concept *Flagging fields to identify fields containing modified values*.

The graphic below shows the BAPI in the BAPI Explorer and the interface of the underlying function module in the Function Builder.

The screenshot displays the BAPI Explorer on the left and the Function Builder code on the right. The BAPI Explorer shows the 'SettlementReqstList' object with a 'Change' method selected. The Function Builder shows the code for 'function bapi_settlementreqslist_change' with various parameters and tables.

```

function bapi_settlementreqslist_change.
**
**-----Lokale Schnittstelle:
**
** IMPORTING
**
**   VALUE(LISTHEADDATAIN) LIKE  BAPISLHC
**                               STRUCTURE BAPISLHC
**   VALUE(LISTHEADDATAINX) LIKE BAPISLHCX
**                               STRUCTURE BAPISLHCX
**   VALUE(DOCUMENTNUMBER) LIKE BAPISLKEY-DOCUMENT_NUMBER
**
** EXPORTING
**
**   VALUE(LISTHEADDATAOUT) LIKE BAPISLHEADO
**                               STRUCTURE BAPISLHEADO
**
** TABLES
**
**   HEADDATAIN STRUCTURE  BAPIACHC
**   HEADDATAINX STRUCTURE BAPIACHCX
**   ITEMDATAIN STRUCTURE  BAPIACIC
**   ITEMDATAINX STRUCTURE BAPIACICX
**   HEADDATAOUT STRUCTURE BAPISRHEADO OPTIONAL
**   ITEMDATAOUT STRUCTURE BAPISRITEMO OPTIONAL
**   LISTITEMDATAOUT STRUCTURE BAPISLITEHO OPTIONAL
**   RETURN STRUCTURE  BAPIRET2 OPTIONAL
**   EXTENSIONIN STRUCTURE BAPIPAREX OPTIONAL
**   EXTENSIONOUT STRUCTURE BAPIPAREX OPTIONAL
**   HEADTEXTIN STRUCTURE BAPIABHEADCTEXT OPTIONAL
**   ITEMTEXTIN STRUCTURE BAPIABITEMCTEXT OPTIONAL
**   HEADTEXTOUT STRUCTURE BAPIABHEADCTEXT OPTIONAL
**   ITEMTEXTOUT STRUCTURE BAPIABITEMCTEXT OPTIONAL
**   VENDORCONDIN STRUCTURE BAPIACCONDC OPTIONAL
**   VENDORCONDINX STRUCTURE BAPIACCONDCX OPTIONAL
**   CUSTOMERCONDIN STRUCTURE BAPIACCONDC OPTIONAL
**   CUSTOMERCONDINX STRUCTURE BAPIACCONDCX OPTIONAL
**

```

Important: The key field DOCUMENTNUMBER of the instance to be changed and which must be defined as an import parameter in the function module is suppressed when you view the BAPI in the BOR.

Programming Delete()/Undelete() BAPIs

Use

The BAPI *Delete()* deletes an instance of an SAP business object type from the database. Likewise, the BAPI *DeleteMultiple()* deletes several instances of a business object type. *Delete()* BAPIs must always delete entire instances, for example, a whole material master.

In contrast, the BAPI *Cancel()* cancels an instance of a business object, that is the instance to be canceled remains in the database and an additional instance is created that is canceled. For more information, see [Programming Cancel\(\) BAPIs \[Page 102\]](#).

A *Delete()* BAPI can delete immediately or at a later time by setting the deletion flag. The type of deletion you use in your BAPI is irrelevant to the caller, so you do not have to include these details in the BAPI interface.

The BAPI *Undelete()* is used to reset a deletion flag that has been set for a specific object.

The BAPIs *Delete()* and *Undelete()* are instance methods, whereas the BAPIs *DeleteMultiple()* and *UndeleteMultiple()* are class methods (instance-independent).

Features

The interfaces of the *Delete()* and *Undelete()* BAPIs should be identical.

BAPI Interface

Import Parameters

You should keep the following points in mind when defining the import parameters:

- If a single instance is to be deleted, you have to create an import parameter in the function module for each key field of the associated business object type. The names of the parameters must be identical to the names of the object keys in the BOR and must have the same data elements.

If several instances are to be deleted, you must create a table for the key fields of the business object type. You can do this using a range table.



For the associated method definition in the BOR, the key fields must not also be specified as parameters. For this reason, the BOR/BAPI Wizard does not include the function module parameters for the key fields in the method definition, when it creates a BAPI.

- The BAPI must also have a *TestRun* parameter that enables it to execute test runs. If this parameter is filled with the value "X", the BAPI executes normally, but does not write the results to the update task. This means that after the *BapiService.TransactionCommit* BAPI executes, all results, such as the application log, for example, can be evaluated, but the BAPI has not modified the database.

For further information see [Test Run Parameters \[Ext.\]](#).

- If customers are to be enabled to enhance the import parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionIn*. For information about extension parameters and recommendations on how the customer enhancement

Programming Delete()/Undelete() BAPIs

concept should be implemented in a *Delete()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).

- When the BAPI *DeleteMultiple()* is used, it must be possible to use the selection criteria to select the object instances to be deleted. To do this you can create selection parameters.
- When the BAPI *DeleteMultiple()* is used, it must be possible to use the selection criteria to select the object instances to be deleted. To do this you can create selection parameters.

For more information about these parameters, see [Selection Parameters \[Ext.\]](#).



Implement the interface of a *Delete()* BAPI so that it is not possible to delete all the instances simply by parameterizing the interface, for example by specifying default settings.

Export Parameters

You should keep the following points in mind when defining the export parameters:

- If customers are to be allowed to extend the export parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionOut*. For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Delete()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).
- To report messages from the method call back to the calling program, you should create only the export parameter *Return*. To ensure the results of a *Delete()* BAPI, called either synchronously or asynchronously, are monitored extensively, the following conventions for filling the *Return* parameter must be observed:
 - If the *Delete()* BAPI executes successfully, in other words, the desired instance is deleted, the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	S
ID	BAPI
NUMBER	004
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

You should keep the following points in mind:

- The field MESSAGE_V1 contains the external name of the business object type, such as *SalesOrder*.
- The key under which the object to be deleted can be accessed at subsequent calls is returned in the field MESSAGE_V2. This is the external key when external number assignment is used, or the internal key when internal number assignment is used. If the object has several key fields, the values of these key fields must be concatenated in MESSAGE_V2. The sequence in which the key fields are defined in the BOR must be adhered to for the concatenation. Moreover, the maximum length must be used for each key field. This may require the use of padding.

Programming Delete()/Undelete() BAPIs



If the complete key exceeds the maximum capacity of MESSAGE_V2 (that is, 50 characters), it is divided up between the fields MESSAGE_V2 and MESSAGE_V3. The first 50 characters are stored in MESSAGE_V2 and the remaining characters in MESSAGE_V3.

When an *Undelete()* BAPI executes successfully, the following standardized T100 message is used:

RETURN	MESSAGE
TYPE	S
ID	BAPI
NUMBER	006
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

- If an error occurs during the execution of the *Delete()* BAPI, in addition to the application-specific error messages the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	E
ID	BAPI
NUMBER	005
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

The meaning of the various fields is the same as for successful execution.

If an error occurs during the execution of an *Undelete()* BAPI, the following standardized T100 message is used:

RETURN	MESSAGE
TYPE	E
ID	BAPI
NUMBER	007
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

- In addition to the standardized message, further messages can be written in the *Return* parameter. It is of particular importance when errors occur that messages describing the errors in detail are returned. You should, therefore, use all of the fields of the structure BAPIRET2. You should, in particular, fill the fields *Parameter*, *Row* and *Field*.

For more information about this parameter, see [Return Parameters \(Error Handling\) \[Ext.\]](#).

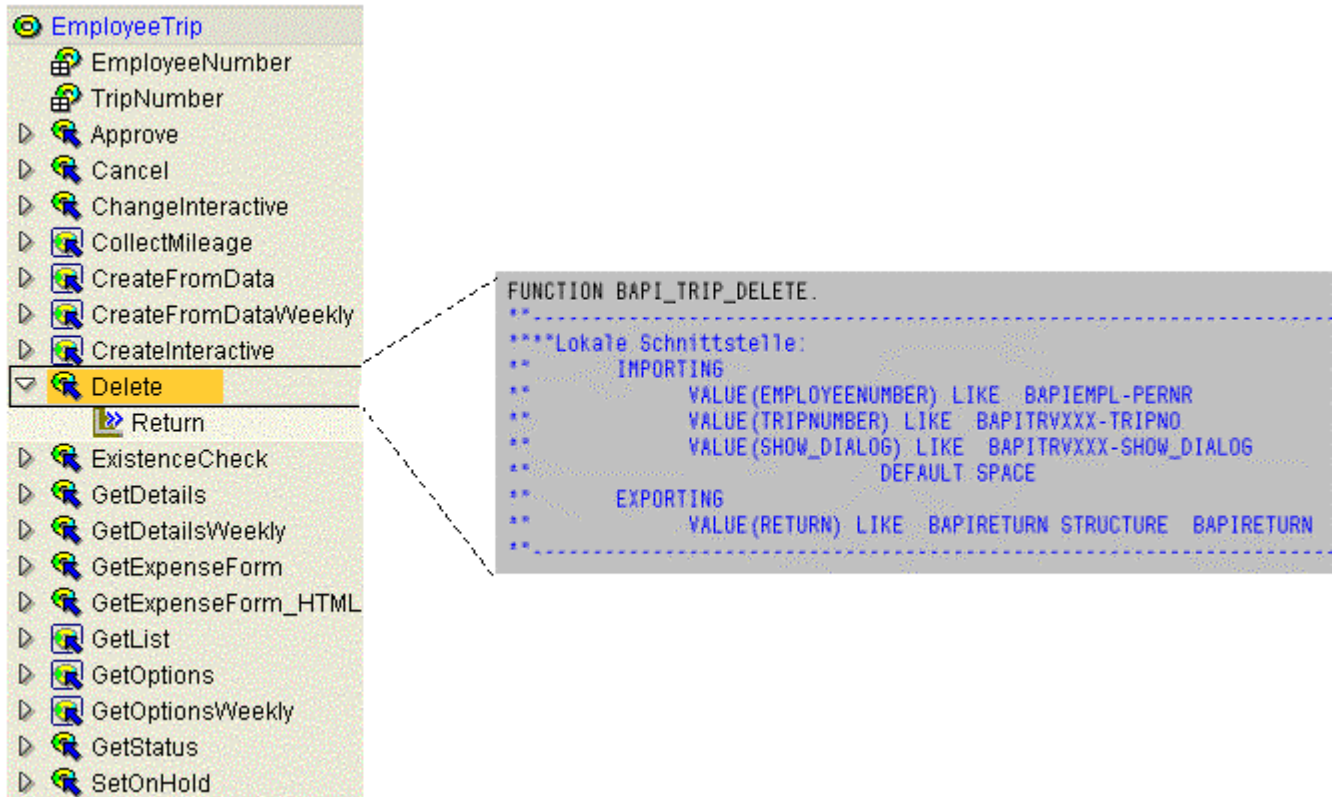
See also:

[Example of a Delete\(\) BAPI \[Page 101\]](#)

Programming Delete()/Undelete() BAPIs

Example of a Delete() BAPI

The BAPI *Delete()* of the business object type *EmployeeTrip* (BUS2089) is used here as an example. The graphic below shows the BAPI in the BAPI Explorer and the interface of the underlying function module in the Function Builder.



Important: The key fields EMPLOYEENUMBER and TRIPNUMBER of the instance to be deleted and which must be defined as import parameters in the function module are suppressed when you view the BAPI in the BOR.

Programming Cancel() BAPIs

Programming Cancel() BAPIs

Use

The *Cancel()* BAPI cancels one instance of a business object.

Unlike the BAPI *Delete()*, which deletes an object instance from the database, with the BAPI *Cancel()*:

- The canceled instance of the business object is not deleted
- An additional instance is created with which the instance of the business object is deleted.

The *Cancel()* BAPI is used to cancel business processes such as goods movements or invoice receipts.

The *Cancel()* BAPI is an instance method.

Features

BAPI Interface

Import Parameters

You should keep the following points in mind when defining the import parameters:

- The import parameters in the function module of the *Cancel()* BAPI must contain the key fields of the corresponding business object type. The names of the parameters must be identical to the names of the object keys in the BOR and must have the same data elements.



For the associated method definition in the BOR, the key fields must not also be specified as parameters. For this reason, the BOR/BAPI Wizard does not include the function module parameters for the key fields in the method definition, when it creates a BAPI.

- The data of the instance to be created comes from the data of the business object instance to be canceled. As such, it must no longer be explicitly transferred too.
- You can also create further parameters to specify information relevant for the actual cancellation process, for example, the name of the user performing the cancellation.
- The BAPI must also have a *TestRun* parameter that enables it to execute test runs. If this parameter is filled with the value "X", the BAPI executes normally, but does not write the results to the update task. In this way, after the BAPI *BapiService.TransactionCommit* executes, all results, such as the application log, for example, can be evaluated, but the BAPI has not modified the database.

For more information, see [Test Run Parameters \[Ext.\]](#).

- If customers are to be enabled to enhance the import parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionIn*. For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Cancel()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).

Export Parameters

You should keep the following points in mind when defining the export parameters:

- If customers are to be allowed to extend the export parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionOut*. For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Cancel()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).
- The key(s) of the cancellation instance created must be returned in further export parameters.
- To report messages from the method call back to the calling program, you should create the parameter *Return*. To ensure the results of a *Cancel()* BAPI, called either synchronously or asynchronously, are monitored extensively, the following conventions for filling the *Return* parameter must be observed:
 - If the *Cancel()* BAPI executes successfully, in other words, a cancellation instance is created, the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	S
ID	BAPI
NUMBER	008
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
MESSAGE_V4	<storno_id>
SYSTEM	<system>

You should keep the following points in mind:

- The field MESSAGE_V1 contains the external name of the business object type, such as *SalesOrder*.
- The key under which the canceled object was accessed is returned in the field MESSAGE_V2. This is the external key when external number assignment is used, or the internal key when internal number assignment is used. If the object has several key fields, the values of these key fields must be concatenated in MESSAGE_V2. The sequence in which the key fields are defined in the BOR must be adhered to for the concatenation. Moreover, the maximum length must be used for each key field. This may require the use of padding.



If the complete key exceeds the maximum capacity of MESSAGE_V2 (that is, 50 characters), it is divided up between the fields MESSAGE_V2 and MESSAGE_V3. The first 50 characters are stored in MESSAGE_V2 and the remaining characters in MESSAGE_V3.

- The key created for the cancellation instance is returned in MESSAGE_V4. This is the external key when external number assignment is used, or the internal key when internal number assignment is used. If the key also consists of several fields, these must likewise be concatenated.

Programming Cancel() BAPIs

- If an error occurs during the execution of the *Cancel()* BAPI, in addition to the application-specific error messages the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	E
ID	BAPI
NUMBER	009
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

Apart from the fact that the field MESSAGE_V4 is not filled, the meaning of the other fields is the same as for successful execution.

- In addition to the standardized message, further messages can be written in the *Return* parameter. It is of particular importance when errors occur that messages describing the errors in detail are returned. You should, therefore, use all of the fields of the structure BAPIRET2. You should, in particular, fill the fields *Parameter*, *Row* and *Field*.

For more information about this parameter, see [Return Parameters \(Error Handling\) \[Ext.\]](#).

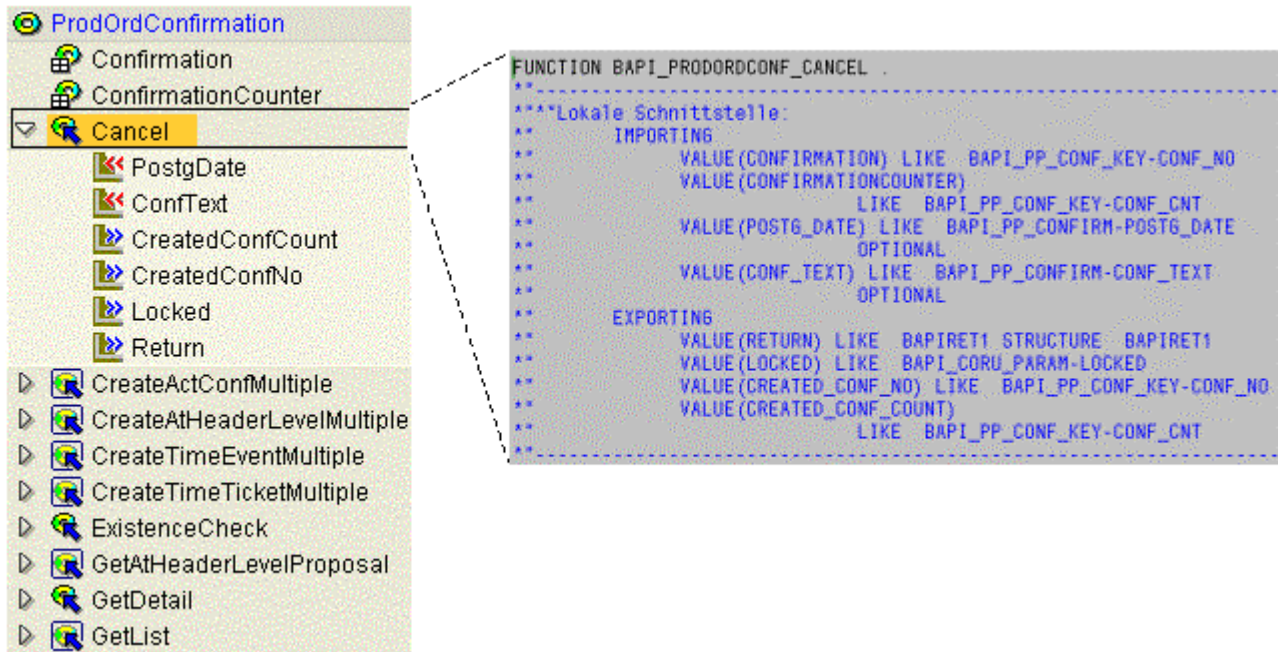
See also:

[Example of a Cancel\(\) BAPI \[Page 105\]](#)

Example of a Cancel() BAPI

The BAPI *Cancel()* of the business object type *ProdOrdConfirmation* (BUS2116) is used here as an example.

The graphic below shows the BAPI in the BAPI Explorer and the interface of the underlying function module in the Function Builder.



Important: The key fields CONFIRMATION and CONFIRMATIONCOUNTER of the instance to be canceled and which must be defined as import parameters in the function module are suppressed when you view the BAPI in the BOR.

Programming Replicate()/SaveReplica() BAPIs

Programming Replicate()/SaveReplica() BAPIs

Use

To replicate a business object instance, for example, to transfer data between two distributed systems in the context of Application Link Enabling (ALE), you can implement specific standardized BAPIs.

The objective of replicating business object types is to make specific instances of an object type available on one or more additional systems. The replicated instances are usually created under the same object key.

The interface of these BAPIs depends on the characteristics and contents of the business object that is to be replicated. For this reason replicate BAPIs must be implemented for each business object.

Business objects instances can be replicated in two ways:

- Upon request ("Pull")
System "A" requests replicates from system "B". Then system "B" replicates the requested business object instances on system "A".
- Via subscription list ("Push")
In this approach system "B" maintains a list of systems requiring replicates. At regular intervals, system "B" replicates the business object instances to all the systems in the list.

Both of the above replication methods can be implemented with the BAPIs *Replicate()*, *SaveReplica()* and *SaveReplicaMultiple()*. These BAPIs are both class methods (instance-independent).

Features

Replicate()

The BAPI *Replicate()* is called in the system which contains the originals of the business object instances to be replicated. The BAPI *Replicate()* is used for:

- Identifying the business object instances to be replicated and to organize the required data
- Calling the *SaveReplica()* methods in the receiving system

The BAPI *Replicate()* can only request the replication of instances of a business object. The actual replication is carried out when the server system calls one of the *SaveReplica()* BAPIs described below on the client system.

You must follow the steps below:

1. Select the data to be replicated.
2. Determine the receiver.

This is done in the ALE distribution model using the function module `ALE_SYNC_BAPI_GET_RECEIVER`. For information about this function module, see the Section *Distribution Using BAPIs* in the ALE Programming Guide. You can also restrict the number of receivers in the parameter *Recipients* of the *Replicate()* BAPI.

3. Loop via the receiver and call the relevant *SaveReplica()* BAPI.
4. Enter a value in the *Return* parameter and complete the *Replicate()* BAPI.

Programming Replicate()/SaveReplica() BAPIs

Import Parameters

The interface of the *Replicate()* BAPI **must** contain the following parameters:

- Parameters which identify the business object instances to be replicated
You could implement these, for example, using a range table for the key fields of the business object types or using selection parameters that enable you to select instances to be replicated. (See also [Selection Parameters \[Ext.\]](#)).
- Parameter *Recipients*
This parameter identifies the logical systems in which the business object instances are to be replicated. This parameter is based on the data element BAPILOGSYS. If this parameter is set to "initial", the receiver is determined using the ALE distribution model.
- If customers are to be enabled to enhance the import parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionIn*.
For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Replicate()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).

The interface of the *Replicate()* BAPI can also contain the following parameters:

- Parameter *RequestChanges*
This parameter enables modifications of object instances already replicated to be copied directly from engineering change management and forwarded to the receiving system. You should only use this parameter if the business object type in question provides engineering change management.
Structure this parameter on the data element RQSTCHNG. This can have the following values:
 - ' ' (no value):
No value is the standard setting and means that engineering change management will not be accessed.
 - 'X':
This value means that the modified data is replicated directly from engineering change management.The parameter *RequestChanges* must not be used together with the *Recipients* parameter, because the change pointers in change management are reset, if change management is accessed by the receiver. Other receivers may then not be allowed access. Documentation on this parameter must refer explicitly to this connectivity.
- Other BAPI-specific import parameters, for example, to specify the data range of the instances to be replicated (for example, material with or without plant data).

Export Parameters

The BAPI *Replicate()* should contain the following export parameters:

- The *Return* parameter in which messages from the method call are returned to the calling program.
For more information about this parameter, see [Return Parameters \(Error Handling\) \[Ext.\]](#).
- A table containing information on the object instances to be replicated.

Programming Replicate()/SaveReplica() BAPIs

- If customers are to be allowed to extend the export parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionOut*.

For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Replicate()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).

SaveReplica() and SaveReplicaMultiple()

The class method *SaveReplica()* and the method *SaveReplicaMultiple()* generate replicates of business object instances. They are used to replicate objects in different systems that are **semantically** identical. For technical reasons these objects may be created with different objects keys (object IDs).

The BAPI *SaveReplica()* is used by a system to replicate one business object instance on a target system or to modify one business object that has already been replicated. Whilst the *SaveReplicaMultiple()* BAPI can replicate several business object instances on the target system or modify several instances that have already been replicated.

For each business object type to be replicated you have to implement one or both of these methods, according to your requirements.

Import Parameters

You should keep the following points in mind when defining the import parameters:

- For the *SaveReplica()* BAPI, all the data required for replicating an individual business object instance must be provided in the import parameters. For the *SaveReplicaMultiple()* BAPI, all the relevant data for replicating several instances must be provided in the import parameters. All business object information may be provided, it must, however, be able to be derived. Since these are class methods, none of the BOR key fields may be a parameter in the function module or of the BAPI.
- If only parts of objects are to be replicated rather than whole objects, you can use other optional import parameters.
- If instances that have already been replicated are to be changed when a *SaveReplica()* or *SaveReplicaMultiple()* BAPI is called, the fields that are to be changed (that is, receive new values) and the fields that are to remain the same must be identified. You can do this by flagging the fields, as described in [Change Parameters \[Ext.\]](#).
- The BAPI must also have a *TestRun* parameter that enables it to execute test runs. If this parameter is filled with the value "X", the BAPI executes normally, but does not write the results to the update task. In this way, after the BAPI *BapiService.TransactionCommit* executes, all results, such as the application log, for example, can be evaluated, but the BAPI has not modified the database.

For further information see [Test Run Parameters \[Ext.\]](#).

Export Parameters

You should keep the following points in mind when defining the export parameters:

- If customers are to be allowed to extend the export parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionOut*.

For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *SaveReplica()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).

Programming Replicate()/SaveReplica() BAPIs

- To report messages from the method call back to the calling program, you should create the parameter *Return*. To ensure the results of a *SaveReplica()* BAPI, called either synchronously or asynchronously, are monitored extensively, the following conventions for filling the *Return* parameter must be observed:
 - If the *SaveReplica()* BAPI executes successfully, the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	S
ID	BAPI
NUMBER	010
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

You should keep the following points in mind:

- The field MESSAGE_V1 contains the external name of the business object type, such as *SalesOrder*.
- The information identifying the replicated object is returned in MESSAGE_V2.



If this information exceeds the maximum capacity of MESSAGE_V2 (that is, 50 characters), the remaining characters can be entered in the field MESSAGE_V3.

- If an error occurs during the execution of the *SaveReplica()* BAPI, in addition to the application-specific error messages the following standardized message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	E
ID	BAPI
NUMBER	011
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

The meaning of the various fields is the same as for successful execution.

- In addition to the standardized message, further messages can be written in the *Return* parameter. It is of particular importance when errors occur that messages describing the errors in detail are returned. You should, therefore, use all of the fields of the structure BAPIRET2. You should, in particular, fill the fields *Parameter*, *Row* and *Field*.

For more information about this parameter, see [Return Parameters \(Error Handling\) \[Ext.\]](#).

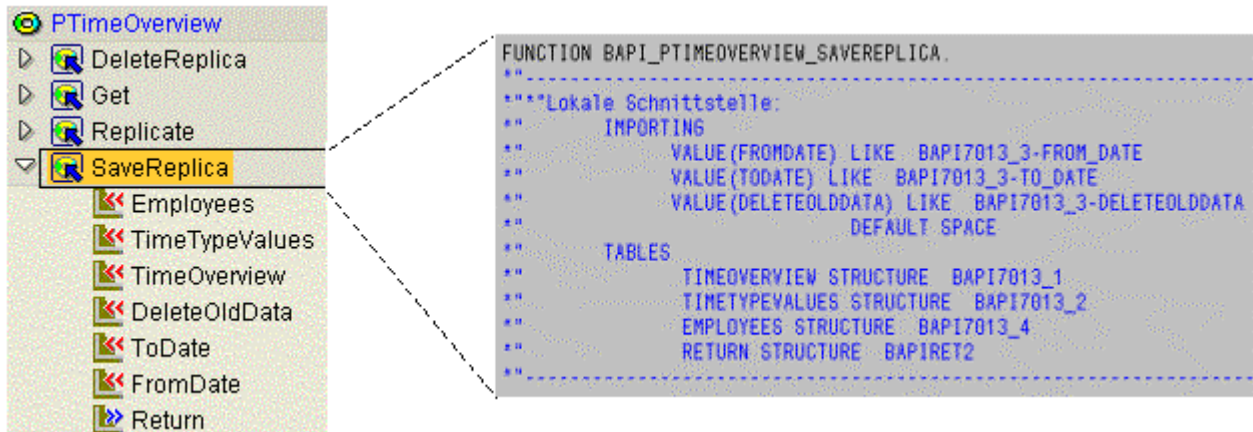
See also:

[Example of a SaveReplica\(\) BAPI \[Page 110\]](#)

Example of a SaveReplica() BAPI

Example of a SaveReplica() BAPI

The BAPI *SaveReplica()* of the business object type *PTimeOverview* (BUS7013) is used here as an example. The graphic below shows the BAPI in the BAPI Explorer and the interface of the underlying function module in the Function Builder.



Programming Methods for Sub-Objects

Use

If a business object type consists of sub-objects, you can implement the following standardized BAPIs to add or remove sub-objects:

- *Add<name of sub-object>*

This method adds a sub-object to an object type.

- *Remove<name of sub-object>*

This method removes a sub-object from an object.

For example, to add or remove the sub-object *purchase order item* to the business object type *purchase order*, the BAPIs *AddItem()* and *RemoveItem()* could be implemented for the object type, *purchase order*.

The BAPIs *Add<Name of sub-object>()* and *Remove<Name of sub-object>()* are instance methods.

Features

BAPI Interface

Import Parameters

You should keep the following points in mind when defining the import parameters:

- The import parameters in the function module of both BAPIs must contain the key fields of the corresponding business object type. The names of the parameters must be identical to the names of the object keys in the BOR and must have the same data elements.
- Parameters are also required for the data that uniquely identifies the sub-object in question.
- The BAPI must also have a *TestRun* parameter that enables it to execute test runs. If this parameter is filled with the value "X", the BAPI executes normally, but does not write the results to the update task. In this way, after the BAPI *BapiService.TransactionCommit* executes, all results, such as the application log, for example, can be evaluated, but the BAPI has not modified the database.

For further information see [Test Run Parameters \[Ext.\]](#).

- If customers are to be enabled to enhance the import parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionIn*. For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in a *Add<name of sub-object>()* or *Remove<name of sub-object>()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).
- You can create a change parameter to identify the values to create the instance with and to differentiate these values from the initial values when the BAPI is called. We recommend that you identify the change-relevant fields by flagging them. For more information, see [Change Parameters \[Ext.\]](#).

Programming Methods for Sub-Objects

Export Parameters

You should keep the following points in mind when defining the export parameters of a *Remove<name of sub-object>()* BAPI:

- To report messages from the method call back to the calling program, you should create only the export parameter *Return*. To ensure the results of a *Remove<name of sub-object>()* BAPI, called either synchronously or asynchronously, are monitored extensively, the following conventions for filling the *Return* parameter must be observed:
 - If the BAPI executes successfully, in other words, the desired sub-object is deleted, the following standardized T100 message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	S
ID	BAPI
NUMBER	014
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

You should keep the following points in mind:

- The field MESSAGE_V1 contains the external name of the business object, such as *SalesOrder*, whose sub-object was deleted.
- The key under which the object, whose sub-object has been deleted, is accessed, is returned in MESSAGE_V2. This is the external key when external number assignment is used, or the internal key when internal number assignment is used. If the object has several key fields, the values of these key fields must be concatenated in MESSAGE_V2. The sequence in which the key fields are defined in the BOR must be adhered to for the concatenation. Moreover, the maximum length must be used for each key field. This may require the use of padding.



If the complete key exceeds the maximum capacity of MESSAGE_V2 (that is, 50 characters), it is divided up between the fields MESSAGE_V2 and MESSAGE_V3. The first 50 characters are stored in MESSAGE_V2 and the remaining characters in MESSAGE_V3.

- If an error occurs during the execution of the *Remove<name of sub-object>()* BAPI, in addition to the application-specific error messages the following standardized message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	E
ID	BAPI
NUMBER	015
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

The meaning of the various fields is the same as for successful execution.

Programming Methods for Sub-Objects

- In addition to the standardized message, further messages can be written in the *Return* parameter. It is of particular importance when errors occur that messages describing the errors in detail are returned. You should, therefore, use all of the fields of the structure BAPIRET2. You should, in particular, fill the fields *Parameter*, *Row* and *Field*.

For more information about this parameter, see [Return Parameters \(Error Handling\) \[Ext.\]](#).

You should keep the following points in mind when defining the export parameters of an *Add<name of sub-object>()* BAPI:

- If customers are to be allowed to extend the export parameters of the BAPI without modifications being necessary, you must create the parameter *ExtensionOut*. For information about extension parameters and recommendations on how the customer enhancement concept should be implemented in an *Add<name of sub-object>()* BAPI, see [Customer Enhancements to BAPIs \[Page 10\]](#).
- To report messages from the method call back to the calling program, you should create the parameter *Return*. To ensure the results of an *Add<name of sub-object>()* BAPI, called either synchronously or asynchronously, are monitored extensively, the following conventions for filling the *Return* parameter must be observed:
 - If the BAPI executes successfully, in other words, the desired sub-object is created, the following standardized message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	S
ID	BAPI
NUMBER	012
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

The meaning of the various fields is the same as described in the programming model for the *Remove<name of sub-object>()* BAPI.

- If an error occurs during the execution of the *Add<name of sub-object>()* BAPI, in addition to the application-specific error messages the following standardized message must be passed in the *Return* parameter:

RETURN	MESSAGE
TYPE	E
ID	BAPI
NUMBER	013
MESSAGE_V1	<objectname>
MESSAGE_V2	<reference_id>
MESSAGE_V3	<reference_id>
SYSTEM	<system>

The meaning of the various fields is the same as for successful execution.

- In addition to the standardized message, further messages can be written in the *Return* parameter. It is of particular importance when errors occur that messages describing the errors in detail are returned. You should, therefore, use all of the fields of the structure BAPIRET2. You should, in particular, fill the fields *Parameter*, *Row* and *Field*.

Programming Methods for Sub-Objects

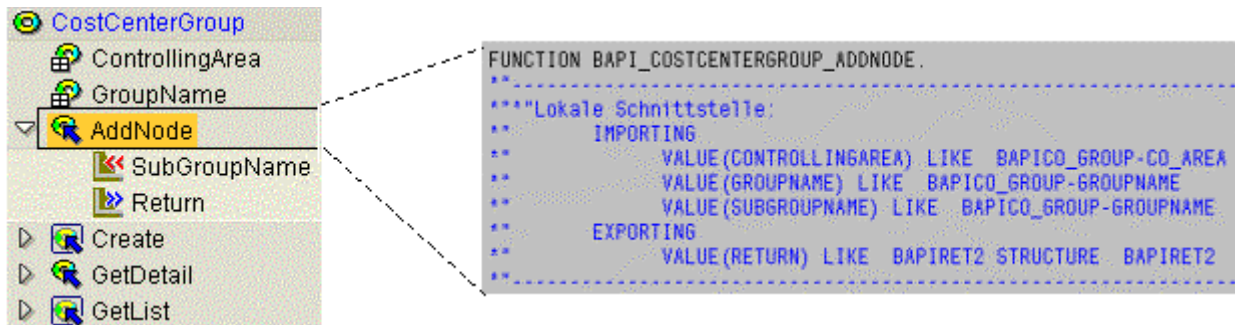
For more information about this parameter, see [Return Parameters \(Error Handling\)](#) [\[Ext.\]](#).

See also:

[Example of an Add<Name of Sub-Object> BAPI \[Page 115\]](#)

Example of an Add<Name of Sub-Object> BAPI

The BAPI *Add<Name of Sub-Object>()* of the business object type *CostCenterGroup* (BUS1112) is used here as an example. The graphic below shows the BAPI in the BAPI Explorer and the interface of the underlying function module in the Function Builder.



Important: The key fields CONTROLLINGAREA and GROUPNAME of the instance to be canceled and which must be defined as import parameters in the function module are suppressed when you view the BAPI in the BOR.

Generating the BAPI-ALE Interface

Generating the BAPI-ALE Interface

Purpose

To allow a BAPI call to be generated automatically for each IDoc contained in the transfer file, the corresponding ALE interface must be generated for all mass data-capable BAPIs. The interfaces are generated using transaction **BDBG**.

Process Flow

A detailed description of the generation process flow is available in [Using ALE Services \[Page 121\]](#) and in the ALE Programming Guide under [Generating the BAPI-ALE Interface \[Ext.\]](#).

The following components are generated for each BAPI:

- A message type
- An IDoc type
- An ALE inbound function module that reads the segments of an inbound IDoc, fills the parameters of the corresponding BAPI, and calls the BAPI
- An ALE outbound function module that generates an IDoc from the parameters of the BAPI and places it in ALE outbound

Writing a report

Use

A report must be written for each BAPI that can be used in the DX Workbench, in order to support the analysis phase. It is needed to generate IDocs for existing R/3 objects in a file. In particular, it generates IDocs for a BAPI call whose parameters have been filled with data from an existing R/3 object.

Integration

The following conventions apply to the report interface:

- A parameter for the receiver of the IDoc is a required import parameter, in addition to the fields for selecting the objects.
- Because the DX Workbench uses this report to write the IDocs to a special file, the receivers are passed on automatically as soon as the report is called. As a result, this field must be defined with “*parameters: receiver like tbdlst-logsys no-display.*” or “*select-options: receiver for tbdlst-logsys no-display.*”
- Additional, optional parameters may be present, but they are not used by the DX Workbench.

The flow logic of the report is written as follows:

- The report first collects all the relevant data for the objects the user has selected. This can be performed, for example, using the corresponding *GetDetail()* BAPIs for the objects.
- It then calls the generated ALE outbound function module to generate the IDoc (also see [Generating the BAPI-ALE Interface \[Page 116\]](#)) with the data collected above. This generates outbound IDocs in the database.

If the user has already maintained outbound partner agreements and a file port, then these outbound IDocs are written to an outbound file – where the DX Workbench finds them and converts them to inbound IDocs, which are available to analyze the underlying structures (also see [Analyzing SAP Structures \[Ext.\]](#)).

Prerequisites

- The report must have parameter RECEIVER of type TBDLST-LOGSYS.
- The report does **not** determine ALE receivers.
No receiver determination may be performed, because although the DX Workbench requires customizing of partner agreement and port, it does not require a logical system to be assigned to the current client.

Registering the BAPI

Registering the BAPI

To use the new BAPI in the DX Workbench, you will have to use transaction **BDLR** to register it and its corresponding report.

Details

Call Bundling

Call bundling improves the performance of instance BAPIs by bundling the individual BAPI calls in the update task. Because the individual BAPI calls are not bundled until they reach the update task, the programming model for BAPIs remains the same, except for one change to the update logic in the update modules.

To achieve call bundling, you have to modify and/or generate the following three components:

1. The BAPI

Each BAPI performs the necessary consistency checks and organizational operations, such as number assignment and lock management. If this application logic is passed free of errors, the update modules are called IN UPDATE TASK.

2. The update module

The enhancement to the programming model involves the update modules – specifically, the database accesses are no longer triggered directly; instead, the pending options are buffered in the function group memory, and a form routine is then called indirectly, “ON COMMIT”, which means that the operations are all performed at the end of the update task.

3. The form routine

The form routine takes the set of flagged operations in the function group memory, and performs the update with only one database access, using techniques such as array insert.

The structure of the three components that are relevant for call bundling can be illustrated in pseudo-coding as follows:

1. The BAPI:

```
function bapi_<object>_create.  
    {consistency checks and error handling}  
    call <object>_post in update task  
        exporting  
        {data}.  
endfunction.
```

2. The update module:

```
function <object>_post.  
    {Operations in buffer}  
    perform <buffersave> on commit.  
endfunction.
```

3. Form routine for the bundled update:

```
form <buffersave>.  
    {Update buffer data}
```

Details

endform.

The call bundling concept offers the following advantages:

- The BAPIs themselves are transactional, since the update tasks buffers the bundling information.
- Traceable number assignment, which is a legal requirement in some scenarios.
- Improved performance through common update.
- The BAPIs can be used in local programs without any other changes.

A major disadvantage of this concept, however, is that technical problems – such as a server failure – will result in the loss of all data.

Using ALE Services (CA-BFA)

Purpose

Customers often use several R/3 Systems. This could be due to organizational or business reasons (if a customer has several plants around the world, for example) or technical reasons (the data volume or system load is too large for a single R/3 System). In many cases, data has to be exchanged between these systems.



Company headquarters uses enterprise-wide applications like accounting, human resources management, and sales planning, while the local plants plan and control production and materials management.

The decentralization of business applications – that is, implementing distributed business processes – can be practical for many reasons:

- Increasing market globalization results in physical separation of organizational units.
- Business processes are not limited to a single company, and customers and vendors are becoming increasingly involved.
- The performance of an R/3 System can be improved by distributing business applications.

To implement distributed business processes, it must be possible to exchange the data in a controlled manner and store it consistently. These tasks are performed by application link enabling (ALE), which integrates distributed business processes both between individual R/3 Systems and between R/3 Systems and non-SAP systems.

This document describes how to use BAPIs in order to distribute business processes using ALE. In particular, it deals with the process flow of communication using BAPIs and explains which steps are necessary to use BAPIs via ALE.

Target Audience

The target audience of this document consists of:

- BAPI developers who want to generate a BAPI-ALE interface for a BAPI
- SAP employees, partners, and customers who want to implement a distributed scenario using BAPIs and ALE services

Implementation Considerations

To implement a BAPI via ALE, you need:

- Basic knowledge of the R/3 System
- Basic knowledge of BAPIs, as described in 'BAPIs – Introduction and Overview'
- Knowledge of ALE

To develop a BAPI that can be used via ALE, you also need:

- Detailed knowledge of BAPI development, as described in the [BAPI Programming Guide \[Ext.\]](#)

Using ALE Services (CA-BFA)

- Knowledge of the ABAP programming language and ABAP Workbench

See also:

[Basic Concepts of ALE Technology \[Page 123\]](#)

[Implementing Narrow Coupling via BAPIs \[Page 127\]](#)

[Implementing Loose Coupling via BAPIs \[Page 131\]](#)

[Developing an ALE Business Process Based on BAPIs \[Page 141\]](#)

Basic Concepts of ALE Technology

Purpose

ALE supports the configuration and operation of distributed applications. It allows the controlled exchange of business messages between distributed applications with consistent data retention. The coupling between the distributed systems can be either narrow or loose.

The application integration is not achieved through a centralized database. Instead, the applications access a local database. Data retention is redundant. ALE ensures the distribution and synchronization of master, control, and transaction data through asynchronous communications. ALE uses synchronous connections to read data.

The use of ALE offers a number of advantages:

- Distribution of application data between R/3 Systems with different releases
- Data exchange can continue after an upgrade without any further adjustment
- Customer-specific enhancements
- The integration of non-SAP systems through communication interfaces
- Coupling R/3 and R/2 Systems

ALE supplies various services and tools for the communication between distributed application systems:

- Options for maintaining a distribution model
- Consistency checks
- Monitoring data transmission
- Error handling
- Synchronization tools
- Tools for defining new ALE business processes

Integration

Types of Coupling

As mentioned above, the coupling between distributed application systems can be either loose or narrow. A **loose coupling** is implemented with asynchronous communications and is used for write accesses. A **narrow coupling**, in contrast, is implemented with synchronous calls, and should only be used for reading data.

Both types of coupling can be implemented by calling BAPIs.

Loose Coupling

In a distributed environment, it is especially important that the systems be loosely coupled and independent of one another. If the called system is down, or a communication error occurs, the calling system can continue working normally. One example in which a loose coupling is essential is inventory management: When an inventory management component is combined with an accounting component, it must be possible to post goods movements even when the accounting component is unavailable.

Basic Concepts of ALE Technology

Loose coupling means that the individual systems for the most part communicate asynchronously with each other. In this type of communication, messages are exchanged between the systems. The data format used for these messages is called the Intermediate Document (IDoc). IDocs are structured data containers in which the data can be stored hierarchically. For more information on IDocs, refer to the document "ALE Introduction and Administration" under [IDoc \[Ext.\]](#).

Different communication technologies can be used to send the IDocs:

- For asynchronous communication between two R/3 Systems, the underlying communication technology is the transactional remote function call (tRFC). The transactional call is not executed immediately; instead, the data to send is first written to a database table. When a COMMIT WORK is triggered in the calling program, the remote call to the receiver system is executed. If the receiver system is currently unavailable, a periodically scheduled background process tries to send the data to the receiver system again later. The tRFC guarantees that the data is only transmitted once.
- If you want to establish asynchronous communications between R/3 Systems and non-R/3 systems, you can also use CPI/C, MPSeries, or other communication techniques to transmit the IDocs.

If an error occurs while the IDoc is processed, the invalid IDoc is saved and a workflow is generated that enables the ALE administrator to correct the error. These ALE error handling routines ensure that the data is updated consistently. As a result, data replication, updates, and inserts of data in other systems should always take place asynchronously. The disadvantage of asynchronous communications via ALE is that only a single return parameter from the called system is available.



Please note that asynchronous communication does not always mean a large time delay between call and execution. If the target system is available, an asynchronous call can be performed immediately after the COMMIT WORK in the called system.

During asynchronous communication via BAPIs, the calling system (client) generates an IDoc with data from the BAPI call (instead of the BAPI call itself) and sends it to the called system (server). The BAPI is then called with the data from this IDoc in the called system. For more information, please refer to [Implementing Loose Coupling via BAPIs \[Page 131\]](#).

Narrow Coupling

In contrast to loose coupling, narrow coupling requires the called system to be available. Narrow coupling is generally implemented using a synchronous call of a remote-capable function module. In contrast to asynchronous communications, the export parameters of the function module can be evaluated.

Synchronous calls are suitable for verifying or reading data in other systems. ALE supports synchronous BAPI calls starting in Release 4.0. Synchronous dialog calls are supported starting in Release 4.5A. In this case, the caller of the BAPI function module is automatically logged on to the other system (remote login to the "destination"). The user sees the transaction in the same window, and can navigate through the called system using menus and so on. Synchronous dialog methods are visible in the BOR and are modeled in the ALE distribution model.



Usually, synchronous writing between two databases is not allowed, to ensure that any transmission errors that may occur do not result in database inconsistencies!

For more information, please refer to [Implementing Narrow Coupling via BAPIs \[Page 127\]](#).

Using the ALE Distribution Model

The ALE distribution model describes the message flow between logical systems. The distribution model defines which messages are exchanged between systems and which BAPIs (starting in Release 4.0) are called.

Filtering

The transmission can be controlled data-specifically through **filters**. Filters are conditions that message types and BAPIs have to meet in order to be distributed by ALE outbound processing.



A customer uses a centralized inventory management system and several accounting systems. If stocks change in company code 0001, accounting data is sent to accounting system 01; if they change in company code 0002, accounting data is sent to accounting system 02.

You can **differentiate** between the following types of **filters** for BAPIs:

1. Receiver Filtering

You can use receiver filters to define dependencies that affect the permitted recipients between BAPIs or between a BAPI and a message type.

For more information on receiver filtering, please refer to the ALE Programming Guide under [Determining Receivers for a BAPI \[Ext.\]](#).

2. Data Filtering

Data filtering offers two filter services: *interface reduction* and *parameter filtering*:

- *Interface reduction* allows field suppression by the BAPI interface, which means these fields will be ignored by the receiver. This can be used to prevent certain fields from being overwritten, for example.
- *Parameter filtering* allows you to control the scope of the dataset for transmission in the BAPI, by filtering out BAPI table parameters that fail to meet the defined conditions. If hierarchical dependencies exist between two table parameters of the BAPI, then additional parameter hierarchies will have to be defined. For further information see the ALE Programming Guide under [Defining Hierarchies between BAPI Parameters \[Ext.\]](#).

For more information on data filtering, see the ALE Programming Guide under [Filtering Data \[Ext.\]](#).

To perform receiver determination and parameter filtering, you must create **filter objects** before you maintain the distribution model. Filter objects consist of a filter object type and an object value. Filter object types for BAPIs correspond to a parameter name during the BAPI call. They check whether a parameter is set to the required object value. The BAPI is only distributed when this condition is met.

Interface reduction, in contrast, does not require any filter objects. However, it can only be used for certain types of BAPIs whose interface explicitly supports this type of filtering.

Maintaining the Distribution Model

The ALE distribution model is used to determine the target system for the remote call in both the synchronous and asynchronous cases. The ALE distribution model, which is a component of

Basic Concepts of ALE Technology

general IMG customizing, can be reached in the R/3 Implementation Guide through path *Basis* → *Application Link Enabling (ALE)* → *Model and Implement Business Processes* → *Maintain Distribution Model*.

To create the required filter object types, first select path *Tools* → *ALE* → *ALE Development* → *BAPI* in the SAP menu. Then you can create filter object types for filtering data using the path *Data filtering* → *Maintain filter object type*, or you can create filter object types for receiver filtering using the path *Receiver determination* → *Maintain filter object type*.

If you want to test the connection between two systems, you will have to make the following settings while maintaining the distribution model:

1. Determine the unique client ID and the logical system
2. Determine the technical communication partner
3. Generate the partner agreements in the sending system
4. Distribute the customer model to the receiving systems
5. Generate the partner agreement in the receiving systems



R/3 Release 99A will include an ALE customizing tool that will help customers to maintain the ALE distribution model. This support takes the form of templates that are available in the SAP menu under path *Tools* → *ALE* → *ALE development* → *Business processes* → *Maintain template*. A template contains a description of an ALE standard scenario based on message types/BAPIs and filter objects. Customers can use the templates, which are designed by the developer of the respective ALE standard scenario, to generate basic entries in the ALE distribution model based on their customer-specific data.

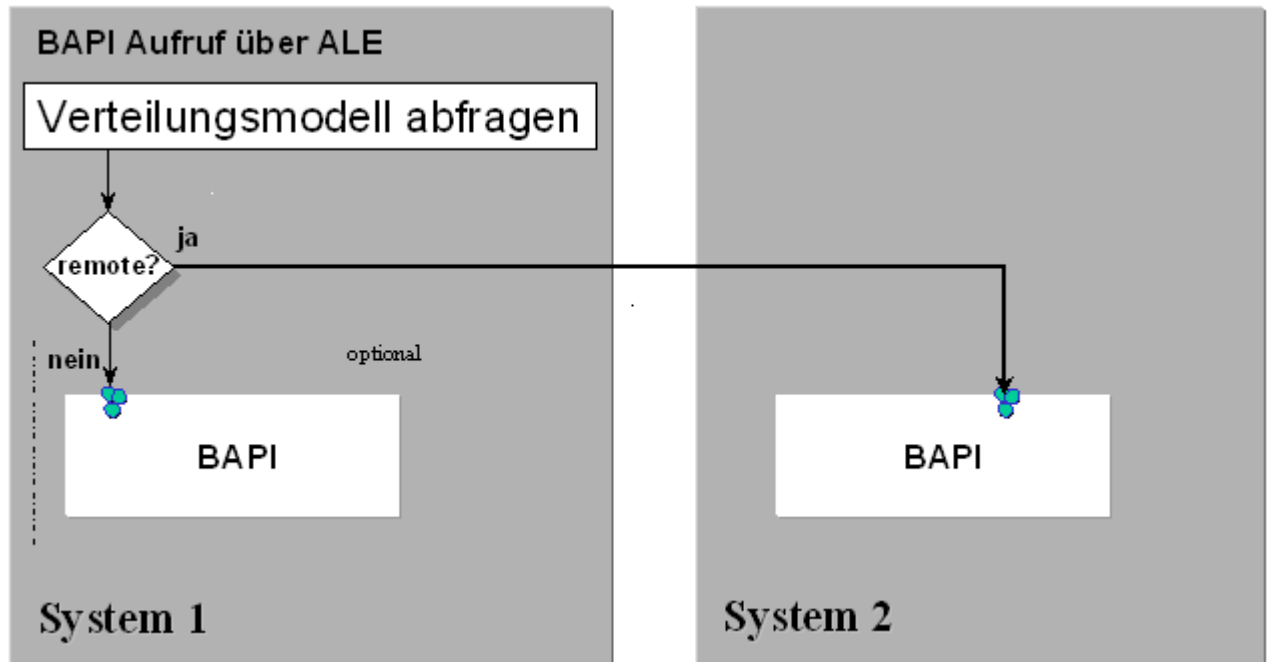
For more information on maintaining the distribution model and the filter objects, refer to document “ALE Introduction and Administration” under [Modeling Distribution. \[Ext.\]](#)

Implementing Narrow Coupling with BAPIs

Process Flow

In a narrow coupling scenario, the BAPI can be called synchronously with a remote function call. The process flow involves the following two steps:

1. [Querying the distribution model \[Page 128\]](#)
2. [Calling the BAPI \[Page 130\]](#)



Querying the Distribution Model

Querying the Distribution Model

Prerequisites

The distribution model defines the potential receivers of a BAPI call. Because the distribution of data can be linked to additional conditions, certain dependencies between BAPIs or between BAPIs and message types can be defined as **receiver filters**.

A filter object is initially created for each receiver filter before the distribution model is maintained; the value of the filter object determines at runtime whether the condition is satisfied or not.

Procedure

The query of the distribution model is divided into the two sub-phases **determining the filter objects** and **performing receiver determination**.

Determining the Filter Objects

Before receiver determination can be performed, the filter objects that have been assigned to the BAPI must be determined.

If the filter objects are known at runtime, they can be specified directly during receiver determination. If not, function module `ALE_BAPI_GET_FILTEROBJECTS` is available: it extracts the corresponding filter objects for a specific BAPI.

Performing Receiver Determination

Function module `ALE_SYNC_BAPI_GET_RECEIVER` is called by the application program to determine the receivers of a synchronous BAPI. This module returns a table with all the receivers. In addition to the logical system, the table also contains the RFC destination. If receiver determination is tied to conditions, then the function module must be given the values of the filter objects that were determined above.

Receiver determination for calling a remote BAPI from the ALE distribution model generally appears as follows:

```
call function 'ALE_SYNC_BAPI_GET_RECEIVER'
  exporting
    object= 'TESTFH01'
    method= 'GETDETAIL'
  tables
    receivers= receivers
    filterobject_values= filterobject_values
  exceptions
    error_in_filterobjects=1
    error_in_ale_customizing=2
    no_rfc_destination_maintained =3.
```

Querying the Distribution Model

If you want to determine only one receiver for a synchronous BAPI, then the application program calls function module ALE_BAPI_GET_UNIQUE_RECEIVER.

Calling BAPIs

Calling BAPIs

Depending on the number of receivers maintained in the distribution model, the remote BAPI is called one or more times:

```

loop at receivers.
  call function 'BAPI_TESTFH01_GETDETAIL'
    destination receivers-rfc_dest
    exporting
      key1          = 'SAP'
      key2          = '007'
    importing
      return        = returnvalue
      testdata      = testdata
    exceptions
      communication_failure = 1 message_text
      system_failure       = 2 message_text.
  If sy-subrc ne 0.
    .....
  else.
    .....
endif.
endloop.

```



Exceptions caused by connection errors to the partner system have to be trapped by the application program. The ALE service does not provide any error handling.



The synchronous RFC automatically triggers a database COMMIT, which means that database changes that were made before the RFC can no longer be rolled back.

For more information on the individual steps, please refer to the ALE Programming Guide under [Determining Receivers for Synchronous BAPIs \[Ext.\]](#).

Implementing Loose Coupling with BAPIs

Purpose

When a BAPI is used to exchange data asynchronously within the context of ALE, the following processes are involved:

- When the asynchronous BAPI function module interface is called in the sending system, it fills the corresponding IDoc with data from the BAPI call (**outbound processing**).
- This IDoc is dispatched to the target system.
- To use BAPIs as asynchronous interfaces, a BAPI-ALE interface that can be used in the distributed business process must be generated for an existing BAPI. This BAPI-ALE interface involves the following components:
 - A message type
 - An IDoc type
 - An ALE inbound function module that reads the segments of an inbound IDoc, fills the parameters of the corresponding BAPI, and calls the BAPI
 - An ALE outbound function module that generates an IDoc from the parameters of the BAPI and places it in ALE outbound

For a detailed description of the process flow during the generation of the BAPI-ALE interface, please refer to [Maintaining the BAPI-ALE Interface \[Page 145\]](#).

Process Flow

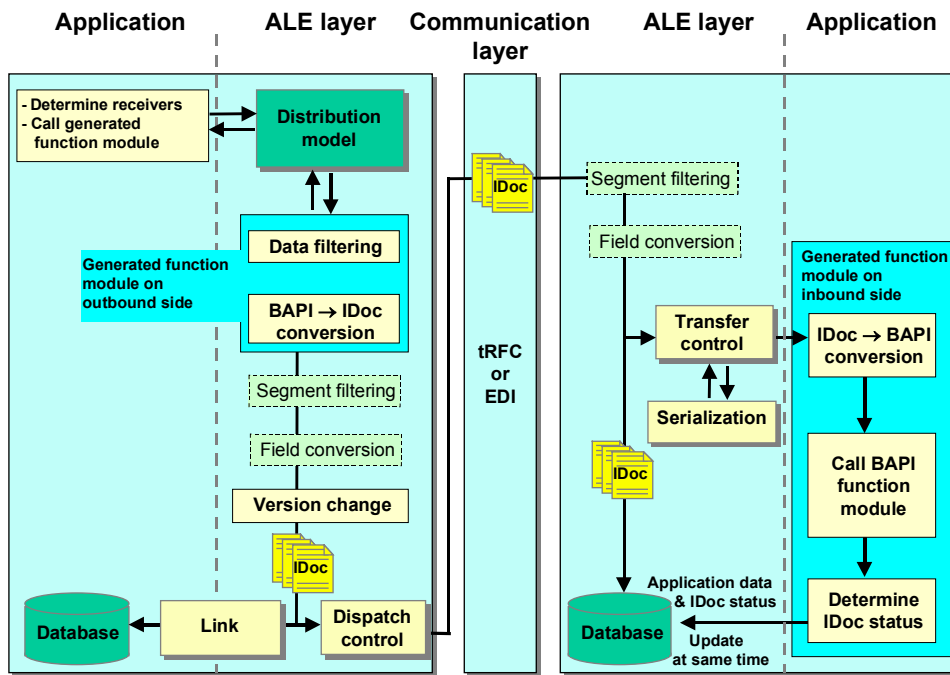
Flow of Asynchronous Communications

The individual steps involved in asynchronous communication using BAPIs are described in detail below. The process flow can be divided into the following phases:

- [Querying the distribution model \[Page 133\]](#)
- [ALE outbound processing \[Page 135\]](#)
- [Dispatching the IDoc \[Page 137\]](#)
- [ALE inbound processing \[Page 138\]](#)
- [Processing the BAPI \[Page 139\]](#)

The diagram below illustrates the steps required to call the BAPI in a remote system.

Implementing Loose Coupling with BAPIs



Querying the Distribution Model

Prerequisites

Analogous to the synchronous call of a BAPI, the distribution model defines the potential receivers of a BAPI call. If distribution of data is linked to additional conditions, these dependencies between BAPIs or between BAPIs and message types are defined as **receiver filters**.

A filter object is initially created for each receiver filter before the distribution model is maintained; the value of the filter object determines at runtime whether the condition is satisfied or not.

Procedure

The query of the distribution model is divided into the two sub-phases **determining receivers** and **calling the generated outbound function module**.

Determining Receivers

Before receiver determination can be performed, the filter objects that have been assigned to the BAPI must be determined. In the process, we have to differentiate whether the filter objects are known at runtime or not.

If the filter objects are known at runtime, they can be specified directly during receiver determination. If not, function module `ALE_BAPI_GET_FILTEROBJECTS` is available: it extracts the corresponding filter objects for a specific BAPI.

Once the filter objects have been resolved, the receivers can be determined. During an asynchronous BAPI call, the application program uses function module `ALE_ASYNC_BAPI_GET_RECEIVER`. This module returns a table with all the receivers. The table contains only the logical receiver systems.

If receiver determination is tied to conditions, then the function module must be given the values of the filter objects that were determined above.

Receiver determination for calling a remote BAPI from the ALE distribution model generally appears as follows:

Program Example: Asynchronous Receiver Determination (Part 1)

```
call function 'ALE_ASYNC_BAPI_GET_RECEIVER'
  exporting
    object = 'TESTFH01'
    method = 'SAVEREPLICA'
  tables
    receivers = receivers
    filterobject_values = filterobject_values
  exceptions
    error_in_filterobjects = 1
    error_in_ale_customizing = 2.
```

Querying the Distribution Model

For more information on receiver determination, please refer to the ALE Programming Guide under [Determining Receivers for Asynchronous BAPIs \[Ext.\]](#).

Calling the Generated Outbound Function Module

Once the receivers have been determined, you have to differentiate whether these receivers are local or remote. The BAPI can be called directly for local receivers. For remote calls, in contrast, the generated ALE outbound function module is executed, which forwards processing to the ALE layer. This function module is given the data for the BAPI call and the list of valid logical receiver systems.

The following program example illustrates the call of the ALE outbound function module:

Program Example: Asynchronous Call of a Remote BAPI (Part 2)

```
call function 'ALE_TESTFH01_SAVEREPLICA'
  exporting
    key1 = 'SAP'
    key2 = '007'
  tables
    receivers = receivers
    application_objects = application_objects
  exceptions
    error_creating_idocs = 1.
  .....
commit work.
```

Important: A COMMIT WORK must follow in the program context after the outbound function module calls the generated BAPI ALE interface. A database COMMIT at the end of the transaction is not sufficient. If no COMMIT WORK is triggered, the IDoc will be generated with the correct status, but it will not be dispatched.

ALE Outbound Processing

Procedure

The outbound processing performed by the ALE layer can be divided into the following steps: **data filtering, converting the BAPI call to an IDoc, segment filtering, field conversion, version change, and dispatch control.**

Data Filtering

If data filters have been defined for the BAPI in the distribution model, then this filtering is performed automatically by the outbound function module. Data filtering offers two filter services: *interface reduction*, which carries no conditions, and *parameter filtering*, which is tied to conditions:

- *Interface reduction* allows field suppression by the BAPI interface, which means these fields will be ignored by the receiver. This can be used to prevent certain fields from being overwritten, for example. However, it can only be used for certain types of BAPIs whose interface explicitly supports this type of filtering.
- *Parameter filtering* allows you to control the scope of the dataset for transmission in the BAPI, by filtering out BAPI table parameters that fail to meet the defined conditions. If hierarchical dependencies exist between two table parameters of the BAPI, then additional parameter hierarchies will have to be defined. For further information, see the ALE Programming Guide under [Defining Hierarchies between BAPI Parameters \[Ext.\]](#).

Because parameter filtering can be linked to conditions, filter objects have to be created for this type of filtering (in contrast to interface reduction). For more information on data filtering, see the ALE Programming Guide under [Data Filtering \[Ext.\]](#).

Converting the BAPI Call to an IDoc

Once data filtering is complete, the outbound function module uses the BAPI call to generate an IDoc with the data to transfer. In the process, note the following:

- If full parameters were suppressed during interface reduction, then these parameters will not be included in the IDoc. In contrast, if you only left out individual fields for structured parameters, the full parameters will still appear in the IDoc.
- Table lines that are filtered out during parameter filtering are not included in the IDoc.

Segment Filtering

Once the IDoc has been created, additional filtering of the IDoc segments is possible. Please note, however, that this type of filtering is only rarely used with BAPIs.

Field Conversion

The field conversion, which takes place after segment filtering, is especially relevant, for example, when a field format differs between the sender and receiver system. This function is especially important for converting the field format during data exchange between R/2 and R/3 Systems. Please note, however, that field conversion only plays a very small role for BAPIs.

ALE Outbound Processing

Version Change

To ensure that ALE functions properly between R/3 Systems with different releases, a conversion of IDoc formats can be performed to modify the message types used by the different releases.

Once the version change is complete, the IDocs are stored in the database and dispatch control is started: it decides which of these IDocs are dispatched immediately.

Dispatch Control

Dispatch control involves both a *time-based control* and a *quantity control*:

- Time-based control
IDocs can be dispatched immediately or in background processing. These settings are made in the partner agreement. If the IDocs are dispatched in background processing, a job must be scheduled to do so. The execution interval of this job can be selected freely.
- Quantity control
By default, ALE immediately writes the result of every IDoc in the BAPI to the database. It is possible, however, to collect IDocs and send them in packages. As a consequence, the results of the IDocs or BAPIs are not updated until the entire package has been processed. The package approach also allows you to bundle multiple BAPIs under a single LUW within ALE. The package size is configured for each specific partner in ALE customizing.

When an IDoc is to be dispatched, dispatch control uses the partner agreement for the logical receiver system to automatically determine the corresponding RFC destination. It then passes the IDoc on to the communication layer.

For more information on ALE outbound processing, see the ALE Programming Guide under [Implementing Outbound Processing \[Ext.\]](#).

Dispatching IDocs

Procedure

In the communication layer, the IDoc is dispatched with a transactional remote function call (tRFC) or through another file interface (EDI, for example).

The transactional call is not executed immediately; instead, the data to send is first written to a database table. When a COMMIT WORK is triggered in the calling program, the remote call to the receiver system is executed. If the receiver system is currently unavailable, a periodically scheduled background process tries to send the data to the receiver system again later. The tRFC guarantees that the data is only transmitted once.

ALE Inbound Processing

ALE Inbound Processing

Procedure

When the IDoc has arrived in the receiver system, the ALE layer of that system assumes inbound processing. Inbound processing involves the following steps: **segment filtering**, **field conversion**, and **transfer control**.

Segment Filtering

IDoc segments can be filtered in inbound, analogous to the outbound processing described above. Please note, however, that this type of inbound filtering is only rarely used with BAPIs.

Field Conversion

Like in outbound processing, field conversion can be performed here when the format of a field differs between the receiver and sender systems.

Once field conversion has taken place, the IDoc is stored in the database and processing is passed on to transfer control.

Transfer Control

Transfer control determines when the BAPIs are called in the application. This can take place immediately when an IDoc is received, or time-controlled in background processing.

If several dependent objects are distributed, **serialization** can be used in transfer control. Serialized distribution of messages means a specific sequence is retained during the generation, dispatch, and update of the corresponding IDocs. This avoids errors during inbound processing of the IDocs.

In the contexts of BAPIs, only object serialization can be used; this serialization ensures that the original sequence of messages for a certain object is always retained on the receiving end.

For more information on object serialization, refer to the document “ALE Introduction and Administration” under [Serialization of Messages \[Ext.\]](#).

When the time has come to process the BAPI, the generated inbound function module is called.

Processing BAPIs

Procedure

When the inbound function module is executed on the application side, the BAPI call is generated from the IDoc, the BAPI function module is called, and the IDoc status is determined. When processing of the BAPI (or the entire package) is complete, the status records of all the IDocs and all application data generated by the successfully completed BAPIs are written together to the database.

Converting the IDoc to a BAPI Call

When the BAPI call is generated, all the data from the IDoc segments is written to the corresponding parameters of the BAPI function module. If interface reduction has been defined for the BAPI, the suppressed fields are not filled with IDoc data.

Calling the BAPI Function Module

The BAPI function module is then executed synchronously with the filled parameters. Because the BAPI does not trigger a COMMIT WORK command, any application data that it creates, modifies, or deletes is not yet updated in the database.

Determining the IDoc Status

Once the execution of the function module is complete, the inbound function module determines the IDoc status, which depends on the result of the call.

If the TYPE field is filled with A (abort) or E (error) in at least one of the entries of the *Return* parameter, this means:

- Type A:
All status records of the corresponding IDoc are assigned status 51 (error, application document not posted), and a ROLLBACK WORK is executed.
- Type E:
All status records of the corresponding IDoc are assigned status 51 (error, application document not posted), and a COMMIT WORK is executed. During package processing, the COMMIT WORK is performed after the entire package has been processed (assuming no other BAPI returns an A message). The results of the invalid BAPIs are not saved, however; only their error status is recorded.
- In all other cases, status 53 (application document posted) is written and a COMMIT WORK is executed. During package processing, the COMMIT WORK is performed after the entire package has been processed (assuming no other BAPI returns an A message).

Posting Application Data and IDoc Status

When every IDoc/BAPI is processed individually, the data is immediately written to the database. If several IDocs are processed within a package, however, then the following situations are feasible:

- If no BAPI within the package aborted with an A message, the *COMMIT WORK* command is executed after the entire package has been processed. This saves the application data of all successfully completed BAPIs to the database together with the status records of all the IDocs.

Processing BAPIs

- As soon as a BAPI in the package aborts with an A message, however, the status of the corresponding IDoc is set to 51, and a ROLLBACK WORK is performed immediately. Inbound processing is then started again for all BAPIs that were completed successfully before (that is, the ones that did not return an E or A message). If no other A message occurs during this run, a COMMIT WORK is performed; the application data of the valid BAPIs is written to the database together with the status records of the IDocs. If further A messages occur, the above procedure is repeated.



Package processing is only possible when no serialization is involved.

Error Handling

If errors occur, the standard ALE error handling can be used. This has the following effects:

- The update of the IDoc and/or BAPI that caused the error is cancelled.
- An event is triggered. This event starts an error task (work item):
- The responsible person receives the task in their workflow inbox.
- Processing the task displays the error message.
- Once the error has been corrected in a separate window, the IDoc can be resubmitted for processing.
- If the error cannot be corrected, the IDoc can be flagged for deletion.
- Once the BAPI/IDoc has been successfully updated, an event is triggered that ends the error task. The task then disappears from the workflow inbox.

Developing an ALE Business Process Based on BAPIs

Process Flow

The following steps are required to develop an ALE business process based on BAPIs:

1. [Develop a new BAPI \[Page 142\]](#) or use an existing one
2. [Define hierarchies between BAPI parameters \[Page 144\]](#)
3. [Maintain the BAPI-ALE interface \[Page 145\]](#)
4. [Maintain the business process in the distribution model \[Page 161\]](#)

In the process, the following activities are optional:

- Defining the filter objects
- Determining receiver and data filters

Implementing the BAPI

Implementing the BAPI

Prerequisites

A BAPI method should only be implemented as an asynchronous interface if at least of the following conditions is satisfied:

- Database changes are consistent in all systems
Data must be updated consistently in both the local and remote systems.
- Looser coupling
With a synchronous interface, the coupling between the client and the server system is too narrow. If the connection is interrupted the client system would not be able to function properly.
- Performance load
The interface is frequently used or the interface handles large volumes of data. In this situation a synchronous interface cannot be used because performance would be too low.

If no suitable BAPI is available in the R/3 System, you can implement your own BAPI. IBUs, partners, and customers can also modify the BAPIs supplied by SAP. For information on modifying BAPIs, please see [Customer Enhancement and Modification of BAPIs \[Page 7\]](#).

Procedure

For more information about implementing BAPIs, see the [BAPI Programming Guide \[Ext.\]](#). IBUs, partners, and customers have to develop their BAPIs in their own namespace and follow certain guidelines that are defined in the BAPI Programming Guide.

If you implement a BAPI as an asynchronous interface, keep the following information in mind in addition to following the standard programming BAPI guidelines:

- The BAPI must not issue a COMMIT WORK command.
- The BAPI *return* parameter must use reference structure BAPIRET2.
- All BAPI export parameters with the exception of the *return* parameter are ignored and are not included in the IDoc that is generated.
- After the function module which converts the IDoc into the corresponding BAPI call in the receiving system has been called, status records are written for the IDoc in which messages sent in the *return* parameter are logged.

If, in at least one of the entries of return parameter, the *Type* field in the *return* parameter is filled with A (abort) or E (error), this means:

- Type A:
Status 51 (error, application document not posted) is written for all status records, after a ROLLBACK WORK has been executed.
- Type E:
Status 51 (error, application document not posted) is written for all status records and a COMMIT WORK is executed.

Otherwise status 53 (application document posted) is written and a COMMIT WORK executed.

Defining Hierarchies Between BAPI Parameters

Defining Hierarchies Between BAPI Parameters

Use

When hierarchical dependencies exist between the table parameters of a BAPI, these hierarchies must be explicitly defined to allow parameter filtering.



A BAPI for material master data contains, for example, the tables for plant data and the corresponding warehouse data. The table for plant data refers to the table for warehouse data through key field WERKS. A hierarchical dependency exists between the plant data and the warehouse data: if plant 001 is not replicated for a material as a result of the parameter filtering, then the warehouse data for plant 001 should not be replicated either.

The hierarchical dependencies are defined using field references between the parameters of the BAPI tables. These references must be defined before you create the BAPI-ALE interface, as this information is generated with the interface.

Procedure

You can define hierarchical dependencies in ALE development using *BAPI → Data Filtering → Maintain Hierarchy of Table Parameters* (transaction **BDBP**).

To do this, enter the object type and method of the BAPI. You can use input help to display the existing BOR object types and corresponding methods and make your selection.

The *Hierarchy* menu contains the following processing activities for creating the hierarchy:

- Create
- Change
- Display
- Delete

For more information on creating parameter hierarchies, see the ALE Programming Guide under [Defining Hierarchies between BAPI Parameters \[Ext.\]](#).

Maintaining the BAPI-ALE Interface

Use

To allow the asynchronous BAPI call in ALE business processes, you have to create a BAPI-ALE interface for the BAPI.

In the process, the following objects are generated for a BAPI:

- A message type
- An IDoc type, including its segments
- A function module that is called on the outbound side (it uses the BAPI data to generate the IDoc and dispatches it)
- A function module that fills the BAPI with the IDoc data on the inbound side.

In contrast to manually maintained message types, the function module that evaluates the change pointers does not create an IDoc; instead, it fills the appropriate BAPI structures, determines the receivers, and calls the generated ALE function module.

The generated message types, IDoc types, and function modules can also be used for distributing master data with the SMD tool.

Prerequisites

The following two prerequisites must be met:

1. The BAPI must exist.
2. The following alternatives are possible:
 - A new BAPI has been developed. The appropriate customer namespace has been used for customer developments.
 - A customer wants to generate a BAPI-ALE interface for a BAPI in the standard shipment, or change a BAPI-ALE interface supplied by SAP after modifying the corresponding BAPI. To do this, proceed as follows:
 - Copy and modify the function module for the original BAPI.
 - In the BOR, create a sub-object type in the customer namespace for the business object type that belongs to the BAPI (transaction **SW01**). When you create a sub-object type, the subtype inherits the methods of the business object.
 - Set the status of the object type to *Implemented* (*Edit* → *Change release status* → *Object type*).
 - All methods of the subtype can then be changed, deleted, or supplemented with your own methods. You then generate the BAPI-ALE interface for the subtype and an assigned method in the customer namespace.

After an upgrade, the customer-generated interface will continue to work like it did in the old release, even if SAP issues a new version of a BAPI-ALE interface with the new release. If SAP has not issued a new interface in the new release, the customer can re-generate the old interface to adjust for any new parameters.

If SAP issues a BAPI-ALE interface for a BAPI for which the customer has already generated an interface, then the customer should use the new interface and delete

Maintaining the BAPI-ALE Interface

the old one. The old interface can still be used, but only in the old release. If you re-generate the old interface, some of the generated objects – like segments of SAP objects, for example – may be overwritten if the interface in the customer BAPI function module refers to BAPI structures that belong to SAP.

2. If you need to take hierarchical dependencies between BAPI table parameters into account, you will have to define them before you generate the BAPI-ALE interface for the BAPI (see [Defining Hierarchies Between BAPI Parameters \[Page 144\]](#)). The defined hierarchy is evaluated during generation and is integrated in the interface coding. As a result, any subsequent change to the hierarchy will require re-generation of the BAPI-ALE interface. Once the generated IDoc type has been released, the defined hierarchy of the asynchronous BAPIs can no longer be changed (for compatibility reasons).

Be sure to read the [Notes \[Page 159\]](#) on related ALE topics. They contain information on data filtering, serializing messages, and other subjects.

Procedure

Choose path *BAPI* → *Maintain ALE interface* under ALE development.

Please note that all object names in customer systems must start with Y, Z, or your own prefix.

Then proceed as follows:

1. In the initial screen, enter the basic business object (object type) for your interface and the method.
2. Then select one of the options in the *Interface* menu to maintain, display, check, or delete an interface:
 - **Creating an interface**
Prerequisite: No interface has been generated for this BAPI yet, or any existing interface has already been deleted with the *Delete* option.
The system proposes names for the objects to generate. You can change these names if necessary.
Creating an interface involves the following process flow:

1.

Maintaining the BAPI-ALE Interface

2.

Maintaining the BAPI-ALE Interface

- **IDoc type:** <message_type>01
Example: MYTEST01
- **Outbound function module:** ALE_<OBJECT>_<METHOD>
Example: ALE_EXAMPLE_TEST
- **Inbound function module:** IDOC_INPUT_<message_type>
Example: IDOC_INPUT_MYTEST

Maintaining the BAPI-ALE Interface

Maintaining the BAPI-ALE Interface

- *Data filtering allowed*
If you want to perform data filtering for the BAPI, you will have to choose option *Data filtering allowed* in the dialog box for creating or changing the interface. This option is generally active for BAPI-ALE interfaces generated by SAP. For more information, see the [Notes \[Page 159\]](#).
- *Call in update task*
You have to set this flag if you want to perform database changes using methods of the update task.
- *Package processing allowed*
You have to select this option if you want to allow package processing of IDocs. The corresponding BAPI must support package processing. Use ALE customizing to configure the package size.

Maintaining the BAPI-ALE Interface

3.

Maintaining the BAPI-ALE Interface

4.

Maintaining the BAPI-ALE Interface

For individual fields as a header segment

Example: E1MYTEST, or

For parameters of structure BAPI_XX...

Example: E1BP_HUGO for BAPI_HUGO

Maintaining the BAPI-ALE Interface

If you define a segment as containing more than 1000 bytes of data, child segments are generated automatically.

Maintaining the BAPI-ALE Interface

– **Changing an interface**

Prerequisite: Objects have already been created for this BAPI.

Choose *Change* to re-generate the objects of an existing ALE interface after changes to the BAPI. The IDoc type and IDoc segments are re-generated when the interface structures and object methods have changed. The function modules are only re-generated when the IDoc type or one of the segments has changed.

A dialog box (similar for the creation case above) is displayed. This dialog box displays the objects that already exist in the system. These fields are not ready for input.

If a field is blank, you can generate the corresponding object.

– **Displaying an interface**

Prerequisite: Objects already exist for this BAPI.

All the existing objects for this BAPI are displayed. This gives you an overview of the relationship between the BAPI method and the IDoc message type.

– **Deleting an interface**

Prerequisite: Objects have already been created for this BAPI.

Any function modules that exist in the system are deleted.

The IDoc structure is deleted if it has not been released yet.

The IDoc segments are only deleted if they have not been released yet and are not used in any other IDocs.

Finally, the message type is deleted if it no longer has an assignment to an IDoc type.

– **Checking the interface**

Prerequisite: Objects have already been created for this BAPI.

All objects of the BAPI are checked to see whether they already exist in the system.

The system also checks whether objects (IDoc type and segments) have already been released.

You can change the release status of the objects (see *Set release* and *Cancel release* under the [Notes \[Page 159\]](#)).

3. You can release the interface.

Developers can change the release status of IDoc types and segments (under *Edit* → *Set release* or *Cancel release*).

To do this, you need authorizations S_IDCDFT_AL+ and S_IDCDFT_ALL of authorization object S_IDOCDEFT.

To release an object, the corresponding BAPI must already be released. When you release an object, the system first checks whether the generated interface under the BAPI method has the current status. If not, it prompts you to re-generate the interface. The system determines which segments and which IDoc type are relevant for release. A new status is then set for objects that have not been released yet.

The release can be cancelled at any time. This action is integrated with the transport system.

The generated function modules are not released.

Result

The processed objects and their status are displayed in the output screen. All changes are recorded in transport requests.

Maintaining the BAPI-ALE Interface

See also:

[Notes \[Page 159\]](#)

Notes

Please also note the following:

- **Namespace extension**

Namespace extension is also supported for customers and partners starting in Release 4.5A.

- **Filtering for data selection**

The filtering of data can be linked to conditions that are defined as filters in the distribution model.

If you want to perform data filtering, you will have to choose option *Data filtering allowed* in the dialog box for creating or changing the interface.

For more information, see the ALE Programmer Guide under [Data Filtering \[Ext.\]](#).

- **Serialization**

The function module on the outbound side has an optional parameter: SERIAL_ID (reference to the channel number). This input parameter controls the allocation of messages to message channels. All the messages in the target system are processed in an object channel in the same order they were created in the source system. An object channel is identified by a key from the BAPI object type and the channel number.

For more information on this subject, refer to the document "ALE Introduction and Administration" under [Serialization of Messages \[Ext.\]](#).

- **Relationships**

The relationship in ALE answers the following questions:

- On the outbound side:

Which application object was used to create the IDoc?

Prerequisite: The application has correctly filled parameter APPLICATION_OBJECTS in the function module on the outbound side.

- On the inbound side:

Which outbound IDoc was used to create the inbound IDoc, and which application object was created from this inbound IDoc?

Prerequisite: A key has been defined in the BOR for the BAPI method, and this key is contained in the exporting or importing parameters in the BAPI function module. This key can consist of several key fields. If no key fields exist in the BOR, then a relationship to a different object can be specified in the corresponding prompt.

- **Documentation on generated function modules**

Documentation has also been written for generated function modules (outbound/inbound).

You can call this documentation within the interface. It describes the meanings of the parameters and the values they can assume.

- **BAPI return parameters and IDoc status**

If the *Return* parameter has been filled by the application, the IDoc status and the corresponding information from the parameter is included in the IDoc. Message types determine the IDoc status.

Notes

If the *Return* parameter is an EXPORTING parameter, than a *single* IDoc status record is written:

Message type A:	Status 51 (application document not posted, with DB rollback)
Message type E:	Status 51 (application document not posted, no DB rollback)
Message type W, I, S:	Status 53 (application document posted)

If the *Return* parameter is a TABLES parameter, then several IDoc status records can be written, depending on the message types in the table:

Message type A:	Status 51 (application document not posted, with DB rollback)
Message type E:	Status 51 (application document not posted, no DB rollback)
Exception:	No IDoc status for type S
No message type A or E:	Status 53 (application document posted)

The IDoc status records are written in the same sequence as the messages in the *Return* parameter.

If the *Return* parameter was not filled, this means the BAPI has been successfully called by the IDoc. In this case, the ALE layer writes an IDoc status record with status 53 (application document posted).

If an error occurs, *only* the first message from the *Return* parameter is included in the text of the corresponding error task (work item).

- **Restrictions to interface generation**

- You should *not* use the generation function when parameters in the BAPI function module (IMPORTING and/or TABLES) have *the same reference structure*. In this situation, the mapping of IDoc segment to parameter is not unique. As a result, the parameters cannot be identified in IDoc inbound.
- *When a data volume of more than 1000 bytes* appears in a reference field for a parameter or a field of the reference structure, you *cannot* use the generation function, as only segments with a data volume of 1000 bytes or less can be loaded.

Maintaining the Distribution Model

Use

The distribution model describes the ALE message flow between logical systems – that is, it determines which messages are distributed to which logical systems. This BAPI-based distribution of data to recipients can be restricted according to your specific requirements.

For example, you can make distribution dependent on conditions that you define as filters in the ALE distribution model (through the R/3 Implementation Guide: *Basis* → *Application Link Enabling (ALE)* → *Model and Implement Business Processes* → *Maintain Distribution Model*).

The prerequisite for this is that a filter object type has been assigned to the corresponding BAPI in the SAP application. SAP has already defined some filter object types and assigned them to different BAPIs. You can also define your own filter object types and assign them to a BAPI. For more information on defining filter objects, refer to the document “ALE Introduction and Administration” under [Modeling Distribution \[Ext.\]](#).

We differentiate between different types of filtering for BAPIs:

Receiver Filtering

Before a BAPI or generated BAPI-ALE interface is called, its receivers have to be determined. To link the receiver determination with conditions, the business object methods are assigned to filter object types during receiver filtering. The values of these objects are used to determine the permitted receivers. The valid filter object values must be defined in the distribution model.

The following dependencies can be modeled in the ALE distribution model:

- Between a BAPI and a message type
- Between BAPIs

If a dependency of this type is defined as a condition in the ALE distribution model, then the receiver of the referenced BAPI or message type is determined.

For more information on receiver filtering and BAPIs, see the ALE Programmer Guide under [Determining Receivers for a BAPI \[Ext.\]](#).

Filtering Data

Two data filtering services are provided for asynchronous BAPI calls via the BAPI-ALE interface:

- **Interface reduction**

The BAPI reduction is without conditions, that is, it involves a projection of the BAPI interface.

In the process, the distribution model suppresses values for optional BAPI parameters and/or fields during data transmission.

The developer of the BAPI whose interface is to be reduced must create the BAPI as reducible using appropriate parameter types.

If you reduce the BAPI interface, you do not have to define any filter object types.

- **Parameter Filtering**

BAPI parameter filtering is linked with content-specific conditions: Lines in table parameters of an asynchronous BAPI are determined depending on the values in the lines (or dependent lines) for the receiver. The table dataset of a BAPI is determined for

Maintaining the Distribution Model

parameter filtering.

Hierarchical relationships can also be defined between table parameters of the BAPI.

To implement parameter filtering, business object methods are assigned to the business object method of filter object types. The valid filter object values must be defined in the distribution model.

For more information on data filtering and BAPIs, see the ALE Programmer Guide under [Filtering Data \[Ext.\]](#).

BAPI filtering is the term used for the shared use of both the filter services of the BAPI interface. BAPI filtering is only implemented as a service in ALE outbound processing.