

ALE Programming Guide



HELP.BCMIDALEPRO

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

Inhalt

ALE Programming Guide	7
Implementing Distribution Using BAPIs	8
Distribution Using BAPIs	10
Implementing Your Own BAPIs	16
Filtering Data	18
Reducing Interfaces	21
Defining and Assigning Filter Object Types	23
Filtering BAPI Parameters	25
Defining Hierarchies Between BAPI Parameters	27
Maintaining BAPI-ALE Interfaces	30
Determining the Receiver of a BAPI	35
Determining Filter Objects of a BAPIs	37
Determining Receivers of Asynchronous BAPIs	38
Determining Filter Objects Using Business Add-Ins	39
Example Programs with Asynchronous BAPI Calls	42
Determining Receivers of Synchronous BAPIs	47
Example Programs with Synchronous BAPI Calls	49
Determining Unique Receivers of Synchronous BAPIs	52
Developing BAPIs for Interactive Processing	53
Enhancing IDocs of BAPI-ALE Interfaces	54
Distribution Using Message Types	55
Distribution Using Message Types	56
Implementing Outbound Processing	60
Developing a Function Module for ALE Outbound Processing	61
Basics	62
Interrogating the Distribution Model	63
Structure of Control Records	64
Structure of the Data Records	65
Converting Currency Amounts	66
Replacing SAP Codes With ISO Codes	67
Left-justified Filling of IDoc Fields	68
Calling MASTER_IDOC_DISTRIBUTE	69
Exceptions and Export Parameters of MASTER_IDOC_DISTRIBUTE	70
Example of Generating an IDoc	71
Example Program to Generate an IDoc	72
Using the Example Coding	79
Customizing ALE Outbound Processing	80
Defining ALE Object Types	81
Assigning the Object Type for the Outbound Link to the Message Type	82
Assigning the Application Object Type for the Outbound Link to the Message Type	83
Outbound Processing Using Message Control	84
Implementing Inbound Processing	85
Inbound Function Module	86
Embedding a Function Module in ALE Inbound Processing	87
Data Consistency	88
Ensuring Data Consistency	89

Serialization	90
Processing IDocs Individually	91
Naming Convention	92
The Function Module's Interface	93
Import Parameters	94
IDoc Processing.....	95
Export Parameters.....	96
The Inbound Function Module's Export Parameters.....	97
Export Parameters When IDoc was Successfully Processed	98
Export Parameters When an Error Occurred in IDoc Processing.....	99
Example of Processing an IDoc	100
Example Program to Process an IDoc	101
Serialization Using Message Types.....	118
Example Program for Serialization	119
Customer Exits.....	125
Example Program for a Customer Exit.....	126
Mass Processing.....	132
Import Parameters	133
Export Parameters.....	134
All Inbound IDocs Processed Successfully	135
Error in One Inbound IDoc	137
Example Program for Mass Processing IDocs.....	139
Using Call Transaction.....	144
ALE-Enabled Transactions.....	145
Call Transaction Succeeds.....	147
Call Transaction Fails	149
Import Parameters in CALL TRANSACTION	150
Export Parameters in CALL TRANSACTION.....	151
Inbound Processing Successful	152
Error During Inbound Processing	153
ALE Settings	154
Declaring the Function Module's Attributes	155
Registering the Function Modules in Inbound Processing	156
Creating an Inbound Processing Code.....	157
Inbound Processing Using SAP Workflow	158
Work Items.....	159
Workflow	160
IDOCXAMPLE as a Reference for IDOC_PACKET	161
IDPKXAMPLE as a Reference for IDOC_PACKET	162
Advanced Workflow Programming.....	163
Setting the Parameter RESULT in the Event Container.....	164
Event inputErrorOccurred	165
Event inputFinished	167
Triggering an Application Event After Successful IDoc Processing	168
Using the Parameter NO_OF_RETRIES.....	170
Master Data Distribution.....	171
Defining the Message	172
Processing Outbound Master Data	173

Distributing Master Data Using the SMD Tool	174
Sending Master Data Directly.....	178
Processing Inbound Master Data.....	179
Connections to Non-SAP Systems	180
Translator Programs for Communication	182
Technical Implementation	183
TCP / IP Settings	184
Sending IDocs to an External System.....	185
Sending IDocs: External System to SAP System	187
Transaction Identification Management (TID)	189
Integrating Dialog Interfaces	191
Calls With References to the Logical System	193
Calls Without References to the Logical System.....	195
Serialization of Messages	197
Serialization by Object Type.....	198
Serialization By Message Type	200
Serialization at IDoc Level	201
Automatic Tests.....	202
Example Scenario for Distributing Master Data	203
Preparing the Test	204
Developing the Test Procedure.....	205
Error Handling.....	207
Objects, Events and Tasks to be Created	209
Object Types and Events	211
Creating IDoc Object Type: IDOCXAMPLE	212
Creating IDoc Packet Object Type: IDPKXAMPLE	214
Creating a Standard Task	215
Maintaining Inbound Methods.....	217
Checking Consistency of Inbound Error Handling	218

ALE Programming Guide

Purpose

You can add your own scenarios to the ALE business processes provided in the standard system. You can use one of two programming models to do this. Each dispatches a different type of message:

- [Distribution Using BAPIs \[Seite 8\]](#): This process has been supported since R/3 Release 4.0A and is the basis for future developments.
- [Distribution Using Message Types \[Seite 55\]](#): ALE developments in R/3 Release 3.x are based on this programming model.



Developers of desktop applications can also implement ALE business processes using the [IDoc Class Library \[Extern\]](#) for C++.

Client copy: logical system name in document header data.

When you are copying clients and logical systems have been defined for a client, logical systems must be converted in the header data of the copied documents, so that these documents can be found again in the new client.

When you are evaluating the statistics of documents, you have to check whether this field contains the logical system of the current client or the value SAPCE.

Only when these two conditions have been met, are documents from the current client included in the statistical evaluation.

Implementing Distribution Using BAPIs

Now that BAPIs and ALE can be integrated you can implement your ALE business processes using BAPIs.

BAPIs are methods of SAP business objects. They are defined in the Business Object Repository (BOR) and are subject to strict design guidelines. BAPIs are implemented as RFC-enabled function modules.

For further information on BAPIs see the [BAPI User Guide \[Extern\]](#) and BAPI Programming.

As of Release 4.5A BAPIs can also be defined that are implemented outside the R/3 System, but can be called from the R/3 System. For further information see [BAPIs Used for Outbound Processing \[Extern\]](#) in the BAPI Programming guide and [BAPIs of SAP Interface Types \[Extern\]](#) in the BAPI User Guide.

ALE provides a complete programming model for implementing BAPIs. ALE supports these method calls:

- Synchronous method calls

Synchronous method calls can also be used in ALE distribution scenarios. These method calls are either BAPIs or [Dialog Methods \[Seite 191\]](#).

In ALE Customizing you can assign the RFC destinations to be used for a synchronous method call.
- Asynchronous method calls

If BAPIs are called asynchronously, ALE error handling and ALE audit can be used.

If an asynchronous BAPI call is to be used for the distribution, the BAPI-ALE interface required for inbound and outbound processing can be automatically generated.

Developing an ALE business process in ABAP is restricted to the programming of the BAPI.

An object-oriented approach has the following advantages:

 - The application only has to maintain one interface
 - The automatic generation of the BAPI-ALE interface avoids programming errors.

Process Flow

If you are not enhancing an SAP BAPI and you are not creating your own BAPI when you are implementing an ALE business process, you can simply follow the steps below:

- [Filtering Data \[Seite 18\]](#)
- [Determining the BAPI Receivers \[Seite 35\]](#)

If, on the other hand, you want to enhance a BAPI or create your own, you have to follow these steps:

- [Implementing Your Own BAPIs \[Seite 16\]](#)
- [Maintaining the BAPI-ALE Interface \[Seite 30\]](#)
- [Determining the BAPI Receivers \[Seite 35\]](#)

Application programs must call a function module for the receiver determination and a generated application function module in the BAPI-ALE interface.

You can verify the quality of the ALE layer and ALE business processes using [Automatic Tests \[Seite 202\]](#).

See also:

[BAPIs for Interactive Processing \[Seite 53\]](#)

[Enhancing IDocs of BAPI-IDoc Interfaces \[Seite 54\]](#)

Distribution Using BAPIs

BAPIs can be called by applications synchronously or asynchronously. ALE functions such as BAPI maintenance in the distribution model and receiver determination can be used for both types of call.

Note that synchronously-called BAPIs are only used for reading external data to avoid database inconsistencies arising from communication errors.



The application synchronously calls a BAPI in the external system to create an FI document. The document is correctly created but the network crashes whilst the BAPI is being executed. An error message is returned to the application and the FI document is created again. The document has been duplicated in the system called.

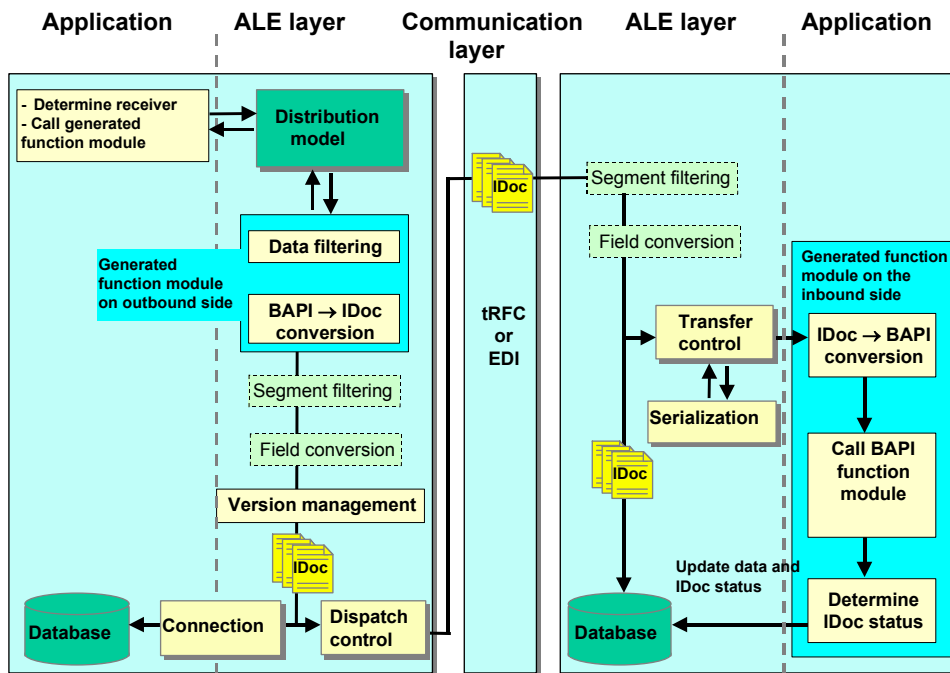
An application program can implement a two-phase commit by thoroughly checking the data in the external system. An easier solution is to call the BAPI asynchronously, as [Error Handling \[Seite 207\]](#) assures that the data remains consistent.

A BAPI should be implemented as an asynchronous interface, if one of the criteria below applies:

- Consistent database changes in both systems
Data must be updated in the local system as well as on a remote system
- Loose coupling
An asynchronous interface would represent too narrow a coupling between the client and the server systems. If the connection fails the client system can no longer function correctly.
- Performance load
The interface is used often or it handles large volumes of data. A synchronous interface cannot be used in this situation because performance would be too low.

If you want to implement a BAPI as an asynchronous interface, you have to generate a BAPI-ALE interface for an existing BAPI. For more information see [Generating BAPI-ALE Interfaces \[Seite 30\]](#).

Data distribution using BAPIs is illustrated in the graphic below:



The processes in the application layer and the ALE layer are completed on both the inbound and outbound processing sides. The communication layer transfers the data by transactional Remote Function Call (tRFC) or by EDI file interface.

The process can be divided into the following sub-processes:

1. Outbound Processing

- Receiver determination
- Calling the generated outbound function module
- Conversion of BAPI call into IDoc
- Segment filtering
- Field conversion
- IDoc version change
- Dispatch control

2. IDoc dispatch

IDocs are sent in the communication layer by transactional Remote Function Call (tRFC) or by other file interfaces (for example, EDI).

tRFC guarantees that the data is transferred once only.

3. Inbound Processing

- Segment filtering
- Field conversion
- Transfer control
- Conversion of IDoc into BAPI call
- BAPI function module call
- Determination of IDoc status

Distribution Using BAPIs

- Posting of application data and IDoc status
- Error handling

The sub-processes in inbound and outbound processing are described below:

Outbound Processing

On the outbound side first of all the receiver is determined from the distribution model. Then the outbound function module that has been generated from a BAPI as part of the BAPI-ALE interface is called in the application layer (see also [Example Programs with Asynchronous BAPI Calls \[Seite 42\]](#)). In the ALE layer the associated IDoc is filled with the filtered data from the BAPI call.

The volume of data and time of the data transfer is controlled by the dispatch control.

The outbound processing consists of the following steps:

Receiver determination

The receivers of a BAPI call are defined in the distribution model in same way as with synchronous BAPI calls.

Before the BAPI or generated BAPI-ALE interface can be called, the receiver must be determined. When the receiver is determined, the filter objects are checked against the specified conditions and the valid receivers are reported back.

If the distribution of the data is also dependent on conditions, these dependencies between BAPIs or between BAPIs and message types are defined as **receiver filters**.

For each of these receiver filters, before the distribution model is defined, a filter object is created whose value at runtimes determines whether the condition is satisfied or not.

For more information see [Determining Receivers of BAPIs \[Seite 35\]](#).

Calling the generated outbound function module

If the receivers have been determined, you have to differentiate between local and remote receivers. The BAPI can be called directly for local receivers. For remote calls the generated ALE outbound function module must be executed so that processing is passed to the ALE layer. The data for the BAPI call and the list of allowed logical receiver systems are passed to this function module.

Programming Notes:

After calling the generated function module the application program must contain the command COMMIT WORK. The standard database COMMIT at the end of the transaction is not sufficient. The COMMIT WORK must not be executed immediately after the call, it can be executed at higher call levels after the function module has been called several times.

The IDocs created are locked until the transaction has been completed. To unlock them earlier, you can call the following function modules:

DEQUEUE_ALL releases all locked objects

EDI_DOCUMENT_DEQUEUE_LATER releases individual IDocs whose numbers are transferred to the function module as parameter values.

Data Filtering

Two filtering services can be used - parameter filtering with conditions and unconditional interface reduction.

- If entire parameters have been deactivated for the interface reduction, they are not included in the IDoc. If, on the other hand, only individual fields are not to be included for structured parameters, the entire parameters are still included in the IDoc.
- With parameter filtering, table rows that have been filtered out are not included in the IDoc.
For more information see [Filtering Data \[Extern\]](#).

Conversion of BAPI call into IDoc

Once the data has been filtered, an IDoc containing the data to be transferred, is created from the BAPI call by the outbound function module

Segment filtering

Once the IDoc has been created, IDoc segments can be filtered again. This filtering is rarely used for BAPIs.

For more information see the R/3 Implementation Guide under:

Basis
Application Link Enabling
Modeling and Implementing Business Processes
Master Data Distribution
Scope of Data for Distribution
Message Reduction

Field conversion

You can define field conversions for specific receivers in the R/3 Implementation Guide:

Basis
Application Link Enabling
Modeling and Implementing Business Processes
Converting Data Between Sender and Receiver

Standard rules can be specified for field conversions. These are important for converting data fields to exchange information between R/2 and R/3 Systems. For example, the field *plant* can be converted from a two character field to a four character field.

Standard Executive Information System (EIS) tools are used to convert fields.

IDoc version change

To guarantee that ALE works correctly between different releases of the R/3 System, IDoc formats can be converted to modify message types to suit different release statuses.

If version change has been completed, the IDocs are stored in the database and the dispatch control is started which decides which of these IDocs are sent immediately.

SAP uses the following rules to convert existing message types:

- Fields can be appended to a segment type
- New segments can be added
ALE Customizing records the version of each message type used in each receiver. The IDoc is created in the correct version in outbound processing.

Dispatch control

Scheduling the dispatch time:

- IDocs can either be sent immediately or in the background. This setting is made in the partner profile.
- If the IDoc is sent in the background, a job has to be scheduled. You can choose how often background jobs are scheduled.

Distribution Using BAPIs

Controlling the amount of data sent:

- IDocs can be dispatched in packets. The packet size is assigned in ALE Customizing in accordance with the partner profile.

Basis

Application Link Enabling

Modeling and Implementing Business Processes

Partner Profiles and Time of Processing

Maintain Partner Profile Manually

or:

Generate Partner Profiles



This setting is only effective if you process the IDocs in the background.

Inbound Processing

On the receiver side the ALE layer continues with the inbound processing.

On the application side when the generated inbound function module is executed, the BAPI call is generated from the IDoc, the BAPI function module is called and the IDoc status is determined.

After the BAPI or the entire packet has been processed, the IDoc status records of all IDocs and the application data created from successfully completed BAPIs, are posted together.

The inbound processing consists of the following steps:

Segment filtering

On the inbound side IDoc segments can be filtered the same as they can on the outbound side. This filtering on the inbound side is also rarely used for BAPIs.

Field conversion

As with outbound processing, fields can be converted if the field format is different in the receiving and sending systems.

After the fields have been converted, the IDoc is saved on the database and it is passed to the transfer control for further processing.

Transfer control

The transfer control decides when the application BAPIs are to be called. This may be either immediately when the IDoc arrives or at a later time in background processing.

If several mutually dependent objects are distributed, **serialization** can be used during the transfer control. IDocs can be created, sent and posted in a specified order by distributing message types serially. Errors can then be avoided when processing inbound IDocs.

If BAPIs are used object serialization is used exclusively. This assures that the message sequence of a particular object is always protected.

For more information about used object serialization see ALE Customizing.

Basis

Application Link Enabling

Modeling and Implementing Business Processes

Master Data Distribution

Converting Data Between Sender and Receiver

Serialization by Object Type

When the time arrives for processing the BAPI, the generated inbound function module is called.

Conversion of IDoc into BAPI call

When the BAPI is called, the entire data from the IDoc segments is written to the associated parameters of the BAPI function module. If an interface reduction has been defined for the BAPI, the hidden fields are not filled with the IDoc data.

BAPI function module call

Next the BAPI function module with the filled parameters is executed synchronously. As the BAPI does not execute a COMMIT WORK command, the application data that it has created, modified or deleted is not yet saved in the database.

IDoc status determination

If the function module has been executed, the IDoc status is determined in the inbound function module from the result of the call.

Posting of application data and IDoc status

If each IDoc or BAPI is processed individually, the data is written immediately to the database.

If several IDocs are processed within one packet, the following may happen:

- The application data of the successfully completed BAPI together with all the IDoc status records is updated, provided that no BAPI call has been terminated within the packet.
- As soon as a BAPI call is terminated within the packet, the status of the associated IDoc will indicate an error. Application data will not be updated. Then inbound processing is run again for all the BAPI calls that had been completed successfully. Provided that there is no termination during this run, the application data of BAPIs and all the IDoc status records are updated. This process is repeated if there are further terminations.

Note: Packet processing is only carried out if there is no serialization.

Error handling

You can use SAP Workflow for ALE error handling:

- The processing of the IDoc or BAPI data causing the error is terminated.
- An event is triggered. This event starts an error task (work item).
- Once the data of the BAPI or IDoc has been successfully updated, an event is triggered that terminates the error task. The work task then disappears from the inbound system.

For more information see [Error Handling \[Seite 207\]](#).

Implementing Your Own BAPIs

SAP provides a large number of BAPIs. If you want to implement your own BAPIs, you have to use your own namespace.

Procedure

You have the following options:

- You can develop your own BAPI in the customer namespace.
- You can modify a BAPI delivered in the standard system.
- 1. Copy and modify the function module belonging to the original BAPI.
- 2. In the Business Object Repository create a subobject type for your BAPI object type in the customer namespace (*Tools → Business Framework → BAPI Development → Business Object Builder*).
When you create the subobject type the methods of the business object inherit the subtype.
- 3. Set the status of the object type to *Implemented* (*Edit → Change release status → Object type*).
- 4. You can change and delete the methods of the subtype or enhance them with your own methods.

For further information about creating new BAPIs and enhancing existing ones refer to [Modifications and Customer Enhancements \[Extern\]](#) in the BAPI Programming guide.

Notes about Asynchronous BAPIs

If you want to implement an asynchronous ALE business process, you have to [Define a BAPI-ALE Interface \[Seite 30\]](#) from the BAPI.

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

- The BAPI must not issue a COMMIT WORK command.
- The method's return parameter must use the 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.
- Status records log the BAPI return parameter values.

After the function module which converts the IDoc into the corresponding BAPI 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 field *Type* 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 ROLLBACK WORK is executed.
Otherwise status 53 (application document posted) is written and a COMMIT WORK executed.

Filtering Data

Filtering Data

There are two filtering services provided for asynchronous BAPI calls using the BAPI-ALE interface.

- Interface Reduction:

If you want to reduce the BAPI interface, you do not have to define any filter object types.

The BAPI reduction does not have any conditions - it is a projection of the BAPI interface.

The developer of the BAPI whose interface is to be reduced must create the BAPI as a reducible using appropriate parameter types.

The optional BAPI parameters and/or BAPI fields are deactivated in the distribution model for the data transfer.

You can reduce an interface in two ways, see [Reducing Interfaces \[Seite 21\]](#)

- By fields (using checkbox lists)
- Fully

- Parameter Filtering

[Filter Object Types \[Extern\]](#) are assigned to the business object method. The valid filter object values must be defined in the distribution model.

The BAPI parameter filtering is linked to conditions, it is therefore content-dependent: The lines in table parameters of asynchronous BAPIs are determined depending on the values in the lines (or dependent lines) for the receiver.

Filters are used to define conditions in the form of parameter values that must be satisfied by BAPIs before they can be distributed in ALE outbound processing.

The table dataset of a BAPI is determined when the parameters are filtered.

Hierarchy relationships between table parameters of the BAPI can also be defined.

Distribution by [Classes \[Extern\]](#) is also supported.

For more information see [Filtering BAPI Parameters \[Seite 25\]](#)

BAPI filtering is the term used for the shared use of both the filter services of the BAPI interface. BAPI filtering is implemented as a service in ALE outbound processing.

Prerequisites for Using Filter Services

The table below lists the prerequisites that the BAPI interface must satisfy, so that ALE filter services can be used.

The BAPI can have the following parameter types:

	Field Reduction	Full Filtering	Parameter Filtering
1. Unstructured without checkbox			
2. Unstructured with checkbox	X		
3. Single-line structured without checkbox			
4. Single-line structured with checkbox	X		

5. Multiple-line structured without checkbox		X	X
6. Multiple-line structured with checkbox	X	X	X
7. Multiple-line unstructured without checkbox		X	
8. Multiple-line unstructured with checkbox			

Note: The fields filled with **X** satisfy the prerequisites.

Explanation of above table:

1. An unstructured parameter without a checkbox is, for example, a BAPI key field (e.g. the parameter *Material* in methods of the business object *Material*). This parameter type cannot be reduced.
2. If there is an unstructured checkbox parameter with the name PX and the data element BAPIUPDATE for an unstructured parameter with the name P, the parameter P is reducible. The parameter is reduced by setting the value of P and of the checkbox parameter PX to EMPTY.
3. A single-line, structured parameter without a checkbox is not reducible.
4. A single-line, structured parameter P with structure S and associated checkbox PX with structure SX can be reduced by fields, provided that:
 - S and SX have the same number of fields, which are identical in name and sequence.
 - The FUNCTION field and the key fields in S and SX each have the same data element.
 - All other fields in SX have the data element BAPIUPDATE.

The FUNCTION field in P and the key fields must be marked as mandatory fields. All the other fields you can chose whether to label them as mandatory. Mandatory fields cannot be reduced. Non-mandatory fields are reduced by setting the field values and the corresponding checkbox to EMPTY.
5. Multiple-line structured parameters (table parameters) without a checkbox cannot be reduced by fields. Parameter filtering and full filtering are possible.
If the hierarchy is maintained and, if dependent tables exist in the hierarchy, records of the dependent tables will also be filtered.
6. A multiple-line structured parameter P with checkbox PX can be reduced by fields, fully filtered or filtered by parameters.
 - For field reduction the prerequisites under 4 must be met.
 - The checkbox PX must lie directly under P in the hierarchy, with identical key fields, so that the corresponding lines from P and PX are filled, when the parameters are filtered.
 - If the hierarchy is maintained and, if dependent tables exist in the hierarchy, records of the dependent tables will also be filtered.
7. A multiple-line, unstructured parameter can only be fully filtered and cannot be used in a hierarchy.
Parameter filtering is not allowed.
8. Multiple-line, unstructured parameters with a checkbox cannot be filtered.

Reducing Interfaces

Use

The purpose of BAPI and ALE integration is to be able to use ALE business process BAPIs as interfaces.

BAPI reductions are particularly necessary in ALE business processes in which master data is replicated asynchronously:

- Part of the BAPI parameter is not required for the receiving system, even though it is declared when the BAPI is called.
- Monitor data transferred into non-SAP systems (non R/3 and/or between business partners) (for example, hide fields).
- Certain data cannot be overwritten in the receiving system.

BAPI reductions can however be used everywhere where asynchronous BAPI calls can be used.

For asynchronous BAPI calls via the BAPI-ALE interface, only the parameters of the BAPI interface relevant for the receiver should be transferred. You can set up BAPI reductions in receiver-dependent filtering in the ALE distribution model. You can create templates for making reductions.



Material master data is replicated from a reference system to a sales and distribution system. As only some of the data on the material is required in the sales and distribution system, a reduction of the BAPI interface, **Material.SaveReplica**, that contains parameters relevant only to sales and distribution, is specified. You can then specify in the distribution model that with **Material.SaveReplica** only data relevant to sales and distribution is transferred to the sales and distribution system.

You can filter BAPIs (parameter filtering and reduction), when you maintain the distribution model. Reduction and filter information are part of ALE Customizing data in the distribution model.

BAPI filtering must be explicitly activated, when the BAPI-ALE interface is generated.

The reduction of the actual (asynchronous) BAPI call is carried out as a service in the ALE layer.

The reduction service retrieves the details of the filter settings from the distribution model at runtime.

For a receiver or a list of receivers the application development can query the list of parameters to be filled before the BAPI-ALE interface is called. This keeps the read-access to the database as low as possible. (Alternatively the call can take place and it does not affect the result of the filtering.)

You can only set up one BAPI reduction for each sender and receiver pair.

Prerequisites

The basic data of the BAPI reduction is maintained by the BAPI developer after the BAPI has been released and before the BAPI-ALE interface is generated. If a parameter hierarchy is to be used, this has to be specified beforehand.

The BAPI developer must create the BAPI as reducible using relevant parameter types.

Mandatory parameters and fields must be specified.

The section [Filtering Data \[Extern\]](#) has a table listing the prerequisites for using filter services.

Prerequisites

Fully Reducible Parameters

Only table parameters of BAPIs can be fully reduced. A fully reduced table is an empty table in the receiving system.

To fully reduce a table parameter T1 with a checkbox, the following prerequisites apply:

Table Parameter	Structure
T1	Q1
T1X	Q1X

T1X is a checkbox parameter.

Reducing Parameter Fields

Fields are reduced by converting the obligatory check fields of a BAPI and initializing the relevant fields in the data parameter. The checkboxes must be assigned to the data parameters following the naming and structure conventions.

The following prerequisites apply for reducing fields of parameter P1:

Table Parameter	Structure
P1	S1
P1X	S1X

Structures S1 and S1X must have the same number of fields, whereby the names of the fields in both parameters must be identical and in the same order.

If P1 has a FUNCTION field or key fields, the FUNCTION field in S1 and S1X and each of the key fields have the same data element. All other fields of the checkbox use the data element BAPIUPDATE.

Procedure

To reduce BAPIs:

1. Create a reducible BAPI that satisfies the above prerequisites.
2. Before generating the BAPI-ALE interface, you have to activate data filtering (*option Data filtering allowed*).

You can set up the filtering in the distribution model in Customizing by choosing *Distribution (ALE) → Modeling and Implementing Business Processes → Maintain Distribution Model*.

Result

The generated BAPI-ALE interface enables BAPIs to be filtered as a service in outbound processing.



To avoid unnecessary accesses to the database, the BAPI parameters required for the receivers can be determined before the BAPI-ALE interface is called. This is optional and will not affect the results of the filtering.

Defining and Assigning Filter Object Types

[Filter object types \[Extern\]](#) are already assigned to some BAPIs in your applications for the receiver and data filtering.

You can also define your own filter object types and assign them to a BAPI or to a parameter of a BAPI.

Process Flow

To define filter object types for BAPIs, follow the steps below:

- Define filter object types

From the SAP menu choose *Tools → ALE → ALE Development → BAPIs*.

You can create filter object types in *Data filtering* or *Receiver Determination*.

Then choose *Define filter object type* (Transaction BD95, table TBD11).

Give the filter object type a name and specify a reference to a table field. The reference to a table field is needed to retrieve the documentation from the data element so that customers can get input help when maintaining the distribution model. For this reason a foreign key must be maintained for the table field.

Use the following conventions to name filter object types:

- Release 3.0/3.1: Domain name (example: KOKRS for the controlling area)
- Release 4.0: Default field name for the data element (example: COMP_CODE for the company code)

For the required data object check whether a name has already been entered in the domain and default field names of the data element. If the fields are empty, a new filter object must be created. Usually the filter object will also appear in the BAPI interface as a field in the transfer structure, for example, `bapiachead-comp_code`.

If this is the case create the filter object as follows:

ALE object type:	Field name in BAPI structure, for example,
<code>comp_code</code>	
table name:	Name of BAPI structure, for example, <code>bapiachead</code>
field name:	Field name in the BAPI structure, for example,
<code>comp_code</code> .	

- Defining filter object types to a BAPI

From the SAP menu choose *Tools → ALE → ALE Development → BAPI Interface*.

You can assign filter object types to a BAPI in *Data filtering* or *Receiver Determination*.

The filter object types allowed for an object method are maintained in each view of Table TBD16.

- *Receiver Determination.*

Choose *Assign filter object type to BAPI*.

You can maintain the entries:

Object type (from table TOJTB),
Method,
Filter object type (from table TBD11)

Keep in mind that for receiver determination you have to implement a business add-in f to determine values for the filter object type you have defined (see [Determining Filter Objects Using Business Add-Ins \[Seite 39\]](#)).

- *Data filtering*

Defining and Assigning Filter Object Types

Choose *Assign filter object type to parameter*.

You can maintain the entries:

- Object type* (from Table TOJTB),
- Method*,
- Filter object type* (from Table TBD11)
- Parameter*
- Field name*

Enter the required data to assign a filter object to an object method for the *receiver determination* or *parameter filtering*.

Filtering BAPI Parameters

Use

Parameter filtering enables you to manage the number of datasets to be replicated in the BAPI interface using filter objects in the ALE distribution model.

The parameters filtered are BAPI table parameters. The lines in the BAPI parameter, that do not match the distribution specifications are filtered out. The filtered table lines are not replicated.



Example: The logical system Q4VCLNT800 is the BAPI server for the BAPI *RetailMaterial.Clone*. Through the parameter filtering only the plant data of the material plant 001 is to be replicated in this system.

Prerequisites

The prerequisite for this filtering is that a [Filter Object Type \[Extern\]](#) is assigned to the relevant BAPI in your SAP applications. For some BAPIs SAP has already defined and assigned filter object types. You can also define your own filter object types and assign them to a BAPI ([Defining Filter Object Types and Assigning Them to a BAPI \[Seite 23\]](#).)

You have to define the valid filter object values in the distribution model. For more information see the R/3 Implementation Guide under *Distribution (ALE) → Modelling and Implementing Business Processes → Maintain Distribution Model*.

Presently, BAPI parameters can *only* be filtered for distributing master data via BAPIs called *asynchronously*. For this reason the required ALE Customizing for parameter filtering is only allowed for asynchronous BAPIs with an ALE IDoc interface.

Parameter filtering is allowed for distributing transaction data via asynchronous BAPIs but for the most cases, it has no purpose.

BAPI parameter filtering for asynchronous parameters is always optional. To generate the [BAPI-IDoc Interface \[Seite 30\]](#) you must select the *Activate* checkbox. Otherwise no coding can be generated in the BAPI-IDoc interface.

If a BAPI-IDoc interface has been generated *without* parameter filtering, you can specify no parameter filtering in ALE Customizing afterwards.

Features

Parameters are filtered dynamically at runtime using the current data in the BAPI table parameters and the distribution conditions specified in the ALE distribution model.

- Reads the specified parameter filter objects in the distribution model
- Reads the interface definition of the BAPI
- Reads the table field values of a table entry for the associated filter objects
- Compares the distribution conditions with the filter objects read out and determines the value of the logical expression
- Deletes table entry
- Examines hierarchy-dependent BAPI parameters and, if applicable, deletes dependent table entries
- For synchronous BAPIs: Calls associated function module and forwards the filtered parameters
- For asynchronous BAPIs: Calls the generated BAPI-ALE interface and forwards the filtered parameters

Filtering BAPI Parameters

The BAPI parameter filtering can also take into account a hierarchical dependency between BAPI table parameters (see [Defining Hierarchies Between BAPI Parameters \[Seite 27\]](#)). You must specify any hierarchical dependencies before you generate the BAPI-ALE interface of the BAPI. The specified hierarchy is evaluated when the interface is generated and incorporated in the interface coding. The BAPI-ALE interface must be regenerated following all subsequent changes made to the hierarchy.

Once the generated IDoc type has been released, the specified hierarchy of the asynchronous BAPI cannot subsequently be changed because of compatibility problems.

Defining Hierarchies Between BAPI Parameters

Use

If you are developing your own ALE business processes, you may have to define dependencies between BAPI table parameters with regard to filtering parameters for data selection.

These dependencies are defined by the field references between the table parameters of BAPIs.

You can filter parameters to determine the dataset and to define dependencies *only* for the distribution of master data via BAPIs that are called synchronously.



A BAPI for material master data contains the tables for plant data and associated storage data. The table containing plant data has a reference to the table containing storage data via the key field PLANTS. There is a hierarchical dependency between the plant and storage data. If plant 001 of a material is not to be replicated due to parameter filtering, then none of the storage data for plant 001 will be replicated.

Prerequisites

You use BAPI parameter filtering to manage the size of the dataset in the BAPI interface.

Procedure

You can define these hierarchical dependencies in ALE Development under *BAPI → Maintain hierarchy of table parameters*.

Enter the object type and the method of the BAPI. You can display existing BOR object types and their associated methods using the input help (F4).

The following processing options are available under the menu *Hierarchy* :

- Create
- Change
- Display
- Delete

Create Hierarchy

This checks whether a hierarchy for the BAPI already exists. Then it checks whether an ALE IDoc interface has already been generated and whether the associated IDoc has been released.

If the IDoc has already been released, then the generated interface has already been delivered to customers and no hierarchy can be created or changed for an existing BAPI because of compatibility problems. In this case you have to create a new BAPI. A corresponding error message is displayed. If the ALE interface already exists, but the IDoc has not yet been released, then the system will inform you that it needs to be regenerated.

A hierarchy tree is displayed on the next screen. For details see *Editing the Hierarchy Display* further below.

Change Hierarchy

The same checks are made as when you create a hierarchy. On the next screen the same processing options are provided as when you create a hierarchy.

Display Hierarchy

The same checks are made as when you create a hierarchy. On the next screen you cannot make any changes to the hierarchy.

Defining Hierarchies Between BAPI Parameters

To display the field references between the tables, double-click on the parent table. The parent table is automatically copied to the next dialog box. Select one of the child tables from the input help.

Select *Field references* to display the field references.

Delete Hierarchy

The same checks are made as when you create a hierarchy. Once you have confirmed you want to delete the BAPI hierarchy, it is deleted.

Editing the Hierarchy Display

The root node in the hierarchy display corresponds to the function module of the BAPI. The root node is used only for display and is not saved. Also, it cannot be changed.

You can edit the hierarchy display as follows:

- Insert table parameters
- Delete table parameters
- Define field references between parent and child tables
- Save hierarchy

Parent tables inserted directly under the root node that do not have child tables are not saved. If only this type of table is created, there is no hierarchy and therefore a hierarchy cannot be saved.

Insert table parameters

Place the cursor on a hierarchy node and choose *Edit → Insert table parameters*.

If you place the cursor on a root node, you can select a parent table of the highest level via the input help.

If a table exists above the marked node, this is copied to the next dialog box and you can add a child table to this table.

In principle a table can only exist once in the hierarchy. You can display the available tables via the input help.

In the dialog box you can display the common fields of the parent and child of the table in which the field references can be defined by selecting *Field references*. You can mark the fields for which a field reference is to be defined. If no field references between the two tables exist, an error message is displayed.

Delete table parameters

To delete a table, place the cursor on the relevant node of the hierarchy with the table names of the child table. Confirm the deletion. All other child tables of the deleted table will also be deleted.

Define field references between parent and child tables

Place the cursor on the node of a child table and choose *Edit → Table parameters → Change field references* or select the pushbutton *Field reference* with the change icon.

The next dialog box contains the parent parameters and provided that a reference exists, the child parameters too.

When you create table parameter, you can select the associated child table via the input help.

You can display common fields by selecting *Field references*. Only field references are displayed that have the same names in the parent and child tables.

You can define the references between the fields by marking the appropriate references. Field references already defined are marked already.

Save hierarchy

To save a hierarchy, choose *Hierarchy* → *Save*.

A transport request is generated to send the associated Customizing table to the correction and transport system.

The hierarchy is not saved if an error occurs when accessing the database. A corresponding error message is displayed.

Maintaining BAPI-ALE Interfaces

The standard R/3 System contains a large quantity of business objects and BAPIs. These include BAPI-ALE interfaces that are generated from BAPIs and enable asynchronous BAPI calls in ALE business processes.

You can develop your own BAPIs in the customer namespace and generate the associated BAPI-ALE interface.

The following objects are generated for a BAPI:

- *Message type*
- *IDoc type* including segments
- Function module called in the outbound processing side. (It creates and sends the IDoc from the BAPI data).
- Function module that calls the BAPI with the IDoc data on the inbound processing side

The difference to manually maintained message types is that the function module that processes the change pointers does not create an IDoc. Instead it fills the corresponding BAPI structures, determines the receivers and calls the generated ALE function module.

The message types, IDoc types and function modules that have been generated can also be used to distribute master data using the SMD tool.

Prerequisites

The essential prerequisite is that a BAPI exists:

- You have developed your own BAPI in the customer namespace.
- You have modified a BAPI from the standard system.

The BAPI-ALE interface is then created in the customer namespace for the new sub-type and a method assigned to it.

For more information see [Implementing Your Own BAPIs \[Seite 16\]](#)



Regardless of whether SAP delivers a BAPI-ALE interface for a BAPI with the new Release, any interface you have generated will continue to function in the same as in the earlier Release. You can regenerate the old interface to adapt newly added parameters, provided that SAP has not delivered a new interface in the new Release.

If SAP delivers a BAPI-ALE interface for a BAPI for which you have already generated an interface, you should use the new interface and delete the interface you generated. You can still use the old interface in the earlier Release. If you regenerate the old interface, some generated objects, such as segments of SAP objects could be overwritten, if the interface in your BAPI function module references BAPI structures but belongs to SAP.

If you want to take into account hierarchical dependencies between BAPI table parameters, then another prerequisite is that you define the hierarchy before generating the BAPI-ALE interface (see [Defining Hierarchies Between BAPI Parameters \[Seite 27\]](#)). The specified hierarchy is evaluated when the interface is generated and incorporated in the interface coding. The BAPI-ALE interface must be regenerated following all subsequent changes made to the hierarchy.

Once the generated IDoc type has been released, the specified hierarchy of the asynchronous BAPI cannot subsequently be changed because of compatibility problems.

Refer also to the notes on related ALE topics at the end of this section. Here you will find information on data filtering and serialization of message types.

Procedure

From the *ALE Development* screen choose *BAPIs → Define ALE interface*.

In customer systems all object names must begin with Y or Z or with their own prefix.

Proceed as follows:

1. Enter the business object (object type) and the method underlying the interface.
2. In the menu path *Interface* choose one of the options - *Create*, *Change*, *Display*, *Check* or *delete*.
 - **Create interface**

Prerequisites An interface has not yet been generated for this BAPI or one already generated has been deleted using the option *Delete*.

Names for the objects to be generated are suggested. You can change these as required.

 1. Enter a name for the message type.

A dialog box appears.

The default names have the following naming convention:

Message type:	Business name
Example:	MYTEST

Confirm your entry.
 2. In the next dialog box you can enter the following:

IDoc type:	<Message type>01
Example:	MYTEST01

Inbound function module: ALE_<OBJECT>_<METHOD>

Example:

ALE_EXAMPLE_TEST

Inbound Function Module: IDOC_INPUT_<Message type>

Example:

IDOC_INPUT_MYTEST

The default development classes and function groups are those that the BAPI function module belongs to. You should use your own development classes and function groups when you generate your own interface.

You have the following options:

 - Data filtering allowed

If you have filtered the data, you have to select the option *Data filtering allowed* in the dialog box when you are creating or changing the filters. With BAPI-ALE interfaces generated by SAP, this option is usually selected. Keep in mind the following notes:
 - Call in update task

If the database changes are carried out by methods using the update task, this indicator must be selected.
 - Packet processing allowed

If you want to allow packet processing, you have to select this option. The associated BAPI must be able to process packets. You can set the packet size in ALE Customizing.

You can only generate the message type (mandatory field) and the IDoc type (mandatory field) by leaving the field blank for which no object is to be generated.
 3. Confirm your entries.

Maintaining BAPI-ALE Interfaces

The dialog box *Enter segment name* appears.

4. Enter a segment name.

The suggested **segment type** is derived from the message type or from the BAPI structure:

- *E1<Message type>* for individual fields as header segments
Example: E1MYTEST or
- *E1BP_XX...* for parameters of the structure BAPI_XX.
Example: E1BP_HUGO for BAPI_HUGO

If a segment contains more than 1000 bytes of data, child segments are automatically generated from it. Child segment names are extended with the digits 1, 2,...appended to the original segment provided that the length of the name is less than 27 digits.

- **Change interface**

Prerequisites Objects have already been created for this BAPI.

To regenerate the objects of an existing ALE interface after an object method has been changed, choose *Change*. The IDoc type and the IDoc segments are regenerated, if the interface structures of the object method have changed. The function modules can only be regenerated, if the IDoc type or one of the segments has changed.

As when you create an object, a dialog box is displayed here also. The objects that already exist in the system are displayed in this dialog box. They are not input fields.

If a field is empty you can generate the associated object.

- **Display interface**

Prerequisites Objects already exist for this BAPI.

All existing objects for this BAPI are displayed. This gives you an overview of the relationship between the BAPI method and the IDoc message type.

- **Delete interface**

Prerequisites Objects have already been created for this BAPI.

The function modules can be deleted provided that they exist in the system.

The IDoc structure is deleted, provided it has not already been released.

The IDoc segments can only be deleted if they have not been released and are not used in other IDocs.

Finally, the message type is deleted, if it is no longer assigned to the IDoc type.

- **Check interface**

Prerequisites Objects have already been created for this BAPI.

The system checks whether all the objects related to this BAPI are available in the system. The system also checks whether objects (IDoc type and segments) have been released.

The release status of objects can be changed (see *set release* and *cancel release* in the *Notes* below).

5. You can release the interface.

Developers can change the release status of IDoc types and segments (Edit → *Set release* or *Cancel release*).

The authorizations required are S_IDCDFT_AL+ and S_IDCDFT_ALL of the authorization object S_IDOCDEFT.

Before an object can be released, the BAPI must have already been released. If the object is released, the system first verifies that the generated interface in the BAPI

method has the current status. If the object is not released, you are asked if you want to regenerate the interface. You are notified of the segments and IDoc type relevant for the release. The new status is assigned to objects not yet released.

The release can be reset at any time. This action is linked to the transport system.

The generated function modules are not released.

Result

The generated objects and their statuses are displayed on the screen. All changes are recorded in transport requests.

Notes

Keep in mind the following notes:

- **Namespace enhancement**

As of 4.5A customers and partners can also enhance namespaces.

- **Filtering the Data Selection**

The distribution of data can be linked to conditions that are defined as filters in the distribution model.

If you have filtered the data, you have to select the option *Data filtering allowed* in the dialog box when you are creating or changing the filters.

For more information see [Data Filters \[Extern\]](#).

The prerequisite for useful filtering at the functional and business levels is the existence of hierarchical dependencies between BAPI table parameters. The dependencies must be defined before the interface is generated. Choose *BAPIs → Data filtering → Maintain hierarchy of table parameters*. Note that changes made to dependencies must be compatible, otherwise the filtering will have a different outcome. For further information see [Defining Hierarchies Between BAPI Parameters \[Seite 27\]](#)

- **Serialization**

The function module on the outbound processing side has an optional parameter SERIAL_ID (reference to the channel number). This input parameter manages the assignment of messages to object channels. In an object channel all messages are processed in the target system in the same sequence they were created in the source system. An object channel is identified by a key from the object type of the BAPI and from the channel number.

For further information see [Serialization Using Object Types \[Extern\]](#)

- **Links**

The links to ALE answers the following questions:

- Outbound processing:
Which application object has the IDoc been created from?
The prerequisite is that the application has correctly filled the parameter APPLICATION_OBJECTS in the function module used for outbound processing.
- Inbound processing:
From which outbound IDoc was the inbound IDoc created from and which application object was created from the inbound IDoc?
The prerequisite is that a key has been defined in the Business Object Repository (BOR) for the BAPI method and that this key is contained in the BAPI function module

Maintaining BAPI-ALE Interfaces

in the export and import parameters. This key may consist of several key fields. If there are no key fields in the BOR, a link to another object can be specified.

- **Documentation on Generated Function Modules**

Documentation has been written on function modules generated for inbound and outbound processing. You can retrieve this documentation from within the interface. It describes the purpose of the parameters and their values.

- **BAPI Return Parameters and IDoc Status**

Provided that the return parameter is filled by the application, the IDoc status and the associated information from the parameter is included in the IDoc. Message types determine the IDoc status.

If the return parameter is an EXPORTING parameter, only *one* IDoc status record is written:

Message type A: Status 51 (Application log not posted, with DB rollback)

Message type E: Status 51 (Application log not posted, no DB rollback)

Message type W, I or S: Status 53 (application document posted).

If the return parameter is a TABLE parameter, *several* IDoc status records can be written, according to the message types in the table:

Message type A: Status 51 (Application log not posted, with DB rollback)

(There is no status for message type S).

Message type E: Status 51 (Application log not posted, no DB rollback)

(There is no status for message type S).

No message type A or E: Status 53 (application document posted).

The IDoc status records are written to the database in the same sequence as the messages in the return parameter.

If the return parameter has not been filled, it means that the BAPI has been successfully called by the IDoc. In this case, an IDoc status record with status 53 (application document posted) is written from the ALE layer.

If there is an error, *only* the first message from the return parameter is copied to the text in the associated error task (work item).

- **Restrictions on Generating Interfaces**

- If the reference structure of parameters in the BAPI function module (IMPORT and/or TABLES) is the same, you should not use the generation function. In this situation the IDoc segment is not uniquely mapped to the parameter. For this reason you cannot identify the parameters when the inbound IDoc is processed.
- If the dataset has more than 1000 bytes in the reference structure of a parameter or in a field in the reference structure, you *cannot* set the generation function because only one segment can be loaded with a dataset of maximum size 1000 bytes.

Determining the Receiver of a BAPI

Use

How data is distributed by an object method can depend on the requirements specified in the distribution model (in the R/3 Implementation Guide). *Basis* → *Application Link Enabling (ALE)* → *Modelling and Implementing Business Processes* → *Maintain Distribution Mode*).

The prerequisite for this filtering is that a filter object type has been assigned to the relevant BAPI in your SAP applications. For some BAPIs SAP has already defined and assigned filter object types. You can also define your own filter object types and assign them to a BAPI ([Defining Filter Object Types and Assigning Them to a BAPI \[Seite 23\]](#).)

The receiver must be determined before the BAPI of a generated BAPI-ALE interface is called. The receiver determination checks whether the filter objects satisfy the specified requirements and verifies the receivers.

In the ALE distribution model the following dependencies can be mapped:

- Between a BAPI and a message type
- Between BAPIs

These dependencies must be implemented by a function module on the application side. In ALE Development the function module must be assigned to the relevant object type (via *Dependencies* → *Define function module for dependent business object*).

If a dependency is defined as a condition in the ALE distribution model, the receiver of the referenced BAPI or message type is determined.



Example of a BAPI dependency on a message type:

The distribution of organization addresses has been integrated in the object maintenance of the vendor. This address data is then distributed together with the object data using ALE. The address data is dependent on the object data and is distributed using BAPIs. The object data is distributed using the message type CREMAS.

So there is a dependency between a BAPI and a message type.

In the distribution model an active receiver filter is assigned to the BAPI to distribute organization addresses (AddressOrg.SaveReplica). The dependency has been activated using the attribute *dependent distribution* in the filter display.

Based on the receiver determination the object data with the BAPI address data is only distributed, if the filter conditions for CREMAS are fulfilled

ALE provides a range of function modules for receiver determination with the function group BDAPI.

Features

The function modules of the function group BDAPI are used for:

Function module	Features
ALE_BAPI_GET_FILTEROBJECTS	Determining Filter Objects of a BAPI [Seite 37]
ALE_ASYNC_BAPI_GET_RECEIVER	Determining Receivers of Asynchronous BAPIs [Seite 38]

Determining the Receiver of a BAPI

ALE_SYNCH_BAPI_GET_RECEIVER	Determining Receivers of Synchronous BAPIs [Seite 47]
ALE_BAPI_GET_UNIQUE_RECEIVER	Determining Unique Receivers of Synchronous BAPIs [Seite 52]



The following examples programs for receiver determination are provided:

- [Example Programs with Asynchronous BAPI Calls \[Seite 42\]](#)
- [Example Programs with Synchronous BAPI Calls \[Seite 49\]](#)

Determining Filter Objects of a BAPIs

Use

To determine the receivers of a BAPI, the relevant filter objects for this BAPI must be specified. If the filter objects maintained in the distribution model are not known at the time of determining the receiver, they can be retrieved using the function module ALE_BAPI_GET_FILTEROBJECTS.

When ALE_BAPI_GET_FILTEROBJECTS is called, the business object, method and a table containing the names of the logical receiver systems are returned. If the BAPI cannot be found in the ALE distribution model or the filter objects have not been maintained in the ALE distribution model, an empty table (FILTEROBJECTS) is returned.

You can only determine the values of filter objects you have defined using defined business add-ins that you have to implement yourself.

For dependencies, the object type through which the dependent ALE distribution object links to the referenced ALE distribution object is always returned. The application must know the object ID of the current object characteristics for this object type.

Input Parameters:

Parameter	Reference Field/Structure	Description
OBJECT	BDI_BAPI-OBJECT	BOR object of the BAPI
METHOD	BDI_BAPI-METHOD	BOR method of the BAPI
RECEIVER_INPUT	BDI_LOGSYS	Default logical receiving system

Output Parameters:

Parameter	Reference Field/Structure	Description
FILTEROBJECTS	BDI_FLTTYP	Filter objects and values

Exceptions

Parameter	Description
ERROR_IN_ALE_CUSTOMIZING	Error in ALE Customizing

Determining Receivers of Asynchronous BAPIs

Determining Receivers of Asynchronous BAPIs

Use

To determine the receivers of an asynchronous BAPI, the application program calls the function module ALE_ASYNC_BAPI_GET_RECEIVER.

The following mechanisms are effective:

- If a dependency for a BAPI is defined as a condition in the ALE distribution model, the receiver of the referenced BAPI or message type is determined. The application program has to pass the object ID (for example, 01815) as the value of the filter object to the receiver determination.

From the object ID the function module ALE_ASYNC_BAPI_GET_RECEIVER determines the current filter object values of the object through which the dependent BAPI references to another BAPI or message type. The application must provide a function module that enables the object data to be read. The name of the function module must be stored in an ALE Customizing table, which the receiver determination can access at runtime (Table TBD18).

- If no receivers are determined or the BAPI cannot be found in the distribution model, an empty table for the receivers is returned.
- If the forwarded filter object types and filter object values are unable to determine the receiver correctly, an error message and an exception are returned (ERROR_IN_FILTEROBJECTS).
- If any inconsistencies have arisen in the distribution model due to Customizing errors, an error message and an exception are returned (ERROR_IN_ALE_CUSTOMIZING).

Input Parameters:

Parameter	Reference Field/Structure	Description
OBJECT	BDI_BAPI-OBJECT	BOR object of the BAPI
METHOD	BDI_BAPI-METHOD	BOR method of the BAPI
RECEIVER_INPUT, optional	BDI_LOGSYS	Default logical receiving system
FILTEROBJECT_VALUES	BDI_FOBJ	Filter objects and values

Output Parameters:

Parameter	Reference Field/Structure	Description
RECEIVERS	BDI_LOGSYS	Receiving systems of the BAPI

Exceptions

Parameter	Description
ERROR_IN_FILTEROBJECTS	Filter objects are incorrect or incomplete
ERROR_IN_ALE_CUSTOMIZING	Error in ALE Customizing

Determining Filter Objects Using Business Add-Ins

You will find out how SAP defines *business add-ins* and how you implement a *business add-in* to query filter objects you have defined when determining receivers.

Use

This description only applies to receiver determination using *business add-ins*.

It is only relevant to BAPIs that exist for ALE interfaces and that are to be implemented for the *business add-ins*.

In the standard system the SAP application provides a range of filter object types for receiver determination. These filter object types are evaluated with the values assigned by you at runtime (for example, filter object *plant 0001, 0002*). You can add more values to the default values provided.

If you do however want to use different filter object types for mapping your own business processes to execute the asynchronous BAPI call under enhanced conditions, you have to implement and activate the *business add-ins* defined by SAP. *Business add-ins* are places in the source code defined by SAP programmers where you can insert code without having to modify the original object.

To find out which of your SAP applications contain *business add-ins* refer to the application documentation.

Prerequisites

The following prerequisites must be fulfilled:

- Your SAP applications contain *business add-ins* defined by SAP (*Tools → ABAP Workbench → Utilities → Business Add-Ins → Definition*, see also the example below, *Creating a Business Add-In in a Form Routine*).
- In the ALE development environment you have defined a filter object type for the receiver determination, for example, filter and assigned it to the appropriate BAPI (from the SAP menu: *Tools → ALE → ALE Development → BAPIs → Receiver Determination*).
- You have implemented and activated the *Business Add-In* (from the SAP menu: *Tools → ABAP Workbench → Utilities → Business Add-Ins → Implementation*).
- You have defined a filter object with specified conditions in the distribution model under the sender and receiver settings (from the R/3 Implementation Guide: *Basis → Application Link Enabling (ALE)*).

Example:

```

Sender
Receiver
  BUSOBJECT.METHOD
Receiver Determination
  Filter Group
    FILTER
      1010
  
```

Structuring a Business Add-In in a Form Routine

Receiver determination for a BAPI (BUSOBJECT.METHOD) can be structured by SAP developers in SAP applications using a form routine (for example, BUSOBJECT_METHOD_RECEIVERS):

Interface:

Determining Filter Objects Using Business Add-Ins

```

TABLES      receivers STRUCTURE bdi_logsys
USING        object TYPE swo_objtyp
              method TYPE swo_method
              parameters LIKE ...
              return_info LIKE syst.

```

In this example the parameters *parameters* contain all the filter object values of the application required for receiver determination. They are application-dependent.

The parameter *receivers* contains the required receivers (or initial value) as the default value and the determined receivers as the return value.

Definition of variables:

```

t_filter_object_type      TYPE bdi_flttp_tab
t_filter_object_value     TYPE bdi_fobj_tab
receivers_output          LIKE bdi_logsys OCCURS 0 WITH
HEADER LINE

```

The following steps are carried out in the code of an SAP application: It involves the same steps as when determining the filter object types defined by SAP, with the additional step (step 3) for determining the filter objects you have defined.

- 1) Query filter object type for the BAPI with the function module **ALE_BAPI_GET_FILTEROBJECTS**:

```

EXPORTING
  object = busobject
  method = method

TABLES
  receiver_input = receivers
  filterobjects = t_filter_object_type

EXCEPTIONS
  error_in_ale_customizing

```

- 2) Assign the current values of the application in parameters *parameters* to the filter object type provided by SAP in the structure *t_filter_object_type*.
- 3) Call the defined *Business Add-In* to evaluate the filter object types defined by the customer in the flow logic of the application.

```

EXPORTING
  object = busobject
  method = method
  parameters = ...
  filterobjtype = t_filter_object_type

CHANGING
  filterobjvalue = t_filter_object_value

```

- 4) Determine the receivers of the asynchronous BAPI call using the function module **ALE_ASYNC_BAPI_GET_RECEIVER**.

```

EXPORTING
  object = busobject
  method = method

TABLES
  receiver_input = receivers

```

Determining Filter Objects Using Business Add-Ins

```
receivers_output = receivers_output  
filterobject_values = t_filter_object_value  
EXCEPTIONS  
    error_in_filterobjects  
    error_in_ale_customizing  
receivers[] = receivers_output[]
```

(You can find the program code of this example under [Example Programs with Asynchronous BAPI Calls \[Seite 42\]](#), *Receiver Determination with Business Add-In*)

Procedure

Implement the object method for the business add-in under the filter object type you have defined.

Result

The receivers of the asynchronous BAPI call have been determined using the defined filter objects.

Example Programs with Asynchronous BAPI Calls

Example Programs with Asynchronous BAPI Calls

The following example programs show how receivers are determined with asynchronous BAPI calls.

- Filter Object Types Are Not Known at Runtime
- Receiver determination with business add-in
- Filter object types are known at runtime



The function ALE_BAPI_GET_FILTEROBJECTS must be used if the application's filter object types are not known at runtime.



A COMMIT WORK must be executed in the program after the outbound function module of the generated BAPI-ALE interface has been called. The database commit at the end of the transaction is not sufficient. If a COMMIT WORK is not executed, the IDoc is created with the correct status but it will not be dispatched.

The IDocs created are locked until the called transaction has been completed. If you want to unlock them earlier, you can call the function module:

DEQUEUE_ALL releases all locked objects

EDI_DOCUMENT_DEQUEUE_LATER releases individual IDocs whose numbers have been transferred to the function module as parameter values.

Filter Object Types Are Not Known at Runtime

* data declaration

```
data: filterobj_values like bdi_fobj occurs 0,
      filterobj_types like bdi_fobjtype occurs 0,
      bapi_logsys like bdi_logsys occurs 0.
```

constants:

```
  c_objtype_plant type c value 'WERKS',
  c_objtype_langu type c value 'SPRAS'.
```

* get filterobjects from ALE distribution model

```
call function 'ALE_BAPI_GET_FILTEROBJECTS'
  exporting
    object          = 'BUS1001'
    method          = 'REPLICATEDATA'
  tables
    filterobjects   = filterobj_types
  exceptions
    error_in_ale_customizing = 1.
```

* fill filterobject values into table

```
loop at filterobj_types.
  case filterobj_values-objtype.
    when c_objtype_plant.
```

Example Programs with Asynchronous BAPI Calls

```

filterobj_values-objtype = c_objtype_plant.
filterobj_values-objvalue = '0002'.
    when c_objtype_langu.
filterobj_values-objtype = c_objtype_langu.
filterobj_values-objvalue = 'D'.
    when others.
    endcase.
append filterobj_values.
endloop.

* get receiver from ALE distribution model

call function 'ALE_ASYNC_BAPI_GET_RECEIVER'
    exporting
        object          = 'BUS1001'
        method          = 'REPLICATEDATA'
    tables
        receivers       = bapi_logsys
        filterobject_values = filterobj_values
    exceptions
        error_in_filterobjects = 1
        error_in_ale_customizing = 2.

* call generated ALE interface function module

if sy-subrc <> 0.
if not bapi_logsys[] is initial.
    call function 'ALE_MATERIAL_REPLICATE_DATA'
        tables
            receivers = bapi_logsys
        ...
    commit work.
endif.
endif.

```

Receiver determination with business add-in

Form routine implemented by SAP

```

*-----
*          FORM ALE_BFA_TEST_RECEIVERS
*-----
FORM ale_bfa_test_receivers TABLES receivers STRUCTURE bdi_logsys
    USING object TYPE swo_objtyp
    method TYPE swo_method
    key1 LIKE tbbfatest-key1
    key2 LIKE tbbfatest-key2
    return_info LIKE syst.

* key1 and key2 are parameters regarding receiver determination
* 2 filter object types were defined by SAP:
*     TEST_KEY1
*     TEST_KEY2

```

Example Programs with Asynchronous BAPI Calls

```

* variables definition
DATA: w_filter_object_type TYPE bdi_flttp,
      t_filter_object_type TYPE bdi_flttp_tab,
      w_filter_object_value TYPE bdi_fobj,
      t_filter_object_value TYPE bdi_fobj_tab,
      receivers_output LIKE bdi_logsys OCCURS 0 WITH HEADER LINE.

CLASS: cl_ex_customer_filter DEFINITION LOAD.
DATA: my_exit TYPE REF TO if_ex_customer_filter.

*> Step 1) get filter object types for a BAPI
CALL FUNCTION 'ALE_BAPI_GET_FILTEROBJECTS'
  EXPORTING
    object          = object
    method          = method
  TABLES
    receiver_input  = receivers
    filterobjects   = t_filter_object_type
  EXCEPTIONS
    error_in_ale_customizing = 1
    OTHERS              = 2.

IF sy-subrc <> 0.
  return_info = syst.
  EXIT.
ENDIF.

*> Step 2) evaluate SAP filter objects
LOOP AT t_filter_object_type INTO w_filter_object_type.
  CASE w_filter_object_type-objtype.
*   evaluate delivered filter objects
    WHEN 'TEST_KEY1'.
      MOVE-CORRESPONDING w_filter_object_type
                        TO w_filter_object_value.
      w_filter_object_value-objvalue = key1.
      APPEND w_filter_object_value TO t_filter_object_value.
    WHEN 'TEST_KEY2'.
      MOVE-CORRESPONDING w_filter_object_type
                        TO w_filter_object_value.
      w_filter_object_value-objvalue = key2.
      APPEND w_filter_object_value TO t_filter_object_value.
*   customers defined filter objects
    WHEN OTHERS.
  ENDCASE.
ENDLOOP.

*> Step 3) evaluate customer-defined filter objects
CREATE OBJECT my_exit TYPE cl_ex_customer_filter.

```

Example Programs with Asynchronous BAPI Calls

```

CALL METHOD my_exit->filtering
  EXPORTING
    object = object
    method = method
    key1 = key1
    key2 = key2
    filterobjtype = t_filter_object_type
  CHANGING
    filterobjvalue = t_filter_object_value.

*> Step 4) determine receivers for all filter objects
CALL FUNCTION 'ALE_ASYNC_BAPI_GET_RECEIVER'
  EXPORTING
    object          = object
    method          = method
  TABLES
    receiver_input  = receivers
    receivers       = receivers_output
    filterobject_values = t_filter_object_value
  EXCEPTIONS
    error_in_filterobjects = 1
    error_in_ale_customizing = 2
    OTHERS = 3.

IF sy-subrc <> 0.
  return_info = syst.
  EXIT.
ENDIF.

receivers[] = receivers_output[].

ENDFORM.
```

Methods Implemented by Customers

```

* The following method was implemented by a customer with
* Business Add-In
* 1 filter object type was defined by customer:
*   ZTEST_KEYS

METHOD if_ex_customer_filter~filtering.
* ...
  DATA: w_filterobjtype TYPE bdi_flgtyp,
        w_filterobjvalue TYPE bdi_fobj.

  LOOP AT filterobjtype INTO w_filterobjtype.
    CASE w_filterobjtype-objtype.
      WHEN 'ZTEST_KEYS'.
        MOVE-CORRESPONDING w_filterobjtype TO w_filterobjvalue.
        w_filterobjvalue-objvalue+0(3) = key1.
```

Example Programs with Asynchronous BAPI Calls

```

        w_filterobjvalue-objvalue+3(3) = key2.
        APPEND w_filterobjvalue TO filterobjvalue.
    WHEN OTHERS.
    ENDCASE.
ENDLOOP.

ENDMETHOD.

#

Filter Object Types are Known at Runtime

* data declaration

data: filterobj_values like bdi_fobj occurs 0,
      filterobj_types like bdi_fobjtype occurs 0,
      bapi_logsys like bdi_logsys occurs 0.

filterobj_values-objtype = 'KKBER'.
filterobj_values-objvalue = '0002'.
append filterobj_values.

* get receiver from ALE distribution model

call function 'ALE_ASYNC_BAPI_GET_RECEIVER'
  exporting
    object          = 'BUS1010'
    method          = 'REPLICATESTATUS'
  tables
    receivers       = bapi_logsys
    filterobject_values = filterobj_values
  exceptions
    error_in_filterobjects = 1
    error_in_ale_customizing = 2.

* call generated ALE interface function module

if sy-subrc <> 0.
if not bapi_logsys[] is initial.
    call function 'ALE_DEBITOR_CREDITACC_REPLICATESTATUS'
      tables
        receivers       = bapi_logsys
        ...
    commit work.
endif.
endif.

```

Determining Receivers of Synchronous BAPIs

Use

The application program calls the function ALE_SYNCH_BAPI_GET_RECEIVER to determine the receiver of a synchronous BAPI.

The following mechanisms are effective:

- If no receivers are determined or the BAPI cannot be found in the ALE distribution model, an empty table for the receivers is returned.
- If a dependency for a BAPI is defined as a condition in the ALE distribution model, the receiver of the referenced BAPI or message type is determined. The application has to pass the object ID (for example, 01815) as the value of the filter object to the receiver determination.

The receiver determination can then read the current filter object values for the object that the dependent BAPI references to another BAPI or message type. The application provides a function module that enables the object data to be read. The name of the function module is stored in an ALE Customizing table, which the receiver determination can access at runtime (Table TBD18).

As well as the logical system the RFC destination is also returned.

- If the forwarded filter object types and filter object values are unable to determine the receiver correctly, an error message and an exception are returned (ERROR_IN_FILTEROBJECTS).
- If the RFC destination for a logical receiving system has not been maintained, an error message and an exception are returned (NO_RFC_DESTINATION_MAINTAINED).
- If any inconsistencies have arisen in the distribution model due to Customizing errors, an error message and an exception are returned (ERROR_IN_ALE_CUSTOMIZING).

Input Parameters:

Parameter	Reference Field/Structure	Description
OBJECT	BDI_BAPI-OBJECT	BOR object of the BAPI
METHOD	BDI_BAPI-METHOD	BOR method of the BAPI
RECEIVER_INPUT	BDI_LOGSYS	Default logical receiving system
FILTEROBJECT_VALUES	BDI_FOBJ	Filter objects and values

Output Parameters:

Parameter	Reference Field/Structure	Description
RECEIVERS	BDI_LOGSYS	Receiver systems of the BAPI and the RFC destination

Exceptions

Parameter	Description
ERROR_IN_FILTEROBJECTS	Filter objects are incorrect or incomplete
ERROR_IN_ALE_CUSTOMIZING	Error in ALE Customizing
NO_RFC_DESTINATION_MAINTAINED	There is no RFC destination for the logical system

Example Programs with Synchronous BAPI Calls

The following example programs show how receivers are determined with Synchronous BAPI calls.



The function ALE_BAPI_GET_FILTEROBJECTS must be used if the application's filter object types are not known at runtime. Otherwise its use is optional.



A database commit is executed by the synchronous RFC, this means that database changes carried out before the RFC cannot be undone.

Filter Object Types Are Not Known at Runtime

```
* data declaration

data: filterobj_values like bdi_fobj occurs 0,
      filterobj_types like bdi_fobjtype occurs 0,
      bapi_server like bdbapidest occurs 0.

constants:
  c_objtype_plant type c value 'WERKS',
  c_objtype_langu type c value 'SPRAS'.

* get filterobjects from ALE distribution model

call function 'ALE_BAPI_GET_FILTEROBJECTS'
  exporting
    object          = 'BUS1001'
    method          = 'GETDETAIL'
  tables
    filterobjects   = filterobj_types
  exceptions
    error_in_ale_customizing = 1.

* fill filterobject values into table

loop at filterobj_types.
  case filterobj_values-objtype.
    when c_objtype_plant.
      filterobj_values-objtype = c_objtype_plant.
      filterobj_values-objvalue = '0002'.
    when c_objtype_langu.
      filterobj_values-objtype = c_objtype_langu.
      filterobj_values-objvalue = 'D'.
    when others.
  endcase.
  append filterobj_values.
endloop.

* get receiver from ALE distribution model
```

Example Programs with Synchronous BAPI Calls

```

call function 'ALE_SYNC_BAPI_GET_RECEIVER'
  exporting
    object          = 'BUS1001'
    method          = 'GETDETAIL'
  tables
    receivers       = bapi_server
    filterobjects_values = filterobj_values
  exceptions
    error_in_filterobjects = 1
    error_in_ale_customizing = 2.

* call synchronous BAPI locally/remotely

if sy-subrc = 0.
if not bapi_server[] is initial.
loop at bapi_server.
call function 'BAPI_MATERIAL_GET_DETAIL'
  destination bapi_server-rfc_dest
  ...
endloop.
else.
call function 'BAPI_MATERIAL_GET_DETAIL'
  ...
endif.
endif.

```

Filter Object Types are Known at Runtime

```

* data declaration

data: filterobj_values like bdi_fobj occurs 0,
      filterobj_types like bdi_fobjtype occurs 0,
      bapi_server like bdibapidest occurs 0.

* fill filterobject values into table

filterobj_values-objtype = 'KKBER'.
filterobj_values-objvalue = '0002'.
append filterobj_values.

* get receiver from ALE distribution model

call function 'ALE_SYNC_BAPI_GET_RECEIVER'
  exporting
    object          = 'BUS1010'
    method          = 'GETSTATUS'
  tables
    receivers       = bapi_server
    filterobjects_values = filterobj_values
  exceptions
    error_in_filterobjects = 1
    error_in_ale_customizing = 2.

```

```
* call synchronous BAPI locally/remotely

if sy-subrc <> 0.
if not bapi_server[] is initial.
loop at bapi_server.
call function 'BAPI_DEBITOR_CREDITACC_GETSTATUS'
            destination bapi_server-rfc_dest
            ...
endloop.
    else.
call function 'BAPI_DEBITOR_CREDITACC_GETSTATUS'
            ...
    endif.
endif.
```

Determining Unique Receivers of Synchronous BAPIs

Determining Unique Receivers of Synchronous BAPIs

To determine a unique receiver of a synchronous BAPI, the application program calls the function ALE_BAPI_GET_UNIQUE_RECEIVER.

The following mechanisms are effective:

- If no receiver is determined or if the BAPI is not found in the ALE distribution model an empty table for the receiver is returned.
- If the forwarded filter object types and filter object values are unable to determine the receiver correctly, an error message and an exception are returned (ERROR_IN_FILTEROBJECTS).
- If more than one receiver for the BAPI is determined an error message and an exception are returned.
- If the RFC destination for a logical receiving system has not been maintained, an error message and an exception are returned (NO_RFC_DESTINATION_MAINTAINED).
- If any inconsistencies have arisen in the distribution model due to Customizing errors, an error message and an exception are returned (ERROR_IN_ALE_CUSTOMIZING).

Input Parameters:

Parameter	Reference Field	Description
OBJECT	BDI_BAPI-OBJECT	BOR object of the BAPI
METHOD	BDI_BAPI-METHOD	BOR method of the BAPI
FILTEROBJECT_VALUES	BDI_FOBJ	Filter objects and values

Output Parameters:

Parameter	Reference Field	Description
RECEIVERS	BDBAPIDEST	Receiving systems of the BAPI

Exceptions

Parameter	Description
ERROR_IN_FILTEROBJECTS	Filter objects are incorrect or incomplete
ERROR_IN_ALE_CUSTOMIZING	Error in ALE Customizing
NOT_UNIQUE_RECEIVER	There is more than one receiver system for the BAPI
NO_RFC_DESTINATION_MAINTAINED	There is no RFC destination for the logical system

Developing BAPIs for Interactive Processing

Prerequisites

Inbound IDoc processing in ALE supports IDoc function modules that execute a CALL TRANSACTION. This enables an IDoc to be posted interactively whereby you work through the transaction screens displayed.

This is not possible with BAPI-ALE interfaces generated from standard BAPIs because the BAPIs do not have a dialog interface. If you require an BAPI-ALE interface with a CALL TRANSACTION to a dialog transaction, you can develop your own BAPI that displays the transaction screens for ALE error handling.

For further information see [Customer Enhancements \[Extern\]](#) in the BAPI Programming guide.

Procedure

The ALE layer puts a parameter in the global memory that can be requested in the BAPI source code as follows:

```
Data: pi_input_method like bdwfap_par-inputmethod.  
...  
Import pi_input_method from memory id 'ALE_INPUT_METHOD'.  
...
```

The parameter pi_input_method can have the following values:

Value	Description
" " (initial)	Request without dialog screens
"E"	Only display screen if an error has occurred on it.
"A"	Display all screens

Enhancing IDocs of BAPI-ALE Interfaces

Prerequisites

The IDoc enhancement concept assumes that there are customer exits in the ABAP code where the BAPI is created or read. If the BAPI-ALE interface has been generated this code does not contain any customer exits.

For further information see [Customer Enhancements \[Extern\]](#) in the BAPI Programming guide.

Procedure

To implement customer enhancements of IDocs of generated BAPI-ALE interfaces you must:

1. Copy and modify the function module belonging to the original BAPI.
2. Create your own BAPI in the BOR by creating a sub-object type in the customer namespace.

When you create the subobject type the methods of the business object inherits the subtype. You can change and delete the methods of the subtype or enhance them with your own methods.

3. Generate a user-defined BAPI-ALE interface from this new BAPI.

To create the enhanced IDoc for outbound processing the application must provide a customer exit.

Distribution Using Message Types

Purpose

In Release 3.x business functions and processes are distributed using message types. A message type represents a business function. The technical structure of the message type is the IDoc type.

The programming model "Distribution using message types" contains the definitions of message types and IDoc types and the ABAP code for processing inbound and outbound IDocs.

Process Flow

Defining message types and IDoc types:

- [Defining New IDoc Types \[Extern\]](#)



If you want to create message type enhancements for master data distribution, you also have to create a new message type for each enhancement.

The ALE interface does not allow you to create different segment data for different IDoc types for the same message type.

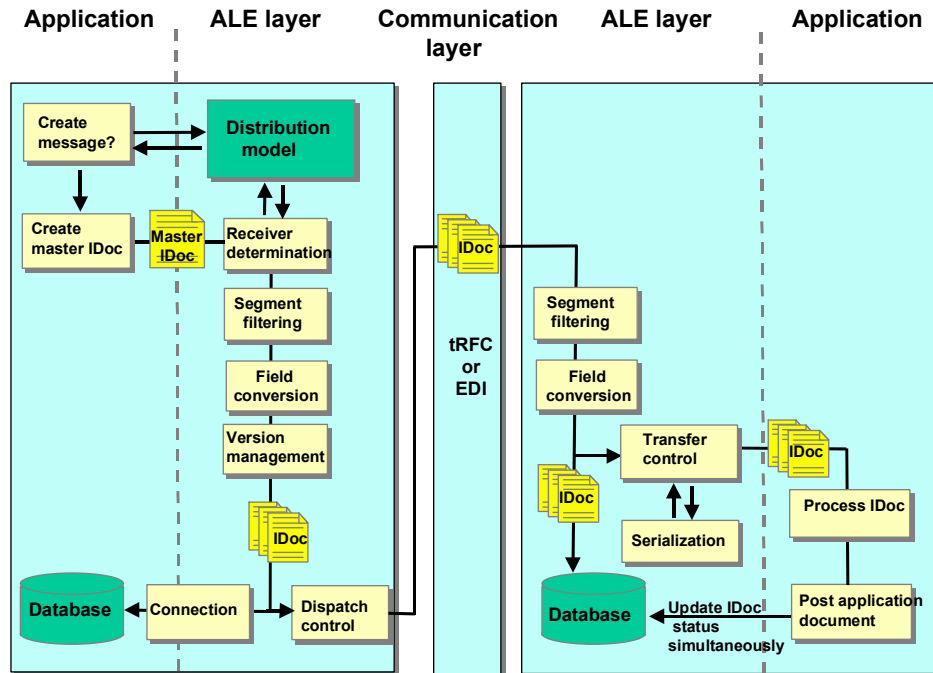
Writing ABAP code:

- [Outbound Processing \[Seite 60\]](#)
- [Inbound Processing \[Seite 85\]](#)

- [Master Data Distribution \[Seite 171\]](#)
- [Communicating with Non-R/3 Systems \[Extern\]](#)

Distribution Using Message Types

When message types are used to transfer data asynchronously in ALE:



Outbound Processing

An application function module creates a master IDoc in outbound processing, the so-called master IDoc.

The following steps are carried out in the ALE layer:

- Receiver determination, if this has not already been done by the application.
- Data selection
- Segment filtering
- Field conversion
- Version change
- Dispatch control

The formatted IDoc is passed to the communication layer and from here sent to the system that was called (server) via a transactional remote function call (RFC) or via file interfaces (for example, EDI).

If an error occurs in the ALE layer, the IDoc containing the error is saved and a workflow task is created. The ALE administrator can use this workflow to correct the error.

For information on programming see the [Implementing Outbound Processing \[Seite 60\]](#).

The individual steps are explained below.

Receiver Determination

Like a normal letter, an IDoc has a sender and a receiver. If the application does not explicitly specify the receiver, the ALE layer uses the distribution model to help determine the receivers of the message.

The ALE layer can find out from the model whether any distributed systems should receive the message and, if so, then how many. The result may be that one, several or no receivers at all are found.

For each of the distributed systems identified as receiver systems, the data specified by the filter objects in the distribution model is selected from the master IDoc. This data is then entered into an IDoc, and the appropriate system is specified as the receiver.

Segment Filtering

Individual segments can be removed from the IDoc before it is dispatched. If you want to remove IDoc segments, in Customizing for ALE choose:

Modelling and Implementing ALE Business Processes
Master Data Distribution
Scope of Data for Distribution
Filter IDoc Segments

The appropriate setting depends on the sending and receiving logical R/3 System.

Field Conversion

You can define field conversions for specific receivers in ALE Customizing:

Modelling and Implementing ALE Business Processes
Converting Data Between Sender and Receiver

Standard rules can be specified for field conversions. One set of rules is created for each IDoc segment and rules are defined for each segment field. These are important for converting data fields to exchange information between R/2 and R/3 Systems. For example, the field *plant* can be converted from a two character field to a four character field.

Standard Executive Information System (EIS) tools are used to convert fields.

IDoc Version Change

SAP guarantees that ALE works correctly between different releases of the R/3 System. By changing the IDoc format you can convert message types from different R/3 releases.

SAP uses the following rules to convert existing message types:

- Fields can be appended to a segment type
- New segments can be added

ALE Customizing records the version of each message type used in each receiver. The communication IDoc is created in the correct version in outbound processing.

Dispatch Control

Time and quantity are the factors that control the dispatch of IDocs in the dispatch control.

Scheduling the dispatch time:

IDocs can either be sent immediately or in the background. This setting is made in the partner profile.

If the IDoc is sent in the background, a job has to be scheduled. You can choose how often background jobs are scheduled.

Controlling the amount of data sent:

Distribution Using Message Types

IDocs can be dispatched in packets. The packet size is assigned in ALE Customizing in accordance with the partner profile.

Modeling and Implementing ALE Business Processes

→ *Partner Profiles and Time of Processing*

→ *Maintain Partner Profiles*



This setting only affects IDocs that are processed in the background.

Inbound Processing

The following processes are carried out on inbound IDocs in the ALE layer:

- Segment filtering
- Field conversion
- Transfer control
- Serialization

For information on programming see the [Implementing Inbound Processing \[Seite 85\]](#).

The individual steps are explained below.

Segment Filtering

You can filter IDoc segments in inbound processing.

In inbound processing this function is principally the same as in outbound processing.

Field Conversion

You can define field conversions for specific receivers in ALE Customizing:

Modelling and Implementing ALE Business Processes

Converting Data Between Sender and Receiver

Standard rules can be specified for field conversions. One set of rules is created for each IDoc segment and rules are defined for each segment field. These are important for converting data fields to exchange information between R/2 and R/3 Systems. For example, the field *plant* can be converted from a two character field to a four character field.

Standard Executive Information System (EIS) tools are used to convert fields.



For reduced message types field values are not overwritten in the receiving R/3 System, if the corresponding IDoc field contains the character "/".

Transfer control

Once the IDocs have been written to the database, they can be posted by the application.

IDocs can be passed to the application either immediately on arrival or at a later time in background processing.

Inbound IDocs can be posted in three ways:

- By calling a function module directly:
The inbound IDocs are posted directly. An error workflow is started, if an error occurs.
- By starting an SAP Business Workflow. A workflow is the sequence of steps required to post an IDoc.

Workflows for ALE are not provided.

- By starting a work item

A single step performs the IDoc posting.

The standard inbound processing setting is for ALE to call a function module directly. For information about the options in SAP Business Workflow see the [Inbound Processing Using SAP Workflow \[Seite 158\]](#).

You can specify the people to be notified for handling IDoc processing errors in SAP Business Workflow. Different people can be responsible for each message type.

Implementing Outbound Processing

This section describes the development steps required to send IDocs. The IDoc structure and the message type must have already been created and must be linked together.

See also:

- [Developing a Function Module for ALE Outbound Processing \[Seite 61\]](#)
- [Customizing ALE Outbound Processing \[Seite 80\]](#)
- [Outbound Processing Using Message Control \[Seite 84\]](#)

Developing a Function Module for ALE Outbound Processing

This section describes how to program a function module that creates an IDoc from an application object and calls ALE outbound processing.

See also:

[Basics \[Seite 62\]](#)

[Interrogating the Distribution Model \[Seite 63\]](#)

[Structure of Control Record \[Seite 64\]](#)s

[Structure of Data Records \[Seite 65\]](#)

[Call of MASTER IDOC DISTRIBUTE \[Seite 69\]](#)

[Coding Example \[Seite 71\]](#)

Basics

An IDoc consists of a control record with the structure *edidc* and one or more data records with the structure *edidd*. The control record is similar to a letter envelope. It contains the sender and receiver of the IDoc, as well as information on the type of message. The data that is being used by the IDoc is contained in the data records as unformatted character strings.

To be able to pass an IDoc to the ALE layer, you must set up a field string with the structure *edidc* and an internal table with the structure *edidd*. With these the function module `master_idoc_distribute` is then called. The module saves the data to the database and if necessary triggers despatch.

All ALE message flows are stored in the ALE distribution model. The distribution model is the central controlling instance for ALE. The application can interrogate the distribution model before the IDoc is created. This makes sense if the actual creation of an IDoc influences the application. As you are not required to set up the internal table for the IDoc if no message flow is maintained in the distribution model, it can also improve performance.

The ALE layer always interrogates the distribution model. If the application does not specify a receiver, all receivers are determined and an IDoc is created for each one. If the application does specify a receiver, a check is made against the distribution model to see whether the receiver has the necessary authorization. In ALE you can use the filter settings in the distribution model to remove parts of the IDoc.

Interrogating the Distribution Model

You do not have to interrogate the distribution model, it is optional.

There are two function modules that can interrogate the ALE distribution model:

`ale_model_determine_if_to_send` and `ale_model_info_get`. `ale_model_determine_if_to_send` is called with the message type and possibly with the logical receiving system if it is already known in the application. A check is made in the ALE distribution model that a message flow has been maintained for the input parameters. If this is not so, the export parameter *idoc_must_be_send* is set to initial; otherwise, an "X" is returned. If there are filter objects in the distribution model that control this message flow, they are not evaluated. An IDoc must only be created if `ale_determine_if_to_send` returns an "X".

Module `ale_model_info_get` is used for more complex queries made to the ALE distribution model. It is called with the message type to be dispatched. In return, you get a table containing all the potential recipients of this message type, as well as the associated filter objects. Note that there may be several entries for one receiver in the table returned. If there are no entries in the distribution model, the exception *no_model_info_found* is issued. If an exception is issued, an IDoc does not have to be created. Otherwise an IDoc does have to be created. You will find the receiving logical system in the *rcvsystem* field in a table entry.

The end result, that is, whether the receivers receives an IDoc and what the IDoc looks like, is only determined after all the filter objects for a message flow in the distribution model have been evaluated. This is carried out in the ALE layer.

Structure of Control Records

The control record consists of a field string for the structure `edidc`. The relevant fields are listed below; all other fields should be left with their initial values.

List of fields for the control record

Field	Description	Comment
mestyp	Logical message type. Conveys the business meaning of the message.	Mandatory field
idoctp	Basic structure of the IDoc. Identifies the layout set that uses this message.	Mandatory field
cimtyp	Structure of customer extension. If the customer extends an SAP basic structure, he must give a name to the structure of his extension.	Mandatory field if customer has made an enhancement. Otherwise initial.
rcvppt	Partner type of the receiver; "LS" (i.e. logical system) for ALE.	Optional field. See below.
rcvprn	Partner number of the receiver; the logical system for ALE.	Optional field. See below.
rcvpfc	Partner function of the receiver; normally initial for ALE.	Optional field. See below.

When the receiving system has been determined from the distribution model, it can be written to field `rcvprn`. Then field `RCVPFC` must be filled with "LS" (for logical system). If necessary, the partner function can be written into the field `RCVPFC`. However, the partner function is not normally used in ALE. What is important, is that either both `rcvppt` and `rcvprn` are left empty or that both are filled. If `rcvppt` and `rcvprn` are passed with their initial values, the receivers are determined entirely in the ALE layer.

Structure of the Data Records

[Replacing SAP Codes with ISO Codes \[Seite 67\]](#)

The data records of an IDoc are created in an internal table with structure EDIDD. The relevant fields are shown below.

Important Table Fields for Creating IDoc Data Records

Field	Description
SEGNAM	Segment type of the IDoc data record
SDATA	1000 byte-long character field for the data used by the IDoc

The remaining fields in EDIDD should be left initial.

All the segment types and their sequence are specified in the IDoc structure. The data records are structured according to this sequence and included in the internal table.

For each segment type of the IDoc structure, there is a DDIC structure with the same name. A field string with this structure is used for creating a data record. The application data is mapped to the field string. The segment type is written to the field SEGNAM, and the field string is written to the field SDATA. This data record is then included in the internal table with the structure edidd.

When creating IDoc data records, note the following design guidelines:

- [Converting Currency Amounts \[Seite 66\]](#)
- [Replacing SAP Codes With ISO Codes \[Seite 67\]](#)
- [Left-justified Filling of IDoc Fields \[Seite 68\]](#)

Converting Currency Amounts

Currency amounts have to be converted from an SAP system format to a format that can be understood externally. In the SAP system, all currency amounts are stored with two decimal places. If a currency has a different number of decimal places, the currency amount has to be converted. You can use function module `CURRENCY_AMOUNT_SAP_TO_IDOC` for this conversion; it performs a suitable currency amount conversion for IDocs.

We recommend that you encapsulate the code in a subroutine `<SEGMENT-TYP>_CURRENCY_SAP_TO_IDOC`.

Replacing SAP Codes With ISO Codes

There are ISO codes for country keys, currency keys, units of measure and shipping instructions. According to SAP design guidelines, you should use ISO codes for an IDoc if they are available. When you set up the IDoc, the SAP codes have to be replaced by ISO codes. To do this, you can use these function modules:

Function modules for converting SAP codes

Domain	Function module
Currency keys	CURRENCY_CODE_SAP_TO_ISO
Country keys	COUNTRY_CODE_SAP-TO_ISO
Units of measure	UNIT_OF_MEASURE_SAP_TO_ISO
Shipping instructions	SAP_TO_ISO_PACKAGE_TYPE_CODE

We recommend that you encapsulate the code in a SUBROUTINE `<SEGMENT-TYP>_CODES_SAP_TO_ISO`.

Left-justified Filling of IDoc Fields

All fields must be filled left-justified. This happens automatically for character fields. If the original field of the application is a non-character field, you must execute a *condense* on the corresponding field in the IDoc segment. To find out which fields require a *condense*, see the documentation structure for a segment type. The name of the documentation structure begins with "E3" or "Z3" (instead of "E1" or "Z1"); otherwise it is the same. This structure contains the same fields as the "E1" or "Z1" structure. But here you will find the original data elements and domains of the application. All fields with a data type unequal to *char*, *cuky*, *clnt*, *accp*, *numc*, *dats*, *tims* or *unit* require a *condense*.

We recommend that you encapsulate the code in a subroutine <SEGMENT-TYP>_CONDENSE. You should set left-justification after converting the currency amounts and ISO codes.

Calling MASTER_IDOC_DISTRIBUTE

After the MASTER_IDOC_DISTRIBUTE has been called, you must specify a COMMIT WORK; the standard Database Commit at the end of the transaction is not sufficient. The COMMIT WORK does not have to directly follow the call; it can be specified at higher call levels or after multiple calls of MASTER_IDOC_DISTRIBUTE.

Note that the IDocs created remain locked until the called transaction has been completed. If you want to unlock them earlier, you can call one of the following function modules:

- **DEQUEUE_ALL** releases all locked objects
- **EDI_DOCUMENT_DEQUEUE_LATER** as a parameter releases the transferred IDocs

If the application document is created via the update program, the call of MASTER_IDOC_DISTRIBUTE must also be performed *in update task* (if an update call has not already been performed at a higher level).

See also:

[Exceptions and Export Parameters of MASTER_IDOC_DISTRIBUTE \[Seite 70\]](#)

Exceptions and Export Parameters of MASTER_IDOC_DISTRIBUTE

The module uses the table parameter COMMUNICATION_IDOC_CONTROL to return the control records of the IDocs that were created in the database. To find out the IDoc number and the current status for example, see fields DOCNUM AND STATUS. In general, this table is not relevant to the calling application.

If the IDoc recipient was passed in the control record when MASTER_IDOC_DISTRIBUTE was called, but the distribution model does not allow the recipient to receive this IDoc, exception ERROR_IN_IDOC_CONTROL is output with an appropriate error message.

If a receiver was not given in the control record and ALE does not find a recipient in the distribution model, an exception is **not** issued. If you want to react to this case, you must query the return table COMMUNICATION_IDOC_CONTROL. If this table is empty, no IDoc was created.

This different behavior for the initial and non-initial receiver has historical reasons. The initial recipient is the standard case for master data replication: here it is of no further interest whether an IDoc was actually created. Presetting the receiver is the standard for dispatching transaction data: if an IDoc is not created, this is interpreted as an error.

List of Exceptions and Their Occurrences

Exception	Occurrence
error_in_idoc_control	Incorrect or no message type specified. Incorrect or no IDoc type specified. No IDoc created, although recipient was preset by application.
error_in_idoc_data	No data records passed.
error_writing_idoc_status	Technical problems when writing status records.
sending_logical_system_unknown	Own logical system could not be determined.

Example of Generating an IDoc

In the [Example Program of Generating an IDoc \[Seite 72\]](#) Parameters APPL_HEADER and APPL_ITEM are used as the source of the application data.

See also:

- [Coding Example \[Seite 100\]](#) of inbound processing
- [Using the Example Coding \[Seite 79\]](#)

Example Program to Generate an IDoc

Example Program to Generate an IDoc

```

FUNCTION MASTER_IDOC_CREATE_XAMPLE.
*"-----
*
*"Local interface:
*".....IMPORTING
*".....VALUE (APPL_HEADER) LIKE XHEAD STRUCTURE XHEAD
D
*".....TABLES
*".....APPL_ITEM STRUCTURE XITEM
*"-----
*
*variables of general interest
*DATA:
*.....control record for the IDoc
*.....IDOC_CONTROL LIKE EDIDC,
*.....data records for the IDoc
*.....T_IDOC_DATA LIKE EDIDD OCCURS 0 WITH HEADER LINE,
*.....table for the IDocs created by MASTER_IDOC_CONTROL
*.....T_COMM_CONTROL LIKE EDIDC OCCURS 0 WITH HEADER LINE,
*.....partner type for logical system
*.....C_PARTNER_TYPE_LOGICAL_SYSTEM LIKE EDIDC-RCVPRT,
*.....help variable for the check if an IDoc has to be created
*.....H_CREATE_IDOC.

*variables specific for this example
*DATA:
*.....field strings with IDoc segment structure
*.....E1XHEAD LIKE E1XHEAD,
*.....E1XITEM LIKE E1XITEM,
*.....data to be put to the control record
*.....C_MESSAGE_TYPE LIKE EDIDC-MESTYP VALUE 'XAMPLE',
*.....C_BASE_IDOC_TYPE LIKE EDIDC-IDOCTP VALUE 'XAMPLE01',
*.....segment types to be put to the data record table
*.....C_HEADER_SEGTYP LIKE EDIDD-SEGNAM VALUE 'E1XHEAD',
*.....C_ITEM_SEGTYP LIKE EDIDD-SEGNAM VALUE 'E1XITEM'.

*check if an IDoc has to be created, read the distribution model

```


Example Program to Generate an IDoc

```

..CALL·FUNCTION·'ALE_MODEL_DETERMINE_IF_TO_SEND'
.....EXPORTING
.....MESSAGE_TYPE.....=·C_MESSAGE_TYPE
*.....SENDING_SYSTEM.....=·'.'
*.....RECEIVING_SYSTEM.....=·'.'
*.....VALIDDATE.....=·SY-DATUM
.....IMPORTING
.....IDOC_MUST_BE_SENT.....=·H_CREATE_IDOC.
*....exceptions
*.....own_system_not_defined.=·1
*.....others.....=·2.

..IF·H_CREATE_IDOC·IS·INITIAL.
*...no·message·flow·maintained·in·the·model,·nothing·to·do
*...EXIT.
*ENDIF.

*put·the·application·header·record·to·the·IDoc
*MOVE-CORRESPONDING·APPL_HEADER·TO·E1XHEAD.

*convert·SAP·codes·to·ISO·codes
*PERFORM·E1XHEAD_CODES_SAP_TO_ISO
*...USING
*.....APPL_HEADER
*...CHANGING
*.....E1XHEAD.

*append·record·to·IDoc·data·table
*T_IDOC_DATA-SEGNAM.=·C_HEADER_SEGTYP.
*T_IDOC_DATA-SDATA.=·E1XHEAD.
*APPEND·T_IDOC_DATA.

..LOOP·AT·APPL_ITEM.
*...put·the·application·item·record·to·the·IDoc·segment
*...MOVE-CORRESPONDING·APPL_ITEM·TO·E1XITEM.

*...convert·currency·amounts·from·SAP·internal·to·neutral·fo
rmat
*...PERFORM·E1XITEM_CURRENCY_SAP_TO_IDOC
*...USING
*.....APPL_HEADER-CURRENCY
*...CHANGING
*.....E1XITEM.

```

Example Program to Generate an IDoc

```

*...convert SAP codes to ISO codes
*...PERFORM E1XITEM_CODES_SAP_TO_ISO
*...USING
*...APPL_ITEM
*...CHANGING
*...E1XITEM.

*...left justify all non character fields
*...PERFORM E1XITEM_CONDENSE
*...CHANGING
*...E1XITEM.

*...append record to IDoc data table
*...T_IDOC_DATA-SEGNAM:=C_ITEM_SEGTYP.
*...T_IDOC_DATA-SDATA:=E1XITEM.
*...APPEND T_IDOC_DATA.
*ENDLOOP.

*CALL FUNCTION 'MASTER_IDOC_DISTRIBUTE'
*in update task... "if application document is posted in update task
*...EXPORTING
*...MASTER_IDOC_CONTROL.....=IDOC_CONTROL
*...TABLES
*...COMMUNICATION_IDOC_CONTROL.....=T_COMM_CONTROL
*...MASTER_IDOC_DATA.....=T_IDOC_DATA.
*...exceptions
*...error_in_idoc_control.....=1
*...error_writing_idoc_status.....=2
*...error_in_idoc_data.....=3
*...sending_logical_system_unknown.....=4
*...others.....=5.

*A commit work has to be done. It could also be done in the
*calling
*application.
*COMMIT WORK.

*READ TABLE T_COMM_CONTROL INDEX 1.
*IF SY-SUBRC <> 0.
*no IDoc was created, you can react here, if necessary
*ENDIF.

ENDFUNCTION.

```

```

*&-----
*-----*
*&.....Form..E1XITEM_CONDENSE
*&-----
*-----*
*.....text.....
*.....*
*-----
*-----*
*..-->..p1.....text
*..<--..p2.....text
*-----
*-----*
FORM..E1XITEM_CONDENSE
..CHANGING
...IDOC_SEGMENT..LIKE..E1XITEM.

*..left..justify..all..non..character..fields
..CONDENSE:..IDOC_SEGMENT-QUANTITY,
.....IDOC_SEGMENT-VALUE.

ENDFORM....."..E1XITEM_CONDENSE
*&-----
*-----*
*&.....Form..E1XITEM_CURRENCY_SAP_TO_IDOC
*&-----
*-----*
*.....text.....
*.....*
*-----
*-----*
*..-->..p1.....text
*..<--..p2.....text
*-----
*-----*
FORM..E1XITEM_CURRENCY_SAP_TO_IDOC
..USING
...CURRENCY_CODE..LIKE..TCURC-WAERS
..CHANGING
...IDOC_SEGMENT..LIKE..E1XITEM.

..CALL..FUNCTION..'CURRENCY_AMOUNT_SAP_TO_IDOC'
.....EXPORTING

```

Example Program to Generate an IDoc

```

.....CURRENCY.....=..CURRENCY_CODE
.....SAP_AMOUNT.....=..IDOC_SEGMENT-VALUE
.....IMPORTING
.....IDOC_AMOUNT.....=..IDOC_SEGMENT-VALUE.
*.....exceptions
*.....others.....=..1.

ENDFORM....."E1XITEM_CURRENCY_SA
P_TO_IDOC
*&-----
-----*
*&.....Form.....E1XHEAD_CODES_SAP_TO_ISO
*&-----
-----*
*.....text.....
*.....*
*-----
-----*
*...-->..p1.....text
*..<--..p2.....text
*-----
-----*
FORM E1XHEAD_CODES_SAP_TO_ISO
..USING
...APPL_DATA LIKE XHEAD
..CHANGING
...IDOC_SEGMENT LIKE E1XHEAD.

*convert a currency code from SAP code to ISO code
..IF NOT APPL_DATA-CURRENCY IS INITIAL.
...CALL FUNCTION 'CURRENCY_CODE_SAP_TO_ISO'
...EXPORTING
...SAP_CODE = APPL_DATA-CURRENCY
...IMPORTING
...ISO_CODE = IDOC_SEGMENT-CURRENCY.
*.....exceptions
*.....not_found = 1
*.....others.....= 2.
..ELSE.
...IDOC_SEGMENT-CURRENCY = APPL_DATA-CURRENCY.
..ENDIF.

*convert a country from SAP code to ISO code
..IF NOT APPL_DATA-COUNTRY IS INITIAL.

```

Example Program to Generate an IDoc

```

.....CALL·FUNCTION·'COUNTRY_CODE_SAP_TO_ISO'
.....EXPORTING
.....SAP_CODE·=·APPL_DATA-COUNTRY
.....IMPORTING
.....ISO_CODE·=·IDOC_SEGMENT-COUNTRY.
*.....exceptions
*.....not_found·=·1
*.....others·=·2.
·ELSE.
·IDOC_SEGMENT-COUNTRY·=·APPL_DATA-COUNTRY.
·ENDIF.

ENDFORM....."·E1XHEAD_CODES_SAP_T
O_ISO
*&-----
-----*
*&.....Form·E1XITEM_CODES_SAP_TO_ISO
*&-----
-----*
*.....text.....
.....*
*-----
-----*
*..-->..p1.....text
*..<--..p2.....text
*-----
-----*
FORM·E1XITEM_CODES_SAP_TO_ISO
·USING
·APPL_DATA·LIKE·XITEM
·CHANGING
·IDOC_SEGMENT·LIKE·E1XITEM.

*convert·a·unit·of·measure·from·SAP·code·to·ISO·code
·IF·NOT·APPL_DATA-UNIT·IS·INITIAL.
·CALL·FUNCTION·'UNIT_OF_MEASURE_SAP_TO_ISO'
·EXPORTING
·SAP_CODE·=·APPL_DATA-UNIT
·IMPORTING
·ISO_CODE·=·IDOC_SEGMENT-UNIT.
*.....exceptions
*.....not_found·=·1
*.....no_iso_code·=·2
*.....others·=·3.

```

Example Program to Generate an IDoc

```
..ELSE.
....IDOC_SEGMENT-UNIT.=.APPL_DATA-UNIT.
..ENDIF.

*.convert.a.package.type.from.SAP.code.to.ISO.code
..IF.NOT.APPL_DATA-SHIP_INST.IS.INITIAL.
....CALL.FUNCTION.'SAP_TO_ISO_PACKAGE_TYPE_CODE'
.....EXPORTING
.....SAP_CODE.=.APPL_DATA-SHIP_INST
.....IMPORTING
.....ISO_CODE.=.IDOC_SEGMENT-SHIP_INST.
*.....exceptions
*.....not_found.=.1
*.....others.....=2.
..ELSE.
....IDOC_SEGMENT-SHIP_INST.=.APPL_DATA-SHIP_INST.
..ENDIF.

ENDFORM....."E1XITEM_CODES_SAP_T
O_ISO
```

Using the Example Coding

- Create your own function module. Suggestion: MASTER_IDOC_CREATE_<MESSAGE TYPE>. In the interface, you substitute the example parameters APPL_HEAD and APPL_ITEM with the application data you are using to create the IDoc.
- Variable definition: The first data statement can be transferred. The second data statement must be adapted to the individual IDoc.
- You can use (or even enhance) the block in which the distribution model is read. In this example, we used module ALE_MODEL_DETERMINE_IF_TO_SEND
- For all IDoc segment types with currency amounts, you can use form routine E1XITEM_CURRENCY_SAP_TO_IDOC as a model for writing your own conversion routine. We suggest the name: <SEGMENT TYPE>_CURRENCY_SAP_TO_IDOC.
- For all IDoc segment types with fields for which ISO codes exist, you can use form routine E1XITEM_CODES_SAP_TO_ISO as a model for writing your own conversion routine. We suggest the name: <SEGMENTTYP>_CODES_SAP_TO_ISO.
- For all IDoc segment types with fields to be left-justified, you can use form routine E1XHEAD_CONDENSE as a model for writing your own routine. We suggest the name: <SEGMENT TYPE>_CONDENSE. You should set left-justification after the conversions.
- You must adapt the program parts which set up the IDoc from the application data to you own application structures and the IDoc segments.
- If the update program is performed on the application, MASTER_IDOC_DISTRIBUTE may also have to be called IN UPDATE TASK.
- Do not forget COMMIT WORK if one is not performed at a higher level. The Database Commit at the end of the transaction is not sufficient.

Customizing ALE Outbound Processing

ALE can note the application object contained for every outbound IDoc. Example: for a material master IDoc, the ALE layer creates a link between the material number and the IDoc number. For this, Customizing steps are necessary; see reference below.

In ALE receiver determination, you can define the cases in which a receiver receives or does not receive an IDoc; you can do this for each receiver individually. Example: for the material master record, you can define that a receiver only receives materials of a certain material type.

You have to define the distribution criteria - the [Filter Objects \[Extern\]](#), as described below:

- [Defining ALE Object Types \[Seite 81\]](#)
- [Assigning the Object Type for the Outbound Link to the Message Type \[Seite 82\]](#)
- [Assigning the Application Object Type for the Outbound Link to the Message Type \[Seite 83\]](#)



The ALE outbound processing functions are under *ALE Development*: To get to them choose *Tools* → *ALE* → *ALE Development*.

Defining ALE Object Types

Both linking to an application object and filtering in receiver determination assume that the relevant ALE objects exist.

If, as in the above example, you want to link the outbound IDoc with the material number, the material number has to be read from the IDoc data part. However the ALE layer takes the IDoc data as unformatted character strings. ALE therefore needs to know from which message type, segment and fields the ALE object "material number" can be read. The same applies to the ALE object "material type", which is used to determine the receiver.

To define ALE objects, choose *ALE Development* → *IDocs* → *Data Filtering* → *Define Filter Object Type*.

Assigning the Object Type for the Outbound Link to the Message Type

To assign the ALE object type that you want to link to each outbound IDoc to your message type, choose *IDoc* → *ALE objects* → *Link and serialization objects* from the *ALE Development* screen.

Assigning the Application Object Type for the Outbound Link to the Message Type

The link between the application object and the outbound IDoc is created using the BOR. For this, the BOR object type must be specified for the message type. Example: BUS1001 for the material number.

To carry out these functions choose *IDoc* → *ALE objects* → *Assign to message type*.



Part of the ALE Customizing for outbound processing is partner-independent. It applies to all the message flows relating to one message type.

Outbound Processing Using Message Control

IDoc dispatch is triggered in Message Control and not in the application.

If you want to use Message Control, note the following points when developing and customizing the program.

Customizing

- In addition to making the settings above, you must create an outbound process code. The process code determines the function module of the application that creates the IDoc.
- For the combination application / message type, you can use the following transmission media: "6" (without ALE model, no conversion of NAST recipient on corresponding logical system) and "A" (with ALE model).
- Transmission medium 6: you should call form routine `edi_processing(rsnasted)` from Message Control. Transmission medium A: you can use form routine `ale_processing(rsnasted)` or write your own form routine for transmitting a call of `master_idoc_distribute`.

Programming

- The interface of the function module that creates the IDoc is preset. For an example, see `idoc_output_orders`.
- You do not need to call `master_idoc_distribute`.
- Do not send a COMMIT WORK.

Implementing Inbound Processing

This section describes the implementation of an ALE-enabling interface for inbound IDocs. The prerequisite is that message types and IDoc types have already been defined.

Components of an ALE interface for processing inbound IDocs

A complete ALE interface for the processing of inbound IDocs consists of:

1. A function module to process inbound IDocs. The ALE layer calls the function module.
2. An SAP Business Workflow task with objects and events for error handling.
3. ALE table entries ('settings').

Processing an inbound IDoc

1. ALE reads the inbound IDoc customizing settings to determine which function module to call.
2. The function module is called and given the IDoc(s) to be processed.
3. The function module returns status information about whether the IDoc was successfully processed.
4. If it was successfully processed, it returns the ID of the application document that was created or changed.
5. If an error occurred while processing the IDoc, the ALE layer triggers the SAP Business Workflow event assigned to the IDoc.

See also:

[Inbound Function Module \[Seite 86\]](#)

[ALE Settings](#)

[Objects, Events and Tasks To Be Created \[Seite 209\]](#)

[Inbound Processing Using SAP Workflow \[Seite 158\]](#)

[Advanced Workflow Programming \[Seite 163\]](#)

Inbound Function Module

This section shows how to implement the function module called to process the inbound IDoc(s). It starts with an explanation of what needs to be considered to ensure consistent data.

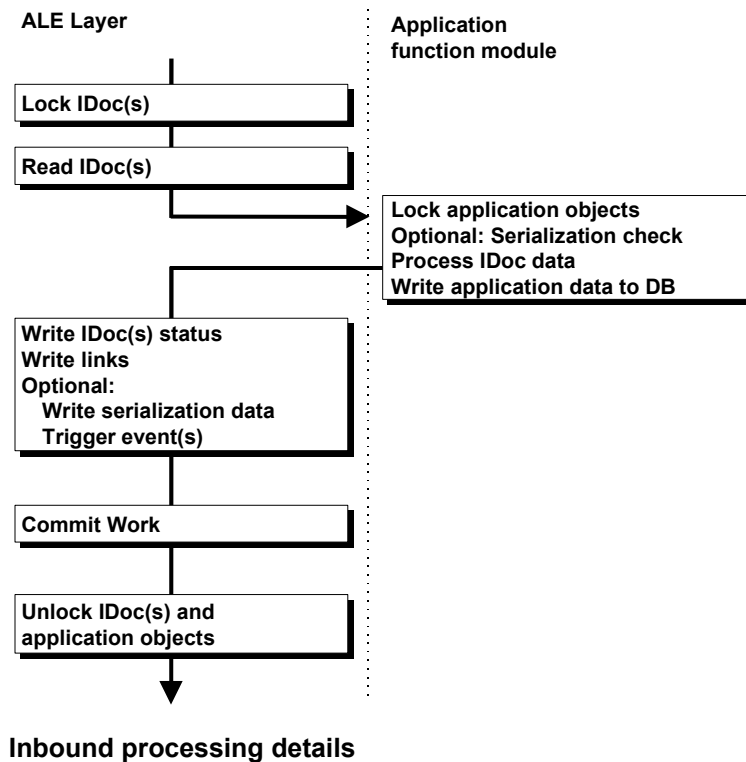
The simplest case is outlined, where the inbound function module only processes one IDoc at a time. The implementation of the serialization function and the addition of customer exits is described. Then, the implementation of an inbound function module that can process more than one IDoc at a time ("mass processing") is outlined. The section ends with an explanation of how to ensure data consistency when using a Call Transaction.

See also:

- [Embedding a Function Module in ALE Inbound Processing \[Seite 87\]](#)
- [Data Consistency \[Seite 88\]](#)
- [Processing IDocs Individually \[Seite 91\]](#)
- [Serialization \[Seite 118\]](#)
- [Customer Exits \[Seite 125\]](#)
- [Mass Processing \[Seite 132\]](#)
- [Using Call Transaction \[Seite 144\]](#)

Embedding a Function Module in ALE Inbound Processing

The following diagram shows how the inbound function module is embedded in ALE's inbound processing. It applies to all cases, except when an IDoc has been successfully processed using a Call Transaction on an ALE-enabled transaction.



Data Consistency

See also:

[Ensuring Data Consistency \[Seite 89\]](#)

[Serialization \[Seite 90\]](#)

Ensuring Data Consistency

In order to ensure data consistency when a database error occurs, the application tables must be posted in the same logical unit of work as the IDoc status table. Otherwise a database error, such as "no tablespace", could lead to the following:

- Application tables update correctly, IDoc status update fails: the IDoc has the status "ready to be passed to application" and would be processed a second time, although the data has already been processed.



Example: the IDoc contains financial postings; processing the IDoc twice would result in two FI documents being created for the same data.

- IDoc status update succeeds, application update fails: the IDoc has the status "successfully processed" although the application tables were not updated. The IDoc cannot be processed again, so the data never gets to the application.

To ensure that only one logical unit of work is used, the application function module must not commit the data to the database. The data is committed by the ALE layer after the function has been called and the IDoc's status records have been updated.

This is not possible when the application data is updated via a Call Transaction, since a Commit Work is automatically set at the end of every Call Transaction. In this case the transaction itself needs to be modified, as shown later.

A Call Dialog instead of a Call Transaction avoids the Commit Work, but can only be used if the dialog step contains no PERFORM ON COMMIT statements - since the commit is set via ALE after the dialog has ended, all global variables set during the dialog are lost. Hence a Call dialog should be used with extreme caution!

When the IDocs contain master data (e.g. customer data) the problem is not so acute, since missing data can always be sent again if need be, and processing an IDoc twice does not lead to duplicate documents being created.

Serialization

If it is important for the inbound IDocs to be processed in the correct order, you can make use of the ALE serialization function to check whether an IDoc has been overtaken or not. This is explained in more detail later.

Processing IDocs Individually

This section describes how individual IDocs are processed. First the steps involved in processing one IDoc at a time are described, starting with the function module's import parameters, then processing the IDoc and filling the function's export parameters. Finally, an example of some coding is shown.

See also:

[Naming Convention \[Seite 92\]](#)

[The Function Module's Interface \[Seite 93\]](#)

[Import Parameters \[Seite 94\]](#)

[IDoc Processing \[Seite 95\]](#)

[Export Parameters \[Seite 96\]](#)

[Coding Example \[Seite 100\]](#)

Naming Convention

We suggest naming the function module used to process incoming IDocs with message type "MSGTYP" "IDOC_INPUT_MSGTYP"; customers could use "Z_IDOC_INPUT_MSGTYP".

The Function Module's Interface

Type	Direction	Parameter	Reference field / structure
Import	-->	INPUT_METHOD	
	-->	BDWFAP_PAR-INPUTMETHOD MASS_PROCESSING	
Export	<--	IN_UPDATE_TASK	
	<--	CALL_TRANSACTION_DONE	
	<--	WORKFLOW_RESULT	
	<--	APPLICATION_VARIABLE	
Tables	-->	IDOC_CONTRL	EDIDC
	-->	IDOC_DATA	EDIDD
	<--	IDOC_STATUS	BDIDOCSTAT
	<--	RETURN_VARIABLES	BDWFRETVAR
	<--	SERIALIZATION_INFO	BDI_SER
Exception		WRONG_FUNCTION_CALLED	

Import Parameters

Import Parameters

Parameter	Description
<i>IDoc_Contrl</i>	This table contains one entry for each IDoc's control record, and is only used by the function module for importing data. Typically the only fields used by the inbound function module are Docnum (the IDoc number) and either Mestyp (Message type) or Idoctp (Basic IDoc type).
<i>Idoc_Data</i>	This table contains one entry for each IDoc data segment. The following fields are relevant to the inbound function module: Docnum the IDoc number Segnam the segment's name; Sdata the segment's data.
<i>Input_Method</i>	Indicates whether the IDoc should be processed in dialog (i.e. via Call Transaction), or not. Possible values are: " " Background (no dialog) "A" Show all screens "E" Start the dialog on the screen where the error occurred Note: the parameter can only take on the values "A" or "E" if the ALE setting for the function module says that it supports dialog processing.
<i>bdwfap_par-inputmethod</i> <i>Mass_Processing</i>	Not used any more (except for advanced workflow programming). It is initial when the methods InputForeground or InputBackground are invoked; otherwise it contains "X".

IDoc Processing

The function module should carry out the following steps:

- Check that the IDoc contains the correct message type (field `Idoc_Contrl-Mestyp`). If it does not, raise the exception `Wrong_Function_Called` with an appropriate message.
 - When implementing an inbound function module for master data that can be "reduced" by customers, don't check the message type; check the basic IDoc type instead (field `Idoc_Contrl-Idoctp`) Initialize/refresh any global variables and/or tables.
- Initialize/refresh any global variables and/or tables. The inbound function module can be called a number of times by the same process, so the global variables will not be empty the second time around.
- Convert the character data in table `Idoc_Data` to internal format in internal tables. See the example coding below to see how to do this. Special attention must be paid to fields containing:
 - Units of measure (ISO code in IDoc)
 - Currency codes (ISO code in IDoc)
 - Country codes (ISO code in IDoc)
 - Shipping instructions (ISO code in IDoc)
 - Monetary amounts (conversion factor needed)
 - Dates and times (see below)

Fields containing dates and times can lead to errors during inbound processing when the field in the IDoc is empty (i.e. blank): In ABAP, moving a blank character field into a date field leaves the date field blank, rather than initial (all zeros) i.e. the date field ends up containing an invalid value. Errors will occur in subsequent processing whenever the field is checked for an initial value "if DateField is initial..." To avoid this, clear the date field if the IDoc field is empty, as shown in the example code.



Remember: the function module should **not** do a Commit Work.

If you have the choice, don't update the database using Call Function "xxx" In Update Task - it is unnecessary for ALE inbound processing, and only increases database load.

Export Parameters

See also:

[The Inbound Function Module's Export Parameters \[Seite 97\]](#)

[Export Parameters When IDoc was Successfully Processed \[Seite 98\]](#)

[Export Parameters When an Error Occurred in IDoc Processing \[Seite 99\]](#)

The Inbound Function Module's Export Parameters

Parameter	Description
<i>In_Update_Task</i>	Flag: Is the Update Task being used to update the database? Is the Update Task being used to update the database? i.e. does your function module update the database using Call Function "xxx" In Update Task?
<i>Call_Transaction_Done</i>	Flag: was a Call Transaction used that has updated the IDoc's status?
<i>Workflow_Result</i>	A parameter that controls whether any events are triggered.
<i>Application_Variable</i>	An optional parameter passed on to workflow.
<i>Idoc_Status</i>	A table that should contain one record for each IDoc processed. Valid values for the field Status are: "51" Error occurred "53" IDoc successfully processed
<i>Return_Variables</i>	A table containing the document numbers of the IDoc and the application object processed. These numbers are used for workflow (e.g. error handling) and for linking the IDoc to the corresponding application object.
<i>Serialization_Info</i>	Not relevant yet - see the section on serialization.

Export Parameters When IDoc was Successfully Processed

Export Parameters When IDoc was Successfully Processed

This example assumes IDoc number 4711 is being processed and that application document number 1234 was created.

Parameter	Value	Description
IN_UPDATE_TASK	" " "X"	Update task not used Update task used
CALL_TRANSACTION_DONE	" " (e.g. initial value)	
WORKFLOW_RESULT	"0"	
APPLICATION_VARIABLE	" " (e.g. initial value)	
IDOC_STATUS	The table must contain one record with fields containing: Docum: 4711 Status: 53 Optionally the fields Msgid etc. can be filled containing the application's success message.	
RETURN_VARIABLES	The table must contain the following two entries:	
	WF_PARAM	Doc_Number
	PROCESSED_IDOCS	4711
	APPL_OBJECTS	1234
	If processing the inbound IDoc does not create or change an application object, the APPL_OBJECTS entry can be omitted. It makes no sense without a document number.	
SERIALIZATION_INFO	Empty	

Export Parameters When an Error Occurred in IDoc Processing

This example assumes that IDoc number 4711 is being processed.

Parameter	Value	
IN_UPDATE_TASK	" " (i.e. initial value) - Update task not used	
CALL_TRANSACTION_DONE	" " (e.g. initial value)	
WORKFLOW_RESULT	"99999"	
APPLICATION_VARIABLE	" " (i.e. initial value)	
IDOC_STATUS	The table must contain one record with fields containing: Docnum: 4711 Status: 51 Msgid, Msgno etc. must be filled with the error message's ID, number etc.	
RETURN_VARIABLES	The table must contain the following entry:	
	WF_PARAM	Doc_Number
	"RROR_IDOCS	4711
SERIALIZATION_INFO	Empty	

Example of Processing an IDoc

Example of Processing an IDoc

The [Example Program to Process an IDoc \[Seite 101\]](#) shows how the fictitious message type XAMPLE, communicated with IDocs of type XAMPLE01, is processed using the inbound function module `Idoc_Input_Xample`. The IDoc type has a header segment, E1xhead, and any number of item segments E1xitem. The data from the IDoc is written to two database tables, XHEAD and XITEM respectively. XHEAD and XITEM contain the same field names as E1xhead and E1xitem respectively. The fields names and data types are shown in the following two tables:

Field Name in XHEAD	Description	Type in E1XHEAD	Type in XHEAD
DOCMNT_NO	Document number	CHAR	NUMC
DATE	Date	CHAR	DATS
CURRENCY	Currency	CHAR	CUKY
COUNTRY	Country	CHAR	CHAR

Field Name in XITEM	Description	Type in E1XITEM	Type in XITEM
ITEM_NO	Item number	CHAR	NUMC
MATERIALID	Material number	CHAR	CHAR
DESCRIPT	Material description	CHAR	CHAR
UNIT	Unit of measure	CHAR	UNIT
QUANTITY	Quantity	CHAR	QUAN
VALUE	Value	CHAR	CURR
SHIP_INST	Shipping instructions	CHAR	UNIT

The data on the database is assigned a new document number (field `DOCMNT_NO`) using number assignment. The field `DOCMNT_NO` is not stored in the newly created table XHEAD.

Example Program to Process an IDoc

```

FUNCTION IDOC_INPUT_XAMPLE.
* "-----
* "
* "Local interface:
* ".....IMPORTING
* ".....VALUE (INPUT_METHOD) LIKE BDWFAP_PAR-INPUTMETHD
* ".....VALUE (MASS_PROCESSING) LIKE BDWFAP_PAR-MASS_PROC
* ".....EXPORTING
* ".....VALUE (WORKFLOW_RESULT) LIKE BDWF_PARAM-RESULT
* ".....VALUE (APPLICATION_VARIABLE) LIKE BDWF_PARAM-
APPL_VAR
* ".....VALUE (IN_UPDATE_TASK) LIKE BDWFAP_PAR-UPDATETASK
* ".....VALUE (CALL_TRANSACTION_DONE) LIKE BDWFAP_PAR-
CALLTRANS
* ".....TABLES
* ".....IDOC_CONTRL STRUCTURE EDIDC
* ".....IDOC_DATA STRUCTURE EDIDD
* ".....IDOC_STATUS STRUCTURE BDIDOCSTAT
* ".....RETURN_VARIABLES STRUCTURE BDWFRETVAR
* ".....SERIALIZATION_INFO STRUCTURE BDI_SER
* ".....EXCEPTIONS
* ".....WRONG_FUNCTION_CALLED
* "-----
* "
* "-----
* "-----
* "-----05 July 1996-----
* "-----
* "-----
* "Example function module for processing inbound IDocs for ALE or
* "EDI.
* "This example applies for processing
* "
* "...with...one IDoc at a time
* "
* "...without...serialization
* ".....customer-exits
* ".....calling an ALE-enabled transaction
* ".....mass processing (more than one IDoc at a time)
* "-----
* "-----Naming conventions-----

```

Example Program to Process an IDoc

```

-----
*Internal tables start with 't_'
*Internal field strings start with 'f_'
*-----
-----

*>>The following line must appear in the global part of your
*>>function group:
*...include mbdconwf....."Report containing the ALE constants.
*The ALE constants start with 'c_'.

..DATA: SUBRC LIKE SY-SUBRC,
.....OBJECT_NUMBER LIKE XHEAD-DOCMNT_NO.

*Initialize variables
..SUBRC = 0.

*Read the IDoc's control record
..READ TABLE IDOC_CONTRL INDEX 1.

*Process the IDoc and post the data to the database
..PERFORM IDOC_PROCESS_XAMPLE TABLES IDOC_DATA
.....IDOC_STATUS
.....USING IDOC_CONTRL
.....CHANGING OBJECT_NUMBER
.....SUBRC.

*Fill the ALE export parameters
..CLEAR IN_UPDATE_TASK.
..CLEAR CALL_TRANSACTION_DONE....."Call Transaction is not used.

..IF SUBRC <> 0....."Error occurred

....WORKFLOW_RESULT = C_WF_RESULT_ERROR.
....RETURN_VARIABLES-WF_PARAM = C_WF_PAR_ERROR_IDOCS.
....RETURN_VARIABLES-DOC_NUMBER = IDOC_CONTRL-DOCNUM.
....APPEND RETURN_VARIABLES.

..ELSE....."IDoc processed successfully

....WORKFLOW_RESULT = C_WF_RESULT_OK.
....RETURN_VARIABLES-WF_PARAM = C_WF_PAR_PROCESSED_IDOCS.
....RETURN_VARIABLES-DOC_NUMBER = IDOC_CONTRL-DOCNUM.
....APPEND RETURN_VARIABLES.
....RETURN_VARIABLES-WF_PARAM = C_WF_PAR_APPL_OBJECTS.

```

Example Program to Process an IDoc

```

...RETURN_VARIABLES-Doc_NUMBER:=OBJECT_NUMBER.
...APPEND RETURN_VARIABLES.

..ELSE.

ENDFUNCTION.

*-----*
*-----*
*.....FORM IDOC_PROCESS_XAMPLE.....
*.....*
*-----*
*-----*
*..This routine creates an application document based on the IDoc
's....*
*..contents..Object_Number contains the new document's number...
*.....*..If an error occurs, subrc is non-
zero, t_idoc_status is filled.....*..Note: if more than one erro
r is detected, t_idoc_status contains...*
*.....*more than one status record.....
*.....*
*-----*
*-----*
*...--
>..F_IDOC_CONTRL.....IDoc control record.....
*
*...--
>..T_IDOC_DATA.....IDoc data records.....
*
*...<--
..T_IDOC_STATUS.....IDoc status records.....*
*...<--
..OBJECT_NUMBER.....Created document's number.....*
*...<--
..SUBRC.....Return code.....*
*-----*
*-----*
FORM IDOC_PROCESS_XAMPLE
.....TABLES...T_IDOC_DATA.....STRUCTURE EDIDD
.....T_IDOC_STATUS...STRUCTURE BDIDOCSTAT
.....USING...F_IDOC_CONTRL...STRUCTURE EDIDC
.....CHANGING OBJECT_NUMBER...LIKE XHEAD-DOCMNT_NO
.....SUBRC.....LIKE SY-SUBRC.

*..Internal field string for the document header.
..DATA: F_XHEAD LIKE XHEAD.

*..Internal table for the document items.

```

Example Program to Process an IDoc

```

..DATA: T_XITEM LIKE XITEM OCCURS 0 WITH HEADER LINE.

*Number given to the created document DOCUMENT_NUMBER LIKE F_XHEAD-DOCMNT_NO.

*Move the data in the IDoc to the internal structures/tables
*f_xhead and t_xitem.
..PERFORM IDOC_INTERPRET TABLES...T_IDOC_DATA
.....T_XITEM
.....T_IDOC_STATUS
.....USING...F_IDOC_CONTRL
.....CHANGING F_XHEAD
.....SUBRC.

*Create the application object if no error occurred so far.
..IF SUBRC = 0.
*...This fictitious function module creates a new object based on
the
*...data that was read from the IDoc. The new object's ID is returned
in the parameter 'document_number'.
*...The function module checks that the data is correct, and raises
an exception if an error is detected.
...CALL FUNCTION 'XAMPLE_OBJECT_CREATE'
.....EXPORTING
.....XHEAD.....= F_XHEAD
.....IMPORTING
.....DOCUMENT_NUMBER = DOCUMENT_NUMBER
.....TABLES
.....XITEM.....= T_XITEM
.....EXCEPTIONS
.....OTHERS.....= 1.

...IF SY-SUBRC <> 0.
.....SUBRC = 1.
*.....Put the error message into 't_idoc_status'
*.....PERFORM STATUS_FILL_SY_ERROR
*.....TABLES...T_IDOC_STATUS
*.....USING...T_IDOC_DATA
*.....SY
*.....'Field name
*.....'idoc_process_xample'....."Form routine

...ELSE.
*.....Fill the remaining export parameters
*.....OBJECT_NUMBER = DOCUMENT_NUMBER....."New document's n

```


Example Program to Process an IDoc

umber

```
.....t_idoc_status-docnum:=f_idoc_ctrl-docnum.
.....t_idoc_status-status:=c_idoc_status_ok.
.....t_idoc_status-msgty:='S'.
.....t_idoc_status-msgid:=your_msgid."Global variable.
.....t_idoc_status-msgno:=msgno_success."Global variable.
.....t_idoc_status-msgvl:=object_number.
.....APPEND T_IDOC_STATUS.
....ENDIF....."if sy-subrc<>0.
..ENDIF....."if subrc=0.
```

ENDFORM.

```
*-----
-----*
*.....FORM IDOC_INTERPRET.....
*.....*
*-----
-----*
*..This routine checks that the correct message type is being use
d, ...*
*..and then converts and moves the data from the IDoc segments to
the.*
*..internal structure f_xhead and internal table t_xitem.....
*.....*
*..If an error occurs, t_idoc_status is filled and subrc<>0.....
*.....*
*-----
-----*
*...--
>..T_IDOC_STATUS.....
*
*...--
>..T_XITEM.....
*
*...--
>..F_IDOC_DATA.....
*
*...--
>..F_XHEAD.....
*
*...--
>..SUBRC.....
*
*-----
-----*
FORM IDOC_INTERPRET TABLES...T_IDOC_DATA...STRUCTURE EDIDD
```

Example Program to Process an IDoc

```

.....T_XITEM.....STRUCTURE·XITEM
.....T_IDOC_STATUS.....STRUCTURE·BDIDOCSTAT
.....USING.....F_IDOC_CONTRL.....STRUCTURE·EDIDC
.....CHANGING·F_XHEAD.....STRUCTURE·XHEAD
.....SUBRC.....LIKE·SY-SUBRC.

*·Check·that·the·IDoc·contains·the·correct·message·type.
*·Note:·if·your·message·type·is·reducible,·check·field·'idoctp'
*.....(IDoc·type)·instead·of·'mestyp'.
*IF·F_IDOC_CONTRL-MESTYP·<>·'XAMPLE'.

.....MESSAGE·ID.....YOUR_MSGID....."Global·variable
.....TYPE.....'E'
.....NUMBER·MSGNO_WRONG_FUNCTION....."Global·variable
.....WITH.....F_IDOC_CONTRL-MESTYP....."message·type
.....'IDOC_INPUT_XAMPLE'....."Your·function·modul
e.
.....F_IDOC_CONTRL-SNDPRT....."Sender·partner·type
.....F_IDOC_CONTRL-SNDPRN....."Sender·number.
.....RAISING·WRONG_FUNCTION_CALLED.

*·ENDIF.

*·Loop·through·the·IDoc's·segments·and·convert·the·data·from·the·
IDoc
*·format·to·the·internal·format.
*LOOP·AT·T_IDOC_DATA·WHERE·DOCNUM·=·F_IDOC_CONTRL-DOCNUM.

.....CASE·T_IDOC_DATA-SEGNAME.

.....WHEN·'E1XHEAD'.
.....PERFORM·E1XHEAD_PROCESS·TABLES.....T_IDOC_STATUS
.....USING.....T_IDOC_DATA
.....CHANGING·F_XHEAD
.....SUBRC.
.....WHEN·'E1XITEM'.
.....PERFORM·E1XITEM_PROCESS·TABLES.....T_XITEM
.....T_IDOC_STATUS
.....USING.....F_XHEAD-CURRENCY
.....T_IDOC_DATA
.....CHANGING·SUBRC.

.....ENDCASE.

*·ENDLOOP.

ENDFORM.

```

Example Program to Process an IDoc

```

*-----*
*-----*
*.....FORM·E1XHEAD_PROCESS.....
*.....*
*-----*
*-----*
*..This routine fills 'f_xhead' out of segment elxhead.....
*.....*..If an error occurs, subrc is non-
zero, t_idoc_status is filled.....*
*-----*
*-----*
*...--
>..F_IDOC_DATA.....IDoc segment containing elxhead fields.....
*
*...<--
..F_XHEAD.....Internal structure containing doc.header...*
*...<--
..T_IDOC_STATUS.....Status fields for error handling.....*
*...<--..SUBRC.....Return code: non-
zero if an error occurred..*
*-----*
*-----*
FORM·E1XHEAD_PROCESS·TABLES··T_IDOC_STATUS··STRUCTURE·BDIDOCSTAT
.....USING··F_IDOC_DATA··STRUCTURE·EDIDD
.....CHANGING·F_XHEAD··STRUCTURE·XHEAD
.....SUBRC····LIKE·SY-SUBRC.

..DATA:·F_E1XHEAD·LIKE·E1XHEAD.

..F_E1XHEAD·=·F_IDOC_DATA-SDATA.

*Process fields that need conversion from ISO-codes to SAP-codes
..PERFORM·E1XHEAD_CODES_ISO_TO_SAP
.....TABLES··T_IDOC_STATUS
.....USING··F_E1XHEAD
.....F_IDOC_DATA
.....CHANGING·F_XHEAD
.....SUBRC.

*Process fields containing dates or times
..PERFORM·E1XHEAD_DATE_TIME·USING··F_E1XHEAD
.....CHANGING·F_XHEAD.

ENDFORM....."elxhead_process

*-----*

```

Example Program to Process an IDoc

```

-----*
*.....FORM·E1XITEM_PROCESS.....
*.....*
*-----*
-----*
*..This·routine·converts·the·data·in·the·segment·'elxitem'·for··
*.....*
*..to·the·format·of·table·'t_xitem'·and·appends·it·to·the·table.·
*.....*..If·an·error·occurs,·subrc·is·non-
zero,·t_idoc_status·is·filled.....*
*-----*
-----*
*...--
>..F_IDOC_DATA.....IDoc·segment.....
*
*...<--
..T_XITEM.....Document·items·to·be·updated·to·database.....*
*...<--
..T_IDOC_STATUS....Status·fields·filled·if·an·error·occurred.....*
*...<--
..SUBRC.....Return·code:·0·if·all·OK.....*
*-----*
-----*
FORM·E1XITEM_PROCESS·TABLES...T_XITEM.....STRUCTURE·XITEM
.....T_IDOC_STATUS·STRUCTURE·BDIDOCSTAT
.....USING...CURRENCY.....LIKE·XHEAD-CURRENCY
.....F_IDOC_DATA...STRUCTURE·EDIDD
.....CHANGING·SUBRC.....LIKE·SY-SUBRC.

..DATA:·F_E1XITEM·LIKE·E1XITEM.

..F_E1XITEM·=·F_IDOC_DATA-SDATA.

*Fields·of·type·CHAR,·NUMC,·QUAN·need·no·conversion.
*..T_XITEM-ITEM_NO.....=·F_E1XITEM-ITEM_NO.
*..T_XITEM-MATERIALID·=·F_E1XITEM-MATERIALID.
*..T_XITEM-DESCRIPT.....=·F_E1XITEM-DESCRIPT.
*..T_XITEM-QUANTITY.....=·F_E1XITEM-QUANTITY.

*Process·fields·that·need·conversion·from·ISO-codes·to·SAP-codes
*..PERFORM·E1XHEAD_CODES_ISO_TO_SAP
*.....TABLES...T_IDOC_STATUS
*.....USING...F_E1XHEAD
*.....F_IDOC_DATA
*.....CHANGING·F_XHEAD
*.....SUBRC.

*Process·fields·that·contain·monetary·values

```

Example Program to Process an IDoc

```

..PERFORM E1XITEM_VALUE_IDOC_TO_SAP TABLES...T_IDOC_STATUS
.....USING...F_E1XITEM
.....CURRENCY
.....F_IDOC_DATA
.....CHANGING T_XITEM
.....SUBRC.

..APPEND T_XITEM.

ENDFORM.

*-----*
*-----*
*.....FORM E1XHEAD_CODES_ISO_TO_SAP.....*
*.....*
*-----*
*-----*
*..Converts ISO-Codes in f_elxhead to SAP-
codes in f_xhead.....*..f_idoc_data, t_idoc_status and sub
rc are used for error handling.*
*-----*
*-----*
FORM E1XHEAD_CODES_ISO_TO_SAP
.....TABLES...T_IDOC_STATUS STRUCTURE BDIDOCSTAT
.....USING...F_E1XHEAD.....STRUCTURE E1XHEAD
.....F_IDOC_DATA.....STRUCTURE EDIDD
.....CHANGING F_XHEAD.....STRUCTURE XHEAD
.....SUBRC.

*f_xhead-currency...Type CUKY=>convert ISO-Code to SAP-Code.
..PERFORM CURRENCY_CODE_ISO_TO_SAP
.....TABLES...T_IDOC_STATUS
.....USING...F_E1XHEAD-CURRENCY
.....F_IDOC_DATA
.....'CURRENCY'
.....CHANGING F_XHEAD-CURRENCY
.....SUBRC.

..CHECK SUBRC=0.

*f_xhead-
country...Contains a country=>convert from ISO to SAP code.
..PERFORM COUNTRY_CODE_ISO_TO_SAP
.....TABLES...T_IDOC_STATUS
.....USING...F_E1XHEAD-COUNTRY
.....F_IDOC_DATA
.....'COUNTRY'
.....CHANGING F_XHEAD-COUNTRY
.....SUBRC.

```

Example Program to Process an IDoc

```

ENDFORM.

*-----*
*-----*
*.....FORM·E1XITEM_CODES_ISO_TO_SAP.....*
*.....*
*-----*
*-----*
*..Converts·ISO-Codes·in·f_elxitem·to·SAP-
codes·in·f_xitem.....*
*..f_idoc_data,·t_idoc_status·and·subrc·are·used·for·error·handli
ng..*
*-----*
*-----*
FORM·E1XITEM_CODES_ISO_TO_SAP
.....TABLES.....T_IDOC_STATUS·STRUCTURE·BDIDOCSTAT
.....USING.....F_E1XITEM.....STRUCTURE·E1XITEM
.....F_IDOC_DATA.....STRUCTURE·EDIDD
.....CHANGING·F_XITEM.....STRUCTURE·XITEM
.....SUBRC.....LIKE·SY-SUBRC.

*f_xitem-unit.....Type·UNIT·=>·convert·ISO-Code·to·SAP-Code.
..PERFORM·UNIT_OF_MEASURE_ISO_TO_SAP
.....TABLES.....T_IDOC_STATUS
.....USING.....F_E1XITEM-UNIT
.....F_IDOC_DATA
.....'unit'
.....CHANGING·F_XITEM-UNIT
.....SUBRC.

*f_xitem-
ship_inst..Contains·shipping·instructions·=>·ISO·to·SAP·code.
..PERFORM·SHIPPING_INSTRUCT_ISO_TO_SAP
.....TABLES.....T_IDOC_STATUS
.....USING.....F_E1XITEM-SHIP_INST
.....F_IDOC_DATA
.....'ship_inst'
.....CHANGING·F_XITEM-SHIP_INST
.....SUBRC.

ENDFORM.

*-----*
*-----*
*.....FORM·E1XITEM_VALUE_IDOC_TO_SAP.....*
*.....*
*-----*

```

Example Program to Process an IDoc

```

-----*
*..Converts fields containing monetary values in f_elxitem to....
*.....*
*..the internal representation in f_xitem.....
*.....*..f_idoc_data, t_idoc_status and subrc are used for error handling.*
*-----
-----*
FORM E1XITEM_VALUE_IDOC_TO_SAP
.....TABLES...T_IDOC_STATUS STRUCTURE BDIDOCSTAT
.....USING...F_E1XITEM.....STRUCTURE E1XITEM
.....CURRENCY.....LIKE XHEAD-CURRENCY
.....F_IDOC_DATA...STRUCTURE EDIDD
.....CHANGING F_XITEM.....STRUCTURE XITEM
.....SUBRC.....LIKE SY-SUBRC.

*f_xitem-
value...Type CURR=> convert IDoc amount to internal amount.
*N.B. the currency code used here must be the SAP-
internal one, not
*.....the one contained in the IDoc!
..CALL FUNCTION 'CURRENCY_AMOUNT_IDOC_TO_SAP'
.....EXPORTING
.....CURRENCY.....= CURRENCY
.....IDOC_AMOUNT = F_E1XITEM-VALUE
.....IMPORTING
.....SAP_AMOUNT = F_XITEM-VALUE
.....EXCEPTIONS
.....OTHERS.....= 1.

..IF SY-SUBRC <> 0.
....SUBRC = 1.
*...Put the error message into 't_idoc_status'
....PERFORM STATUS_FILL_SY_ERROR
.....TABLES...T_IDOC_STATUS
.....USING...F_IDOC_DATA
.....SY
.....'value'....."Field name
.....'elxitem_value_idoc_to_sap'....."Form routine
..ENDIF....."if sy-subrc <> 0.

ENDFORM.

*-----
-----*
*.....FORM E1XHEAD_DATE_TIME.....
*.....*

```

Example Program to Process an IDoc

```

*-----*
*-----*
*..Moves date and time fields in f_elxhead to the fields in f_xhead..*
*-----*
*-----*
FORM E1XHEAD_DATE_TIME USING ..... F_E1XHEAD STRUCTURE E1XHEAD
..... CHANGING F_XHEAD STRUCTURE XHEAD.

*f_xhead-date.....Type DATS=>initial value is not 'blank'.
..IF E1XHEAD-DATE IS INITIAL.
....CLEAR F_XHEAD-DATE.
....F_XHEAD-DATE=F_E1XHEAD-DATE.
..ENDIF.

ENDFORM.

*-----*
*-----*
*.....FORM CURRENCY_CODE_ISO_TO_SAP.....*
*.....*
*-----*
*-----*
*..Converts ISO currency code 'iso_currency_code' to SAP code in.....*
*.....*
*..'sap_currency_code'.....*
*.....*
*..f_idoc_data, field_name, t_idoc_status and subrc are used for.....*
*.....*
*..for error handling.....*
*.....*
*-----*
*-----*
FORM CURRENCY_CODE_ISO_TO_SAP
..... TABLES T_IDOC_STATUS..... STRUCTURE BDIDOCSTAT
..... USING ISO_CURRENCY_CODE LIKE TCURC-ISOCD
..... F_IDOC_DATA..... STRUCTURE EDIDD
..... FIELD_NAME..... LIKE BDIDOCSTAT-SEGFLD
..... CHANGING SAP_CURRENCY_CODE LIKE TCURC-WAERS
..... SUBRC..... LIKE SY-SUBRC.

..IF ISO_CURRENCY_CODE IS INITIAL.
....CLEAR SAP_CURRENCY_CODE.
..ELSE.
....CALL FUNCTION 'CURRENCY_CODE_ISO_TO_SAP'
.....EXPORTING
.....ISO_CODE=ISO_CURRENCY_CODE
.....IMPORTING

```


Example Program to Process an IDoc

```

.....SAP_CODE*=SAP_CURRENCY_CODE
.....EXCEPTIONS
.....OTHERS...=1.

....IF SY-SUBRC<>0.
.....SUBRC*=1.
*....Put the error message into 't_idoc_status'
.....PERFORM STATUS_FILL_SY_ERROR
.....TABLES...T_IDOC_STATUS
.....USING....F_IDOC_DATA
.....SY
.....FIELD_NAME
.....'currency_code_iso_to_sap'....."Form routine
utine
..ENDIF....."if sy-subrc<>0.

..ENDIF....."if iso_currency_code is initial.

ENDFORM.

*-----
*-----*
*.....FORM CURRENCY_CODE_ISO_TO_SAP.....
*.....*
*-----
*-----*
*..Converts ISO currency code 'iso_currency_code' to SAP code in
*.....*
*..'sap_currency_code'.....
*.....*
*..f_idoc_data, field_name, t_idoc_status and subrc are used for
*.....*
*..for error handling.....
*.....*
*-----
*-----*
FORM COUNTRY_CODE_ISO_TO_SAP
.....TABLES...T_IDOC_STATUS...STRUCTURE BDIDOCSTAT
.....USING....ISO_COUNTRY_CODE LIKE T005-INTCA
.....F_IDOC_DATA.....STRUCTURE EDIDD
.....FIELD_NAME.....LIKE BDIDOCSTAT-SEGFLD
.....CHANGING SAP_COUNTRY_CODE LIKE T005-LAND1
.....SUBRC.....LIKE SY-SUBRC.

*..Only convert if the field is not initial.
*..IF ISO_COUNTRY_CODE IS INITIAL.
*...CLEAR SAP_COUNTRY_CODE.

```

Example Program to Process an IDoc

```

...ELSE.
...CALL FUNCTION 'COUNTRY_CODE_ISO_TO_SAP'
...EXPORTING
...ISO_CODE = ISO_COUNTRY_CODE
...IMPORTING
...SAP_CODE = SAP_COUNTRY_CODE
...EXCEPTIONS
...OTHERS = 1.

...IF SY-SUBRC <> 0.
...SUBRC = 1.
*...Put the error message into 't_idoc_status'
...PERFORM STATUS_FILL_SY_ERROR
...TABLES T_IDOC_STATUS
...USING F_IDOC_DATA
...SY
...FIELD_NAME
...'country_code_iso_to_sap'....."Form routine
...ENDIF....."if sy-subrc <> 0.

...ENDIF....."if iso_country_code is initial.

ENDFORM.

*-----
*-----*
*...FORM UNIT_OF_MEASURE_ISO_TO_SAP.....
*.....*
*-----
*-----*
*...Converts ISO unit of measure code 'iso_unit_of_measure' to SAP
*.....*
*...code in 'sap_unit_of_measure'.....
*.....*
*...f_idoc_data, field_name, t_idoc_status and subrc are used for
*.....*
*...for error handling.....
*.....*
*-----
*-----*
FORM UNIT_OF_MEASURE_ISO_TO_SAP
...TABLES T_IDOC_STATUS.....STRUCTURE BDIDOCSTAT
...USING ISO_UNIT_OF_MEASURE LIKE T006-ISOCODE
...F_IDOC_DATA.....STRUCTURE EDIDD
...FIELD_NAME.....LIKE BDIDOCSTAT-SEGFLD
...CHANGING SAP_UNIT_OF_MEASURE LIKE T006-MSEHI
...SUBRC.....LIKE SY-SUBRC.

```

```

*Only convert the field if it is not empty.
*IF ISO_UNIT_OF_MEASURE IS INITIAL.
...CLEAR SAP_UNIT_OF_MEASURE.
*ELSE.
...CALL FUNCTION 'UNIT_OF_MEASURE_ISO_TO_SAP'
...EXPORTING
...ISO_CODE:=ISO_UNIT_OF_MEASURE
...IMPORTING
...SAP_CODE:=SAP_UNIT_OF_MEASURE
...EXCEPTIONS
...OTHERS...=1.

...IF SY-SUBRC <> 0.
...SUBRC:=1.
*...Put the error message into 't_idoc_status'
...PERFORM STATUS_FILL_SY_ERROR
...TABLES...T_IDOC_STATUS
...USING...F_IDOC_DATA
...SY
...FIELD_NAME
...unit_of_measure_iso_to_sap'...'Form routine
*ENDIF...if sy-subrc <> 0.

*ENDIF...if iso_unit_of_measure_code is initial
.

ENDFORM.

*-----
*-----*
*...FORM SHIPPING_INSTRUCT_ISO_TO_SAP...
*...*
*-----
*-----*
*...Converts ISO package code 'iso_package_type' to SAP code for...
*...*
*...purchasing shipping instructions in 'sap_shipping_instructions'...
*...*
*...f_idoc_data, field_name, t_idoc_status and subrc are used for...
*...*
*...for error handling...
*...*
*-----
*-----*
FORM SHIPPING_INSTRUCT_ISO_TO_SAP
...TABLES...T_IDOC_STATUS...STRUCTURE BDIDOCSTAT
...USING...ISO_PACKAGE_TYPE...LIKE T027A-IVERS

```

Example Program to Process an IDoc

```

.....F_IDOC_DATA.....STRUCTURE EDIDD
.....FIELD_NAME.....LIKE BDIDOCSTAT-SEGFLD
.....CHANGING SAP_SHIPPING_INSTRUCTIONS LIKE T027A-EVERS
.....SUBRC.....LIKE SY-SUBRC.

*Only convert the field if it is not empty.
..IF ISO_PACKAGE_TYPE IS INITIAL.
...CLEAR SAP_SHIPPING_INSTRUCTIONS.
..ELSE.
...CALL FUNCTION 'ISO_TO_SAP_PACKAGE_TYPE_CODE'
...EXPORTING
...ISO_CODE = ISO_PACKAGE_TYPE
...IMPORTING
...SAP_CODE = SAP_SHIPPING_INSTRUCTIONS
...EXCEPTIONS
...OTHERS = 1.

...IF SY-SUBRC <> 0.
...SUBRC = 1.
*...Put the error message into 't_idoc_status'
...PERFORM STATUS_FILL_SY_ERROR
...TABLES T_IDOC_STATUS
...USING F_IDOC_DATA
...SY
...FIELD_NAME
...shipping_instruct_iso_to_sap'."Form routi.
..ENDIF....."if sy-subrc <> 0.

..ENDIF....."if iso_unit_of_measure_code is initial
.

ENDFORM.

*-----
*-----*
*.....FORM STATUS_FILL_SY_ERROR.....
*.....*
*-----
*-----*
*..Fills the structure t_idoc_status with the import parameters..
*.....*
*..plus the relevant sy fields.....
*.....*
*-----
*-----*
*.....
>..IDOC_NUMBER.....IDoc number.....

```

Example Program to Process an IDoc

```

*
*...--
>..SEGNUM.....Segment.number.....
*
*...--
>..SEGFLD.....Field.in.segment.....
*
*...--
>..ROUTID.....Name.of.routine.....
*
*...<--
..T_IDOC_STATUS.....Status.fields.....*
*-----*
-----*
FORM STATUS_FILL_SY_ERROR TABLES ..T_IDOC_STATUS STRUCTURE BDIDOC
STAT
.....USING.....F_IDOC_DATA...STRUCTURE EDIDD
.....VALUE (F_SY) ...STRUCTURE SY
.....SEGFLD.....LIKE BDIDOCSTAT-
SEGFLD
.....ROUTID.....LIKE BDIDOCSTAT-
ROUTID.

..t_idoc_status-docnum:=f_idoc_data-docnum.
..t_idoc_status-status:=c_idoc_status_error.
..t_idoc_status-msgty:=f_sy-msgty.
..t_idoc_status-msgid:=f_sy-msgid.
..T_IDOC_STATUS-MSGNO:=F_SY-MSGNO.
..t_idoc_status-msgv1:=f_sy-msgv1.
..t_idoc_status-msgv2:=f_sy-msgv2.
..t_idoc_status-msgv3:=f_sy-msgv3.
..t_idoc_status-msgv4:=f_sy-msgv4.
..t_idoc_status-segnum:=f_idoc_data-segnum.
..t_idoc_status-segfld:=segfld.
..t_idoc_status-repid:=f_sy-repid.
..t_idoc_status-routid:=routid.
..APPEND T_IDOC_STATUS.

ENDFORM.

```

Serialization Using Message Types

The ALE serialization function module `SERIALIZATION_CHECK` flags each IDoc that has been overtaken. An overtaken IDoc is defined as follows: assuming IDocs A and B contain information about object/document X (e.g. order number 4711). If A is created by the R3 sending system before B, but B has already been successfully processed by the R3 receiving system, A is said to have been overtaken.

To use serialization you need to:

- Define the serialization object for your message type - ALE extracts the object/document number from the IDoc's data segments, and hence needs to know which field to use.
- Call the function module `SERIALIZATION_CHECK` at the beginning of your inbound function module.
- Handle overtaken IDocs according to your needs.
- Ensure that the inbound function module's export table `SERIALIZATION_INFO` contains the serialization table from the function module `SERIALIZATION_CHECK` (see the example below).



The [Example Program for Serialization \[Seite 119\]](#) shows the additional coding necessary in the function module `Idoc_Input_Xample` to recognize overtaken IDocs and to return an appropriate error message. The example assumes that overtaken IDocs need to be manually dealt with, i.e. they cannot be automatically processed.

Example Program for Serialization

```

FUNCTION IDOC_INPUT_XAMPLE2.
* "-----
* "
* "Local interface:
* ".....IMPORTING
* ".....VALUE (INPUT_METHOD) LIKE BDWFAP_PAR-INPUTMETHD
* ".....VALUE (MASS_PROCESSING) LIKE BDWFAP_PAR-MASS_PROC
* ".....EXPORTING
* ".....VALUE (WORKFLOW_RESULT) LIKE BDWF_PARAM-RESULT
* ".....VALUE (APPLICATION_VARIABLE) LIKE BDWF_PARAM-APPL_VAR
* ".....VALUE (IN_UPDATE_TASK) LIKE BDWFAP_PAR-UPDATETASK
* ".....VALUE (CALL_TRANSACTION_DONE) LIKE BDWFAP_PAR-CALLTRANS
* ".....TABLES
* ".....IDOC_CONTRL STRUCTURE EDIDC
* ".....IDOC_DATA STRUCTURE EDIDD
* ".....IDOC_STATUS STRUCTURE BDIDOCSTAT
* ".....RETURN_VARIABLES STRUCTURE BDWFRETVAR
* ".....SERIALIZATION_INFO STRUCTURE BDI_SER
* ".....EXCEPTIONS
* ".....WRONG_FUNCTION_CALLED
* "-----

* .-----
* .-----05 July 1996-----
* .-----

* Example function module for processing inbound IDocs for ALE or EDI.
* This example applies for processing
*
* ...with ...one IDoc at a time
* .....serialization
*
* ...without ...customer-exits
* .....calling an ALE-enabled transaction
* .....mass processing (more than one IDoc at a time)

* .-----Naming conventions-----
* Internal tables start with 't_'
* Internal field strings start with 'f_'
* .-----

* >> The following line must appear in the global part of your
* >> function group:
* ....include mbdconwf. .... "Report containing the ALE constants.
* The ALE constants start with 'c_'.

* DATA: SUBRC LIKE SY-SUBRC,
* .....OBJECT_NUMBER LIKE XHEAD-DOCMNT_NO.

```

```
*·Initialize·variables
**SUBRC:=0.

*·Read·the·IDoc's·control·record
**READ·TABLE·IDOC_CTRL·INDEX·1.

**PERFORM·IDOC_PROCESS_XAMPLE2·TABLES···IDOC_DATA
*****SERIALIZATION_INFO
*****IDOC_STATUS
*****USING···IDOC_CTRL
*****CHANGING·OBJECT_NUMBER
*****SUBRC.

*·Fill·the·ALE·export·parameters

*·In·this·example·we·assume·that·'call·function·'xxx'·in·update·task'·is
*·not·used·to·update·the·database.
**CLEAR·IN_UPDATE_TASK.
**CLEAR·CALL_TRANSACTION_DONE....."Call·Transaction·is·not·used.

**IF·SUBRC<>0....."Error·occurred

****WORKFLOW_RESULT:=C_WF_RESULT_ERROR.
****RETURN_VARIABLES-WF_PARAM:=C_WF_PAR_ERROR_IDOCS.
****RETURN_VARIABLES-DOC_NUMBER:=IDOC_CTRL-DOCNUM.
****APPEND·RETURN_VARIABLES.

**ELSE....."IDoc·processed·successfully

****WORKFLOW_RESULT:=C_WF_RESULT_OK.
****RETURN_VARIABLES-WF_PARAM:=C_WF_PAR_PROCESSED_IDOCS.
****RETURN_VARIABLES-DOC_NUMBER:=IDOC_CTRL-DOCNUM.
****APPEND·RETURN_VARIABLES.
****RETURN_VARIABLES-WF_PARAM:=C_WF_PAR_APPL_OBJECTS.
****RETURN_VARIABLES-DOC_NUMBER:=OBJECT_NUMBER.
****APPEND·RETURN_VARIABLES.

**ENDIF.

ENDFUNCTION.
```

·····FORM·IDOC_PROCESS_XAMPLE2·····

·This·routine·creates·an·application·document·based·on·the·IDoc's···

·contents·Object_Number·contains·the·new·document's·number·····

·If·an·error·occurs,·subrc·is·non-zero,·t_idoc_status·is·filled····

·Note:·if·more·than·one·error·is·detected,·t_idoc_status·contains···

·····more·than·one·status·record·····

·-->·F IDOC CTRL···IDoc·control·record·····

Example Program for Serialization

```

*-->T_IDOC_DATA.....IDoc·data·records.....*
*-<--T_IDOC_STATUS.....IDoc·status·records.....*
*-<--OBJECT_NUMBER.....Created·document's·number.....*
*-<--SUBRC.....Return·code.....*
*-----*
FORM IDOC_PROCESS_XAMPLE2
.....TABLES.....T_IDOC_DATA.....STRUCTURE·EDIDD
.....T_SERIALIZATION_INFO·STRUCTURE·BDI_SER
.....T_IDOC_STATUS.....STRUCTURE·BDIDOCSTAT
.....USING.....F_IDOC_CONTRL.....STRUCTURE·EDIDC
.....CHANGING·OBJECT_NUMBER.....LIKE·XHEAD-DOCMNT_NO
.....SUBRC.....LIKE·SY-SUBRC.

*Internal·field·string·for·the·document·header.
*DATA:F_XHEAD·LIKE·XHEAD.

*Internal·table·for·the·document·items.
*DATA:T_XITEM·LIKE·XITEM·OCCURS·0·WITH·HEADER·LINE.

*Number·given·to·the·created·document
*DATA:DOCUMENT_NUMBER·LIKE·F_XHEAD-DOCMNT_NO.

*Move·the·data·in·the·IDoc·to·the·internal·structures/tables
*f_xhead·and·t_xitem.
*PERFORM IDOC_INTERPRET2·TABLES.....T_IDOC_DATA
.....T_SERIALIZATION_INFO
.....T_XITEM
.....T_IDOC_STATUS
.....USING.....F_IDOC_CONTRL
.....CHANGING·F_XHEAD
.....SUBRC.

*Create·the·application·object·if·no·error·occurred·so·far.
*IF SUBRC=0.
*...This·fictitious·function·module·creates·a·new·object·based·on·the
*...data·that·was·read·from·the·IDoc..The·new·object's·ID·is·returned
*...in·the·parameter·'document_number'.
*...The·function·module·checks·that·the·data·is·correct,·and·raises
*...an·exception·if·an·error·is·detected.
*...CALL·FUNCTION·'XAMPLE_OBJECT_CREATE'
*...EXPORTING
*...XHEAD.....=·F_XHEAD
*...IMPORTING
*...DOCUMENT_NUMBER=·DOCUMENT_NUMBER
*...TABLES
*...XITEM.....=·T_XITEM
*...EXCEPTIONS
*...OTHERS.....=·1.

*...IF SY-SUBRC·<>·0.
*...SUBRC=1.

```

Example Program for Serialization

```

*.....Put the error message into 't_idoc_status'
*.....PERFORM STATUS_FILL_SY_ERROR
*.....TABLES....T_IDOC_STATUS
*.....USING....T_IDOC_DATA
*.....SY
*.....' '....."Field name
*.....'idoc_process_xample'....."Form routine

....ELSE.
*.....Fill the remaining export parameters
*.....OBJECT_NUMBER:=DOCUMENT_NUMBER....."New document's number

*.....t_idoc_status-docnum:=f_idoc_ctrl-docnum.
*.....t_idoc_status-status:=c_idoc_status_ok.
*.....t_idoc_status-msgty:='S'.
*.....t_idoc_status-msgid:=your_msgid.."Global variable.
*.....t_idoc_status-msgno:=msgno_success.."Global variable.
*.....t_idoc_status-msgv1:=object_number.
*.....APPEND T_IDOC_STATUS.
*.....ENDIF....."if sy-subrc <> 0.
*.....ENDIF....."if subrc = 0.

```

ENDFORM.

```

*-----*
*.....FORM IDOC_INTERPRET2.....*
*-----*
*..This routine checks that the correct message type is being used, ...*
*..then checks that the IDoc has not been overtaken (serialization), ...*
*..and then converts and moves the data from the IDoc segments to the ...*
*..internal structure f_xhead and internal table t_xitem.....*
*..If an error occurs, t_idoc_status is filled and subrc <> 0.....*
*-----*
*..-->..T_IDOC_STATUS.....*
*..-->..T_XITEM.....*
*..-->..F_IDOC_DATA.....*
*..-->..F_XHEAD.....*
*..-->..SUBRC.....*
*-----*
FORM IDOC_INTERPRET2 TABLES....T_IDOC_DATA....STRUCTURE EDIDD
.....T_SERIALIZATION_INFO.STRUCTURE BDI_SER
.....T_XITEM.....STRUCTURE XITEM
.....T_IDOC_STATUS..STRUCTURE BDIDOCSTAT
.....USING....F_IDOC_CTRL..STRUCTURE EDIDC
.....CHANGING F_XHEAD.....STRUCTURE XHEAD
.....SUBRC.....LIKE SY-SUBRC.

DATA:..BEGIN OF T_IDOC_CTRL OCCURS 1.
.....INCLUDE STRUCTURE EDIDC.
DATA:..END OF T_IDOC_CTRL.

```

[illegible]

[illegible]

Customer Exits

Customer exits in the inbound function module allow customers to:

- Change the way SAP segments are processed, e.g. by changing the value of some of the fields
- Access the customer segments they added to the SAP IDoc using the IDoc definition tool

The following points should be fulfilled by the customer exits:

- Can all the data in SAP segments in the IDoc be read via customer exits?
- Can all the customer segments in the IDoc be read via customer exits?
- Is a customer exit called every time a segment has been processed?
- Does the customer exit support error handling?
- Can the exit fill the parameters *Workflow_Result*, *Idoc_Status* and *Return_Variables*?
- Is there a customer exit for serialization?
- This makes sense if the inbound function module does not support serialization.
- If the customer has appended fields to an SAP table, are these fields easily accessible in the exit?
- Are the exits long-lived? Customers expect exits to be called with the same meaning in future releases.
- Is it easy for the customer to navigate through the data in the exit? For example, to determine which item is being processed now, etc.
- If the same coding is used for different message types: Can the customer tell which message type is being used?



The [Example Program for a Customer Exit \[Seite 126\]](#) shows the implementation of a customer exit.

Example Program for a Customer Exit

Example Program for a Customer Exit

```

FUNCTION IDOC_INPUT_XAMPLE3.
*"-----
*"
*"Local interface:
*".....IMPORTING
*".....VALUE (INPUT_METHOD) LIKE BDWFAP_PAR-INPUTMETHD
*".....VALUE (MASS_PROCESSING) LIKE BDWFAP_PAR-MASS_PROC
*".....EXPORTING
*".....VALUE (WORKFLOW_RESULT) LIKE BDWF_PARAM-RESULT
*".....VALUE (APPLICATION_VARIABLE) LIKE BDWF_PARAM-APPL_VAR
*".....VALUE (IN_UPDATE_TASK) LIKE BDWFAP_PAR-UPDATETASK
*".....VALUE (CALL_TRANSACTION_DONE) LIKE BDWFAP_PAR-CALLTRANS
*".....TABLES
*".....IDOC_CONTRL STRUCTURE EDIDC
*".....IDOC_DATA STRUCTURE EDIDD
*".....IDOC_STATUS STRUCTURE BDIDOCSTAT
*".....RETURN_VARIABLES STRUCTURE BDWFRETVAR
*".....SERIALIZATION_INFO STRUCTURE BDI_SER
*".....EXCEPTIONS
*".....WRONG_FUNCTION_CALLED
*"-----

*-----
*-----05 July 1996-----
*-----

*Example function module for processing inbound IDocs for ALE or EDI.
*This example applies for processing
*
*...with...one IDoc at a time
*...serialization
*...customer-exits
*
*...without...calling an ALE-enabled transaction
*...mass processing (more than one IDoc at a time)

*-----Naming conventions-----
*Internal tables start with 't_'
*Internal field strings start with 'f_'
*-----

*>>>The following line must appear in the global part of your
*>>>function group:
*...include mbdconwf....."Report containing the ALE constants.
*The ALE constants start with 'c_'.

*DATA: SUBRC LIKE SY-SUBRC,
*.....OBJECT_NUMBER LIKE XHEAD-DOCMNT_NO.

```

[illegible]

Example Program for a Customer Exit

```
*...WORKFLOW_RESULT.=C_WF_RESULT_OK.  
*...RETURN_VARIABLES-WF_PARAM.=C_WF_PAR_PROCESSED_IDOCS.  
*...RETURN_VARIABLES-DOC_NUMBER.=IDOC_CTRL-DocNum.  
*...APPEND RETURN_VARIABLES.  
*...RETURN_VARIABLES-WF_PARAM.=C_WF_PAR_APPL_OBJECTS.  
*...RETURN_VARIABLES-DOC_NUMBER.=OBJECT_NUMBER.  
*...APPEND RETURN_VARIABLES.  
  
**ENDIF.  
  
*>>>>>>>>>Customer.exit.3.(Start) *<<<<<<<<<<<<<<<<<<<<<<<<<<<  
*.This.exit.gives.the.customer.access.to.the.export.parameters.  
.CALL CUSTOMER-FUNCTION.'003'  
.EXPORTING  
.SUBRC.....=SUBRC  
.WORKFLOW_RESULT_IN.....=WORKFLOW_RESULT  
.APPLICATION_VARIABLE_IN....=APPLICATION_VARIABLE  
.IN_UPDATE_TASK_IN.....=IN_UPDATE_TASK  
.IMPORTING  
.WORKFLOW_RESULT_OUT.....=WORKFLOW_RESULT  
.APPLICATION_VARIABLE_OUT..=.APPLICATION_VARIABLE  
.IN_UPDATE_TASK_OUT.....=IN_UPDATE_TASK  
.TABLES  
.RETURN_VARIABLES.....=RETURN_VARIABLES.  
*>>>>>>>>>Customer.exit.3.(End) *<<<<<<<<<<<<<<<<<<<<<<<<<<<  
  
ENDFUNCTION.  
  
*-----*  
*FORM IDOC_PROCESS_XAMPLE3.....*  
*-----*  
*..This.routine.creates.an.application.document.based.on.the.IDoc's...*  
*..contents.Object_Number.contains.the.new.document's.number.....*  
*.If.an.error.occurs,.subrc.is.non-zero,.t_idoc_status.is.filled.....*  
*..Note:if.more.than.one.error.is.detected,.t_idoc_status.contains...*  
*..more.than.one.status.record.....*  
*-----*  
*-->.F_IDOC_CTRL...IDoc.control.record.....*  
*-->.T_IDOC_DATA....IDoc.data.records.....*  
*-<-.T_IDOC_STATUS...IDoc.status.records.....*  
*-<-.OBJECT_NUMBER...Created.document's.number.....*  
*-<-.SUBRC.....Return.code.....*  
*-----*  
FORM IDOC_PROCESS_XAMPLE3  
.....TABLES..T_IDOC_DATA.....STRUCTURE EDIDD  
.....T_IDOC_STATUS.....STRUCTURE BDIDOCSTAT  
.....USING...F_IDOC_CTRL.....STRUCTURE EDIDC  
.....CHANGING OBJECT_NUMBER.....LIKE XHEAD-DOCMNT_NO  
.....SUBRC.....LIKE SY-SUBRC.  
  
*.Internal.field.string.for.the.document.header.  
..DATA:F XHEAD.LIKE XHEAD.
```


Example Program for a Customer Exit

```

*Internal table for the document items.
*DATA:T_XITEM LIKE XITEM OCCURS 0 WITH HEADER LINE.

*Number given to the created document
*DATA:DOCUMENT_NUMBER LIKE F_XHEAD-DOCMNT_NO.

*Move the data in the IDoc to the internal structures/tables
*f_xhead and t_xitem.
*PERFORM IDOC_INTERPRET3 TABLES...T_IDOC_DATA
*...T_XITEM
*...T_IDOC_STATUS
*...USING...F_IDOC_CONTRL
*...CHANGING F_XHEAD
*...SUBRC.

*Create the application object if no error occurred so far.
*IF SUBRC=0.
*...This fictitious function module creates a new object based on the
*...data that was read from the IDoc. The new object's ID is returned
*...in the parameter 'document_number'.
*...The function module checks that the data is correct, and raises
*...an exception if an error is detected.
*...CALL FUNCTION 'XAMPLE_OBJECT_CREATE'
*...EXPORTING
*...XHEAD.....=F_XHEAD
*...IMPORTING
*...DOCUMENT_NUMBER=DOCUMENT_NUMBER
*...TABLES
*...XITEM.....=T_XITEM
*...EXCEPTIONS
*...OTHERS.....=1.

*...IF SY-SUBRC<>0.
*...SUBRC=1.
*...Put the error message into 't_idoc_status'
*...PERFORM STATUS_FILL_SY_ERROR
*...TABLES...T_IDOC_STATUS
*...USING...T_IDOC_DATA
*...SY
*...'"Field name
*...'"idoc_process_xample'.....'"Form routine

*...ELSE.
*...Fill the remaining export parameters
*...OBJECT_NUMBER=DOCUMENT_NUMBER.."New document's number

*...t_idoc_status-docnum=f_idoc_contrl-docnum.
*...t_idoc_status-status=c_idoc_status_ok.
*...t_idoc_status-msgty='S'.
*...t_idoc_status-msgid=your_msgid.."Global variable.

```

Example Program for a Customer Exit

```

.....t_idoc_status-msgno==msgno_success."Global variable.
.....t_idoc_status-msgv1==object_number.
.....APPEND T_IDOC_STATUS.
....ENDIF....."if sy-subrc<>0.
..ENDIF....."if subrc=0.

ENDFORM.

*-----*
*.....FORM IDOC_INTERPRET3.....*
*-----*
*..This routine checks that the correct message type is being used, ...*
*..then checks that the IDoc has not been overtaken (serialization), ...*
*..and then converts and moves the data from the IDoc segments to the ...*
*..internal structure f_xhead and internal table t_xitem.....*
*..If an error occurs, t_idoc_status is filled and subrc<>0.....*
*-----*
*..--> T_IDOC_STATUS.....*
*..--> T_XITEM.....*
*..--> F_IDOC_DATA.....*
*..--> F_XHEAD.....*
*..--> SUBRC.....*
*-----*
FORM IDOC_INTERPRET3 TABLES T_IDOC_DATA STRUCTURE EDIDD
.....T_XITEM STRUCTURE XITEM
.....T_IDOC_STATUS STRUCTURE BDIDOCSTAT
.....USING F_IDOC_CONTRL STRUCTURE EDIDC
.....CHANGING F_XHEAD STRUCTURE XHEAD
.....SUBRC LIKE SY-SUBRC.

*..Check that the IDoc contains the correct message type.
*..Note: if your message type is reducible, check field 'idoctp'
*.....(IDoc type) instead of 'mestyp'.
*..IF F_IDOC_CONTRL-MESTYP<>'XAMPLE'.

.....MESSAGE ID YOUR_MSGID....."Global variable
.....TYPE.....'E'
.....NUMBER MSGNO_WRONG_FUNCTION....."Global variable
.....WITH F_IDOC_CONTRL-MESTYP....."message type
.....'IDOC_INPUT_XAMPLE'....."Your function module.
.....F_IDOC_CONTRL-SNDPRT....."Sender partner type
.....F_IDOC_CONTRL-SNDPRN....."Sender number.
.....RAISING WRONG_FUNCTION_CALLED.

..ENDIF.

*..Loop through the IDoc's segments and convert the data from the IDoc
*..format to the internal format.
*..LOOP AT T_IDOC_DATA WHERE DOCNUM=F_IDOC_CONTRL-DOCNUM.

.....CASE T_IDOC_DATA-SEGNAM.

```


Mass Processing

Processing more than one IDoc at a time can improve throughput because:

- More than one IDoc is processed per COMMIT WORK
- The function can be coded to add multiple entries to a table with one update command ("array insert").

See also:

[Import Parameters \[Seite 133\]](#)

[Export Parameters \[Seite 134\]](#)

[Example Program for Mass Processing IDocs \[Seite 139\]](#)

Import Parameters

The import parameters have the same meaning as when Idocs are processed one at a time. The difference is that the table `Idoc_Contrl` can contain the control records of more than one IDoc, and `Idoc_Data` the data records of more than one IDoc.

Export Parameters

The export parameters need to be filled in much the same way as when processing one IDoc at a time. The difference is that some of the IDocs may be processed successfully, and some might have an error.

The following examples show how to fill the export parameters. In each case three IDocs, numbers 4711, 4712 and 4713, are passed to the inbound function module. In the first case all IDocs are processed successfully, and application documents number 1234, 1235 and 1236 are created respectively. In the second case, the first and third IDoc are processed successfully, but the second IDoc causes an error.



[All Inbound IDocs Processed Successfully \[Seite 135\]](#)

[Error in One Inbound IDoc \[Seite 137\]](#)

All Inbound IDocs Processed Successfully

If all IDocs were processed successfully, the export parameters should be filled as follows:

How to fill the export parameters when processing packets of IDocs, when all the IDocs were successfully processed. IDoc numbers 4711 - 4713 created application object numbers 1234 - 1236.

Parameter	Value	
IN_UPDATE_TASK	" " Update task not used "X" Update task used	
CALL_TRANSACTION_DONE	" "	
WORKFLOW_RESULT	"0"	
APPLICATION_VARIABLE	" " (e.g. initial value)	
IDOC_STATUS	The table must contain three records with fields containing:	
	Docnum	Status
	4711	53
	4712	53
	4713	53
	Optionally the fields Msgid etc. can be filled containing the application's success message.	
RETURN_VARIABLES	The table must contain the following six entries:	
	WF_PARAM	Doc_Number
	PROCESSED_IDOCS	4711
	APPL_OBJECTS	1234
	PROCESSED_IDOCS	4712
	APPL_OBJECTS	1235
	PROCESSED_IDOCS	4713
	APPL_OBJECTS	1236

All Inbound IDocs Processed Successfully

	If processing the inbound IDoc does not create or change an application object, the "Appl_Objects" entries can be omitted - they make no sense without a document number.
SERIALIZATION _INFO	Empty if not using serialization.

Error in One Inbound IDoc

If one of the three IDocs causes an error, the export parameters should be filled as follows:

How to fill the export parameters when processing packets of IDocs, when only IDoc no. 4712 contains an error. IDoc numbers 4711 and 4713 created application objects number 1234 and 1235.

Parameter	Value	
IN_UPDATE_TASK	" " (i.e. initial value) - Update task not used	
CALL_TRANSACTION_DONE	" " (e.g. initial value)	
WORKFLOW_RESULT	"99999"	
APPLICATION_VARIABLE	" " (e.g. initial value)	
IDOC_STATUS	The table must contain three records with fields containing:	
	Docnum	Status
	4711	53
	4712	51
	4713	53
	The Msgid etc. fields of the status record for IDoc 4712 must contain the error message.	
RETURN_VARIABLES	The table must contain the following five entries:	
	WF_PARAM	Doc_Number
	PROCESSED_IDOCS	4711
	APPL_OBJECTS	1234
	ERROR_IDOCS	4712
	PROCESSED_IDOCS	4713
	APPL_OBJECTS	1235
	If processing the inbound IDoc does not create or change an application object, the "Appl_Objects" entries can be omitted - they make no sense without a document number.	

Error in One Inbound IDoc

SERIALIZATION _INFO	Empty if not using serialization.
------------------------	-----------------------------------

Example Program for Mass Processing IDocs

```

FUNCTION IDOC_INPUT_XAMPLE4.
* "-----
* "
* "Local interface:
* ".....IMPORTING
* ".....VALUE (INPUT_METHOD) LIKE BDWFAP_PAR-INPUTMETHD
* ".....VALUE (MASS_PROCESSING) LIKE BDWFAP_PAR-MASS_PROC
* ".....EXPORTING
* ".....VALUE (WORKFLOW_RESULT) LIKE BDWF_PARAM-RESULT
* ".....VALUE (APPLICATION_VARIABLE) LIKE BDWF_PARAM-APPL_VAR
* ".....VALUE (IN_UPDATE_TASK) LIKE BDWFAP_PAR-UPDATETASK
* ".....VALUE (CALL_TRANSACTION_DONE) LIKE BDWFAP_PAR-CALLTRANS
* ".....TABLES
* ".....IDOC_CONTRL STRUCTURE EDIDC
* ".....IDOC_DATA STRUCTURE EDIDD
* ".....IDOC_STATUS STRUCTURE BDIDOCSTAT
* ".....RETURN_VARIABLES STRUCTURE BDWFRETVAR
* ".....SERIALIZATION_INFO STRUCTURE BDI_SER
* ".....EXCEPTIONS
* ".....WRONG_FUNCTION_CALLED
* "-----

* .-----
* .-----05 July 1996-----
* .-----

* Example function module for processing inbound IDocs for ALE or EDI.
* This example applies for processing
*
* ...with ... mass processing (more than one IDoc at a time)
*
* ...without ... serialization
* .....customer-exits
* .....calling an ALE-enabled transaction

* .-----Naming conventions-----
* Internal tables start with 't_'
* Internal field strings start with 'f_'
* .-----

* >> The following line must appear in the global part of your
* >> function group:
* ....include mbdconwf. .... "Report containing the ALE constants.
* The ALE constants start with 'c_'.

* Internal table for the document headers.
* DATA: T_XHEAD LIKE XHEAD OCCURS 0 WITH HEADER LINE.

```

Example Program for Mass Processing IDocs

```

* Internal table for the document items.
** DATA: T_XITEM LIKE XITEM OCCURS 0 WITH HEADER LINE.

** DATA: SUBRC LIKE SY-SUBRC,
** ..... OBJECT_NUMBER LIKE XHEAD-DOCMNT_NO.

* Initialize variables
** SUBRC := 0.

* Fill the ALE export parameters prior to loop through IDocs.
** CLEAR IN_UPDATE_TASK.
** CLEAR CALL_TRANSACTION_DONE. .... "Call Transaction is not used.
** WORKFLOW_RESULT := C_WF_RESULT_OK.

* Loop through the IDocs' control records
** LOOP AT IDOC_CONTRL.

*** Process the IDoc and check the data; no database updates!
*** PERFORM IDOC_PROCESS_XAMPLE4 TABLES IDOC_DATA
*** ..... IDOC_STATUS
*** ..... t_xhead
*** ..... t_xitem
*** ..... USING IDOC_CONTRL
*** ..... CHANGING OBJECT_NUMBER
*** ..... SUBRC.

*** Fill the ALE export parameters.
*** IF SUBRC <> 0. .... "Error occurred

*** WORKFLOW_RESULT := C_WF_RESULT_ERROR.
*** RETURN_VARIABLES-WF_PARAM := C_WF_PAR_ERROR_IDOCS.
*** RETURN_VARIABLES-DOC_NUMBER := IDOC_CONTRL-DOCNUM.
*** APPEND RETURN_VARIABLES.

** ELSE. .... "IDoc processed successfully

*** RETURN_VARIABLES-WF_PARAM := C_WF_PAR_PROCESSED_IDOCS.
*** RETURN_VARIABLES-DOC_NUMBER := IDOC_CONTRL-DOCNUM.
*** APPEND RETURN_VARIABLES.
*** RETURN_VARIABLES-WF_PARAM := C_WF_PAR_APPL_OBJECTS.
*** RETURN_VARIABLES-DOC_NUMBER := OBJECT_NUMBER.
*** APPEND RETURN_VARIABLES.

** ENDIF.

** ENDLOOP. .... "loop at idoc_contrl.

* Once all IDocs have been processed, insert the application data to
* the database (as long as there is some data to insert).

** read table t_xitem index 1.
** if sy-subrc = 0. .... "i.e. at least one entry

```

Example Program for Mass Processing IDocs

```

*...This fictitious function module inserts the data in tables
*...t_xhead and t_xitem to the database tables xhead and xitem.
*...It has no exceptions, because a failed insert leads to a run-time
*...error.
*...CALL FUNCTION 'XAMPLE_OBJECTS_INSERT_TO_DATABASE'
*...TABLES
*...XHEAD = T_XHEAD
*...XITEM = T_XITEM.

..endif....."if sy-subrc = 0.

ENDFUNCTION.

*-----*
*...FORM IDOC_PROCESS_XAMPLE4.....*
*-----*
*...This routine adds an application document to tables t_xhead and...*
*...t_xitem based on the IDoc's contents.....*
*...Object_Number contains the new document's number.....*
*...If an error occurs, subrc is non-zero, t_idoc_status is filled.....*
*...Note: if more than one error is detected, t_idoc_status contains...*
*...more than one status record.....*
*-----*
*...--> F_IDOC_CONTRL...IDoc control record.....*
*...--> T_IDOC_DATA...IDoc data records.....*
*...<-- T_XHEAD...Application document's header records.....*
*...<-- T_XITEM...Application document's line item records.....*
*...<-- T_IDOC_STATUS...IDoc status records.....*
*...<-- OBJECT_NUMBER...Created document's number.....*
*...<-- SUBRC...Return code.....*
*-----*
FORM IDOC_PROCESS_XAMPLE4
.....TABLES T_IDOC_DATA...STRUCTURE EDIDD
.....T_IDOC_STATUS...STRUCTURE BDIDOCSTAT
.....T_XHEAD...STRUCTURE XHEAD
.....T_XITEM...STRUCTURE XITEM
.....USING F_IDOC_CONTRL...STRUCTURE EDIDC
.....CHANGING OBJECT_NUMBER...LIKE XHEAD-DOCMNT_NO
.....SUBRC...LIKE SY-SUBRC.

*...Internal table string for the document headers.
*...DATA: F_XHEAD LIKE XHEAD OCCURS 0 WITH HEADER LINE.

*...Internal table for one document's items.
*...DATA: T_ONE_XITEM LIKE XITEM OCCURS 0 WITH HEADER LINE.

*...Number given to the created document
*...DATA: DOCUMENT_NUMBER LIKE XHEAD-DOCMNT_NO.

*...Move the data in the IDoc to the internal structures/tables
*...f_xhead and t_xitem.

```

Example Program for Mass Processing IDocs

```

..PERFORM IDOC_INTERPRET TABLES...T_IDOC_DATA
.....T_ONE_XITEM
.....T_IDOC_STATUS
.....USING...F_IDOC_CONTRL
.....CHANGING F_XHEAD
.....SUBRC.

*Create the application object if no error occurred so far.
..IF SUBRC.=0.
*...This fictitious function module checks the new object based on the
*...data that was read from the IDoc.
*...If the checks succeed, the new object's ID is returned in the
*...parameter 'document_number'.
*...If the checks fail, an exception is raised.
*...Note: this function must not insert or modify database records!
....CALL FUNCTION 'XAMPLE_OBJECT_CHECK'
.....EXPORTING
.....XHEAD.....=F_XHEAD
.....IMPORTING
.....DOCUMENT_NUMBER.=DOCUMENT_NUMBER
.....TABLES
.....XITEM.....=T_ONE_XITEM
.....EXCEPTIONS
.....OTHERS.....=1.

....IF SY-SUBRC.<>0.
.....SUBRC.=1.
*.....Put the error message into 't_idoc_status'
*.....PERFORM STATUS_FILL_SY_ERROR
*.....TABLES...T_IDOC_STATUS
*.....USING...T_IDOC_DATA
*.....SY
*.....'.....'Field name
*.....'idoc_process_xample'.....'Form routine

....ELSE.
*.....Fill the remaining export parameters
*.....OBJECT_NUMBER.=DOCUMENT_NUMBER..New document's number
*.....append f_xhead to t_xhead.
*.....APPEND LINES OF T_ONE_XITEM TO T_XITEM.

*.....t_idoc_status-docnum.=f_idoc_contrl-docnum.
*.....t_idoc_status-status.=c_idoc_status_ok.
*.....t_idoc_status-msgty.='S'.
*.....t_idoc_status-msgid.=your_msgid..Global variable.
*.....t_idoc_status-msgno.=msgno_success..Global variable.
*.....t_idoc_status-msgv1.=object_number.
*.....APPEND T_IDOC_STATUS.
*.....ENDIF....."if sy-subrc.<>0.
..ENDIF....."if subrc.=0.

ENDFORM.

```


Using Call Transaction

In general, you should try to use function modules to update the database rather than resorting to a Call Transaction; a Call Transaction has a significant performance overhead. The advantage of using a Call Transaction is that, if an error occurs, the user can reprocess the IDoc by going through the transaction's input screens.

Implementing a 'one IDoc at a time' function module as described above, but where the function module uses a Call Transaction to post the application data to the database, results in the application data being posted in a different logical unit of work from that in which the IDoc status data is posted - a Call Transaction does a 'commit work' when successfully executed. (See the above section on 'Ensuring one logical unit of work').

The solution to this problem is to modify the transaction so that it updates the IDoc status data when the application data is updated, i.e. when the IDoc is successfully processed. Note: this only applies to successfully processed IDocs; errors are handled in the same way as before.

See also:

[ALE -Enabled Transactions \[Seite 145\]](#)

[Call Transaction Successful \[Seite 147\]](#)

[Call Transaction Failed \[Seite 149\]](#)

[Import Parameters in CALL TRANSACTION \[Seite 150\]](#)

[Export Parameters in CALL TRANSACTION \[Seite 151\]](#)

ALE-Enabled Transactions

A transaction is 'ALE-enabled' when the following two prerequisites are met:

- At the beginning of the first screen's PBO module, it must read the IDoc number from a memory variable and call the ALE function module **Idoc_Input_Open**, passing it the IDoc's number (Parameter *Document_Number*).
- Before the data is updated to the database, the transaction must call the ALE function module **Idoc_Input_Close**. This function module must **not** be called from within a function module that is processed in update task, i. e. "Call Function "xxx" In Update Task". Instead, it should be called directly before or after the other update function modules "xxx". This is because it accesses global data written by the function module **Idoc_Input_Open**.

Idoc_Input_Close will update the IDoc's status in the same update task if the interface parameter *In_Update_Task* is set to "X".

Example of coding to update database:

- call function "UPDATE_APPL_TABLES " in update task tables...
- call function "IDOC_INPUT_CLOSE" exporting...
- commit work.

The parameters of the function module **Idoc_Input_Close** must be filled as follows:

How to Fill the Import and Table Parameters of the Function Module IDOC_INPUT_CLOSE.

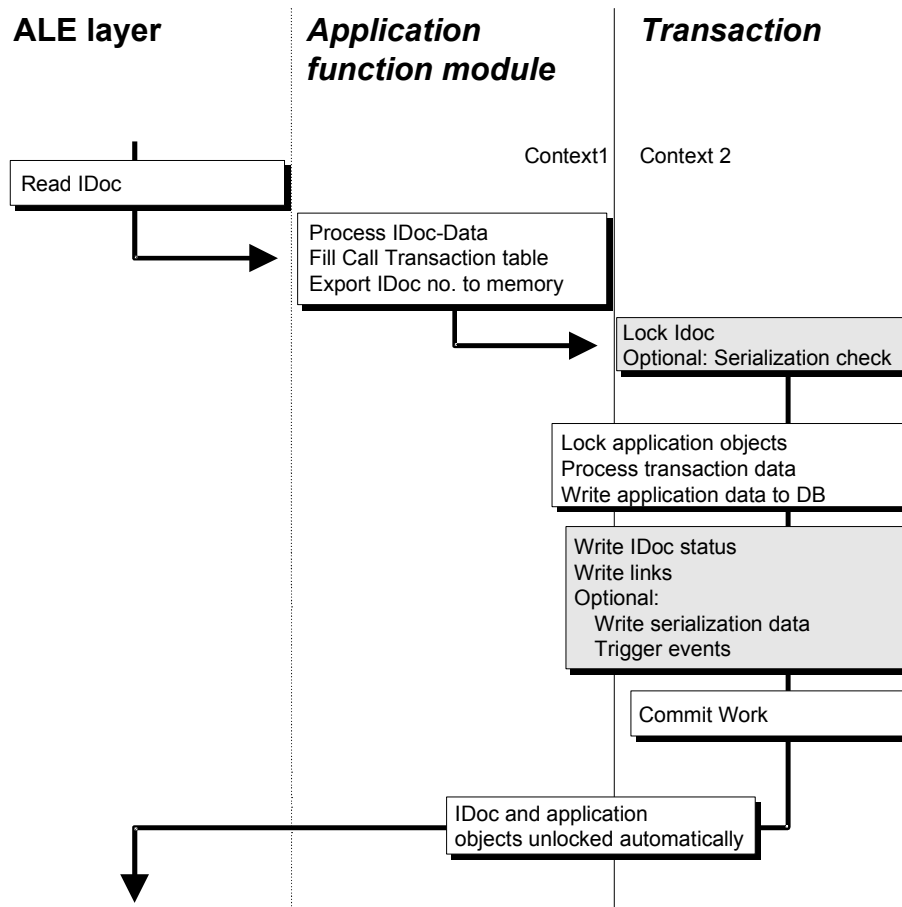
Parameter	Value	
WORKFLOW_RESULT	"0"	
APPLICATION_VARIABLE	" " " " (i.e. initial value)	
IN_UPDATE_TASK	" " <i>Update Task</i> not used by the transaction 'X' <i>Update Task</i> used by the transaction	
IDOC_CONTROL	The contents of Idoc_Input_Open's export parameter <i>Idoc_Control</i>	
IDOC_STATUS	The table must contain one record with fields containing: Docnum: 4711 Status: 53 Optionally the fields Msgid etc. can be filled containing the application's success message.	
RETURN_VARIABLES	The table must contain the following two entries:	
	WF_PARAM	Doc_Number
	PROCESSED_IDOCS	4711

ALE-Enabled Transactions

	APPL_OBJECTS	1234
	If processing the inbound IDoc does not create or change an application object, the "Appl_Objects" entry can be omitted - it makes no sense without a document number.	
SERIALIZATION_INFO	The contents of Idoc_Input_Open's table parameter SERIALIZATION_INFO.	

Call Transaction Succeeds

The inbound function module that uses an ALE enabled transaction must pass the IDoc number to the transaction's IDoc memory variable before calling the transaction.



Inbound Processing with ALE-Enabled Transaction: Transaction Successful.



- The transaction's code is executed in a separate context from that of the ALE layer and inbound function module;
- The shaded boxes represent the ALE function modules `Idoc_Input_Open` (above) and `Idoc_Input_Close` (below).

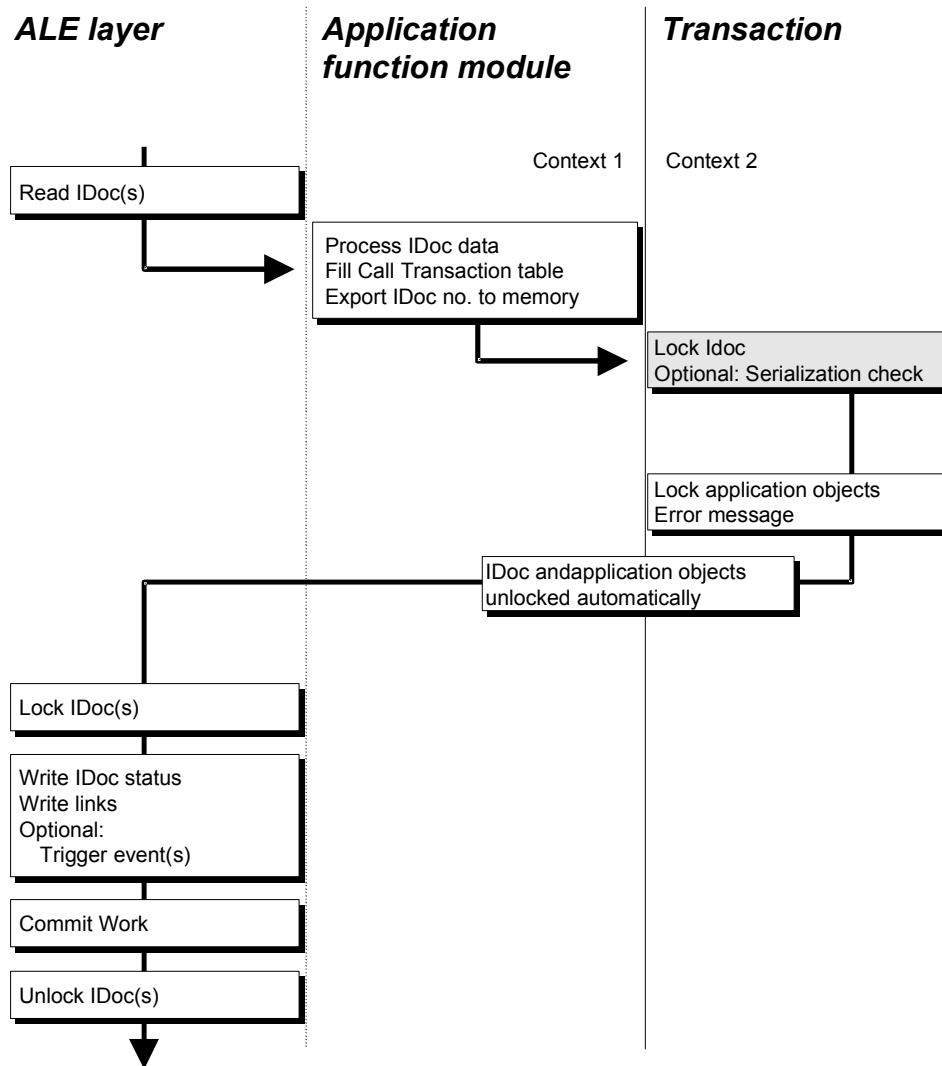
Call Transaction Succeeds**How can you tell whether the Call Transaction has succeeded?**

On the face of it, you would think that the Call Transaction has succeeded if "Sy-Subrc = 0" after the call. In an ALE environment this is only half the story, because when the import parameter *Input_Method* takes on the value "A" or "E", the inbound function module must call the transaction using *Imode* = "A" or "E" ("show all screens" or "show the screens starting with the one where the error occurred"). In this case, the user sees the screens and can cancel the transaction using OK-code "/n", **which also leads to "Sy-Subrc = 0" after the call!** Hence the only reliable way to tell whether the Call Transaction succeeded or not is to check the message ID (Sy-Msgid) and number (Sy-Msgno). Beware that some transactions have more than one "success" message.

Call Transaction Fails

The following graphic shows in detail how the ALE layer, inbound function module and transaction interact when the Call Transaction fails, i.e. when "Sy-Subrc" <> 0 or "Sy-Subrc" = 0 and a non-success message was returned.

Inbound Processing with ALE-enabled Transaction: Transaction Failed



Import Parameters in CALL TRANSACTION

The import parameters have the same meaning as those used when individual IDocs are processed.

Export Parameters in CALL TRANSACTION

See also:

[Inbound Processing Successful \[Seite 152\]](#)

[Error in Inbound Processing \[Seite 153\]](#)

Inbound Processing Successful

Inbound Processing Successful

If the Call Transaction was successful, the inbound function module's export parameters need to be filled as follows:

How to fill the export parameters when processing one IDoc at a time via an ALE enabled Call Transaction, when the IDoc was successfully processed.

Parameter	Value
IN_UPDATE_TASK	" " (e.g. initial value)
CALL_TRANSACTION_DONE	"X"
WORKFLOW_RESULT	"0"
APPLICATION_VARIABLE	" " (e.g. initial value)
IDOC_STATUS	Empty
RETURN_VARIABLES	Empty
SERIALIZATION_INFO	Empty

Error During Inbound Processing

If the Call Transaction failed, the export parameters must be filled in exactly the same way as when processing one IDoc at a time:

How to fill the export parameters when processing one IDoc at a time via an ALE enabled Call Transaction, when an error occurred during inbound processing.

Parameter	Value	
IN_UPDATE_TASK	" " (i.e. initial value) - Update task not used	
CALL_TRANSACTION_DONE	" " (e.g. initial value)	
WORKFLOW_RESULT	"99999"	
APPLICATION_VARIABLE	" " (i.e. initial value)	
IDOC_STATUS	The table must contain one record with fields containing: Docum: 4711 Status: 51 Msgid, Msgno etc. must be filled with the error message's ID, number etc.	
RETURN_VARIABLES	The table must contain the following entry:	
	WF_PARAM	Doc_Number
	"ERROR_IDOCS"	4711
SERIALIZATION_INFO	Empty	

ALE Settings

ALE Settings

In order to ensure that the inbound functional module is called via ALE, a number of ALE settings (table entries) need to be made. To get to ALE outbound processing settings choose *Tools* → *Business Framework* → *ALE* → *Development*.

The graphic below gives an overview of the relationships between the settings. Each row represents a field; the row "Error events" represents five fields. Only the relevant fields of each table are listed. The columns represent the following (bold entries represent primary key fields):

- The fields that are read from an inbound IDoc's control record
- The six IDoc fields that determine the process code to be used
- The attributes associated with the process code
- The attributes associated with the inbound function module
- The fields used to check that the function module is registered in ALE

The Relationships Between the Fields in IDoc Control Record and the Tables Containing ALE Inbound Settings

IDoc control record	Partner Profile inbound	Process code tables (inbound)	Function module's attributes	ALE function module registry
IDoc type				→ IDoc type
Send. partn. type	→ Sender type			
Sender partner no.	→ Sender no.			
Message type	→ Message type			→ Message type
Message variant	→ Message var.			→ Message var.
Message function	→ Message func.			→ Message func.
Test flag	→ Fest flag			
	Process code	→ Process code		
		Process type		
		Error events		
		Appl.obj.type		→ Appl. obj. type
		Inbound func.	→ Inbound func.	→ Inbound func.
			Input type	
			Dialog?	

See also:

- [Declaring the Function Module's Attributes \[Seite 155\]](#)
- [Registering the Function Module in Inbound Processing \[Seite 156\]](#)
- [Creating an Inbound Processing Code \[Seite 157\]](#)

Declaring the Function Module's Attributes

The inbound function module has two ALE attributes: "Dialog possible?" and "Input type".

Dialog possible?

If the function module supports a Call Transaction, it can be programmed to show the user the input screens as described above. This ability must be declared here, otherwise the user will not be given the option of choosing to display the screens.

Inbound type

There are three types of function module in inbound processing:

1. Those that support mass processing
2. Those that can only process one IDoc at a time and **do not** use an ALE-enabled transaction
3. Those that can only process one IDoc at a time and that use an ALE-enabled transaction

For the latter two types, the ALE layer splits up an incoming packet of IDocs and calls the function module once for each IDoc. The latter two need to be distinguished from one another because the ALE processing before the function is called differs in each case.



For an example, look at the entry for the inbound function module IDOC_INPUT_MATMAS01 used for material master data.

Registering the Function Modules in Inbound Processing

ALE keeps track of which message type, variant and function, IDoc type and application object types are applicable for a given inbound function module. Before a function module can be used in inbound processing, it must be registered here.

If the function module is used for processing master data, only the entry for the "reference" message type is needed here; entries are automatically generated when the "reference" message type is "reduced".

For an example, look at the entry for the inbound function module IDOC_INPUT_MATMAS01 and reference message type MATMAS used for material master data.

Creating an Inbound Processing Code

The inbound process code can be thought of as the name given to the ways and means of processing an incoming IDoc, i.e. the process code's attributes. A process code has the following attributes:

- Processing type (e.g. should the inbound function module be called immediately, or should a Workflow or work item be started?);
 - Standard: "ALE, function module called directly"
- Name of inbound function module;
- Error processing attributes (objects and events; see the section on objects etc. for error handling);
- Application object type used for ALE links;
 - the object type created or changed by the inbound function module. Example: An inbound ORDERS IDoc, containing a customer's purchase order, creates a customer order in the receiving R/3 System. Here the application object type is BUS2032, the object type for customer orders in the BOR (Business Object Repository).
- The application event to be triggered (dealt with in the section on advanced techniques) is not generally used.



For an example, have a look at the process code MATM used for material master data.

Naming Convention

The naming convention for process codes is to use the first four letters of the message type to which it applies. Example: MATM for message type MATMAS; if there is a conflict, use the first three letters and a number, e.g. MAT1.

In our case, the process code would be XAMP.

Inbound Processing Using SAP Workflow

This section shows how to use Workflow to process inbound IDocs, rather than using the direct input via function call.

We recommend using direct input via function call rather than Workflow for inbound processing because of the performance overhead of invoking Workflow for each inbound IDoc or IDoc packet.

Using Workflow makes sense when each IDoc of a given message type and/or from a given sender needs to be manually processed, or the subsequently created application object needs to be manually processed, but only when created from an IDoc. An example is with the message type ORDCHG, "sales order change"; whereby a company might decide that sales order changes from some customers (senders) need to be reviewed by a person before they are posted.



Inbound function modules implemented as described above support inbound processing via workflow in addition to error handling via workflow. No changes are required.

The two subsections deal with the two different ways of processing inbound IDocs with workflow: Work items and Workflow.

See also:

[Work Items \[Seite 159\]](#)

[Workflow \[Seite 160\]](#)

Work Items

You can define your inbound process code to start a specified work item, rather than using the direct function call. The object that is passed to the work item's container is the IDoc object (e.g. an object of type IDOCMATMAS).

Note: if a packet of IDocs containing more than one IDoc is to be processed, only the first IDoc will be passed to the work item. The other IDocs will not be processed.

Workflow

Workflow

You can define your inbound process code to start a specified Workflow, rather than using the direct function call. The Workflow's container must contain the following two import parameters:

Mandatory Import Parameters in the Workflow Container

Parameter name	Type	Multi-line	Reference (object type/table field)
IDOC_PACKET	Object	No	IDOCXAMPLE or IDPKXAMPLE
UNPROCESSED_IDOCS	Variable	Yes	Edidc-Docnum

IDOC_PACKET is filled with an object of the type as defined in the workflow container (this means that ALE reads the workflow container definition to determine the type); the object's ID is the number of the first IDoc in the packet.

Unprocessed_IDOCS is a list of the IDoc numbers in the packet.

See also:

[IDOCXAMPLE as a Reference for IDOC_PACKET \[Seite 161\]](#)

[IDPKXAMPLE as a Reference for IDOC_PACKET \[Seite 162\]](#)

IDOCXAMPLE as a Reference for IDOC_PACKET

If you want to use the IDoc object type's methods `InputForeground` and `InputBackground`, you need to use your IDoc object type (IDOCXAMPLE) as a reference for IDOC_PACKET.

Doing this means that you must ensure that only one IDoc at a time is passed to ALE, i.e. packets of size 1. The reason is simple: the methods `InputForeground` and `InputBackground` deal with a single IDoc object, so you need to define the binding so that it can deal with the object contained in IDOC_PACKET. The methods do not use the variable `Unprocessed_IDOCs`, and hence are unaware of any other IDocs in the packet.

IDPKXAMPLE as a Reference for IDOC_PACKET

If you want to use the IDoc packet object type's (IDPKXAMPLE) methods `MassInput` or `Display`, you need to use IDPKXAMPLE as a reference for IDOC_PACKET.

Both the methods `MassInput` and `Display` have an importing variable `Unprocessed_IDOCs`, which needs to be bound to the variable `Unprocessed_IDOCs` in the workflow container. This allows all the IDocs in the packet to be processed.

Typically it will only make sense to use IDPKXAMPLE if you also define your own methods to process the IDoc packet.

Advanced Workflow Programming

See also:

[Setting the Parameter RESULT in the event container \[Seite 164\]](#)

[Triggering an Application Event After Successful IDoc Processing \[Seite 168\]](#)

[Using the Parameter NO_OF_RETRIES \[Seite 170\]](#)

Setting the Parameter RESULT in the Event Container

Both IDoc-object-events inputErrorOccurred and inputFinished contain the parameter RESULT in their containers. Its value is affected by:

See also:

[Event inputErrorOccurred \[Seite 165\]](#)

[Event inputFinished \[Seite 167\]](#)

Event inputErrorOccurred

ALE will trigger the event inputErrorOccurred when the inbound function module's import parameter MASS_PROCESSING is set to "X". In this case ALE is treating the inbound IDoc(s) as a packet, even if it only contains one IDoc.

Since some IDocs in a packet could be processed successfully while others fail, the export parameter WORKFLOW_RESULT cannot be used to set the container parameter RESULT. Instead, the parameter takes on different values according to the parameters used in the table Return_Variables:

How to Set the Parameter RESULT when MASS_PROCESSING = "X". Note that for a given IDoc number (in field Doc_Number), you should only use one of the above three names. The names are case sensitive.

Wf_param	Value of RESULT in event container
ERROR_IDOCS	99999
RETRY_IDOCS	1
CONTINUE_IDOCS	2

You could use these three values as follows: inputErrorOccurred could be used to trigger a Workflow, which branches according to the value of the parameter RESULT:

Possible Values of RESULT in a Workflow

RESULTValue	Action defined in Workflow
1	Retry the IDoc n minutes later, using a new task using the method InputBackground. This could be used to retry IDocs that failed because an application was temporarily locked by another user or process.
2	Process the IDoc in some other manner.
99999	Error handling, using the standard/customer task

Export parameter values when processing packets of IDocs, when IDoc 4711 was successfully processed, creating application object no 1234; IDoc 4712 caused an error for which the event's parameter RESULT = 99999; IDoc 4713 caused an error for which RESULT = 2; IDoc 4714 caused an error for which RESULT = 1.

Parameter	Value
IN_UPDATE_TASK	" " (i.e. initial value) - Update task not used
CALL_TRANSACTION_DONE	" " (e.g. initial value)
WORKFLOW_RESULT	"99999"

Event inputErrorOccurred

APPLICATION_VARIABLE	" " (e.g. initial value)	
IDOC_STATUS	The table must contain four records with fields containing:	
	Docnum	Status
	4711	53
	4712	51
	4713	51
	4714	51
	The Msgid etc. fields of the status record for IDocs 4712, 4713 and 4714 must contain the error message.	
RETURN_VARIABLES	The table must contain the following five entries:	
	Wf_param	Doc_Number
	PROCESSED_IDOCS	4711
	APPL_OBJECTS	1234
	ERROR_IDOCS	4712
	CONTINUE_IDOCS	4713
	RETRY_IDOCS	4714
	If processing the inbound IDoc does not create or change an application object, the "Appl_Objects" entry can be omitted - it makes no sense without a document number.	
SERIALIZATION_INFO	Empty if not using serialization	

Event inputFinished

ALE triggers the event inputFinished when the inbound function module's import parameter MASS_PROCESSING is set to " ", that is, set to initial.

If the method InputForeground is used, the event will only be triggered if the IDoc passed to the inbound function module was successfully processed, i.e. ends up with status 53, or it was marked for deletion or had already been processed and the user marks the work item as being completed (via the IDoc menu).

If the method InputBackground is used, the event will always be triggered.

The parameter RESULT takes on the value of the inbound function module's export parameter WORKFLOW_RESULT. By convention the values 0, 1, 2 and 99999 should be used as with the event inputErrorOccurred.

The following table summarizes the values for which conventions exist. Other values can be used as needed, starting with 3.

Possible Values of RESULT in the IDoc container for object event inputFinished

RESULT Value	Description
0	IDoc successfully processed
1	IDoc not successfully processed; retry later
2	IDoc not successfully processed; application-specific workflow action should be started as next step
99997	Set by ALE: IDoc has already been processed; mark work item as completed
99998	Set by ALE: IDoc marked for deletion
99999	IDoc not successfully processed

Triggering an Application Event After Successful IDoc Processing

Triggering an Application Event After Successful IDoc Processing

Any number of events can be triggered within the inbound function module by calling the Workflow "create event" function module; doing this gives you full control of the event's parameters.

If you simply want to have an event triggered for your application object with a single parameter RESULT in its container, you can let ALE do this for you. An example where this is used is the message type EDLNOT, process code EDLN.

To let ALE trigger the event you need to

- Fill the field "application event" for your process code's inbound methods
- Enter a value in the inbound function module's table Return_Variables, where Wf_Param is filled as outlined in the following table, and Doc_Number is the same as your application object ID
- Set the parameter WORKFLOW_RESULT to a value not equal to zero. Customers can use alphanumeric values beginning with Y or Z. In the example below the value 3 is used.

Triggering your application object event using ALE and Specifying a Value in the RESULT Parameter.

Wf_Param	Doc_Number	Parameter RESULT in event container
CONTINUE_OBJECTS1	1234	1
CONTINUE_OBJECTS2	1234	2
CONTINUE_OBJECTS3	1234	3
CONTINUE_OBJECTS4	1234	4
CONTINUE_OBJECTS5	1234	5

How to let ALE trigger one of your application object's events, and how to affect the value of the parameter RESULT in its container. How to fill the export parameters when processing packets of IDocs, when all the IDocs were successfully processed and an application object event should be triggered for each successfully processed IDoc. IDoc numbers 4711 and 4712 created application objects 1234 and 1235. In both cases the RESULT parameter in the event container is set to 1.

Parameter	Value
IN_UPDATE_TASK	" " Update task not used "X" Update task used
CALL_TRANSACTION_DONE	" "
WORKFLOW_RESULT	"3"
APPLICATION_VARIABLE	" " (e.g. initial value)

Triggering an Application Event After Successful IDoc Processing

IDOC_STATUS	The table must contain three records with fields containing:	
	Docnum	Status
	4711	53
	4712	53
	Optionally the fields Msgid etc. can be filled containing the application's success message.	
RETURN_VARIABLES	The table must contain the following six entries:	
	Wf_param	Doc_Number
	PROCESSED_IDOCS	4711
	APPL_OBJECTS	1234
	CONTINUE_OBJECTS1	1234
	PROCESSED_IDOCS	4712
	CONTINUE_OBJECTS1	1235
SERIALIZATION_INFO	Empty if not using serialization	

Using the Parameter NO_OF_RETRIES

If you want to implement a Workflow that allows you to reprocess an IDoc a fixed number of times before an error-handling work item is started, for example for those errors that are temporary, for example, caused by an object being locked, ALE provides a means of keeping track of how often the IDoc has been processed. ALE allows customers to specify the number of retries individually.

The object IDOCAPPL's (and hence all child object type's) method InputBackground has an import parameter NO_OF_RETRIES which is set to zero by ALE when an IDoc is first processed. The method processes this parameter as follows:

- If it is larger than the maximum number of retries set in the inbound process code's input methods, the parameter RESULT in the event inputFinished's container is set to "99999".
- Otherwise, it is incremented by one and written to the parameter NO_OF_RETRIES in the event inputFinished.

To make use of this feature, you need to define your Workflow so that it has a parameter NO_OF_RETRIES in its container which is bound to the import and to the event parameter NO_OF_RETRIES, and so that it loops back to the InputBackground step unless the event parameter RESULT = "99999".

Master Data Distribution

Master data distribution using an asynchronous IDoc interface consists of three steps:

1. [Defining the Message \[Seite 172\]](#) by specifying the message type and IDoc type.
2. Developing the program or function module that creates the IDoc from the application object and dispatches it via the ALE interface ([Outbound Processing \[Seite 173\]](#))
3. Developing the function module for IDoc processing on the receiver side ([Inbound Processing \[Seite 179\]](#)).

Defining the Message

Defining the Message

For general rules on defining a new message (message type, IDoc type), refer to the documentation "Guidelines for Designing IDoc Types and IDoc Segments". When you define your new message, note these specific points for master data distribution:

- Define the segment contents and segment hierarchy in the IDoc type according to the logical hierarchy of the data in the master data object, which normally matches the database table hierarchy for the master data object in the SAP system.



The material master consists of tables MARA, MAKT, MARC, MARD etc. The contents of IDoc segments E1MARAM, E1MAKTM, E1MARCM and E1MARDM in IDoc type MATMAS02 for the material master correspond to the most extent to the tables listed above. The hierarchy of these segments in IDoc type MATMAS02 corresponds to the database table hierarchy:

E1MARAM: Material master general data (MARA)

E1MAKTM: Material master short texts (MAKT)

E1MARCM: Material master C segment (MARC)

E1MARDM: Material master warehouse/batch segment (MARD)

- In each IDoc segment, define field MSGFN as the first field with data element MSGFN. Information is transmitted to field MSGFN on whether segment data is to be created, changed, deleted or updated in the target system.



As an example, refer to the definition of the segments E1MARAM, E1MAKTM, E1MARCM, and E1MARDM for the IDoc type MATMAS02 (material master).

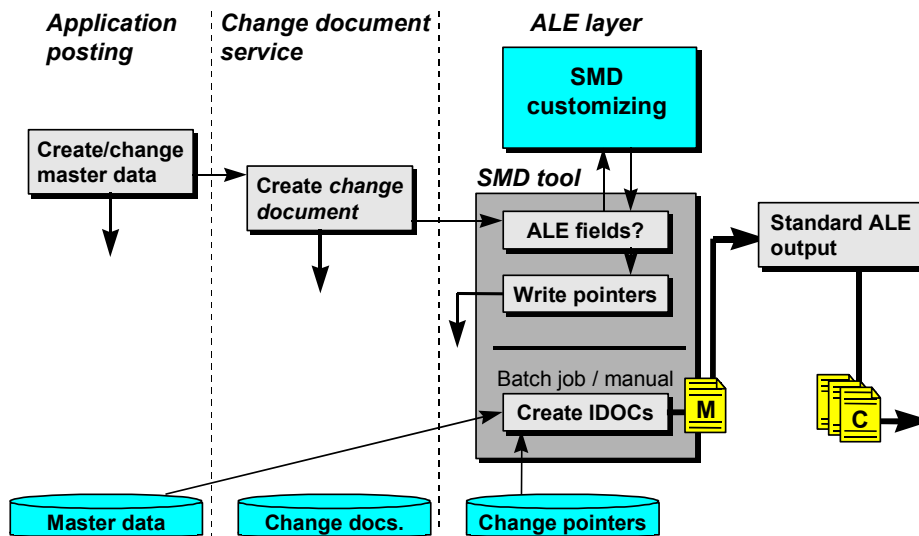
We recommend that you name the IDoc segments as follows: E1XXXXX, where XXXXX is the name of the corresponding database table. If the name of the table has less than five places, fill the remaining places with the letter M (For example, E1MARAM for table MARA).

Processing Outbound Master Data

There are two procedures for distributing master data: In both cases, an IDoc receives data for only one master data object.

- Master data objects are sent directly.
With this function, the dispatch of master data objects is triggered explicitly. The IDocs are filled with complete data for master data objects.
- Master data is distributed using the SMD tool (Shared Master Data tool).
In this case, changes to master data objects in the R/3 System are first logged in the form of "change pointers". The writing of change pointers is directly connected to the change document interface. The change pointers can then be evaluated, and the IDocs for the changed master data objects can be set up and dispatched. Only changed data is set up in the IDoc, passed to the ALE layer and then dispatched to other systems. The diagram below shows each individual step for distributing master data using the SMD tool.

Steps for Distributing Master Data Using the SMD Tool



Distributing Master Data Using the SMD Tool

The *Shared Master Data Tool* (SMD tool) logs changes to master data objects in change pointers. It is linked to the change document interface. To distribute master data using the SMD tool, change documents must be written when the master data object is changed, created and deleted. The master data object must be linked to the change document interface.

Change pointers are used to determine changes and distribute them to master data objects.

Change pointers are created in the following way:

The application program calls the change document interface for a change document object. To do this the generated function module `xyz_DOCUMENT_WRITE` is called. Its interface contains two parameters - the old records and the new records - for each table and structure for which change document deltas are to be determined. The change document interface creates a list of changes (table name, table key, field name, change type, old value, new value).

There are three types of changes:

- **Insert**
With the insert change type precisely one record is written (table name, table key, field name = "KEY", change type = insert, old value = empty, new value = empty). The field values are not documented as they can be found in the database. With this change type it is important that the special field name KEY is used.
- **Update**
With the update change type one record is written for each changed field (table name, table key, field name = <field name>, change type = update, old value = ABC, new value = DEF).
This change type has one variant:
In the definition of the change document object you can also specify that a delete record and an insert record are written rather than an update record. This is why the change type is also a key field in the CDPOS table. Two records for delete and insert can therefore be written for one table name, table key and field name.
- **Delete**
With the delete change type precisely one record is written (same as with insert). You can also specify in the definition of the change document object that all values are saved. Then there will be one record for each field.

When change documents are written, the interface for writing change pointers is called. Records which have message types to be sent are determined from table TBD62 for the table name, table key and field name. TBD62 consists of the table name, table key, field name and message type. A change pointer is written for each active message type in table TBDA2. A record with the change information is saved in table BDCP and a status record for each message type is saved in table BDCPS.

For this reason you have to add to table TBD62 all the changed fields that will cause a message (IDoc) to be sent.

Procedure

To be able to activate the writing of change pointers via the SMD tool for your master data object, you must carry out these steps:

- *Activate change pointers for each message type*
- *Maintain change-relevant fields for message type*
- *Activating Change Pointers Generally*

You also have to:

- *Implement function module for evaluating change pointers*
- *Defining the ALE Object Type MSGFN and Maintaining It as a Filter Object Type*



If you distribute master data using an asynchronous BAPI, all the settings below apply to the generated message type of the BAPI-ALE interface.

Activate change pointers for each message type

In table TBDA2 you can activate or deactivate the writing of change pointers for a specific master data object.

To maintain table TBDA2, in Customizing choose:

Basis Components

Distribution (ALE)

Modeling and Implementing Business Processes

Distribution of Master Data

Replication of Modified Data

Activate Change Pointers for Message Type (Transaction BD50).

Enter data in table TBDA2 with the message type for your master data object. For the delivery to the customer, the "Active" flag for the entry should not be set. If, for test purposes, you want to activate the writing of change pointers for the master data object, you must set the 'Active' flag for the TBDA2 entry for your message type.



As an example, look at the TBDA2 entry for the message type MATMAS for the material master.

Maintain change-relevant fields for message type

Change document fields from the change document object are entered in table TBD62. Change pointers are written when change document fields are logged via the change document interface.



To maintain table TBD62, from the ALE Development screen choose *Master Data* → *Activate change pointers for each field* (Transaction BD52).

In table TBD62, define for your message type all the change document fields, for which change pointers are to be written, so that changes to your master data object can be distributed to other systems.

When you add an entry in a table for your master data object, a change document field with the imaginary field name KEY is used to log this kind of change (examples: MATERIAL MARA KEY for the creation of a material, MATERIAL MARC KEY for the addition of a plant data to a plant). You must also include these entries in table TBD62, and you must do so for all tables from the change document object for your master data object.

Additions or changes to long text are logged with an entry in the change document item. This entry has the text object for the table name, and a value comprising the text ID and the language key for the field name. If changes to long texts for a master data object are to be distributed, these entries must also be included in table TBD62. You are not required to do this for every possible language key. It is sufficient to include an entry in table TBD62 for the text ID concerned, which has a value for the field name comprising the text ID and the character * (e.g. MATERIAL MATERIAL BEST* for the purchase order text in the material master).



Distributing Master Data Using the SMD Tool

Take a look at the TBD62 entries for the message type MATMAS for the material master in the R/3 System.

Activating change pointers generally

To activate master data distribution using change pointers, in Customizing choose:

Basis Components

Distribution (ALE)

Modeling and Implementing Business Processes

Distribution of Master Data

Replication of Modified Data

Activate Change Pointers - General (Transaction BD61).

Implement function module for evaluating change pointers

When change pointers are processed, IDocs for the master data objects are generated and dispatched. Change pointers are processed and IDocs subsequently generated and dispatched for each message type (for example, MATMAS for the material master). A function module carries out this process (for example, MASTERIDOC_CREATE_SMD_MATMAS for the material master).

You need to implement the function module for each message type. The naming convention for the function module is MASTERIDOC_CREATE_SMD_XXXXXX, where XXXXXX is the name of the message type.

Below is a description of how to implement the function module that processes change pointers and generates and sends the IDoc. Refer to the function module

MASTERIDOC_CREATE_SMD_MATMAS for the material master as an example. You can also use function module MASTERIDOC_CREATE_SMD_MATCOR for the core material master as a model for your function module. The IDoc type MATCOR01 for the core material master consists of merely the two segments: E1MARAC and E1MAKTC and contains only the core data from the material master. Function module MASTERIDOC_CREATE_SMD_MATCOR for the core material master has a more simple structure than the function module MASTERIDOC_CREATE_SMD_MATMAS for the complete material master.

Now create a function module for your message type. The interface of the function module is preset and consists of the parameter for the message type.



To call the function module, in ALE Administration choose *Services* → *Change pointers* → *Evaluate* (Transaction BD21).

Before you can start the transaction for your message type, you must maintain the message type assignment to the function module in control table TBDME. Define an entry for your message type in table TBDME, the reference message type being the same as your message type, and assign the function module. For examples, take a look at the entries for message type MATMAS (Material Master) and MATCOR (Core Material Master).



To maintain table TBDME, from the *ALE Development* screen choose *Master data* → *Additional data for message type*.

Implement the following steps in your function module to process change pointers and to generate and send IDocs:

1. Read all the change pointers that have not yet been processed for your message type using the function module CHANGE_POINTERS_READ.

2. Create an IDoc for every modified master data object. In the IDoc, only fill the segments that, according to the change pointers, were changed. In every segment, fill the first field MSGFN as follows:
 - 009, if the segment was added
 - 004, if segment fields were changed
 - 003, if the segment was deleted
 - 018, if segment fields were not changed, but the segment must be included in the IDoc, because hierarchically subordinate segments in the IDoc have to be dispatched.
3. Pass the IDoc to the ALE layer by calling function module MASTER_IDOC_DISTRIBUTE.
4. For the master data object that has just been processed, set the change pointers to 'Finished'. This is done by calling function module CHANGE_POINTERS_STATUS_WRITE.
5. Execute the COMMIT WORK command and call the DEQUEUE_ALL function module. For performance reasons, do not perform this step after every IDoc. Wait until you have created, for example, 50 IDocs.

Defining the ALE object type MSGFN as a filter object type

In the receiver determination for message types that have not been generated by the BAPI-ALE interface, IDoc segments containing modified data may be filtered off. Sometimes there may still be IDoc segments that do not contain any modified data at the end of segment chains. These segments have only been included due to the segment hierarchy in the IDoc.

For ALE outbound processing to be carried out, these attached segments must be suppressed by assigning the value 018 to them in the field MSGFN when the IDoc is generated.

You must also make the following settings:

- The ALE object type MSGFN must be defined In ALE Development choose *IDoc interface* → *Data filtering* → *Maintain filter object type* (Transaction BD95). Generally the ALE object type MSGFN has already been defined and the field MSGFN assigned to the table BDIPARAM.
- MSGFN must be specified as the filter object type for the new message type and for the new segment types in table TBD21. The entry in table TBD21 consists of the message type name under MESTYP, the segment type SEG Typ and the field name **MSGFN** under FLDNAM. From the *ALE Development* screen, choose *IDoc* → *ALE objects* → *Assign to message type* (Transaction BD59).



When distributing changes with asynchronous BAPIs, for compatibility you should use the FUNCTION field with the data element BAPIFN and the fixed values INS, DEL, UPD, REF and IGN. This identifies the type of change to be distributed with the BAPI.

Copy the FUNCTION field to each structured parameter. Then you can, for example, distribute the deletion of a data record.

Sending Master Data Directly

In this procedure, master data objects are sent immediately. The entire data in the master data objects is sent. The function is triggered via a report (e.g. RBDSEMAT for the material master).

In ALE Administration of master data distribution, you will find all the objects for which the "Send directly" function is already implemented.

Procedure

Now create a program for your master data object. In the program, define parameters to select objects to be sent and a parameter for the logical system.

Then implement the following steps in the program:

1. Create an IDoc for each master data object to be sent Enter all the data of the master data object into the IDoc In each segment, the first field MSGFN should contain the value 005.
2. Pass the IDoc to the ALE layer by calling function module MASTER_IDOC_DISTRIBUTE.
3. Execute the COMMIT WORK command and call the DEQUEUE_ALL function module. To improve performance, this step should only be carried out after several IDocs have been created.

Processing Inbound Master Data

For a general description of IDoc posting on the receiver side, see [Inbound Processing \[Seite 85\]](#). When you distribute master data keep in mind the following points:

- When posting the IDoc segments, process the first field (i.e. MSGFN) in each segment to determine the function to be performed for the segment: 009 for add, 004 for change, 003 for delete, 018 for unchanged data and 005 for a refresh function.
 - For functions 009, 004, 018, and 005, the data must be added to the receiving system (if not already available) or changed (if it already exists in the receiving system).
 - For function 003, the data must be deleted in the receiving system.
- For master data with reducible message types you must not change the field in the database if the corresponding IDoc segment field has the contents "/".
- When master data is posted, master data objects can be assigned to a class using the function module `CACL_OBJECT_ALLOCATION_MAINT`

Connections to Non-SAP Systems

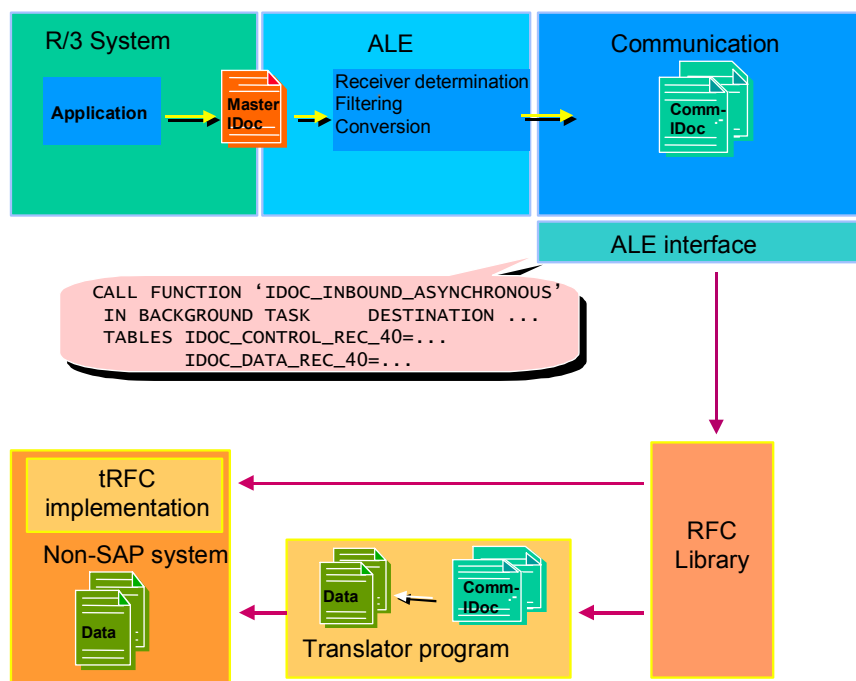
ALE is not restricted to communication between SAP systems, it can also be used for connecting R/3 Systems to non-SAP systems.

By using IDocs as universal information containers, ALE can reduce the number of different application interfaces to one single interface that can either send IDocs from an R/3 System or receive IDocs in an R/3 System.

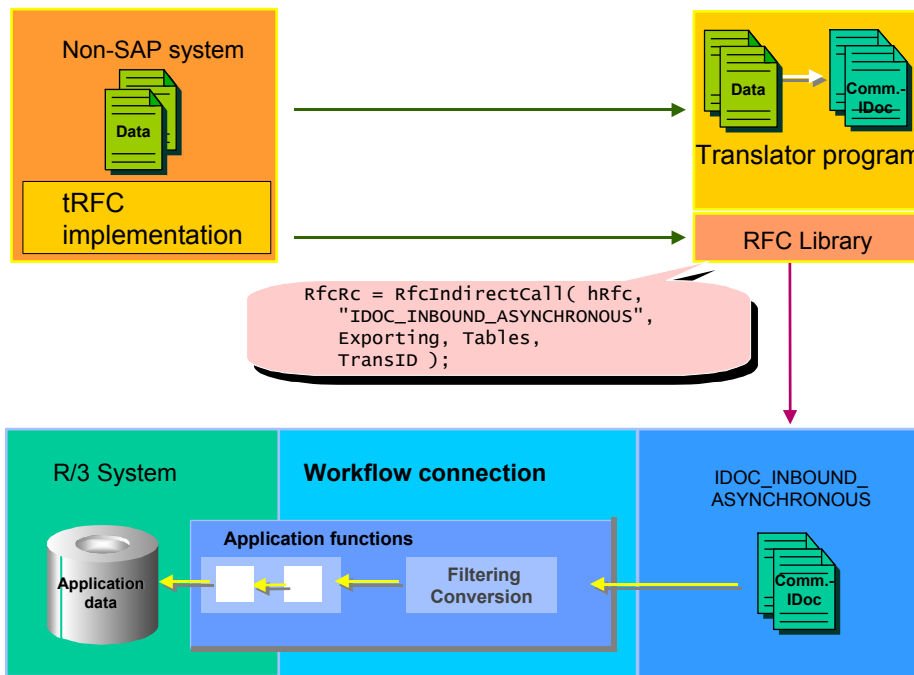
SAP certified [Translator Programs \[Seite 182\]](#) can convert IDoc structures into customer-defined structures.

Alternatively, the RFC interface for sending and receiving IDocs can be used in non-SAP systems. In both cases you need the *RFC Library* of the *RFC Software Development Kit* (RFC-SDK).

Communication from an R/3 System to a Non-SAP System



Communication from a Non-SAP System to an R/3 System



You can find an example of an IDoc interface to non-R/3 Systems in the documentation [Interfaces to Link Mobile Data Entry and Warehouse Control Unit \[Extern\]](#).



- For information on the technical implementation see [ALE Programming Guide \[Extern\]](#).
- You can find the requirements for the certification of interfaces in SAPnet under <http://www.sap.com/csp/scenarios>. Choose *Cross Application*, *CA-ALE* and *CA-AMS*.

Translator Programs for Communication

Definition

Translator programs are used to connect non-SAP systems to ALE. They must be certified by SAP.

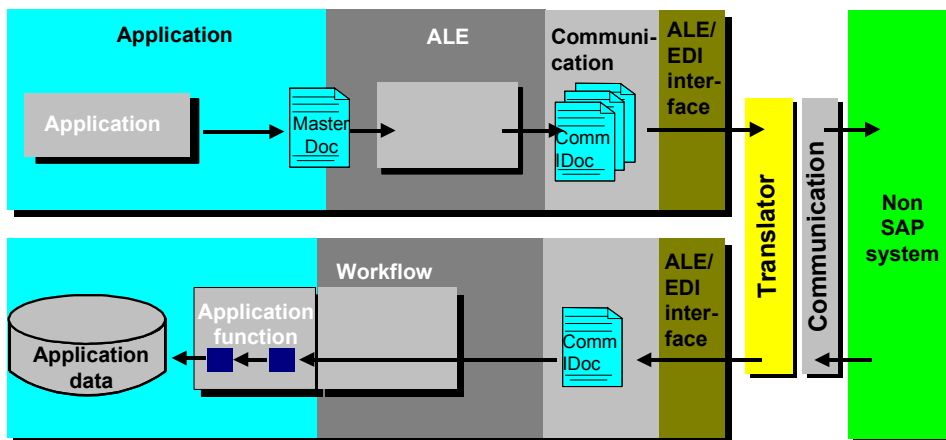
Use

Translators are typically used for:

- Mapping IDocs to any structure required in non-SAP systems
- Controlling communication such as establishing and restarting connections.

Structure

Using A Translator Between R/3 Systems And Non-SAP Systems



Integration

Translators are supplied from external vendors. SAP certifies the programs to ensure that communication between the ALE interface and the translator is functioning correctly.

The following criteria is checked for the certification:

- Can the translator automatically copy the IDoc structures into its own repository?
- Can the translator take an IDoc from an R/3 System and interpret the information based on its repository data?
- Does the translator have adequate mapping functionality?
- Can the translator pass the IDoc created back to R/3?

The certification itself does not evaluate the functions provided in the program.

Technical Implementation

This section provides you with an overview of the technical implementation of an interface.

Communication is executed through the SAP interface *Remote Function Call* (RFC).

As of Release 3.0, data can be transmitted between R/3 systems and external programs reliably and safely using the **transaction Remote Function Call** (tRFC).

The function module is executed **once** in the RFC server system. The remote system does not have to be available at the time when the RFC client program executes a tRFC. The tRFC component stores the called RFC function together with the respective data in the R/3 database under a unique transaction ID (TID).

For a detailed description of the RFC interface, refer to the documentation [Remote Communications \[Extern\]](#).

For details on the required TCP/IP settings, refer to the documentation [BC - SAP Communication: Configuration \[Extern\]](#).

This section gives you an overview of the program techniques involved. It is not a complete description.

If you wish to set up a connection yourself, you must refer to the documentation listed above.

TCP / IP Settings

The following TCP/IP settings are required to start the communication process:

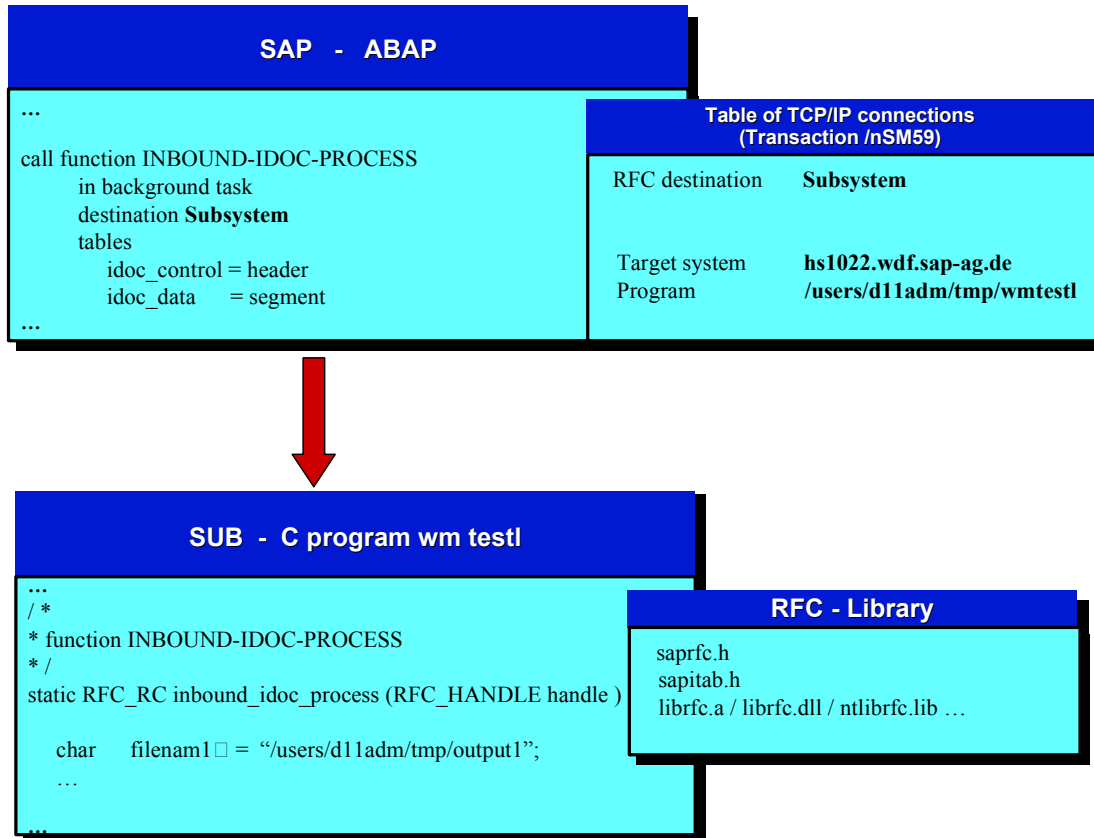
- So that the R/3 System can find the destination system, these TCP/IP prerequisites must be fulfilled, in particular the IP addresses in the respective file *hosts* must be known.
- The name of the gateway and the dispatcher must be entered in the file *services*, for example, *sapgw00* and *sapdp00*.
- In the R/3 System, Idocs are transmitted from the actual posting (update). Therefore, the TCP/IP link must also be created for the posting system.
- The SAP Gateway must have the right to start the external program (RFC server) via Remote Shell.

As of release 3.0C, you can work in register mode. In this way, the connection between the external system program and Gateway remains open (see [Registering Server Programs with the SAP Gateway \[Extern\]](#) in *The RFC API*).

For details on the TCP/IP settings, refer to the documentation [BC - SAP Communication: Configuration \[Extern\]](#).

Sending IDocs to an External System

The following diagram illustrates the program logic.



You transmit IDocs from the R/3 System by calling one of the two following functions modules with a destination:

- **IDOC_INBOUND_ASYNCHRONOUS**

You use this function module **from** release 4.0 upwards. It processes IDocs in record types that are valid for 4.x releases. Longer IDoc segment names are thus supported.

- **INBOUND_IDOC_PROCESS**

You use this function module for releases **up to** 4.0. It processes IDocs in record types that were valid for 3.x releases. For compatibility reasons, it should also be possible to use this function module in 4.x. External programs, too, should be able to support this function module.

The additional statement IN BACKGROUND TASK for the function call indicates the transaction RFC.

As with synchronous calls, the parameter DESTINATION defines the destination system and the destination program with the path (program context) in the remote system through a table in R/3.

Refer also to the ABAP test program SRFCTEST.

In the remote system, the destination program maintained in SM59 must exist. This program must also contain a function with the name of the function module call.

Sending IDocs to an External System

In R/3, the application data in the internal table is transmitted to the structure EDI_DD40 (EDI_DD before 4.0). For each IDoc, a control record of the structure EDI_DC40 (EDI_DC before 4.0) is also transmitted with the administrative data of the IDoc. In the example given, this data is transmitted in the form of internal tables.

For further information on this topic, refer to the documentation [RFC Programming in ABAP \[Extern\]](#).

For examples of tRFC programs, refer to the documentation *RFC Software Development Kit* (RFC-SDK):

- trfctest.c (client program)
- trfcserv.c (server program)

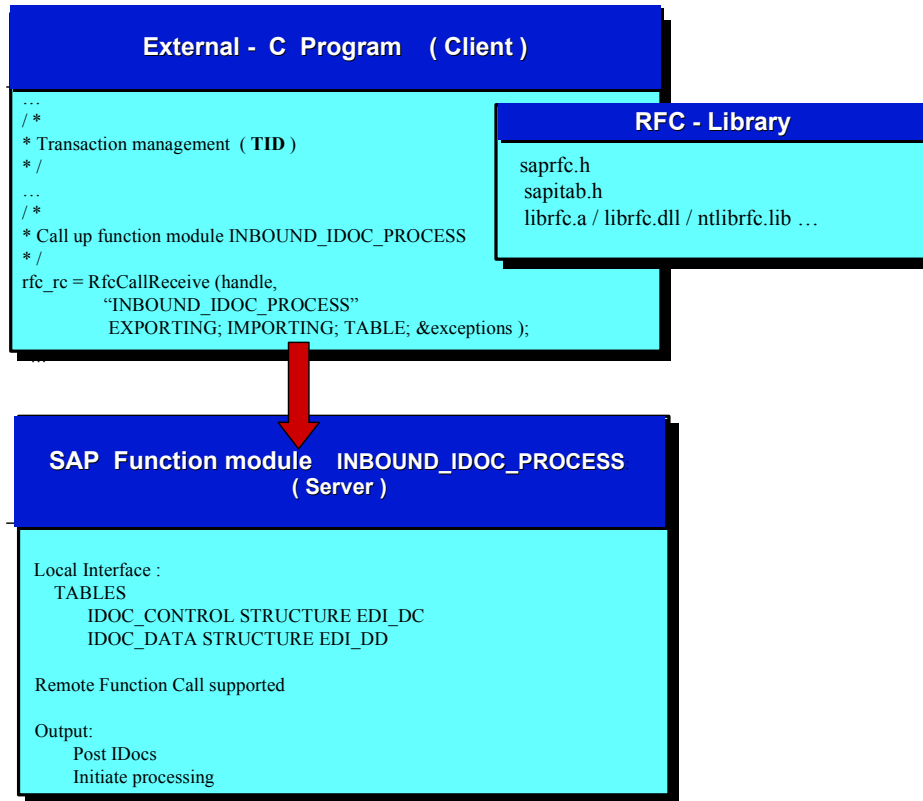
For details on the required functions, refer to the documentation [The RFC API \[Extern\]](#) or to the documentation of the RFC-SDK.

You can use these programs as examples for your own.

To interpret the useful data in the IDoc, you also need the data structures of the IDoc at the C program level. If you have an R/3 System available, you can **generate a header file of the IDoc** directly from the transaction WE60 (*Documentation for IDoc types*).

Sending IDocs: External System to SAP System

The following diagram illustrates the program logic.



The calling, external program uses the following functions of the *RFC Software Development Kit* (RFC-SDK):

- **RfcOpen**

Using this call, the system sets up an RFC connection to the server system. You can define the logon to the SAP System, including the server name of the SAP destination system, SAP logon, user ID, and so on in the C program or in the file *saprfc.ini*.

As soon as the connection to the server system has been set up, you must call the two following functions for the tRFC in the client program:

- **RfcCreateTransID**

The transaction ID that was created in the server system is determined with this call.

- **RfcIndirectCall**

The RFC data, together with the TID, is transmitted to the server system with this call.

If there is an error, the client program repeats this call.

Here the system must use the old TID with the call **RfcCreateTransID**. Otherwise, it will not be guaranteed that the RFC function is executed only once in the R/3 System.

The transaction is completed after successful execution of this call. The calling program can then update its own TID administration data (for example, delete the TID entry).

Sending IDocs: External System to SAP System

For more information, refer to the documentation [The RFC API \[Extern\]](#) or to the documentation of the RFC-SDK.

The useful data must be structured in the same way as the IDoc and placed in the internal table of the structure EDI_DD40 (EDI_DD before 4.0). The control record must be generated for each IDoc and placed in the internal table of the structure EDI_DC40 (EDI_DC before 4.0). The form in which the data is transferred is also described in detail in the documentation.

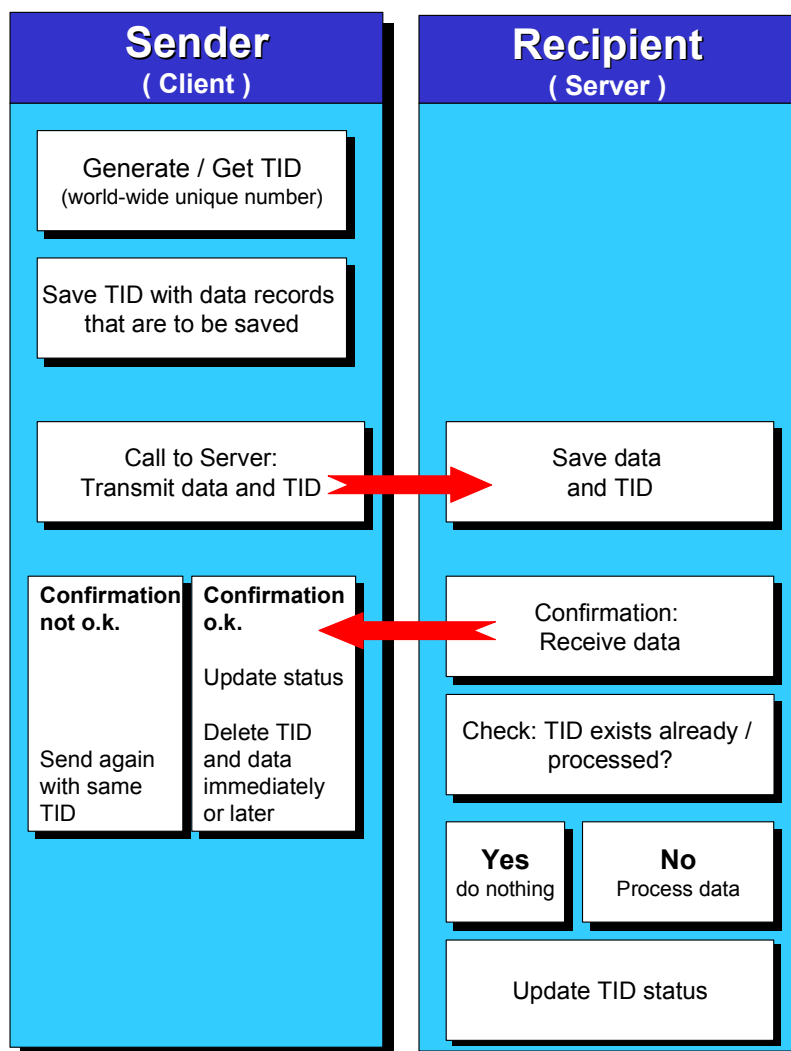
Transaction Identification Management (TID)

A unique code must be used for a communication process in order to guarantee the integrity of the data to be transferred. The receiving system can then use this code to decide whether this data has already been received and processed.



For example, communication may break down during data transmission when goods receipts are entered on mobile data entry devices. The person handling the data would then have to send it again to make sure that it is posted in the SAP system. If, however, the data was successfully received and processed the first time it was sent to the SAP system, the system must be able to recognize this and then not process the second data record.

This example inevitably results in the following sequence of operations between the sending and receiving system.



Integrating Dialog Interfaces

Use

Data exchange in distributed environments is carried out without user interaction.

Sometimes, however, you may need to call dialog interfaces from remote systems. A typical example is to display the source document of a document held in another system.



HR forwards payroll results to Accounting. Here the appropriate documents are posted. Each document in Accounting has a reference to the source document in HR. In the integrated system the document display is programmed in Accounting so that you can jump to the document display of the source document in HR.

If HR and Accounting are running on different systems, there is no purpose in Accounting reading the data via a BAPI and reprogramming the HR document display again. Instead the HR document display is called from the remote system.

Dialog interfaces are not provided in place of BAPIs, they are *additional* to BAPIs. We recommend that for each dialog interface you provide a relevant BAPI that returns the data as a background interface.

Currently dialog interfaces are primarily for display functionality in R/3 to R/3 distribution scenarios. Dialog methods with change functionality are currently *not* supported.

Dialog interfaces can also be called from external platforms.

Procedure

RFC is used to call dialog functionality from remote systems. Images can be transferred with RFC.

A dialog method is implemented through an RFC-enabled function module. The dialog is then called in this function module.

This function module must fulfill the same quality requirements as function modules for BAPIs:

- English names for fields and parameters
- A return parameter and no exceptions
- Documentation must be provided
- The interface must be frozen

The function module is modeled in the BOR. The same rules apply as for BAPIs, with two exceptions:

- The function module can only be released internally
- The dialog flag must be set for the object method.

A dialog method is created as an API method, the same as a BAPI in the BOR. This ensures that the required coding in the BOR is generated and the dialog method can also be called via the BOR runtime environment. If changes are later made to a dialog method ensure that the BOR coding is also changed appropriately. A dialog method must be able to be called via the BOR runtime environment.

The internal release indicates that this method is used for R/3 to R/3 scenarios. The API flag indicates that this method can be called from a remote system. The dialog flag is set because a dialog interface is used as opposed to a background interface. The dialog flag delimits the dialog methods from the BAPIs.

Integrating Dialog Interfaces



- When you implement an object display ensure that users cannot branch into the change mode.
- Currently, technical problems may arise if you want to call a pop-up from remote rather than an entire screen.
- If you are working with reports, the command `submit...` and `return` must be used. `submit...` alone is not possible.

Naming Dialog Methods

Each object type in the BOR can be accessed through the method *Display*. However, in many cases the method has not been implemented. If the dialog method to be implemented is to display the object, *Display* should be used, if possible,

If *Display* has already been implemented, and changes have resulted that are incompatible according the BAPI quality requirements, a different method must be used.

Apart from the *Display* method, we recommend you use the suffix *WithDialog* for dialog methods when you are modeling them in the BOR.

Calling Dialog Methods

Dialog methods are called from an R/3 System in the same way as BAPIs are called through RFC.

There are two different types of calls:

- [Calls with References to the Logical System \[Seite 193\]](#)
- [Calls Without References to the Logical System \[Seite 195\]](#)

The logical system must be determined from the distribution model.



The examples do not claim to be appropriate for all possible situations where methods are called. In particular, a standard procedure cannot be recommended for error handling because it depends on the application involved and the circumstances the call is made in. Before you incorporate a dialog method call into your coding, you should carefully consider how errors will be handled.

If, for example, no destination for dialog calls can be determined for a server system it may be because the server does not have dialog functionality for technical or security reasons. This situation can also occur in productive systems. You cannot interpret this as an inconsistently configured system.

You should also keep in mind that the RFC implicitly executes a database commit, that is the LUW is completed. If you are not familiar with using RFC calls, you should first read the on-line help on the ABAP language element `call function` with the addition, `destination`.

Calls With References to the Logical System



HR forwards payroll results to Accounting. Here the appropriate documents are posted. Each document in Accounting has a reference to the source document in HR. The logical system in which the HR document was created is included in the reference.

The method to be called is named, for example, *HRDoc.Display*. The field BKPF-AWREF contains the ID of the reference document in HR and the field BKPF-AWSYS contains the name of the logical system.

The source code below shows how the HR document display is called the from the document display in Accounting:

```
...
DATA:
  HEAD LIKE BKPF,
  RETURN LIKE BAPIRET2,
  SERVER_DEST LIKE TBLSYSDEST-RFCDEST,
  MSG_TXT(80) TYPE C.
...

* get RFC destination for remote method call

CALL FUNCTION 'OBJ_METHOD_GET_RFC_DESTINATION'
  EXPORTING
    OBJECT_TYPE      = 'HRDOC'
    METHOD            = 'DISPLAY'
    LOGICAL_SYSTEM   = HEAD-AWSYS
  IMPORTING
    RFC_DESTINATION = SERVER_DEST
  EXCEPTIONS
    NO_RFC_DESTINATION_MAINTAINED = 1
    ERROR_READING_METHOD_PROPS    = 2
    OTHERS                        = 3.

IF SY-SUBRC <> 0.
  IF SY-SUBRC = 1.
    * application specific message saying document cannot be displayed
    ...
  ELSE.
    * hard error
    MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
      WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
  ENDIF.
ENDIF.

* call display function. If SERVER_DEST is initial, it's a local call.
CALL FUNCTION 'BAPI_HRDOC_DISPLAY'
  DESTINATION SERVER_DEST
  EXPORTING
```

Calls With References to the Logical System

```
    DOCUMENT_ID = HEAD-AWREF
IMPORTING
    RETURN = RETURN
EXCEPTIONS
    COMMUNICATION_FAILURE = 1 MESSAGE MSG_TXT
    SYSTEM_FAILURE       = 2 MESSAGE MSG_TXT.

IF SY-SUBRC <> 0.
* handle remote exceptions
    MESSAGE E777(B1) WITH
        'HRDoc.Display' HEAD-AWSYS HEADER-AWSYS
        MSG_TXT(50) MSG_TXT+50(30).
ELSEIF NOT RETURN-TYPE IS INITIAL.
* handle return parameter
    ...
ENDIF.
```

The message B1 777 is a generic message. An application-specific message could be used instead.

Calls Without References to the Logical System



HR forwards payroll results to Accounting. Here the appropriate documents are posted. The HR document does not have a reference to the document created in Accounting.

The method to be called is named, for example, *ACDoc.Display*. *ACDoc.Display* does not have any filter objects. The field HRKPF-DOCNR contains the ID of the HR document.

As the logical system cannot be identified here, the logical target system has to be determined from the distribution model.

The source code below shows how the Accounting document display is called from the HR document display:

```

...
DATA:
  HEAD LIKE HRKPF,
  SERVER LIKE BDBAPIDEST,
  RETURN LIKE BAPIRET2,
  MSG_TXT(80) TYPE C,
  FILTER_VALUES LIKE BDI_FOBJ OCCURS 0 WITH HEADER LINE.
...

* get logical system and RFC destination for remote method call

* no filter objects are used
REFRESH FILTER_VALUES.

* get server system from ALE distribution model
CALL FUNCTION 'ALE_BAPI_GET_UNIQUE_RECEIVER'
  EXPORTING
    OBJECT          = 'ACDOC'
    METHOD          = 'DISPLAY'
  IMPORTING
    RECEIVER       = SERVER
  TABLES
    FILTEROBJECTS_VALUES = FILTER_VALUES.
  EXCEPTIONS
    ERROR_IN_FILTEROBJECTS = 1
    ERROR_IN_ALE_CUSTOMIZING = 2
    NOT_UNIQUE_RECEIVER = 3
    NO_RFC_DESTINATION_MAINTAINED = 4
    OTHERS = 5.

IF SY-SUBRC <> 0.
  IF SY-SUBRC = 4.
    * application specific message saying document cannot be displayed
    ...
    ELSE.
    * hard error
    MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO

```

Calls Without References to the Logical System

```
        WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.  
    ENDIF.  
ENDIF.
```

*** call display function. If SERVER_DEST is initial, it's a local call.**

```
CALL FUNCTION 'BAPI_ACDOC_DISPLAY'  
  DESTINATION SERVER-RFCDEST  
  EXPORTING  
    DOCUMENT_ID = HEAD-DOCNR  
  IMPORTING  
    RETURN = RETURN  
  EXCEPTIONS  
    COMMUNICATION_FAILURE = 1 MESSAGE MSG_TXT  
    SYSTEM_FAILURE       = 2 MESSAGE MSG_TXT.
```

```
IF SY-SUBRC <> 0.
```

*** handle remote exceptions**

```
  MESSAGE E777(B1) WITH  
    'HRDoc.Display' HEADER-AWSYS MSG_TXT(50) MSG_TXT+50(30).  
ELSEIF NOT RETURN-TYPE IS INITIAL.
```

*** handle return parameter**

```
  ...  
ENDIF.
```



This example does not use filter objects. If filter objects exist for the object method to be called, the distribution model must be accordingly interrogated.

Serialization of Messages

Use

Serialization plays an important role in distributing interdependent objects, especially when master data is being distributed.

IDocs can be created, sent and posted in a specified order by distributing message types serially. Errors can then be avoided when processing inbound IDocs.

Features

Interdependent messages can be serially distributed in the following ways:

- [Serialization by Object Type \[Extern\]](#)
- [Serialization by Message Type \[Seite 118\]](#)
- [Serialization at IDoc Level \[Seite 201\]](#)
(not for IDocs from generated BAPI-ALE interfaces)

Serialization by Object Type

Use

Serialized messages can be of different types (for example, create, change or cancel messages). All messages here relate to one special application object.

The messages can contain both master data and transaction data.

With object serialization the messages a given object are always processed in the correct order on the receiver system. The order messages are posted in on the receiver system is the same as they were created on the sender system.

Prerequisites

You have to activate serialized distribution in both systems in ALE Customizing:

Tools → AcceleratedSAP → Customizing → Project Management

SAP Reference IMG

Basis Components

Distribution (ALE)

Modeling and Implementing Business Processes

Master Data Distribution

Serialization for Sending and Receiving Data

Serialization by Object Type

Features

Object type serialization is carried out using object channels.

All messages are processed in an object channel in the target system in the same order they were sent from the source system. An object channel contains a number of ordered IDocs and is defined by an object type (BOR) and precisely one channel number. A channel number is a message attribute. It is generated by the function module ALE_SERIAL_KEY2CHANNEL.

All messages with the same channel number and object type are serialized when the messages are processed.

The current number of each object channel is recorded. This process is takes place in what is known as the registry. There is an outbound registry and an inbound registry. Serialization must be activated in both registries (see prerequisites).

Outbound Processing (Source System)

When outbound IDocs are processed, for each object channel (field CHNUM) a unique serial number is assigned to each IDoc created (field CHCOU). This number and the object channel are transferred with the IDoc in the SERIAL field.

An unique serial number is assigned to each message for the application object in question.

Inbound Processing (Target System)

When inbound IDocs are processed, a unique serial number is generated for each object channel (field CHNUM) and communication link. The ALE layer determines whether a given IDoc can now be posted or whether other IDocs have to be posted first. The serial number for each received IDoc is exactly one whole number lower. Any gaps in the sequence will mean that IDocs are missing, either because the transfer did not work, or because earlier IDocs were not posted successfully.

In this case the IDoc is assigned status 66 and must be posted again with the program RBDAPP01.

Objects are assigned to messages and channels by the application.

Transfer errors (IDoc sequence mixed up) and inbound posting errors (IDoc cannot be posted due to Customizing errors) no longer affect the sequential order, because serialization corrects these errors.

Serialization By Message Type

Use

IDocs can be created, sent and posted in a specified order by distributing message types serially. Object interdependency is important at the message type level.



Consider a purchasing info record with a vendor and a material. To avoid any processing errors, the vendor and material must be created in the receiving system before the purchasing info record.

Prerequisites

You have to activate serialized distribution of message types in both systems in ALE Customizing (Transaction SALE).

Basis Components

Distribution (ALE)

Modeling and Implementing Business Processes

Master Data Distribution

Serialization for Sending and Receiving Data

Serialization by Message Type

Features

Serialized distribution is only used to transfer changes to master data. IDoc message types are assigned to serialization groups according to the order specified for their transfer. Master data is distributed in exactly the same order. If all the IDocs belonging to the same serialization group are dispatched successfully, the sending system sends a special control message to the receiving system. This control message contains the order IDocs are to be processed in and starts inbound processing in the receiving systems.

Serialized distribution of message types is not a completely new way of distributing master data; it uses existing ALE distribution mechanisms whilst adhering to a specified order of message type. This distribution could also be carried out manually using existing ALE programs. However, serialized distribution automates the single steps and can schedule them in a batch job.

In the serialization menu selection criteria determine how certain parts of the serialized distribution will be processed, for example, control message dispatch and inbound processing.

Serialization at IDoc Level

Use

Delays in transferring IDocs may result in an IDoc containing data belonging to a specific object arriving at its destination before an "older" IDoc that contains different data belonging to the same object. Applications can use the ALE Serialization API to specify the order IDocs of the same message type are processed in and to prevent old IDocs from being posted if processing is repeated.

SAP recommends that you regularly schedule program RBDSRCLR to clean up table BDSER (old time stamp).

Prerequisites

IDocs generated by BAPI interfaces cannot be serialized at IDoc level because the function module for inbound processing does not use the ALE Serialization API.

Features

ALE provides two function modules to serialize IDocs which the posting function module has to invoke:

- IDOC_SERIALIZATION_CHECK to check the time stamps in the serialization field of the IDoc header.
- IDOC_SERIAL_POST updates the serialization table.

Automatic Tests

You can check the quality of the ALE layer and the ALE business processes using automatic tests.

Although automatic tests cannot replace manual tests, you can still use them to check the basic functions of ALE business processes. This is especially useful for upgrade tests that should be regularly carried out without manual intervention. The logical system can also be automatically customized so that subsequent tests are easier and quicker to perform.

The automatic tests are developed entirely in R/3 using the CATT environment. They comprise mainly test runs and test modules. For further information see [CA - Computer Aided Test Tool \[Extern\]](#).

These instructions apply to all applications responsible for one or more ALE business processes. You need to be familiar with developing CATT modules.

IDES data should be used in the test modules and test runs so that they can be run regularly in new test systems without requiring manual action.

Example Scenario for Distributing Master Data

The example scenario for distributing master data via ALE is used to illustrate how a test procedure is set up. The procedure for transaction data is essentially the same, the test runs may be more extensive because several messages flow between the different logical systems.

Creating an ALE Scenario

To distribute master data (material, cost center, etc.) a master data is created or changed in a central R/3 System and then sent in an IDoc to a decentralized system.

This scenario can be divided into four steps:

Application

1. Create/change master data
2. Create IDoc

ALE layer

3. Send IDoc
4. Post IDoc

Application

Preparing the Test

Preparing the Test

Before you can test run an ALE scenario, the ALE layer must be customized. The customizing is carried out by a leader CATT procedure based on the existing IDES customizing. The test module P3013119 is provided for this. This module enhances the distribution model and creates partner profiles for the message types.

You must also establish connections from the system running the test to the central and decentralized systems, as both systems are started during the test procedure.

Test module P3017940

The test module P3017940 itself consists of number of test modules and function modules.

Starting the sender system

Finds IDoc number: FB GET_IDOCNR_FROM_OBJECT
Sends IDoc: TB P3015649
Reads IDoc status: FB GET_STATUS_FROM_IDOCNR
Converts IDoc status: TB P3013115

Starting the receiver system

Finds IDoc: FB GET_IDOCNR_FROM_IDOCNR
Posts IDoc: TB P3013464

Examples:

The test procedure P3013121 is an example of a master distribution process, P3017156 an example of a complex process with transaction data.

The module P3017940 is used in both procedures.

Developing the Test Procedure

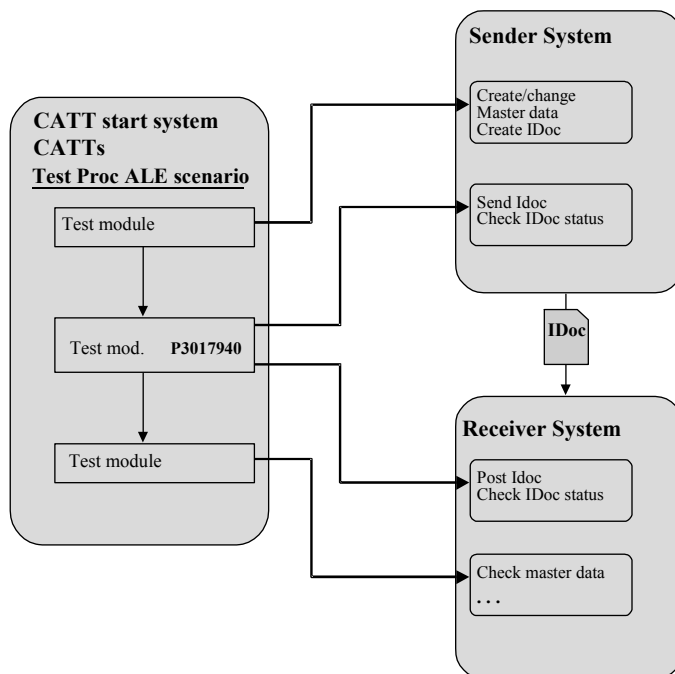
Remote Call to

A test procedure consists of three parts:

1. Creating/changing master data and creating the IDoc
2. Executing the test module P3017940
(Sending and posting the IDoc and checking its status).
3. Checking the master data resulting from the IDoc

Process Schema

The test procedure for an ALE scenario to distribute master data consists of the following test modules:



The test module **P3017940** is provided for the ALE layer steps. This module can be used for all scenarios in which object links are created by the application.

This test procedure can be called from each test procedure on local systems. Appropriate values need to be entered in the import parameters.

Details of links are held in Table TBD14.

To run a test module in another system, you have to specify an RFC destination in the procedure with CTRL/F2. The test module must be available in the target system. For further information refer to the on line help on CATT.

Error Handling

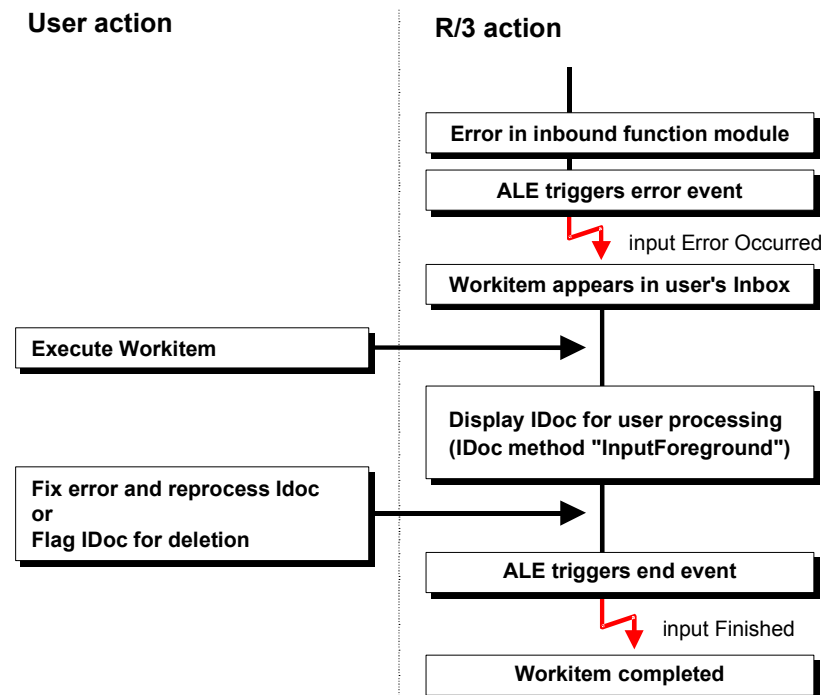
ALE error handling makes use of the SAP Business Workflow technology in the R/3 System. SAP Business Workflow organizes and manages a work process whereby tasks are assigned to individual agents. After completing a task, agents are informed of subsequent work items in their inbox.

SAP Business Workflow is object oriented; ALE error handling objects are IDocs and their methods and events.



If there is an error, *only* the first message from the return parameter is copied to the text in the associated work item.

Error Handling Process



The example below of an inbound error for a material master message shows the steps in ALE error handling:

1. The inbound function module passes message to the ALE layer that an error has occurred.
2. ALE triggers the object event "inputErrorOccurred" from the IDCOMATMAS category.
3. This event is linked to standard task number 00007947, long name "MATMAS_inbound error"
4. A work item appears in the user's inbox, the work item's short text is the first fifty characters of the error message contained in the IDoc's status record.
5. When the user processes the work item, the IDOCMATMAS method "IDOC.InputForeground" is processed.

Error Handling

IDoc status record is displayed and the user can display the error message's long text. If the user was able to remedy the error, the IDoc can be submitted for updating. If the error cannot be remedied, the user can flag the IDoc for deletion.

6. If the IDoc was either successfully submitted or flagged for deletion, IDOCMATMAS's event "inputFinished" is triggered indicating that the task has been carried out.

Objects, Events and Tasks to be Created

How to implement error handling for a message type, (XAMPLE):

- Create a new object type IDOCXAMPLE as a child of the object type IDOCAPPL, in the Business Object Repository (BOR). Customers should use the name ZDOCXAMPLE.
- Create a new standard task, named "XAMPLE_Error".
- Create event-couplings linking IDOCXAMPLE's event *inputErrorOccurred* to your standard task, and event *inputFinished* to the function module for completing work items.

In each case it is easier to copy an existing object type or standard task.

To provide a fully ALE-compatible interface, you will also need to:

- Create a new object type IDPKXAMPLE, as a child of the object type IDOCPACKET. Customers should use the name ZDKXAMPLE.
- Maintain your inbound process code to refer to the above objects and events

The example of the material master record IDoc MATMAS explains how the above objects are created.

The attributes of object type IDOCMATMAS are used to define the standard task so that the error message and the material number appear in the work item text.

The methods and events used are described above.

Object type IDOCMATMAS:

Attributes, methods and events relevant to the inbound function module.

	Name	From IDOCAPPL	Description
Attribute	ShortMessage	Yes	First 50 characters of IDoc's status message
	ApplicationObjectID	Yes	ID of ALE link object in IDoc
Method	InputForeground	Yes	Processes IDoc starting with status display
	InputBackground	Yes	Processes IDoc without any dialog
Event	inputErrorOccurred	Yes	Triggered when direct application handover failed; not triggered by the methods InputForeground and InputBackground
	inputFinished	Yes	Triggered when IDoc successfully processed, or user flags IDoc for deletion

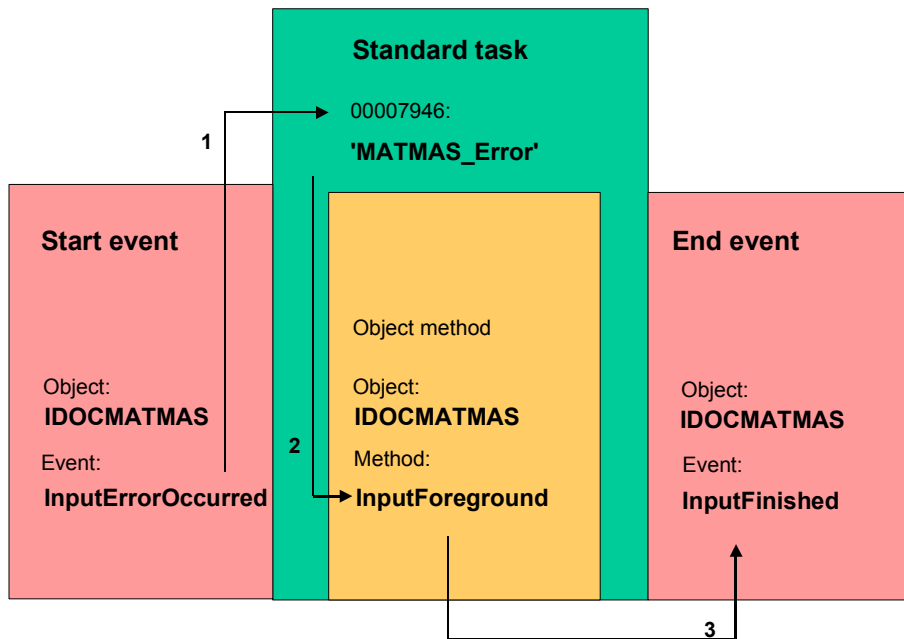


An example of an ALE error handling.

The arrows show the three stages:

1. The event *inputErrorOccurred* causes a work item to be created.
2. When the user executes the work item, the method *InputForeground* is invoked.
3. If the IDoc has been successfully processed, or flagged for deletion, the event *inputFinished* is triggered, which terminates the work item.

Objects, Events and Tasks to be Created



Before continuing, have a look at the object type IDOCMATMAS in the Business Object Builder and at the standard task 7946. Both the Business Object Repository and standard tasks are reached via the menu path:

Tools → *Business Workflow* → *Development* → and the menu *Definition Tools* →
 → *Business Object Builder* and
 → *Tasks/Task Groups*

For further information see the application help.

Object Types and Events

The naming convention for the object types for message type XAMPLE is IDOCXAMPLE and IDPKXAMPLE. These object types are created in the Business Object Repository, under object type IDOCAPPL and IDOCPACKET respectively. The steps to carry out are described next.

For each object type you create you are asked for the name of a report that will also be created. This can also be deleted. The standard naming conventions apply, that is they should be within the customer name range and begin with Y or Z. Make sure you use a different report for each object type.



The object types IDOCAPPL and IDOCPACKET contain documentation that describes their methods and events.

See also:

[Creating IDoc Object Type IDOCXAMPLE \[Seite 212\]](#)

[Creating IDoc Packet Object Type IDPKXAMPLE \[Seite 214\]](#)

Creating IDoc Object Type: IDOCXAMPLE

Creating IDoc Object Type: IDOCXAMPLE

From the initial R/3 screen you can get to the Business Object Repository (BOR) via menu path via *Tools → SAP Business Workflow → Development*, then select *Object repository*.

In the hierarchy, go to object type IDOCMATMAS under *Cross-Application Components → IDoc Interface/Electronic Data Interchange → IDOC → IDOCAPPL*.

Copy the object type IDOCMATMAS as follows:

1. Select object type IDOCMATMAS and then select *Copy*
2. A popup window appears: enter the name of your object type (e.g. IDOCXAMPLE) and your report (e.g. RXAMPLE1) and select *Copy*

The naming convention for SAP is IDOC<Message type>, e.g. IDOCXAMPLE

The naming convention for customers is ZDOC<Message type> e.g. ZDOCXAMPLE

3. A popup window appears. Enter your development class

Edit the object type you have created (e.g. IDOCXAMPLE) as follows:

1. Select the object type (e.g. IDOCXAMPLE) and then select *Change*
2. A popup window appears: **do not** choose any of the options, just press *Enter*.
3. Select *Basic data* and change the object's short text and description to fit your needs

The naming convention for the short text is "IDOC <Message type>", e.g. "IDOC XAMPLE".

Select *Back*.
4. Change the description of the event *inputFinished* as follows: Look at the events by expanding the hierarchical list under the heading *Events*; double-click on the event *inputFinished* and now change the description and press *Enter*
5. Change the event's parameter *Appl_Object* as follows: Select menu options *Goto → Obj. type components → Parameters*, and then select parameter *Appl_Object* by double-clicking on it. Change the object type BUS001 to the object type that has been processed by your inbound function module. Also change the short text and the description.



If a suitable application object type does not exist for your inbound function module, then delete the parameter *Appl_Object* rather than changing it. To do this, select the parameter *Appl_Object* and then select *Delete* rather than double-clicking on the parameter.

- Select *Back* and save the object type (*Object type → Save w/o check*).
- Generate the object type by selecting *Generate*.
- Release the object type as follows: Go back to the BOR hierarchy view and select *Object type → Release*.

As an option, you can add a further parameter *Application_Variable* to the *inputFinished* event's container. This variable is filled with the contents of the inbound function module's export parameter *Application_Variable*. To see an example where this parameter is used, look at the object type IDOCORDERS.

Creating IDoc Packet Object Type: IDPKXAMPLE

Creating IDoc Packet Object Type: IDPKXAMPLE

In the hierarchy, go to the object type IDPKMATMAS under *Cross-application components* → *IDoc interface/Electronic Data Interchange* → IDOCPACKET.

Copy the object type IDPKMATMAS as follows:

1. Click on object type IDPKMATMAS and then select *Copy*.
2. A dialog box appears: enter the name of your object type (IDPKXAMPLE) and your report (RXAMPLE2) and then select *Copy*.

The naming convention for SAP is IDOC<message type>, e.g. IDPKXAMPLE

The naming convention for customers is ZDOC<message type>, for example, ZDPKXAMPLE

3. A dialog box appears: enter your development class;

Edit the object type you have created (IDPKXAMPLE) as follows:

1. Select the object type (IDPKXAMPLE) and then select *Change*.
2. A dialog box appears: **do not** choose any of the options, just select *Enter*.
3. Select *Basic data* and change the object's short text and description as required.
 - The naming convention for the short text is IDPK<message type>, for example, IDPKXAMPLE.
4. Select *Back* and then *Object type* → *Save w/o check*.
5. To generate the object type select *Generate*.
6. Release the object type as follows: Go back to the BOR hierarchy and select *Object type* → *Release*.

Creating a Standard Task

To create a new standard task copy an existing task. The standard task 7947 "MATMAS_TASK" is used as an example.

From the R/3 menu choose *Tools → SAP Business Workflow → Development*.

1. To create a standard task, choose *Definition tools → Tasks/Task groups → Copy*
2. In the *Task type* field enter TS and in the *Task* field enter 7947. Then choose *Task → Copy*.
3. In the window *Task:Copy* enter the *Abbreviation* (XAMPLE_Error) and *Name* (XAMPLE input error) of your object and then select *Copy*.
4. A *Create object catalog entry* window appears: enter your development class and select *Save*.
5. Note the number of the task you have created (99900000).

Edit the new task as follows:

1. Choose *Definition tools → Tasks/Task groups → Change*
2. Delete the triggering events:
 - Select *Triggering events*
 - Select the event `inputErrorOccurred` and choose *Edit → Delete event*.
 - Select *Back*.
3. Delete the terminating events:
 - Select *Terminating events*
 - Select the event `inputFinished` and choose *Edit → Delete event*.
 - Select *Back*.
4. Replace the application component HLA0006031 with the application component applicable for your inbound processing.



This is used for documentation and for finding the tasks appropriate to a given application component.

5. Replace the object type IDOCMATMAS with your newly created object type (IDOCXAMPLE).
6. Add the triggering event applicable to your object type:
 - Select *Triggering events*
 - Select *Insert event*
 - A dialog box appears: type in your object type (IDOCXAMPLE), and use F4 in the *event* field to choose the event `inputErrorOccurred`
 - Select *Binding definition*
 - Enter `&_EVT_OBJECT&` in the field *_WI_Object_Id*.
 - Enter `&EXCEPTION&` in the field *Exception*.
 - Save your entries.
 - Choose *Goto → Event linkage*, then *Activate*.
 - Select *Back* twice.
7. Add the terminating event applicable to your object type:
 - Select *Terminating events*

Creating a Standard Task

- Select *Insert event*
 - A dialog box appears. enter your object type (IDOCXAMPLE) and use F4 in the *Event* field for the event inputFinished. Use F4 on the event field and choose the element *_WI_OBJECT_ID*
 - Select *Back*. Note: You do not need to maintain the binding definition for this event.
8. To save your changes select *Save*.
- Select *Save*.

All the tasks should now be correctly copied. To be sure, check the following three settings:

1. Check the binding to the object:
 - Select *Binding OM*
 - &EXCEPTION& should be assigned to the field *Exception*.
 - Select *Back*.
2. Check the default role:
 - Select *Default roles*.
 - The standard role for the agent should be 134. The standard role for the agent should be 134.
 - Select *Binding editor*. &_WI_OBJECT_ID& should be assigned to the parameter IDOCNUMBER.
 - Select *Back* twice.
3. Check the work item text:
 - – Select *Work item text*
 - A dialog box appears containing the work item text, which should contain two entries:
 - &_WI_Object_Id.ShortMessage& This entry ensures that the first 50 characters in the work item text contain the IDoc's attribute "ShortMessage", which is the first 50 characters of the IDoc's error short text.
 - &_WI_Object_Id.ApplicationObjectID&: This entry ensures that the remaining work item text contains the IDoc's attribute *ApplicationObjectID*, which is the ID of the application object contained in the IDoc. In the case of MATMAS it is the material number. The attribute is determined using the ALE link object.
 - Select *Back*.



An agent is assigned to a work item via Customizing for SAP Business Workflow. These customizing functions can be found in ALE Customizing under *Error Processing* → *Create organizational units and assign standard tasks*.

Maintaining Inbound Methods

The event fields in your inbound method mentioned in the section ALE Settings under Process codes can now be maintained. From the ALE Development screen choose *IDoc* → *Inbound* → *Process code Processing type*



Process codes are client-dependent Make sure you maintain them in the correct client.

1. Select your process code (XAMP).
2. Enter your packet object type (IDPKXAMPLE) and the end event `massInputFinished` in the IDoc packet fields.
3. Enter the start event `inputErrorOccurred` and the end event `inputFinished` in the IDoc fields of your IDoc object type (IDOCXAMPLE),
4. Enter your application object type, for example, BUS1001 for materials, in the application object type field.
5. Save your entries.

Checking Consistency of Inbound Error Handling

Checking Consistency of Inbound Error Handling

You have now maintained everything needed for error handling via Workflow. To verify that everything is correct, choose IDoc → *Inbound* → *Consistency Check* from the *ALE Development* screen.

The line containing your process code should appear white if everything is OK, yellow if you are not using an application object type and everything is OK, and red if at least one setting is incorrect. (These colors assume the default color settings.)

To see details, double-click on the line containing your process code.



Process codes and event linkages are client-dependent, so make sure you carry out the check in the correct client.