

# Cookbook for enhancing the SAP Business Partner with additional customer/vendor fields

Customer Edition 1.14 – February 12<sup>th</sup>, 2018

## Source

The source of this document is SAP note [2309153](#). Please check this note regularly to get the latest version of this document.

The example coding used in this cookbook is provided by SAP note [2295823](#). It might be necessary to implement this note to have this coding available in your system.

## Change History

Version 1.14 (February 12<sup>th</sup>, 2018)

- Chapter 3.10 – Field modification for blocked customer/vendor

Version 1.13 (May 04<sup>th</sup>, 2017)

- Chapter 3.5.2 – Necessary section to screen assignments in BDT customizing

Version 1.12 (April 20<sup>th</sup>, 2017)

- Wrong structure BUS\_EI\_EXTERN in section 2.3.2 corrected to CVIS\_EI\_EXTERN
- Wrong structure BUS\_EI\_EXTERN in section 5.2 corrected to CVIS\_EI\_EXTERN

Version 1.11 (April 19<sup>th</sup>, 2017)

- Hyperlinks of document references in section 1.3 updated

Version 1.10 (October 06<sup>th</sup>, 2016)

- New chapter 6 for mass processing (Central API) created
- Removing scenario where appended fields are not available in complex structure BUS\_EI\_EXTERN

Version 1.00 (June 13<sup>th</sup>, 2016)

## 1 General Information

In SAP Business Suite (ERP 600 and Enhancement Packages), customer master data and vendor master data transactions such as FD01, FD02, FD03, XD01, XD02, XD03, FK01, FK02, FK03, XK01, XK02 and XK03 have been enhanced by customers with additional fields using the Business Add-In (BAI) technology. In the customer and vendor master dialog transactions, these fields were integrated by adding additional sub-screens to the existing screens.

Moving to a SAP S4HANA release, traditional customer/vendor master transactions are made obsolete and replaced by the business partner transaction BP. Because of this, all extension-specific fields have to be integrated into the business partner.

This document provides a guideline how customers can transfer the enhancements they have made in the customer/vendor transactions to transaction BP, so that a maintenance of these fields remains possible after the upgrade to a S4HANA release.

### Important

There are three different scenarios, which have to be considered differently:

- |                    |  |
|--------------------|--|
| <b>Scenario A</b>  | Integration of complete tables in customer namespace that refer to customer / vendor master            |
| <b>Scenario B</b>  | Integration of own appends of customer/vendor master core tables (e.g. appends to tables KNA1 or LFA1) |
| <b>Scenario C*</b> | <i>Integration of a new development into the SAP Business Partner*</i>                                 |

*\*Scenario C is not in scope of this document. New requirements with the goal to enhance master data in S4 releases with own fields should be developed completely within the SAP Business Partner.*

*Please refer to the guidelines for enhancing the SAP Business Partner:*

- *BDT (Business Data Toolset) – Developer’s Manual*
- *Help Portal – Business Partner Extensibility*
- *Easy Enhancement Workbench (EEW)*

*Please find document links to these topics in reference table in chapter 1.3.*

## 1.1 Used tools and frameworks

Enhancements to the SAP Business Partner can be made by using the frameworks / tools mentioned below:

- BDT (Business Data Tool Set) – Enhance dialog screens in transaction BP
- XO Framework (Extensible Objects) – Validate data and store them in memory
- CVI Synchronization – Map the data from BP to customer/vendor master and save them to the database

## 1.2 Idea in a nutshell

The BDT will be used to enhance the existing screens in transaction BP with the additional required fields, tables or checkboxes. The XO framework will be used to validate and to store the data in the memory. BAdI implementations within the CVI synchronization will collect the data from XO memory and save it to the database, in case additional data is not part of the complex interface structure.

For mass processing the “Central API” is available. This interface provides the functionality for creating and updating master data including both core and application data.

## 1.3 Document references

Topic	Reference
BDT (Business Data Tool Set)	BDT Developer’s Manual <a href="#">Link to SAP User Assistance</a>
XO Framework (Extensible Objects)	SAP note “1623809 - Developer documentation for the XO framework” <a href="#">Link to SAP note 1623809</a>
CVI (Customer-Vendor-Integration)	SAP note “956054 - BP_CVI: Customer/vendor integration as of ERP 6.00” <a href="#">Link to SAP note 956054</a> Help Portal <a href="#">Link to SAP Documentation</a>
Easy Enhancement Workbench (EEW)	Help Portal <a href="#">Link to SAP Documentation</a>
Business Partner Extensibility	Help Portal <a href="#">Link to SAP Documentation</a> → Functions → Extensibility

## Table of Contents

<b>1</b>	<b>General Information</b>	<b>2</b>
1.1	Used tools and frameworks	3
1.2	Idea in a nutshell	3
1.3	Document references	3
<b>2</b>	<b>General integration overview</b>	<b>5</b>
<b>2.1</b>	<b>BDT Integration</b>	<b>5</b>
2.2	XO overview	6
2.2.1	Scenario A – Integration of own tables in customer namespace	6
2.2.2	Scenario B – Integration of core table appends	6
2.3	Saving data to the database	6
2.3.1	Scenario A – Saving data in own tables	6
2.3.2	Scenario B – Saving data in core table appends	6
<b>3</b>	<b>Settings in BDT – Business Data Toolset</b>	<b>7</b>
3.1	General information about naming conventions in BDT	7
3.2	Own BDT application – Transaction BUS1	7
3.3	Separate BDT datasets – Transaction BUS23	7
3.4	Registering tables – Transaction BUSG	8
3.5	BDT Assignments of screen->section->view->field group->field – Transaction BUS4	8
3.5.1	BDT screen sequences – Transaction BUS6	8
3.5.2	BDT screens – Transaction BUS5	8
3.5.3	BDT sections – Transaction BUS4	8
3.5.4	BDT views – Transaction BUS3	8
3.5.5	BDT field groups – BUS2	9
3.6	BDT events – BUS7	9
3.7	Additional functions – BUS9	10
3.8	Assignments Dynpro field->DB field and DI field->DB field – BUSB and BUSB_DI	11
3.9	BP_Views – BUSD	11
<b>4</b>	<b>XO Framework – Extensible Objects</b>	<b>12</b>
4.1	Scenario A – Integration of own tables in your namespace	12
4.1.1	Create Memory Object (MO) class	12
4.1.2	Create Persistence Object (PO) class	12
4.1.3	XO Customizing	13
4.2	Scenario B – Integration of core table appends	15
4.2.1	Create Validation Object (VO) classes	15
4.2.2	XO Customizing	18
<b>5</b>	<b>CVI – Saving data to database</b>	<b>19</b>
5.1	Scenario A – Saving data in own tables in your namespace	20
5.2	Scenario B – Saving data in core table appends	22
5.2.1	Foreign key checks of appended fields	22
<b>6</b>	<b>Mass Data Processing – Central API</b>	<b>24</b>
6.1	Scenario A – Processing data of non-core tables	24
6.2	Scenario B – Processing data in core table appends	24
6.3	Data validation	24
6.4	Committing changes and error handling	25
<b>7</b>	<b>Debugging</b>	<b>26</b>

## 2 General integration overview

For details about the Extensible Objects (XO) tool see note 1623809 (the note contains a developer documentation as attachment). The complete XO customizing is maintained in transaction XO80. Only the XO business object type BUSINESS\_PARTNER is relevant for the CVI enhancement.

The purpose of this chapter is to provide an overview over the necessary activities using the tools/frameworks mentioned in chapter 1.1. In chapters 3, 4 and 5 the detailed steps to be executed per tool/framework are described. The relation between the overview chapter and the detail chapters is as follows:

- chapter 2.1 (BDT integration) → details see chapter 3
- chapter 2.2 (XO integration) → details see chapter 4
- chapter 2.3 (Saving data to DB) → details see chapter 5

### 2.1 BDT Integration

This sub-chapter provides the overview of the BDT integration of the CVI. You can find the detailed activities in Chapter 4 below.

The integration of XO into BDT has been implemented in a generic way. The major part of the integration is implemented generically. All further new datasets or core table appends that are integrated into BDT using XO do not need to care about this part. If the integration is done as described below, the framework will take care of the generic logic automatically. Especially most of the BDT events are treated generically by the so-called BDT-adapter. The generic implementation is provided by class

- XO\_BDT\_ADAPTER (generic class for BDT integration).

On top of this, there is class:

- Class CVI\_BDT\_ADAPTER\_INTERN with specific enhancements for CVI Business Partner datasets

Within the BDT adapter classes you find methods (generic\_\*; e.g. GENERIC\_ISDAT or GENERIC\_ISSTA), that are called by the BDT event function modules (XO\_GENERIC\_EVENT\_\* in transaction BUS7). These methods are generically responsible for processing the corresponding events for all Financial Services and CVI datasets. For these events, no individual implementation is necessary.

In addition, there are BDT settings that need to be implemented individually. For example in the corresponding view in BDT, a PBO and a PAI module need to be registered for every view. Within these two function modules the selection of data for this dataset from XO memory as well as the saving of the changed data into XO memory later on are implemented. Further individual BDT settings that need to be considered are for example a new BDT application per extension (separate for customer and vendor enhancements), and of course the complete screen construction (screens, sections, views, field groups, datasets).

In addition dynpros (SE80-screens) need to be implemented in transaction SE80 including PBO and PAI logic within the BDT function group that is responsible for the corresponding BDT application. The already mentioned PBO- and PAI-function modules need to be implemented there as well.

You can find more details including a precise step-by-step description about the BDT implementation in chapter 3 below.

## 2.2 XO overview

In this sub-chapter, a rough overview of the XO implementation is provided. For more details have a look into chapter 4.

### 2.2.1 Scenario A – Integration of own tables in customer namespace

For every dataset (database table) of customer or vendor to be supported in transaction BP, a memory object needs to be assigned in XO customizing. Basic elements of a memory object are the database table that shall be integrated into CVI (which usually already exists) and the class in which the memory of this database table is kept during runtime. This class will not yet be available and has to be implemented inheriting from one of the classes CVI\_MO\_CUSTOMER or CVI\_MO\_VENDOR

Within every memory object class, you need to make sure that:

- for every field for which a validation check is needed a separate method `VALIDATE_<FIELD_NAME>` is created. Of course you can also create validation methods with combined field checks like: If field1 is not initial -> verify field2 is not initial
- method `VALIDATE_INTERN` is redefined in a way that it calls `SUPER->VALIDATE_INTERN` at the beginning and afterwards all individual validation methods that were added in the current inheritance level.

Additionally a persistence object is needed to retrieve data from the database including a reference class representing the persistence object. A persistence object needs to be assigned to every memory object according to the following logic:

- If your table uses KUNNR or LIFNR as key field you can refer to one of the existing persistence objects `CUSTOMER` for table key KUNNR or `VENDOR` for table key LIFNR.
- If the key field in your table is a different one you have to create a new persistence object referring to a new class, which has to inherit from either `CVI_PO_CUSTOMER` or `CVI_PO_VENDOR`.

### 2.2.2 Scenario B – Integration of core table appends

Appends to customer/vendor core tables (KNA1, KNB1, LFA1, LFB1, ...) are treated differently. For these fields the memory object class already exists and it would lead to technical problems if a new enhancement was created. Because of this for the integration of core table appends instead a validation object (VO) represented by a VO-class needs to be implemented in XO. The VO has the purpose of providing static validation methods, which can be used to validate the data of the append fields (e.g. in the corresponding PAI modules). Additionally the validation methods will automatically be called during the overall business partner validation to validate the entries in your fields.

## 2.3 Saving data to the database

### 2.3.1 Scenario A – Saving data in own tables

For saving data stored in your own tables, the CVI synchronization provides several BADIs, which have to be implemented in this scenario. Within these BADI implementations, the data has to be retrieved from the XO memory and written to the database via an update task module.

For more details about the BADI implementations see chapter 5.

### 2.3.2 Scenario B – Saving data in core table appends

In this scenario, you also have to enhance the complex structure `CVIS_EI_EXTERN` of the CVI with your appended fields. Make sure that you have enhanced both the `DATA` and the `DATA_X` structure. In this case, the database persistence is provided generically and there is no additional implementation effort.

## 3 Settings in BDT – Business Data Toolset

The Business Data Toolset (BDT) is used to integrate new fields into the business partner UI (transaction BP). The BDT integration involves the creation of various field groups, screens, views and sections so that the corresponding tabs and the screens appear in transaction BP.

The following aspects need to be considered, when implementing enhancements in BDT (the general transaction for starting the BDT business partner control menu is /NBUPT).

### 3.1 General information about naming conventions in BDT

The central part in the naming for all BDT objects is the application name (<APPL\_NAME>) defined in transaction BUS1 (example: ZCUS).

All datasets, screens, sections and views are named with <APPL\_NAME>number (example: ZCUS01...ZCUS99).

All event function modules should have the following naming:  
APPL\_NAME>\_BUPA\_EVENT\_<EVENT\_NAME> (example: ZCUS\_BUPA\_EVENT\_FCODE).

All PBO/PAI function modules should have the following naming:  
<APPL\_NAME>\_BUPA\_PBO\_<VIEW\_NAME> (example: ZCUS\_BUPA\_PBO\_ZCUS01)  
<APPL\_NAME>\_BUPA\_PAI\_<VIEW\_NAME> (example: ZCUS\_BUPA\_PAI\_ZCUS01).

Rules for the use of position numbers (e.g. when assigning function modules to BDT events)

- The objects assigned by help of position numbers will be processed in the order of the numbering (e.g. the views assigned to a section will be constructed in a way that the view with the lowest number is displayed first and all higher numbers below / same for the event function modules the module with the lowest number assigned to an event is processed first and all others later according to their number)
- Especially for the event function modules conflicts can occur. Function modules from SAP and function modules from in customer name space are maintained and delivered in the same table. As the position number is the main key field of this table it is essential that customers use a position number that cannot be overwritten by SAP customizing. This is especially critical in case you have to add one or more event function modules in transaction BUS7.
- In order to make sure that customer-specific entries are not overwritten by SAP customizing delivery please stick to the following rule:  
Use a position number where one of the last two digits are unequal to 0. All other digits of the position number can then be freely chosen.
- Example: If you would like to register a new FCODE-function module (e.g. ZCUS\_BUPA\_EVENT\_FCODE) you can use one of the following position 3.600.010 or 4.000.001. In the normal use case, it is essential that the SAP standard modules are processed before, so please use position numbers that are higher than the ones of SAP-standard modules that are already registered in this event in transaction BUS7.

### 3.2 Own BDT application – Transaction BUS1

You need to create your own BDT application(s) and assign them to your objects (views, bp views ...). Do not add the generic CVI applications CVIC or CVIV to your objects. You can also decide to split your enhancement into different BDT applications. It necessary to use different BDT applications for customer and vendor enhancements.

### 3.3 Separate BDT datasets – Transaction BUS23

You need to create your own datasets. Do not use the generic CVI datasets CVIC\*/CVIV\*. A separate dataset is needed at least for the differentiation between general data, company code dependent data and sales area dependent data for each customer and vendor master, however you should split your enhancement into different datasets according to the business context. In SAP standard usually there is one dataset per table (although in special cases deviations are possible). All data belonging to one view can only be assigned to the same dataset (because the assignment is made by the dataset assignment in the view).

### 3.4 Registering tables – Transaction BUSG

Every table that is added to CVI needs to be registered in this transaction with the corresponding attributes (change document object, function module to get data, function module to collect data). If for the enhanced table (or set of tables) a change document object exists, this needs to be added in the corresponding field. In case no change document object exists, a new one might be created and integrated.

### 3.5 BDT Assignments of screen->section->view->field group->field – Transaction BUS4

For customer-specific fields always create a new section with its own frame title (maintained in column “Title” in transaction BUS4). In case you have fields that do not belong to the already existing tabs you can create new tabs (screens) and assign them to the corresponding screen sequence (transaction BUS6 – screen sequence BUP001 for general data, screen sequence FS0001 for company code data customer and vendor, screen sequence CVIC01 for sales data customer, screen sequence CVIV01 for sales data vendor).

#### 3.5.1 BDT screen sequences – Transaction BUS6

In case you have a completely separate set of screens (tabs), you can create your own subheader-id (button in the subheader line in transaction BP). For this, you need a new screen sequence, which can be defined here. In addition, the relevant screens need to be assigned to the screen sequence here.

#### 3.5.2 BDT screens – Transaction BUS5

In case you have fields that belong to a screen sequence, but do not fit into any of the existing tabs, you can create a new tab (screen) and assign it to an existing screen sequence. Assign your sections to the corresponding screens also in transaction BUS5.

When creating a new screen, it is necessary to assign the correct header section to this screen. The header section must be the first section in the screen, which means it must have the lowest item number. There are different header sections for general, company code, sales area and purchasing organization data which must be assigned:

Data Segment	Header Section	Header Section
Customer General	BUP009	Header Data (General Screens)
Customer Company Code	FI0201	Header Data (Company Code-Dependent Screens)
Customer Sales Area	CVIC00	Header Data (Sales Area-Dependent Screens)
Vendor General	BUP009	Header Data (General Screens)
Vendor Company Code	FI0201	Header Data (Company Code-Dependent Screens)
Vendor Purchasing Org.	CVIV00	Vendor: Header Data (Purchasing Organization)

#### 3.5.3 BDT sections – Transaction BUS4

For any new field or set of fields, you need to create a section of your own with a speaking frame title (see column “Title” in transaction BUS4). Assign only those views to a section that fit together. If the business meaning is different, create a new section. Assign the views to the sections here as well.

#### 3.5.4 BDT views – Transaction BUS3

The view is the central object in BDT. It contains the connection to the SE80-screen (in a function group) where the input fields are defined. For differentiation between the BDT elements and the SE80 elements, the German name for the SE80-screen, **dynpro**, is used here. The view also contains the PBO/PAI-logic as well as an assignment of application and dataset. In addition, the fields are assigned to the views via the field groups here.

The general rule should be: Do not assign too many field groups to a view. Only assign more than one field group into one view if the fields belong to the same context. If in doubt, create different views for different field groups.



In addition, you need to add a PBO and PAI function module here in which the PBO and PAI logic is processed. The coding in the PBO and PAI-modules can be mainly copied from the standard modules. Compare for example:

- CVIC\_BUPA\_PBO\_CVIC01 / CVIC\_BUPA\_PA\_I\_CVIC01 for general customer data
- CVIC\_BUPA\_PBO\_CVIC15 / CVIC\_BUPA\_PA\_I\_CVIC15 for sales area customer data
- CVIC\_BUPA\_PBO\_CVIC30 / CVIC\_BUPA\_PA\_I\_CVIC30 for company code customer data
- CVIV\_BUPA\_PBO\_CVIV01 / CVIV\_BUPA\_PA\_I\_CVIV01 for general vendor data
- CVIV\_BUPA\_PBO\_CVIV30 / CVIV\_BUPA\_PA\_I\_CVIV30 for company code vendor data
- CVIV\_BUPA\_PBO\_CVIV71 / CVIV\_BUPA\_PA\_I\_CVIV71 for purchasing vendor data

All these modules are constructed in the following way: In PBO the data is read from the XO memory and transferred to the dynpro structure. In PAI module, the changed data is saved back into the XO memory for later saving. In addition, in PBO modules, the texts for F4-fields are also determined and in PAI modules at the end the validation methods from the XO memory or validation object are called to validate the changes.

In addition to the PBO and PAI function modules there are – of course – also the PBO and PAI modules in the dynpro flow logic. Here you need to call the standard function modules provided by BDT:  
BUS\_PBO – to be called in a perform BPO, which in turn is called in the PBO module  
BUS\_PA\_I – to be called in a perform PAI, which in turn is called in the PAI module.  
For an example, check the dynpros in function group CVI\_FS\_UI\_CUSTOMER.

Attention: When you miss to call function modules BUS\_PBO and BUS\_PA\_I in the dynpro flow logic, there will be problems with field modification and the behavior of the fields in dialog (e.g. fields available for input in display mode or similar problems).

### 3.5.5 BDT field groups – BUS2

On field group level the fields from the dynpro are assigned.

General rule: only one field per field group! Only in exceptional cases, you should assign more than one field to one field group. Field modifications are done on the basis of field groups and two fields that belong to the same field group can only be set to “required”, “display”, “hidden”, “change” or “unspecified” together.

In addition, you need to add a function module for FMOD2 event in the field group. If you do not have any specific field modification logic for your field groups you can add the standard FMOD-function modules here:

- CVIC\_BUPA\_EVENT\_FMOD2/CVIV\_BUPA\_EVENT\_FMOD2 for general data
- CVIC\_BUPA\_EVENT\_FMOD2\_SALES/CVIV\_BUPA\_EVENT\_FMOD2\_PORG for sales data/purchasing org data
- CVIC\_BUPA\_EVENT\_FMOD2\_CC / CVIV\_BUPA\_EVENT\_FMOD2\_CC for company code data.

In case you have an additional requirement for field modification setting copy the coding of the corresponding standard module to a function module in your name space and enhance it by your logic.

### 3.6 BDT events – BUS7

The big majority of BDT events is handled automatically by class XO\_BDT\_ADAPTER and corresponding enhancements (classes inheriting from XO\_BDT\_ADAPTER, see chapter 2.1), provided you have implemented your XO-customizing correctly.

You only need to care about BDT events in case you do anything special like for example adding some function codes (buttons) for navigating to a popup or similar. In this case you need a separate event function module (in above example an FCODE-function module) and assign it to the corresponding event (e.g. FCODE).

In addition, it is necessary to implement additional event function modules for displaying the change documents for your table(s) in transaction BP. Please have a look into standard CHGD\* function modules and implement an analogous logic for your tables

- CVIC\_BUPA\_EVENT\_CHGD1 / CVIV\_BUPA\_EVENT\_CHGD1
- CVIC\_BUPA\_EVENT\_CHGD3 / CVIV\_BUPA\_EVENT\_CHGD3
- CVIC\_BUPA\_EVENT\_CHGD4 / CVIV\_BUPA\_EVENT\_CHGD4

For event CHGD2, no function module needs to be assigned. If your tables are correctly integrated in XO, function module XO\_GENERIC\_EVENT\_CHGD2 will process the necessary logic generically.

### 3.7 Additional functions – BUS9

You need this transaction in case you have buttons (function codes) in your application. In order to integrate a function code in the application the following steps are necessary:

1. Enter the function on your dynpro. Naming convention: The name of the push button needs to have the prefix “PUSH\_”. Example: PUSH\_BUTTON1.
2. In the screen painter details in field “FctCode” you need to enter the name of the push button again.
3. In order to react to the selection of the button during runtime by a user you need to implement an FCODE function module (naming convention <APPL\_NAME>\_BUPA\_EVENT\_FCODE) and implement the following coding here

```
"set fcode as processed per default
e_xhandle = true.
case i_fcode.
  when 'BUTTON1'.
    "implement application-specific logic here
  when others.
    clear e_xhandle.
endcase.
```

Please note that you need to remove the prefix “PUSH\_” when catching the function code in your FCODE function module (==> WHEN ‘BUTTON1’ and **not** WHEN ‘PUSH\_BUTTON1’).

4. In order to make sure that the push button can be influenced using the field modification adjustments, create a new field group (transaction BUS2) and assign the button to the field group using the “Field Group -> Fields” sub-menu. The name of the button (including the prefix) needs to be entered in column “field name”. Column “table” remains empty. In addition you should assign an FMOD2-function module according to the rules described in chapter “BDT field groups – BUS2”.
5. You have to create a separate view for the button according to the rules above in chapter “BDT views – BUS3”. Maintain the dynpro in which the button has been implemented and assign the field group created in “4” to the view.
6. Finally, the function code needs to be made known to the BDT. For this it has to be defined in transaction BUS9. The following steps need to be executed for this:
  - a. Press button “New entries”.
  - b. In field “Function code”, enter the name of your function code **without** the prefix (“PUSH\_”).
  - c. In field “Function text”, enter a text for your function code. You can leave field “screen sequence cat.” empty.
  - d. After that, you have to decide if you would like to rely the visibility of the function code on either field group or view. Recommendation: use function “Activate using Status of Field Group” and maintain visibility in section “Active per Status of Field Group” at the end of the page.

To have a reference to copy from you can have a look at the push button PUSH\_CVIS\_SEPA (BUS9), which is assigned to field group 309 and view CVIS01. The corresponding FCODE function module is CVIS\_BUPA\_EVENT\_FCODE.

### 3.8 Assignments Dynpro field->DB field and DI field->DB field – BUSB and BUSB\_DI

You need this transaction in case you have dynpro fields that have a different name than the corresponding database fields. BDT needs to be able to assign the fields. Otherwise for example the required-fields check will not work.

### 3.9 BP\_VIEWS – BUSD

This is the technical view on the business partner roles (see views V\_TB003 / V\_TB003A in transaction SM30). You may think about creating your own roles instead of enhancing the standard roles.

In order to make sure that your enhanced fields are available in the corresponding roles add your datasets and applications to the BP views. Differentiate between customer and vendor roles here.

### 3.10 Field modification status for blocked customer/vendor master

S/4 releases support the blocking of customer and vendor master without blocking the assigned business partner. In such a case where an assigned customer/vendor is blocked while the business partner is not, all users with standard authorization will not have any access to exclusive customer/vendor based data in transaction BP. General data like names or addresses are still accessible and editable in the business partner, but any changes to these data will no longer be synchronized with the blocked customer/vendor master.

A user with “Data Privacy Officer” (DPO) authorizations, who opens a business partner which is not blocked itself but assigned to a blocked customer or vendor in transaction BP, has access to all customer/vendor data fields and sections. When the DPO-user switches transaction BP to change mode, all fields you have added to transaction BP in context of customer-vendor-integration will be editable (please note that changes to these fields will not be transferred to the blocked customer/vendor when saving and will therefore not be saved!). Nevertheless, this is not a consistent UI behavior and you should take the below measures to prevent the fields to be open for change. You need to create a function module which sets the field status of your fields to “display” in that case and assign this function module in BDT customizing, so that it is processed in FMOD1 event.

The function module should have the following naming convention:

<APPL\_NAME>\_BUPA\_EVENT\_FMOD1\_DPP

Execute the following steps:

1. Make sure that SAP notes 2592806 and 2590430 are implemented in your system.
2. Create a new function module taking function module CVIS\_BUPA\_EVENT\_FMOD1\_DPP as a reference. If you copy the code and adjust replace the name of the BDT application to your own BDT application, your field groups will also be switched to display mode in case an assigned customer/vendor is blocked.
3. Afterwards register your function module in BDT:
  - a. Open maintenance view V\_TBZ3Q in SM30 and select application object BUPA
  - b. Add a new entry with
    - i. FGroupCrit: <APPL\_NAME>000
    - ii. Description: CustVEnd Blocked -> Display
    - iii. Read function module: <Name of your function module>
4. Save this entry.

## 4 XO Framework – Extensible Objects

This chapter provides detailed instructions about the integration into the XO framework. If you need any details about XO, please see note 1623809 (contains a developer documentation as attachment).

### XO customizing

For adaption, the XO customizing transaction SE80 is used. The relevant business object type for all customer/vendor data is BUSINESS\_PARTNER.

### 4.1 Scenario A – Integration of own tables in your namespace

This chapter describes the XO-related steps that are necessary to integrate additional (new / customer-owned) tables into transaction BP.

#### 4.1.1 Create Memory Object (MO) class

For each database table that shall be integrated into transaction BP a memory object class is needed. The class has to inherit from class CVI\_MO\_CUSTOMER for customer data or from class CVI\_MO\_VENDOR for vendor data.

Naming convention: CVI\_MO\_<TABLE\_NAME>  
Example: CVI\_MO\_ZCUST1

#### Validation

In the new class, it is necessary to create separate validation methods (static methods with public visibility) for every table field that needs to be validated. You can also create methods to perform combined validations using multiple fields. However, it is strongly recommended to have a low granularity when creating the methods.

In addition, method VALIDATE\_INTERN must be redefined. Within the redefinition first of all method VALIDATE\_INTERN of the super-class has to be called, before all new validation methods within this redefinition are processed. You can compare for example the implementation in method CVI\_MO\_KNA1->VALIDATE\_INTERN. Method VALIDATE\_INTERN is called in an overall check on saving the business partner or when you explicitly press button “check” in business partner maintenance.

In addition, the created validation methods must be called in the corresponding PAI function modules for dialog input validation.

#### Class examples

CVI\_MO\_KNA1, CVI\_MO\_KNB1, CVI\_MO\_LFA1, CVI\_MO\_LFB1

#### 4.1.2 Create Persistence Object (PO) class

If your table uses KUNNR or LIFNR as key field, no separate PO-class is needed. You only need to assign the existing persistence object CUSTOMER or VENDOR to your memory object in the next section.

If the key field of your table is not KUNNR or LIFNR you have to create a separate persistence object class. The new class has to inherit from CVI\_PO\_CUSTOMER or CVI\_PO\_VENDOR. Now perform the following steps:

1. Redefine method IF\_XO\_PERSISTENCE\_OBJECT~READ\_DATA.  
Copy the coding from the inherited method (superclass method).  
In the copied coding, change the select statement at the end according to your table key.
2. Redefine method IF\_XO\_PERSISTENCE\_OBJECT~SORT\_DATA\_BY\_KEY.  
Copy the coding from the inherited method (superclass method).  
Exchange the name of the assigned component ‘KUNNR’ / ‘LIFNR’ with the key field of your table at the end of the method.

### 4.1.3 XO Customizing

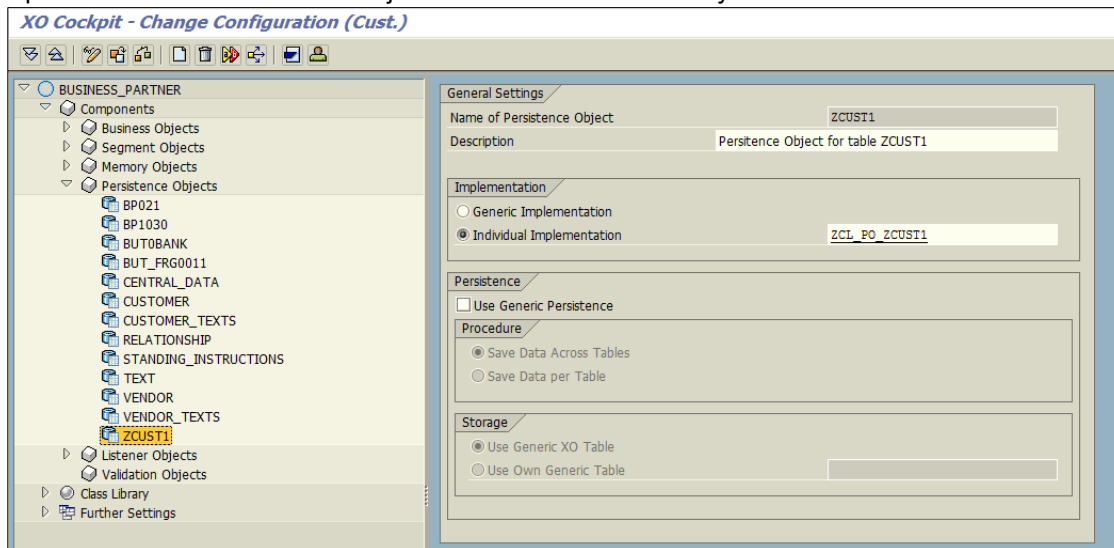
Execute the following steps to maintain the XO customizing for your objects:

1. Start transaction XO80.
2. Select Business Object Type BUSINESS\_PARTNER from the initial popup.
3. Switch to change mode.
4. Make sure that you are in mode “Individual Settings” (ctrl+F4, 11<sup>th</sup> button in button line)

#### a) Persistence Object

This step is only needed when you have created an own persistence object in chapter 4.1.2. If you use one of the standard persistence objects CUSTOMER or VENDOR, you can skip this step.

Open tree node “Persistence Objects” and create a new entry.

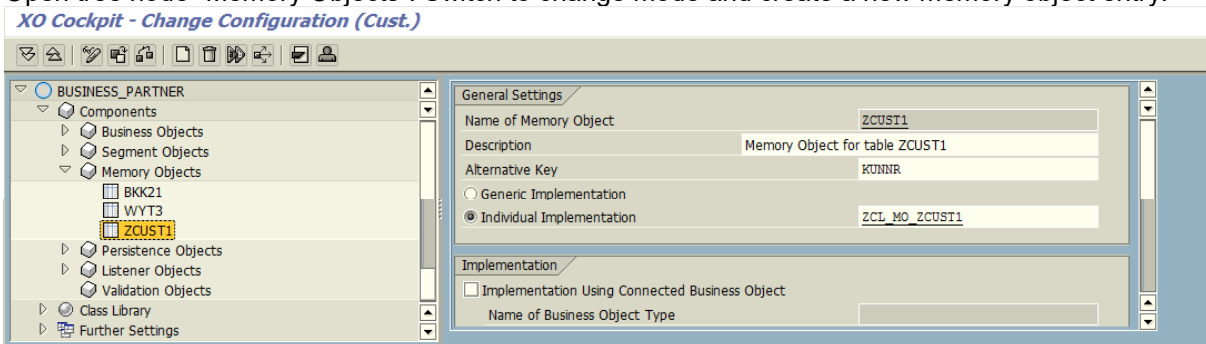


- Name of Persistence Object = can be chosen arbitrarily
- Description = can be chosen arbitrarily
- Individual Implementation = Name of persistence object class created in step 4.1.2

Save this step.

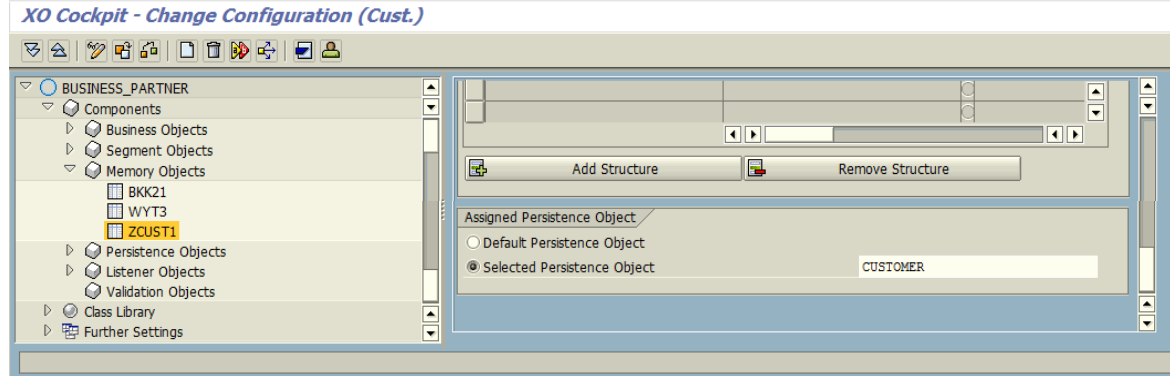
#### b) Memory Object

Open tree node “Memory Objects”. Switch to change mode and create a new memory object entry.



- Name of Memory Object = Name of your table name (as in SE11)
- Description = can be chosen arbitrarily
- Alternative Key = KUNNR or LIFNR or other key field name of your table
- Individual Implementation = Name of memory object class created in step 4.1.1.

Scroll down to...

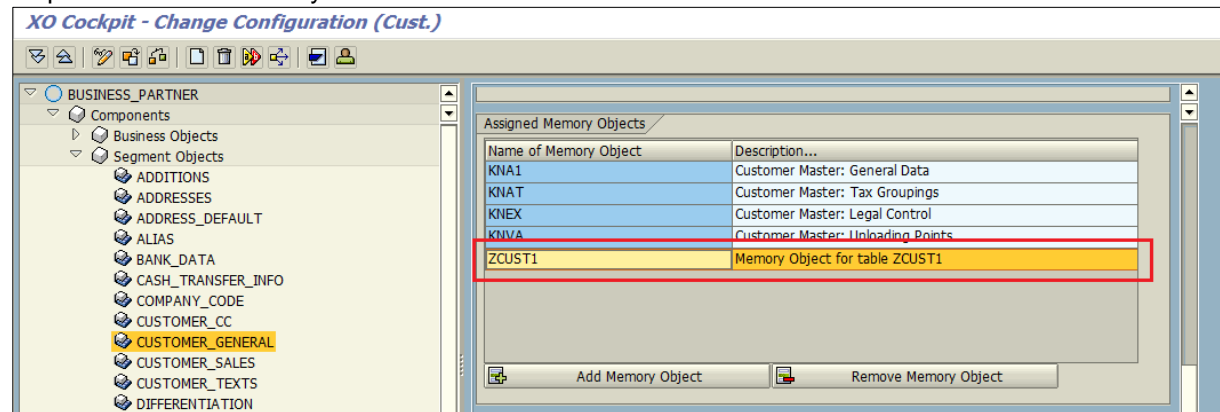


Selected Persistence Object = CUSTOMER or VENDOR when your table uses KUNNR or LIFNR or name of your own persistence object from last step

Save this step.

### c) Segment Object assignment

After creating the memory object this has to be assigned to the correct segment object. Open tree node “Segment Objects” and assign your memory object to the segment object by drag and drop or button functionality.



The correct segment object depends on the customer/vendor segment to which your table belongs:

CUSTOMER_GENERAL	= General customer data	– e.g. KNA1, KNAT
CUSTOMER_CC	= Company Code data	– e.g. KNB1, KNB5
CUSTOMER_SALES	= Sales Area data	– e.g. KNVV, KNVI
VENDOR_GENERAL	= General vendor data	– e.g. LFA1, LFAT
VENDOR_CC	= Company Code data	– e.g. LFB1, LFB5
VENDOR_PURCHASE	= Purchasing Org. data	– e.g. LFM1, LFM2

Save this step.

## 4.2 Scenario B – Integration of core table appends

All appends of customer/vendor core tables like KNA1, KNB1, LFA1 or LFB1 are available in the corresponding XO memory object out of the box. There is no need to create any new memory object or any new class. The only thing to do here is to implement the validation logic for your appended fields. Needed validations will be handled in a validation class, which is registered in XO customizing as a so-called Validation Object (VO).

### 4.2.1 Create Validation Object (VO) classes

The maintenance of new fields from the customer/vendor core tables is already possible after integrating the fields into BDT and making sure that the changes are transferred to XO-memory in the PAI function modules. The only missing part is the validation of the new fields. In order to implement the validation the following steps need to be processed. As an example we refer to table KNA1 in the below screen shots. If you integrate appends of another core table unequal KNA1 replace objects, parameters and variable names accordingly.

1. With SAP note 2295823 class CVI\_FS\_ENH\_VO\_TEMPLATE is available in the system. It is recommended to create a copy of this class to implement an own Validation Object class.
2. Create an implementation for method IF\_XO\_VALIDATION\_OBJECT~ON\_VALIDATE\_ME. Refer to the implementation in class CVI\_FS\_ENH\_VO\_TEMPLATE as an entry point. This example refers to appends on table KNA1 and has to be adjusted with the name of the table which was extended with your append.

Example code from CVI\_FS\_ENH\_VO\_TEMPLATE ->ON\_VALIDATE\_ME :

```

Class Builder: Class CVI_FS_ENH_VO_TEMPLATE Change
-----
Ty. Parameter Typing Descri...
-----
> SENDER LIKE

Method IF_XO_VALIDATION_OBJECT~ON_VALIDATE_ME Active
-----
1 method if_xo_validation_object-on_validate_me.
2
3 data:
4 lr_so type ref to xo_segment_object,
5 lr_mo type ref to xo_memory_object,
6 lv_partner type bu_partner,
7 lt_results type tty_xo_message,
8 lt_results_all type tty_xo_message,
9
10 lv_table_name type xo_table_name value 'KNA1', "---- replace with correct table name
11 lt_data_new type table of knal, "---- replace with correct table name
12 lt_data_old type table of knal. "---- replace with correct table name
13
14 field-symbols:
15 <result> type str_xo_message.
16
    
```

1. Use the correct table type of the customer/vendor table for these variables

- If you need further data from other memory objects (database tables) have a look at section “Further data needed...” in the example method above:

```

Class Builder: Class CVI_FS_ENH_VO_TEMPLATE Change
-----
Ty. Parameter      Typing      Descri...
-----
x SENDER           LIKE        [Empty]

Method IF_XO_VALIDATION_OBJECT-ON_VALIDATE_ME Active
-----
32
33 *****
34 * Further data needed for validation? Example KNVP
35 *****
36 data:
37   lt_knvp_new type table of knvp,           "<--- replace with correct table name
38   lt_knvp_old type table of knvp.         "<--- replace with correct table name
39
40   lv_table_name = 'KNVP'.                 "<--- replace with correct table name
41
42   lr_mo ?= fshp_memory_factory=>get_instance( i_partner = lv_partner i_table_name = lv_table_name ).
43   lr_mo->get_data(
44     importing
45       e_data_new = lt_knvp_new           "<-- contains new data (updated data)
46       e_data_old = lt_knvp_old         "<-- contains old data (database level)
47   ).
48
    
```

2. Use the correct table type of the customer/vendor table for these variables

- The example code calls three static methods VALIDATE\_KNA1\_FIELD1/2/3 for performing the field validations. If possible, it is recommended to implement one single method for each appended field to be validated.

Example code:

```

Class Builder: Class CVI_FS_ENH_VO_TEMPLATE Change
-----
Ty. Parameter      Typing      Description
-----
x I_KNA1           TYPE CVIS_KNA1_T OPTIONAL  Customer Master (General Part)
x I_KNA1_LINE     TYPE KNA1 OPTIONAL        General Data in Customer Master
value( R_RESULTS ) TYPE TTY_XO_MESSAGE        Messages

Method VALIDATE_KNA1_FIELD1 Active
-----
1 method validate_kna1_field1.
2
3 data:
4   lt_kna1 type table of kna1,
5   lt_result type tty_xo_message,
6   lv_error type c.
7
8 field-symbols:
9   <result> like line of lt_result,
10  <kna1> type kna1.
11
12 * Combine imported data
13 lt_kna1 = i_kna1.
14 if i_kna1_line is not initial.
15   append i_kna1_line to lt_kna1.
16 endif.
17
18 * Check we have data to validate
19 check lt_kna1 is not initial.
20
21 * Perform validation
22 loop at lt_kna1 assigning <kna1>.
23   "Do your validation here and set lv_error = true if validation fails
24   "
25   "
26   check lv_error = true.
27
28   append initial line to lt result assigning <result>.
29   "Raise message into <result>-message
30   "Example: message id 'FSBP' type 'E' number '000' with 'Error Message from VO method' into <result>-message.
31
32   <result> = xo_services=>new_message(
33     i_from_system = true
34     i_table       = 'KNA1'
35     i_field       = 'FIELD1' ).
36
37   clear lv_error.
38 endloop.
    
```

- Adjust all parameters and used variables accordingly to the appended core table
- Implement your validation logic here and set lv\_error = true in error case
- Append error message to <result>-message



5. Call all created static validation methods in the interface method IF\_XO\_VALIDATION\_OBJECT~ON\_VALIDATE\_ME like it is done in the example class.
6. Now implement method IF\_XO\_VALIDATION\_OBJECT~REGISTER\_HANDLER. Simply take over the example code. No adjustment of this code is necessary.

Example code:

The screenshot shows the SAP Class Builder interface for the class `CVI_FS_ENH_VO_TEMPLATE`. The class is in the state of 'Change'. The interface `IF_XO_VALIDATION_OBJECT` is visible, with parameters `I_SENDER` (TYPE REF TO IF\_XO\_VALIDATION\_TARGET) and `I_ACTIVATION` (TYPE XO\_BOOLE DEFAULT 'X'). The method `IF_XO_VALIDATION_OBJECT~REGISTER_HANDLER` is active and contains the following code:

```

1  method if_xo_validation_object~register_handler.
2
3      set handler:
4          on_validate_me for i_sender activation i_activation.
5
6  endmethod.
    
```

The created single static validation methods from step 4. shall be called in your PAI modules for the on screen validation within the BP dialogue when data is entered by the user.

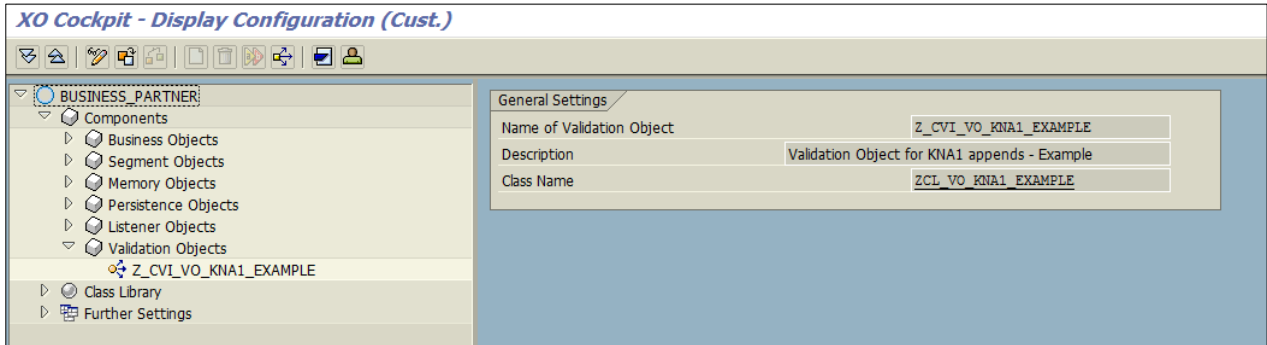
Please also have look at SAP note 2293713 if want to integrate foreign key checks provided by classes `CMD_EI_API_CHECK` and `VMD_EI_API_CHECK`.

**Attention** Make sure that your implementation does not contain any BDT specific logic, like calling methods from class `CVI_BDT_ADAPTER ADAPTER` or calling function module `BUS_MESSAGE_STORE` to process an error message! Validation Objects are also processed in mass processes (Central API), so that BDT-specific coding could lead to a short-dump during runtime.

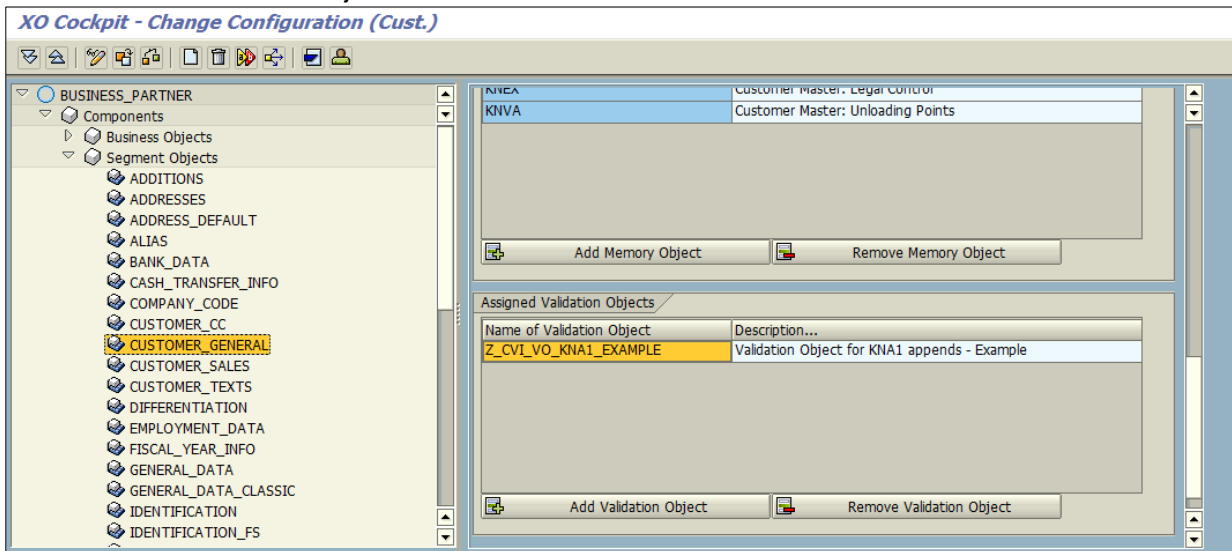
### 4.2.2 XO Customizing

In the XO customizing (transaction XO80) open business object type BUSINESS\_PARTNER.

Open tree node “Validation Objects” and create a new validation object referring to your validation class.



Open the corresponding segment object (for KNA1 this is CUSTOMER\_GENERAL) and add the created validation object.



The correct segment object depends on the customer/vendor segment to which the memory object of the appended table is assigned:

CUSTOMER_GENERAL	= General customer data	– e.g. appends on KNA1, KNAT
CUSTOMER_CC	= Company Code data	– e.g. appends on KNB1, KNB5
CUSTOMER_SALES	= Sales Area data	– e.g. appends on KVVV, KNVI
VENDOR_GENERAL	= General vendor data	– e.g. appends on LFA1, LFAT
VENDOR_CC	= Company Code data	– e.g. appends on LFB1, LFB5
VENDOR_PURCHASE	= Purchasing Org. data	– e.g. appends on LFM1, LFM2

## 5 CVI – Saving data to database

While maintaining business partner data in transaction BP the data are only retrieved and transferred back to the XO memory and validations are executed. Saving of changed customer/vendor data to database is collectively triggered on commit after the button “Save” has been pressed. The reason for this is that the customer/vendor data need to be saved together (either save all data or no data). As the central data needs to be mapped from the corresponding business partner structure before they are available in the customer/vendor structure, the data can only be saved to database after the mapping has been performed. Mapping and saving to database is triggered by the business partner outbound processing which starts the synchronization

The CVI offers two enhancement spots for saving data:

CUSTOMER\_EXTENSION and VENDOR\_EXTENSION.

These enhancement spots contain several BAdI definitions, which provide initialize, validate and save functionality.

	CUSTOMER_EXTENSION	VENDOR_EXTENSION
BAdI	CUSTOMER_EXTENSION_AUTH_CHECK	VENDOR_EXTENSION_AUTH_CHECK
	CUSTOMER_EXTENSION_CHECK	VENDOR_EXTENSION_CHECK
	CUSTOMER_EXTENSION_COMPLETE	VENDOR_EXTENSION_COMPLETE
	CUSTOMER_EXTENSION_INITIALIZE	VENDOR_EXTENSION_INITIALIZE
	CUSTOMER_EXTENSION_OUTBOUND	VENDOR_EXTENSION_OUTBOUND
	CUSTOMER_EXTENSION_UPDATE	VENDOR_EXTENSION_UPDATE

### How-to recommendation

It is necessary to create one enhancement implementation only for each enhancement spot. Each of these two enhancement implementations should use only one single implementing class containing implementations for all BAdIs that are marked in red in the above table. So in case you integrate both customer and vendor data in your application you will have only two enhancement implementations and only two implementing classes after finishing the implementation.

Both classes must contain at least one public static attribute to store KUNNR / LIFNR.

### Example

A created enhancement implementation containing all three BAdIs should look like this:

The screenshot shows the SAP Enhancement Implementation 'ZCF\_CUSTOMER\_EXTENSION\_EXAMPLE' in 'Change' mode. The 'Enh. Implementation Elements' tab is active, displaying a tree view of BAdI implementations and their corresponding methods in the 'Implementing Class'.

BAdI Implementation	Description	Method	Short Description
ZCF_CUSTOMER_COMPLETE	BAdI COMPLETE - Example Implementation	IF_EX_CUSTOMER_EXTENSION_COMPL~COMPLETE	Data Completion
ZCF_CUSTOMER_INITIALIZE	BAdI INITIALIZE - Example Implementation	IF_EX_CUSTOMER_EXTENSION_INITI~INITIALIZE	Data Initialization
ZCF_CUSTOMER_UPDATE	BAdI UPDATE - Example Implementation	IF_EX_CUSTOMER_EXTENSION_UPDAT~UPDATE_MODU...	Data Update

## 5.1 Scenario A – Saving data in own tables in your namespace

In this scenario, the data of the newly integrated tables need to be retrieved from the memory object and then saved to database. To save the data into the corresponding database table BAdIs CUSTOMER\_EXTENSION\_UPDATE or VENDOR\_EXTENSION\_UPDATE are needed.

In addition it is necessary to implement BAdIs CUSTOMER\_EXTENSION\_COMPLETE respectively VENDOR\_EXTENSION\_COMPLETE. The reason for this is that the UPDATE BAdIs are called without KUNNR / LIFNR of the currently processed customer/vendor. For supporting mass data processing, it is important to make sure that the UPDATE BAdIs only save the data of the currently processed customer/vendor.

With SAP note 2295823 class CVI\_FS\_ENH\_CUSTOMER\_BADI\_SCENA is available in the system containing implementation examples for all three BAdI definitions. This class should be used as copy source for creating own implementations.

Interface	Abstract	Final	Model...	Description
IF_EX_CUSTOMER_EXTENSION_UPDAT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Interface for BAdI: CUSTOMER_EXTENSION_UPDATE
IF_EX_CUSTOMER_EXTENSION_INITII	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Interface for BAdI: CUSTOMER_EXTENSION_INITIALIZE
IF_EX_CUSTOMER_EXTENSION_COMPL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Interface for BAdI: CUSTOMER_EXTENSION_COMPLETE
IF_BADI_INTERFACE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Tag Interface for BAdIs

### 1. Implementation CUSTOMER\_EXTENSION\_COMPLETE / VENDOR\_EXTENSION\_COMPLETE

To have the current customer/vendor number available in the update BAdIs implement the following logic in your implementation of CUSTOMER\_EXTENSION\_COMPLETE respectively VENDOR\_EXTENSION\_COMPLETE.

1. Read customer/vendor number from the transferred structure CS\_CUSTOMER / CS\_VENDOR
2. Store customer/vendor number in a corresponding static attribute of your implementing class
3. Read customer/vendor number in your implementation of BAdI CUSTOMER\_EXTENSION\_UPDATE / VENDOR\_EXTENSION\_UPDATE from the corresponding attribute of your implementing class

Example from CVI\_FS\_ENH\_CUSTOMER\_BADI\_SCENA->IF...~COMPLETE

Ty.	Parameter	Typing	Description
	CS_ERROR	TYPE CVIS_MESSAGE	Error Indicator and System Messages
	CS_CUSTOMER	TYPE CMDS_CUSTOMER_S	Customer Data

Method	IF_EX_CUSTOMER_EXTENSION_COMPL~COMPLETE	Active
1	<code>method if_ex_customer_extension_compl~complete.</code>	
2		
3	<code>if cs_customer-kna1-new_data-kunnr is not initial.</code>	
4	<code>    kunnr = cs_customer-kna1-new_data-kunnr.</code>	
5	<code>endif.</code>	
6		
7	<code>endmethod.</code>	

Remark: This BAdI will be called multiple time in case of simultaneous changes on different data segments (KNA1, KNB1, KNVV ...). But only in the first call you can rely on a filled KNA1 / LFA1 segment containing the needed KUNNR / LIFNR. Therefore the check that KUNNR / LIFNR is not initial is needed here.

## 2. Implementation CUSTOMER\_EXTENSION\_UPDATE / VENDOR\_EXTENSION\_UPDATE

In the implementation of method IF...~UPDATE\_MODULES execute the following steps:

1. Retrieve the data from the corresponding Memory Object
2. Ensure that you process only data of the current customer/vendor
3. Call your database update function module in update task.
4. In addition the creation of change documents has to be triggered here.

**Attention:** Database updates must be performed in update task!

Example from CVI\_FS\_ENH\_CUSTOMER\_BADI\_SCENA->IF...~UPDATE\_MODULES

```

Class Builder: Class CVI_FS_ENH_CUSTOMER_BADI_SCENA Change
-----
Ty. Parameter      Typing      Description
-----
CS_ERROR          TYPE CVIS_MESSAGE  Error Indicator and System Messages

Method IF_EX_CUSTOMER_EXTENSION_UPDATE~UPDATE_MODULES      Active
-----
1  method if_ex_customer_extension_updat-update_modules.
2
3
4  data:
5  lv_table_name type fsbp_table_name value 'ZCUST1',  <!-- replace with name of your table  1.
6  lt_data_new   type table of zcust1,                <!-- replace with name of your table
7
8  lv_kunnr      type kunnr,
9  lt_data_old   like lt_data_new,
10 lt_inserts    like lt_data_new,
11 lt_updates    like lt_data_new,
12 lt_deletes    like lt_data_new,
13 lt_unchanged  like lt_data_new.
14
15 * Retrieve current kunnr
16 lv_kunnr = kunnr.
17
18 check lv_kunnr is not initial.
19
20 * Retrieve data from XO Memory Object
21 fsbp_memory_factory->get_data_all(

```

1. Use the table type of your own table for these variables

The provided example coding will determine which entries have to be inserted, updated or deleted from the database.

## 5.2 Scenario B – Saving data in core table appends

You have to extended the complex structure CVIS\_EI\_EXTERN with your appended fields in the structures DATA and DATAX in the corresponding core table section.

After that, there is nothing more to do here. Since all of your appended fields exist now in CVIS\_EI\_EXTERN these will be processed by the CVI standard functionality.

### Remark

With implementation of SAP note 2295823 class CVI\_FS\_ENH\_CUSTOMER\_BADI\_SCENB might be available in your system delivered. This was needed for an earlier enhancement concept supporting a constellation where appended fields are not added to the complex structure CVIS\_EI\_EXTERN. Since this concept is no longer recommended, this example coding is not needed any more.

### 5.2.1 Foreign key checks of appended fields

If your appended fields could have a foreign key check definition, you have to implement the corresponding BAdI so that it will be performed automatically.

CVI_CUSTOM_MAPPER_ENH		
BAdI Method	MAP_CUSTOMER_FOREIGN_KEY_TABLE	Map customer relevant foreign key check data
	MAP_VENDOR_FOREIGN_KEY_TABLE	Map vendor relevant foreign key check data

### Details on Foreign key check enablement

#### Requirement

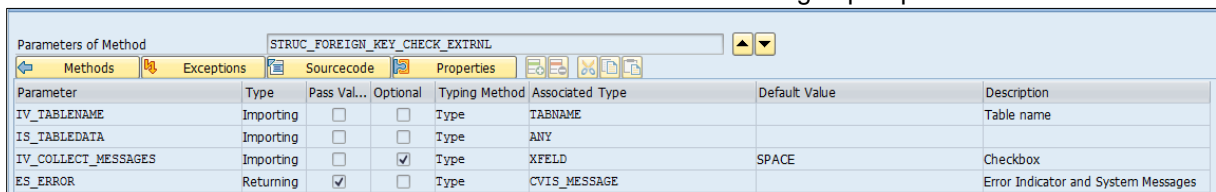
Industry specific CVI enhancements need to perform a foreign key check at the PAI level for the fields of their application. There is no public method available which executes the foreign key check.

#### Solution

For processing generic foreign key checks in each of the two classes VMD\_EI\_API\_CHECK and CMD\_EI\_API\_CHECK for vendor master and customer master the following two methods have been introduced:

STRUC\_FOREIGN\_KEY\_CHECK\_EXTRNL and FOREIGN\_KEY\_CHECK\_EXTERNAL.

1. Method STRUC\_FOREIGN\_KEY\_CHECK\_EXTRNL: This method is used to perform foreign key check on one line of a database table. This method has the following import parameters:

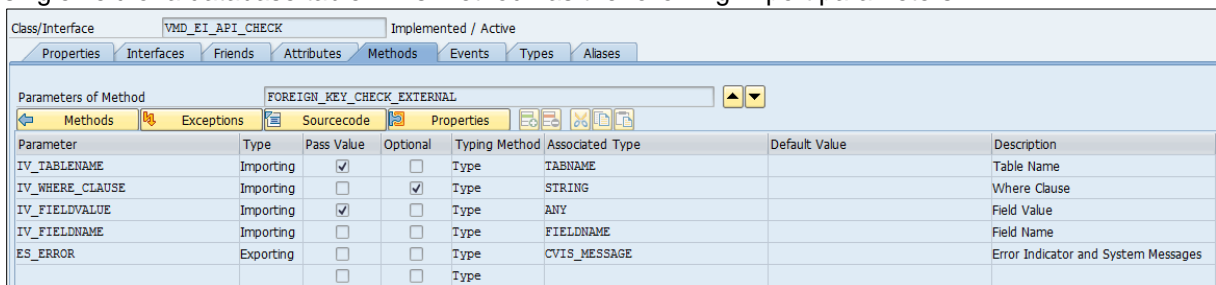


Parameter	Type	Pass Val...	Optional	Typing Method	Associated Type	Default Value	Description
IV_TABLENAME	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	TABNAME		Table name
IS_TABLEDATA	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	ANY		
IV_COLLECT_MESSAGES	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	XFELD	SPACE	Checkbox
ES_ERROR	Returning	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Type	CVIS_MESSAGE		Error Indicator and System Messages

IV\_TABLENAME: Table name as string.  
 IS\_TABLEDATA: One line of table IV\_TABLENAME.

This method calls the FOREIGN\_KEY\_CHECK\_EXTERNAL method internally transferring the where clause (in parameter IV\_WHERE\_CLAUSE).

2. FOREIGN\_KEY\_CHECK\_EXTERNAL: This method is used to perform foreign key check on a single field of a database table. This method has the following import parameters:



Parameter	Type	Pass Value	Optional	Typing Method	Associated Type	Default Value	Description
IV_TABLENAME	Importing	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Type	TABNAME		Table Name
IV_WHERE_CLAUSE	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	STRING		Where Clause
IV_FIELDVALUE	Importing	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Type	ANY		Field Value
IV_FIELDNAME	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	FIELDNAME		Field Name
ES_ERROR	Exporting	<input type="checkbox"/>	<input type="checkbox"/>	Type	CVIS_MESSAGE		Error Indicator and System Messages

- IV\_TABLENAME: Table name as string.
- IV\_WHERE\_CLAUSE: Optional parameter that must not be provided when called from outside the class. This parameter is only provided when the method is called from method STRUC\_FOREIGN\_KEY\_CHECK\_EXTRNL.
- IV\_FIELDNAME: Name of the field in table <IV\_TABLENAME> for which the foreign key check has to be performed.
- IV\_FIELDVALUE: Value of the field in table <IV\_TABLENAME> for which the foreign key check has to be performed.

## 6 Mass Data Processing – Central API

Maintaining new integrated fields in mass processing can be done by calling method CL\_MD\_BP\_MAINTAIN=>MAINTAIN.

Before passing data to CL\_MD\_BP\_MAINTAIN=>MAINTAIN it is highly recommended to perform the pre-requisite field checks with method CL\_MD\_BP\_MAINTAIN=>VALIDATE\_SINGLE. To get all the relevant validation checks the BP grouping in the BP complex structure and the account grouping in the customer/vendor complex structure have to be transferred along with all the other key data.

Again, the same two scenarios as in dialog processing need to be distinguished:

### 6.1 Scenario A – Processing data of non-core tables

The following steps need to be executed:

- Call method CL\_MD\_BP\_MAINTAIN=>MAINTAIN\_NON\_CORE\_VALUE with the table name and the data. This method converts the transferred data into a data type (RAWSTRING) which can be transferred to method CL\_MD\_BP\_MAINTAIN=> MAINTAIN. The converted data is returned in parameter ET\_TABLE\_EXT.
- The mass processing is started by calling method CL\_MD\_BP\_MAINTAIN=> MAINTAIN. Before calling this method the converted data (ET\_TABLE\_EXT) needs to be integrated into the complex structure that is transferred to CL\_MD\_BP\_MAINTAIN=>MAINTAIN (sub-structure CVIS\_EI\_EXTERN-EXT\_APPL\_DATA)
- The transferred data of application owned tables will be automatically saved in the XO memory and can be retrieved in the update BAdI later on as described in section 5.1.

### 6.2 Scenario B – Processing data in core table appends

If your fields are available in the complex structure you can transfer them directly to method CL\_MD\_BP\_MAINTAIN=>MAINTAIN to update your fields.

### 6.3 Data validation

For both scenarios A and B you have to create an implementation of the corresponding Check-BAdI in an enhancement implementation that belongs to your area. In the following table, you find the available BAdIs that can be implemented for the two enhancement spots CUSTOMER\_EXTENSION and VENDOR\_EXTENSION.

	CUSTOMER_EXTENSION	VENDOR_EXTENSION
BAdI	CUSTOMER_EXTENSION_AUTH_CHECK	VENDOR_EXTENSION_AUTH_CHECK
	<b>CUSTOMER_EXTENSION_CHECK</b>	<b>VENDOR_EXTENSION_CHECK</b>
	CUSTOMER_EXTENSION_COMPLETE	VENDOR_EXTENSION_COMPLETE
	CUSTOMER_EXTENSION_INITIALIZE	VENDOR_EXTENSION_INITIALIZE
	CUSTOMER_EXTENSION_OUTBOUND	VENDOR_EXTENSION_OUTBOUND
	CUSTOMER_EXTENSION_UPDATE	VENDOR_EXTENSION_UPDATE



Example taken from the customer extension BAdI:

Ty.	Parameter	Typing	Description
▶	IS_CUSTOMER_EXT	TYPE CMDS_EI_EXTERN	Complex External Interface for Customers
▶▶	CS_ERROR	TYPE CVIS_MESSAGE	Error Indicator and System Messages

Method	IF_EX_CUSTOMER_EXTENSION_CHECK-CHECK	Active
1	<code>method IF_EX_CUSTOMER_EXTENSION_CHECK-CHECK.</code>	
2	<code>* Interface</code>	
3	<code>* IS_CUSTOMER_EXT TYPE CMDS_EI_EXTERN</code>	
4	<code>* CS_ERROR TYPE CVIS_MESSAGE</code>	

Call your validation methods within your implementation to validate your data that are passed in parameter IS\_CUSTOMER\_EXT. In case of an error, fill the returning parameter CS\_ERROR accordingly with the error indicator and the error message(s) to be returned.

Please also have look at SAP note 2293713 if want to integrate foreign key checks provided by classes CMD\_EI\_API\_CHECK and VMD\_EI\_API\_CHECK.

## 6.4 Committing changes and error handling

In order to execute the data changes and to trigger the database update, function module BAPI\_TRANSACTION\_COMMIT must be called after the call of method CL\_MD\_BP\_MAINTAIN=> MAINTAIN. A simple COMMIT WORK is not sufficient, because it does not refresh the buffers. Since the CVI is running on commit, all errors raised during synchronization will create an entry in the PPO (Post-Processing-Office), in case PPO is set to active in customizing, or lead to a short dump in case PPO is set to inactive. In case of an error with active PPO, the BP will be saved, whereas the corresponding customer/vendor data will not be saved/updated (which means a data inconsistency between the objects until the error has been solved and the changes have been reloaded). In case of a short dump due to inactive PPO, all changes will be rolled back and no changes will be saved in any of the objects.

In case PPO is active, you can call method CL\_MD\_BP\_MAINTAIN-> GET\_PPO\_MESSAGES after function module BAPI\_TRANSACTION\_COMMIT has been executed. This method will return all logged PPO entries of the current LUW in parameter E\_RETURN.

There is also transaction MDS\_PPO2 to display the created PPO entries.

## 7 Debugging

Places to have a look in debugger, when problems occur:

**Class CVI\_BDT\_ADAPTER\_INTERN** (or corresponding sub-classes): set breakpoints in methods “event\_\*”, for example:

- EVENT\_DSAVC: xo event finalize is processed to replace the preliminary number ##1 by the final number
- EVENT\_DSAVE: xo-events ON\_COMMIT\_START and ON\_COMMIT\_END are triggered (responsible for saving the data to database)  
-> Exception: for CVI datasets the saving to database is done in CVI BAdIs
- EVENT\_DLVE2: xo event cleanup is triggered to clear the complete memory and all object instances (in dialog in transaction BP the XO\_EVENT is registered on commit level 99, so that it is processed after the synchronization has been finished – otherwise the memory-object methods ON\_DELIVER\_CUSTOMER\_DATA data would not return any data when called during mapping)

**XO\_BUSINESS\_FACTORY=>GET\_INSTANCE**: here the XO-instance for the current business partner is created (or returned)

**Memory object classes** (like CVI\_MO\_KNA):

- in method GET\_DATA(\_NEW) the data are read from memory, on first call the data are selected from database by communication with persistence layer
- in method SET\_DATA(\_NEW) changed data are transferred to the memory for later saving
- in methods VALIDATE\_\* the validations for the different fields of the corresponding database table are processed
- in method VALIDATE\_INTERN all VALIDATE\_\* methods are processed for an overall check of all fields of the current table

**Mapping** (is processed on commit!)

- class CVI\_MAPPER triggers the mapping (Customer: Method MAP\_BPS\_TO\_CUSTOMERS, Vendor: MAP\_BPS\_TO\_VENDORS)
- actual mapping of different datasets is done in class CVI\_FM\_BP\_CUSTOMER
- Interesting method in class CVI\_FM\_BP\_CUSTOMER is GET\_ENHANCEMENT\_DATA, here event DELIVER\_CUSTOMER\_DATA is raised, on which all memory object classes of CVI-standard have registered themselves; as a consequence in all memory classes the method ON\_DELIVER\_CUSTOMER\_DATA is triggered, by this the data from business partner XO memory is transferred to the complex customer structure

Logic for vendor is analogous in class CVI\_FM\_BP\_VENDOR

**Save customer/vendor master data to database**

- method CMD\_EI\_API= >MAINTAIN / VMD\_EI\_API= >MAINTAIN  
Core tables are saved here automatically
- Implementation in Enhancement Spots CUSTOMER\_EXTENSION and VENDOR\_EXTENSION