Infor LN Extensions Development Guide

Release 10.6.x

Important Notices

The material contained in this publication (including any supplementary information) constitutes and contains confidential and proprietary information of Infor.

By gaining access to the attached, you acknowledge and agree that the material (including any modification, translation or adaptation of the material) and all copyright, trade secrets and all other right, title and interest therein, are the sole property of Infor and that you shall not gain right, title or interest in the material (including any modification, translation or adaptation of the material) by virtue of your review thereof other than the non-exclusive right to use the material solely in connection with and the furtherance of your license and use of software made available to your company from Infor pursuant to a separate agreement, the terms of which separate agreement shall govern your use of this material and all supplemental related materials ("Purpose").

In addition, by accessing the enclosed material, you acknowledge and agree that you are required to maintain such material in strict confidence and that your use of such material is limited to the Purpose described above. Although Infor has taken due care to ensure that the material included in this publication is accurate and complete, Infor cannot warrant that the information contained in this publication is complete, does not contain typographical or other errors, or will meet your specific requirements. As such, Infor does not assume and hereby disclaims all liability, consequential or otherwise, for any loss or damage to any person or entity which is caused by or relates to errors or omissions in this publication (including any supplementary information), whether such errors or omissions result from negligence, accident or any other cause.

Without limitation, U.S. export control laws and other applicable export and import laws govern your use of this material and you will neither export or re-export, directly or indirectly, this material nor any related materials or supplemental information in violation of such laws, or use such materials for any purpose prohibited by such laws.

Trademark Acknowledgements

Publication Information

Release: Infor LN 10.6.x
Publication Date: July 10, 2018
Document code: ln_10.6.x_lnstudextdg__en-us

Contents

About this guide

This guide describes the Extensibility features of Infor Enterprise Server 10.5.2. To be able to use all features you must at least apply KB 1860730.

Intended audience

This guide is intended for IT professionals working in implementation projects or IT optimization phases for Infor LN. Basic knowledge about the Infor LN software structure and Infor LN's 4GL programming language is a pre-requisite.

Related documents

You can find the documents in the product documentation section of the Infor Xtreme Support portal, as described in "Contacting Infor".

- *Infor ES Programmer's Guide*
- *Infor LN Studio Application Development Guide*
- *Infor LN Studio Integration Development Guide*
- *Infor LN Document Output Management User Guide*
- *Infor LN UI Infor Ming.le-LN Plug-in User Guide*
- *Infor Enterprise Server Connector for Infor Reporting Development Guide*
- *Infor LN Report Designer Development Guide*
- *Infor LN - Development Tools Development Guide*
- *Infor Enterprise Server - Administration Guide*
- *Infor Enterprise Server - Cloud Edition Administration Guide*

Contacting Infor

If you have questions about Infor products, go to the Infor Xtreme Support portal.

If we update this document after the product release, we will post the new version on this website. We recommend that you check this website periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com.

Chapter 1: Introduction

The main goal of extensibility is to develop the last-mile functionality for your organization without changing the core standard software components and using only the public interfaces of the standard application. In this way, you can develop the extensions separate from the standard components. This leads to a situation that upgrading the standard software does not result in additional costs for upgrading customizations. Extensions “survive” the upgrades.

This table shows the types of extensibility in Infor LN:

Type	Features	Tool
Personalize	Hide/unhide fields, add customer-defined fields, conditional coloring, personalize menus and forms, suppress dialog boxes / messages, set defaults	LN standard (through User Interface)
Tailor	Add fields and logic to existing forms / BODs / web services; add field hooks and commands to existing tables and forms; add secondary table to existing forms; add customer defined fields to existing IR push reports. **Note:** Not all features are available in Infor LN 10.5.1.	LN Extension Modeler
Extend	Create new tables, domains, labels, screens, sessions, modules, libraries, messages, etc.	LN Studio
Integrate	Create new BODs and web services, and call SOAP web services from extensions	LN Studio

Supported LN versions

The extensibility concept is available with LN 10.5 or later, and with some limitations*) it is also available for LN 10.3 and 10.4.x if Enterprise Server 10.5 (Tools) is installed.

*) Limitations extensibility in 10.3 and 10.4.x:

- Report extensibility: Native LN reports must be copied to own VRC, TIV-number must be increased to at least 2020 and the report must be recompiled.
- Session extensibility: (Easy) filtering on additional form fields is not possible.
- BOD extensibility: The BODs must be on 10.5 level to be able to extend them. Check KB 22945150. The related KBs with “Extension Modeler” in the description are the ones that must be applied.

Licensing

To use the full Extensibility features of LN the development license (product ID 10146) is required.

Without this development license, you cannot create new tables, domains and script components.

Chapter 2: Personalization

With the personalization features you change more the look and feel of the application. With the extension features you add functionality to the application.

An overview of the personalization features of LN is supplied. Those features are not described in detail, but references are made to other guides and online help where you can find more information.

Personalization possibilities are to prevent that you choose to build an extension to achieve a process improvement, that can be achieved by a personalization. On the other hand, applying some personalizations can be necessary to complete the required functionality, which was built with an extension. For example, an extension can add additional fields to a session, but the personalization features are required to position those fields at the desired location in the screen.

For more information, see the *Infor LN UI Infor Ming.le-LN Plug-in User Guide*.

Features

This table shows the personalization features in LN:

Table heading	Table heading
Conditional formatting	Rows and fields can be formatted based on certain conditions.
Field location	Fields can be moved to another location within the screen.
(Un)hide fields	Fields can be (un)hidden. Unhiding of fields applies to fields that are already present in the session's form (but hidden), but also the other fields of the main table of the session, which are not yet in the form.
Mandatory field	Fields can be made mandatory. If a field must be conditionally mandatory, you can achieve that with an extension.
Read-only field	Fields can be made read-only. If a field must be conditionally read-only, you can achieve that with an extension.
Label change	Field prompts and session titles can be changed to make them clearer for the user.
Color change	Fields and field prompts can be colored differently.
Size change	Fields and field prompts can get a larger size.

Table heading	Table heading
Saved filters	Filters can be saved and one can be set as default.
Toolbar modification	Buttons in the toolbar can be removed, the order can be changed, specific commands can be added with a custom icon.
Menu modification	Menu items in forms can be hidden and the default action for the icon can be selected.
Saving defaults	Values filled on dialog boxes (for example selection criteria and options) can be saved. Next time the dialog box pops up, the fields are filled with those values.
Quick flow	Once defaults are saved for a dialog box, it can be suppressed.
Export to Excel	The set of fields to be included in the Export to Excel can be specified and is retained.
Suppress Messages	Message boxes, with **OK** button, can be suppressed.
Suppress Questions	Questions, with other buttons than **OK** button, can be suppressed; the user defines his default answer.
Menu structure	(Un)hide options in the menus.

Chapter 3: Customer Defined Fields

Tailoring is adding functionality to existing components. Tailoring includes the concept of Customer Defined Fields (CDFs). Those fields can be added to tables, screens, reports and BODs and validation and calculation logic can be defined around those fields.

Use the Customer Defined Fields (CDF) concept to store additional data in the standard Infor LN tables. The CDF definitions are stored separately from the table definitions in the Data Dictionary. For the end user, the CDFs behave in the same way as the standard fields, if defaulting, validations, etc. are built using the CDF logic of the table extension point. The session extension point also has features for the CDFs.

See Customer defined field logic on page 33 and Customer Defined Field .

CDFs are configured per package combination. This implies that when moving your companies from one package combination to another, the CDF definitions must be present in the target package combination. Otherwise you lose the data in the CDFs.

New export and import procedure

From Enterprise Server 10.6.0.1, PMC is used to export and import CDFs through extensions.

See Exporting and importing CDFs through PMC on page 17.

Old export and import procedure

The old procedure to export and import CDFs should only be used to copy CDF definitions from or to LN 10.5 and older LN versions. In the old procedure, these sessions are used:

- **Export Customer Defined Fields (ttadv4291m000)**
- **Import Customer Defined Fields (ttadv4292m000)**

See Exporting and importing CDFs through ttadv4291m000 and ttadv4292m000 on page 18.

CDF types

CDFs can be defined with standard domains, or your own domains that you can create with LN Studio. If you do not use a standard or own domain, the CDFs are added to the table with an implicit domain that is dependent on the data type.

This table shows the supported data types:

Data Type	Remark	Implicit Domain
String	Multibyte string, length between 1 and 999; default length is 30	<pk>cdf___str<lll> (3 underscores)
Integer	Integer number (long)	<pk>cdf_____int (6 underscores, for integers with 10 positions and format ZZZZZZZZZ9) <pk>cdf____i<ll><ss> (4 underscores)
Numeric	Numeric number (double)	<pk>cdf_____num (6 underscores, for doubles with 9 digits before and 5 digits after decimal point and format ZZZZZZZZ9VD99999) <pk>cdf__n<bb><aa><ss> (2 underscores)
Date	Date/Time, local time on screen, stored as UTC	<pk>cdf_____dat (6 underscores, for dates with format %u001 %U001) <pk>cdf_____u<ss>, 6 underscores)
Checkbox	true or false	<pk>cdf_____chk (6 underscores)
List	A predefined list of choices	<pk>cdf_lst<list> (1 underscore)
Text	Text field	<pk>cdf_____txt (6 underscores)

Explanation for the domain codes:

- <pk> the package code of the table to which the CDF is added
- <ll[l]> length of the string or integer
- <ss> sequence number, per format a different sequence number is generated
- <bb> digits before
- <aa> digits after
- <list> list code that holds the predefined list of choices

The domain codes can be required during the development of the extensions with the Extension Modeler.

See Extension Modeler on page 19.

You can also add calculated CDFs; those are not physically stored in the table, but calculated based on other table fields and presented in the UI. This type of CDF is deprecated. We recommend that you use the Calculated Field extension type of the session extension point.

See Calculated Field on page 66

CDFs, except the calculated ones and text fields, can be defined with multiple elements (arrays).

CDF Configuration

Go to **Tools > Application Configuration** for the sessions to define CDFs.

To configure customer defined fields:

1. Start the **Customer Defined Fields Parameters (ttadv4590m000)** session.
2. Select **CDF Active** and click **OK** .
3. Define customer defined fields in one of these ways:
   - Use the **Customer Defined Fields** option in the **Settings** (gear icon) menu in a session you started in LN UI.
   - Use the **Customer Defined Fields (ttadv4591m000)** session.
   - To create customer defined fields of type ‘List’, specify the lists and their constants in the **Lists (ttadv4592m000)** and **List Constants (ttadv4593m000)** sessions.
4. In the **Customer Defined Fields (ttadv4591m000)** session, click **Actions** and select **Convert to Runtime**.

   The **Convert to Runtime Data Dictionary (ttadv5215m000)** session starts. Convert the customer defined fields and the related implicit domains to the runtime data dictionary.

   Note: Converting to runtime changes the physical structure of the LN tables. The users must be logged out from the system.

   Infor Enterprise Server - Cloud Edition Administration Guide

Exporting and importing CDFs through PMC

1. Ensure that a table extension exists. Alternatively, create an empty table extension for the tables that include the CDFs.

   See Table extension point on page 27.
2. Export or import the solutions that contain these table extensions through PMC.

   See Extension Deployment on page 131.

The old way of exporting and importing CDFs through **ttadv4291m000** and **ttadv4292m000** is still supported. CDFs that are imported with PMC are not overwritten with CDFs from **ttadv4292m000**.

If CDFs already exist on the system, they may be overwritten by PMC. If you then uninstall through PMC, the original CDFs are still available.

The “Customer Defined Fields” table has a new field: **Origin** (**orig**). This **orig** field is used to distinguish between CDFs that are manually created and CDFs that are imported through PMC. This field is displayed in the **Customer Defined Fields** session.

You can import a PMC solution for a table extension that did not exist before. If the same CDFs for that table already existed with origin “Manual”, they are restored if you perform an uninstall of this solution.

Exporting and importing CDFs through ttadv4291m000 and ttadv4292m000

Note: You should perform the export and import through PMC. CDFs that are imported through PMC are not overwritten with CDFs from the **Import Customer Defined Fields (ttadv4292m000)** session.

The format of CDF files has been changed. If you use a 10.6 version of LN, you can export CDFs to a pre-10.6 version. To do this, you must select **Before 10.6 format** in the **Export Customer Defined Fields (ttadv4291m000)** session. Otherwise you cannot import CDFs from a 10.6 LN version into a pre-10.6 LN version.

In LN versions before 10.6, all lists were exported and imported. From 10.6 onwards, only the lists that are used by the exported and imported CDFs are exported and imported.

Suppose that a CDF with origin “FromPMC” was exported through **ttadv4291m00** and then imported into a system without this CDF. In that case, the CDF origin is set to the default value: “Manual”. If the CDF already existed on the import system and was “FromPMC”, it remains “FromPMC”.

CDF Limitations

- You cannot define customer defined fields for tables within Tools (the `tl` and `tt` packages).
- External integrations, such as Infor Integration, EDI, Office Integration, and SOA-based integration, do not support customer defined fields.
- You can use customer defined fields within 4GL reports, if editing the 4GL report layouts is still supported in your environment or by using the report personalization features of Infor LN Report Designer. For external reporting, only Infor Reporting and Microsoft Reporting (SSRS) support customer defined fields.
- Customer defined fields cannot store application data in multiple data languages.
- There is no direct limitation on the number of CDFs in a table. The actual number of fields in a table and the total length of all fields may be limited by the RDBMS you use.
- Only super users can run the **Convert to Runtime Data Dictionary (ttadv5215m000)** session to convert the customer defined fields and the related domains to the runtime data dictionary.

Note: The full functionality of customer defined fields is only available within Web UI and LN UI. Customer defined fields are not displayed in the classic Infor LN BW UI.

Chapter 4: Extension Modeler

Use the Infor LN Extension Modeler to add the logic around CDFs and how to tailor standard components. Infor LN 10.6 contains these extension points:

- Table
- Session
- Report
- BOD / BDE
- Menu

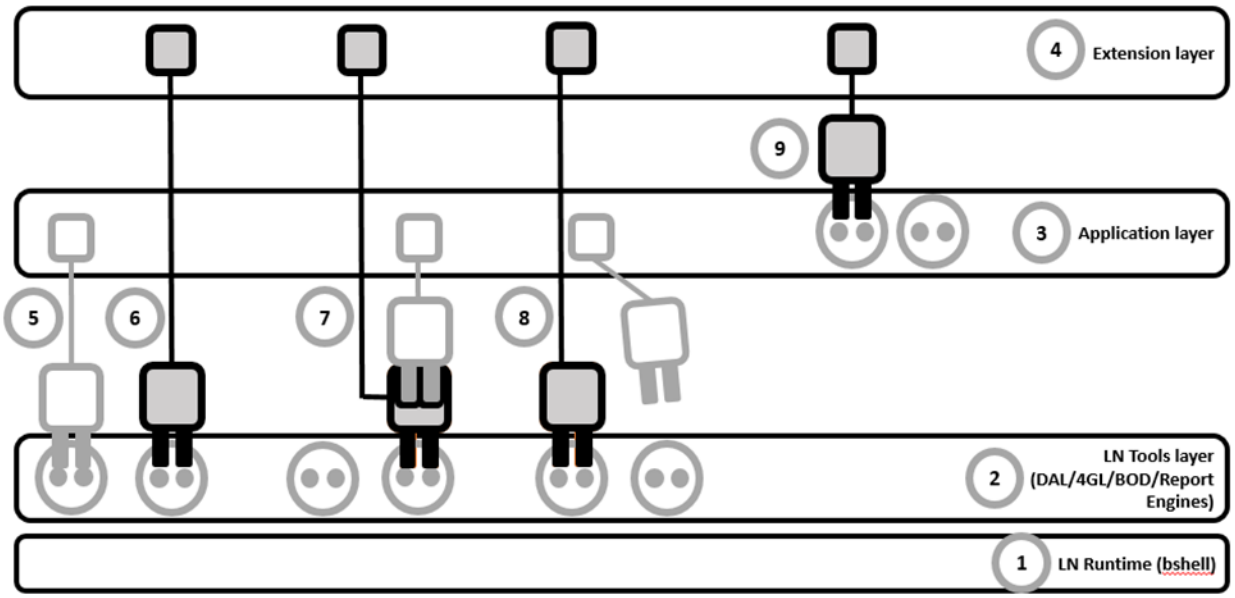
The Extension Modeler in 10.6 can show the extension point "Process" as well. This is for future use.

In the Extension Modeler, you can set properties and hooks for those components. The implementation of the extension point for one component is called an extension. With an extension built for an extension point you change the behavior of a component. For example, by creating an extension for a session you can add additional fields to that session.

LN's extensibility is built upon LN's pluggable architecture. The standard application components of LN are plugged into the sockets of the runtime layers, which perform all common tasks, such as database access, screen handling, etc. Extensions are additional plugs into the runtime layers; sometimes, an extension can also handle as an adapter.

You can find the Extensibility sessions in **Tools > Application Extensibility**

This diagram shows this architecture:



1. The LN Runtime layer (bshell) runs the LN programs and handles all RDBMS and operating system actions.
2. The LN Tools layer is responsible for all common tasks regarding tables, screens, reports and BODs. This layer has several sockets where the Application layer can plug-in with properties and hooks into to perform the specific application functionality. For the Extension layer, additional sockets are available in the Tools layer.
3. The Application layer has a set of standard components that have properties and pieces of code. This produces the desired behavior or results in the different engines (DAL / 4GL / BOD / Report).
4. The Extension layer has a set of components with properties and pieces of code. This produces different behavior or results in the different engines (DAL / 4GL / BOD / BDE / Report).
5. The standard Application layer has coded actions that must be performed on a certain event. Those actions are plugged into the socket that is meant for this event. Those actions are executed by one of the tools engines when that event occurs. Examples:
   - When a record is updated in a table, also another table must be updated. For example, when the quantity is changed in a sales order line, the inventory allocation also must be updated. In this case the DAL application component has an `after.save.object()` hook to perform the update for the inventory allocation.
   - When a report is printed, for each detail line also a percentage must be calculated and printed. The report script has a `before.field` hook to calculate the percentage.
6. The Tools layer has also specific sockets for the Extension layer. The plug is created by adding an extension in the Extension Modeler. Examples:
   - An overview session must show some additional fields (directly from database or a complex calculation). For example, for business partners the number of open purchase orders should be displayed. This field, with the code to calculate the value, must be added in the session extension.
   - An Infor Reporting report requires to print sub-details. Those sub-details must be added to the XML data source by adding additional rows. This is done in the report extension with the `write.row()` hook.
7. Next to the specific sockets for the extensions, extensions can also act as an adapter. In this case the standard plug is adapted. There is a standard plug that does specific actions when a record is

saved. For example an update on the inventory allocation when a sales order line is inserted. The extension plug can do additional actions, for example inserting data in an own table. Adapters cannot bypass the standard behavior.

8 The extension can remove a standard plug and connects its own plug to the socket. For example, some form commands of a session can be removed, and other form commands can be added.

9 Another concept of extensibility is that the standard application itself has sockets. Functionality that had to be customized often in the past, can be influenced by plugging in some own pieces of code. **Note:** The concept of application sockets is not widely implemented in LN 10.5. The Document Output Management example, mentioned on the third bullet, is available.

For more information about custom plug-ins in Document Output Management, see *Infor LN Document Output Management User Guide*.

Examples of application sockets that can be implemented or are implemented:

- For export control, you can use your own application. This application exposes a web service to check whether shipment of an item to a certain country is allowed. In the application extension, you can call the web service; if shipping is not allowed, the application blocks the shipment.
- To change the compose invoices algorithm: items that have a different value in a specific customer defined field, must not be combined in one invoice. In the application extension, you can check whether the CDF has a different value. If so, inform the standard application that this line cannot be added to the invoice. It must be on a separate one.
- LN's Document Output Management is flexible. If you require an output channel that is not supported in the standard application, add your own output channel in the application extension.

Cloud readiness

We recommend that you build your extensions in a way that they are ready for the cloud. Although you may not consider to make that move with LN on short term, you save a lot of effort in migrating your extensions when you decide to move.

In general, cloud readiness is related to these topics:

- Upgradability.

  Upgrades to new versions must not be impeded by the presence of extensions. This applies both to efforts required to upgrade extensions and the possibility that extensions can break the upgrade itself.

- Stability and performance.

  Extensions must not impact the infrastructure in such a way that other customers within the cloud environment are experiencing adverse effects.

- Security.

  Ensure that extensions cannot have access to information of the infrastructure that is a security risk.

The mechanisms that are built in LN's Extensibility layer to govern the extensions are described in Governance on page 126.

Getting started with extensions

To get started with extensions:

1. Start the **Initialize Extensibility (ttext0200m000)** session.

   If your current package combination already has the Extensions (`tx`) package, this package VRC is displayed. You cannot change this VRC. If your current package combination does not have the Extensions package, you can use the default VRC (`B610_a_ext`). If required you can change the VRC name. Do not choose an existing VRC that is already used in another package combination. All package combinations require a different VRC for the Extensions (`tx`) package.

   Note: Developing extensions always applies to the current package combination. There is no inheritance through a VRC-derivation structure.

2. To use Software Configuration Management (SCM), select **Use SCM** and specify a Development VRC to be created. The **Use SCM** option is only available if your LN server is prepared to use SCM.

   For more information about Software Configuration Management, see the *Infor LN Studio Application Development Guide*.

   Note: Activating SCM is not required to keep the revisions of your extensions. History of extensions is always available and you can restore old revisions; see Extension history. If you use SCM, you can isolate checked-out changes from other extension developers when you share development activities.

3. Click **Initialize**. Close the session.
4. Select **Restart** in the **Options** menu, to restart your LN environment.
5. By default, your Extensibility environment is setup with the **Extensions Ready for Cloud** setting. See Cloud readiness.

   If you are not running LN in a cloud environment, you can switch off this setting: Start the Extensibility Parameters (ttext0100m000) session. Clear the **Extensions Ready for Cloud** check box and click **Save**.

6. Start the **Extensions (ttext1500m000)** session to create extensions. For the procedure to create the extensions, see the Extension development procedure.

Extension development procedure

For the development of extensions, you must set a current activity. An activity groups the different extensions, which you must create for a functional unit. Multiple developers can work in the same activity.

For more information about activity based development, see the *Infor LN Studio Application Development Guide*.

Setting a current activity

To set a current activity:

1. Start the **Extensions (ttext1500m000)** session.
2. Click **Actions** and select **Select Current Activity**.

   The screen to select a current activity is always displayed if you have not yet selected a current activity and the required action requires one.
3. Select the activity and click **OK**.
4. If your activity is not in the list, you can create a new activity. Continue with the next step, otherwise this procedure is finished and you can start to build an extension.
5. To create a new activity, click **New**.
6. Specify at least **Activity Name**. The other fields are optional. **Activity Documentation** is used as default revision text during check-in of extensions.
7. Click **Save changes and exit**.
8. Click **OK**.

Building an extension

To build an extension:

1. Start the **Extensions (ttext1500m000)** session and click **New**.
2. Select the **Extension Point** and specify the **Component Name**.

   Accept the proposed default in **Library** or specify your own Library code. Note that the package (`tx`) and the proposed module (esb, esm, esr, ess, est; see Extension scripts) cannot be changed.
3. Click **Save changes and Exit**.
4. Select the Extension. Click **Actions** and select **Check-Out**.
5. Click **Extension Modeler**.
6. In the Extension Modeler specify the **Properties** on component level, if applicable.
7. To implement a hook, right-click the hook and select **Add Implementation** or double-click the hook.
8. Click **Add** to add other extension types for the extension and fill the properties and hooks for those levels.

   The extension types, hooks and properties depend on the extension point to build an extension for.
9. Click **Save** to save the extension. The extension script is automatically generated during save.

   Compilation issues can be displayed in the Problems view. Solve those problems and click **Save**.

   Note: If a compilation problem must be solved in another component, for example a library which you created with LN Studio, click **Generate and Compile** after changing that other component.
10. Test the extension by starting the session(s) that would open the extension functionality. For testing a BOD extension, we recommend that you run the relevant BOD publishing session in simulation mode.

    Go to the **Common** menu under **BOD Messaging > Publish BODs**.

11 Close the Extension Modeler.

12 Click **Actions** and select **Check-In**.

13 Accept the default revision text or type your own text and click **Save changes and exit**.

Before you checked-in the extension, the new extension or the new version of the extension was only available for you. After check-in, the most recent version of the extension is available to all users who set their activity context to your activity.

14 Click **Actions** and select **Commit**.

The, new version of the, extension is available to all users.

Activity context

When starting the **Extensions (ttext1500m000)** session, the activity context is automatically set to your current activity. The activity context is changed when you select another current activity. After the activity context is set, the sessions that are started, run within this context. The sessions include the functionality that is added in the extensions.

With **Options** and **Debug and Profile 4GL, y**ou can also set activity context.

When the extensions are committed, the sessions include the extension functionality without the requirement to set the activity context.

Hint: To ensure your session runs in the correct activity context, add the activity to the title that is used for the session tab in LN UI. To achieve this:

1 Select **Options > Settings** and select your current profile.

2 Add **`-set BAAN_WIN_TITLE="%S-%a`** to the **Command** field in your User Profile Details. The **`%a`** shows the activity context; the result is, for example, Item Defaults-act0001.

Extension scripts

For each extension, an extension script is generated. This extension script contains the hooks that are programmed in the Extension Modeler and other generated functions. They are called by the different tools engines to do the required actions of the extension. Those extension scripts are DLLs (libraries), which are stored in the Extension package (`tx`).

This table shows the modules within the tx-package that are reserved for extension scripts:

Module	Description
esb	Extension Scripts for BODs
esm	Extension Scripts for Menus
esp	Extension Scrips for Processes
esr	Extension Scripts for Reports

Module	Description
ess	Extension Scripts for Sessions
est	Extension Scripts for Tables

Note that all module codes starting with es are reserved for future use.

Extension scripts are visible in Infor LN Studio and can be debugged using LN Studio, see Extension debugging on page 119.

We do not recommend that you make changes in the generated scripts. The changes are lost after a change of the extension in the Extension Modeler.

Extension history

History of extensions is kept in the extension history table.

To view the history:

1. Start the **Extensions (ttext1500m000)** session.
2. Click **References** and select **History**.

History has two levels:

- Activity level
- Extension level

The activity level history is updated each time an extension is checked-in; the revision of the extension is stored in the history. The extension level history is updated each time an extension is committed or imported into the environment. Note that during commit of an extension, the activity revisions are removed. The revision text of the last revision within the activity is used to create the revision on extension level.

Activation and deactivation

After developing and committing an extension, the extension is active.

To deactivate the extension, go to the **Extensions (ttext1500m000)** session.

Click **Actions** and select **Deactivate** .

The extension component itself remains in the system, but the functions of the generated extension script are not executed anymore by the tools engines. You can use this deactivation to check whether problems with the system are caused by your extension or by the standard software.

Click **Actions** and select **Activate** to activate the extension again.

Note: Restarting your sessions can be required to see the result of (de)activation.

Note: During import of extensions deactivated extensions always remain deactivated. Active extensions can be deactivated during import if the extension is not active in the file being imported.

See Extension Deployment on page 131.

Chapter 5: Table extension point

A table extension is used to react on the table events such as insert, update and delete for standard LN tables. You can also control whether those actions on the table are allowed. For CDFs, you can set defaults, add validations, etc. For standard fields, you can also add validations, etc.

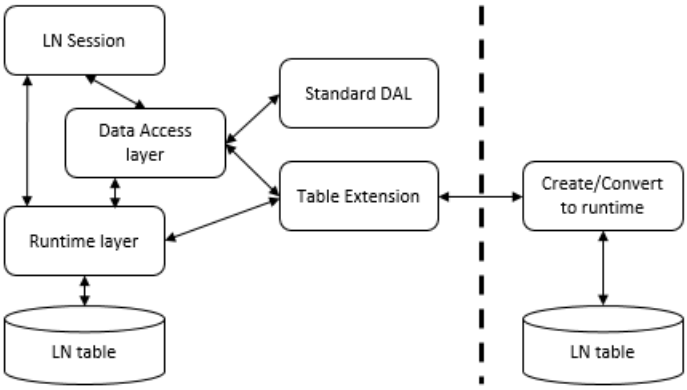
Examples:

- When an Item is added to or updated in the Item table, to update a CDF that holds the last modification date.
- Make a CDF a mandatory field if the Item is of a certain type.
- Block the adding of new Sales Order Lines for a Sales Order when a CDF on Sales Order level has a certain value.
- Do additional validation on a standard field.

A table extension is an extension to the Data Access Layer (DAL) of the table, although for the table itself no DAL must be implemented.

For background on hooks, validations, setting error messages, return values, etc. see the DAL chapters and functions in the *Infor ES Programmer's Guide*.

This diagram shows the position of the table extension:



LN sessions manipulate data in the LN tables. The hooks of the table extension are executed both from the data access layer and the runtime layer. The latter happens in the case the data access layer has not been implemented or is bypassed (for performance reasons) for certain LN tables. The table extension is applied regardless of the techniques used in the standard application.

The Create/Convert to runtime process also invokes the table extension to determine whether custom indexes must be created, changed or deleted.

For the table extension point, there are these extension types:

- Table
- Customer defined field logic
- Standard field logic
- Custom index

Table

With the hooks defined for the extension type Table, you can react on events that occur on table level. This table shows the available hooks:

Name	Signature
Declarations	
Functions	
Before Open Object Set	long `before.open.object.set()`
Set Object Defaults	long `set.object.defaults()`
Method is Allowed	boolean `method.is.allowed(long method)`
Before Save	long `before.save.object(long mode)`
After Save	long `after.save.object(long mode)`
Before Destroy	long `before.destroy.object()`
After Destroy	long `after.destroy.object()`

Declarations hook

Use this hook to declare tables and variables that must be globally available in all hooks of the extension. Also, the references to include files and DLLs that are used by the extension must be coded in this hook with `#include` and `#pragma`.

Example:

```
#include        <bic_text>
        table   tccom100          |* Business Partners
        domain  tcnama  old.nama
                string  date.string(14)
```

```
                boolean retb
#pragma used dll "otxprcdll0001"
```

Functions hook

Use this hook to code (common) functions to use in the other hooks of the table extension. This helps you in reusing code and to keep the other hooks small and clear.

Functions that are called through `with.old.object.values.do()` and `with.object.set.do()` in the other hooks of the extension must be coded in this hook.

Example:

```
function get.old.nama()
{
        old.nama = tccom100.nama
}
function string format.date(long i.date)
{
        return(utc.to.iso(i.date, UTC_ISO_DIFF))
}
```

Before Open Object Set hook

Use this hook to initialize variables for this extension. You can also use this hook to disallow access to the table.

See `before.open.object.set()` of the standard Data Access Layer in the *Infor ES Programmer's Guide*. The possibility of extending the query is not supported in the extension.

Example:

```
function extern long before.open.object.set()
{
   if txprcdll0001.pricebooks.blocked() then
           dal.set.error.message("@Pricebooks blocked for maintenance")
           return(DALHOOKERROR)
   endif
   return(0)
}
```

Set Object Defaults hook

Use this hook to set default values for CDFs.

Example:

```
function extern long set.object.defaults()
{
        tcmcs004.cdf_date = utc.num()
        return(0)
}
```

Method is Allowed hook

Use this hook to control whether new records can be inserted, existing records can be updated or deleted. The input argument for this hook is the method.

This tables shows the Method values:

Method	Description
DAL_NEW	The hook is called to know whether records can be added. There is no current record, but in case of a session with a view, the view fields are available.
DAL_UPDATE	The hook is called to know whether the current record can be updated.
DAL_DESTROY	The hook is called to know whether the current record can be deleted.

For more information, see `method.is.allowed()` of the standard Data Access Layer in the *Infor ES Programmer's Guide*. Note that this hook can only be used to set more restrictions. If the standard functionality does not allow a certain action, the extension cannot allow it either.

Example:

```
function extern boolean method.is.allowed(long method)
{
        on case method
        case DAL_NEW:
              select  tdsls400.cdf_blck
              from    tdsls400
              where   tdsls400.orno = :tdsls401.orno
              as set with 1 rows
              selectdo
                   if tdsls400.cdf_blck = tdcdf_____chk.yes then
                         dal.set.error.message(
                           "@Order is blocked, you cannot add Lines to
it.")
                         return(false)
                   endif
              endselect
              break
        case DAL_UPDATE:
              break
        case DAL_DESTROY:
              break
        endcase
```

```
        return(true)
}
```

Before Save hook

Use this hook to perform additional actions before the current (new or existing) record is saved. Think of updating fields in other tables, validations that could not be done on field level, etc. The input argument for this hook is the mode.

This table shows the Mode values:

Mode	Description
DAL_NEW	A new record is being inserted.
DAL_UPDATE	An existing record is being updated.

This hook is executed before the `before.save.object()`hook of the standard Data Access Layer is executed. If the standard hook must be executed before the extension hook is executed, you can force the standard hook to execute anytime you prefer. This can be achieved by calling the `table.super()` function.

All field values of the current record of the table are available.

For more information, see `before.save.object()` of the standard Data Access Layer in the *Infor ES Programmer's Guide*.

Example:

```
function extern long before.save.object(long mode)
{
        table.super()
        tcmcs004.cdf_lcdt = utc.num()
        tcmcs004.cdf_user = logname$
        return(0)
}
```

After Save hook

Use this hook to perform additional actions after the current (new or existing) record is saved. Think of updating fields in other tables, etc. The input argument for this hook is the mode.

This table shows the Mode values:

Mode	Description
DAL_NEW	A new record is being inserted.
DAL_UPDATE	An existing record is being updated.

This hook is executed before the `after.save.object()` hook of the standard Data Access Layer is executed. If the standard hook must be executed before the extension hook is executed, you can force the standard hook being executed anytime you prefer. This can be achieved by calling the `table.super()` function.

All field values of the current record of the table are available.

For more information, see `after.save.object()` of the standard Data Access Layer in the *Infor ES Programmer's Guide*.

Example:

```
function extern long after.save.object(long mode)
{
        table.super()
        with.old.object.values.do(get.old.values)
        if tcmcs004.cdf_city <> old.city then
                ret = txcomdll0001.log.city.change(
                                old.city, tcmcs004.cdf_city)
                if ret < 0 then
                        dal.set.error.message(
                               "@Error during logging city change")
                        return(DALHOOKERROR)
                endif
        endif
        return(0)
}
```

Before Destroy hook

Use this hook to perform additional actions before the current record is deleted. Think of updating fields in other tables, additional checks whether it can delete the record, etc.

This hook is executed before the `before.destroy.object()`hook of the standard Data Access Layer is executed. If the standard hook must be executed before the extension hook is executed, you can force the standard hook being executed anytime you prefer. This can be achieved by calling the `table.super()` function.

All field values of the current record of the table are available.

For more information see `before.destroy.object()`of the standard Data Access Layer in the *Infor ES Programmer's Guide*.

Example:

```
function extern long before.destroy.object()
{
        table.super()
        select  txcom001.*
        from    txcom001
        where   txcom001.crou = :tcmcs004.crou
        as set with 1 rows
```

```
        selectdo
                dal.set.error.message(
                        "@Route still being used in Carrier Plan")
                return(DALHOOKERROR)
        endif
        return(0)
}
```

After Destroy hook

Use this hook to perform additional actions after the current record is deleted. Think of updating fields in other tables, etc.

This hook is executed before the `after.destroy.object()` hook of the standard Data Access Layer is executed. If the standard hook must be executed before the extension hook is executed, you can force the standard hook being executed anytime you prefer. This can be achieved by calling the `table.super()` function.

All field values of the current record of the table are available.

For more information, see `after.destroy.object()` of the standard Data Access Layer in the *Infor ES Programmer's Guide*.

Example:

```
function extern long after.destroy.object()
{
        table.super()
        txcomdll0001.log.deleted.sales.order(
                  tdsls400.orno, tdsls400.crep,
                  tdssl400.otbp, tdsls400.oamt)
        return(0)
}
```

Customer defined field logic

With the hooks on CDF level you can let the CDFs behave like standard fields. This applies to making the field mandatory, update them automatically based on changes of other fields, validations, and so on.

This table shows the hooks that are available for each individual CDF:

Name	Signature
Is Never Applicable	boolean <cdf field>.is.never.applicable(long mode)
Is Applicable	boolean <cdf field>.is.applicable(long mode)

Name	Signature
Is List Entry Applicable	boolean <cdf field>.<constantname>.is.applicable(long mode)
Is Derived	boolean <cdf field>.is.derived(long mode)
Is Mandatory	boolean <cdf field>.is.mandatory(long mode)
Is Read-only	boolean <cdf field>.is.readonly(long mode)
Make Valid	long <cdf field>.make.valid(long mode)
Is Valid	boolean <cdf field>.is.valid(long mode)
Update	long <cdf field>.update(long mode)

The input argument for all hooks is the mode.

This table shows the Mode values:

Mode	Description
DAL_NEW	A new record is being inserted.
DAL_UPDATE	An existing record is being updated.

In all hooks, except for the <cdf field>.is.never.applicable() hook, all field values of the current table record are available.

If CDFs are dependent on standard fields or other CDFs – in other words if in the hooks the values of other fields are used – the hooks are re-executed when the field(s) on which the CDF depends are changed. Those dependencies are registered automatically.

For more information about the hooks, see the corresponding -field.<hook>()of the standard Data Access Layer in the *Infor ES Programmer's Guide*.

Is Never Applicable hook

Use this hook to indicate if the field is never applicable. If a field is never applicable the field is made invisible at startup of a session. A field can become never applicable based on a static constraint, such as a parameter setting.

Example:

```
function extern boolean tcmcs004.cdf_city.is.never.applicable(long mode)
{
        select  txmcs000.icty
        from    txmcs000
        where   txmcs000.sequ = 0
        as set with 1 rows
        selectdo
                if txmcs000.icty = tcyesno.no then
```

```
                        return(true)
                endif
        endselect
        return(false)
}
```

Is Applicable hook

Use this hook to indicate whether the field is applicable. If a field is not applicable, then the field is disabled and the field is cleared.

Example:

```
function extern boolean tcmcs004.cdf_city.is.applicable(long mode)
{
        return(tcmcs004.crou(1;1) = "U")
}
```

Is List Entry Applicable hook

Use this hook to indicate whether a certain list constant is applicable. If the list constant is not applicable it is not displayed in the field's drop down list box, so the end-user cannot select it.

The constant names to be used in the hooks are the constants that are defined in the CDF **Lists (ttadv4592m000)** session. If a standard enum domain is used for the CDF, instead of a List, the constant names can be found in the **Domains (ttadv4500m000)** session, Enum/Set data.

Example:

```
function extern boolean tdsls400.cdf_brsn.export.is.applicable(long mode)
{
        if tdsls400.orno(1;3) = "EXP" or
           tdsls400.orno(1;3) = "SLE" then
                return(true)
        endif
        return(false)
}
```

Is Derived hook

Use this hook to indicate whether the field is derived. If a field is derived, then the field is made read-only in the UI. The difference with the `<cdf field>.is.readonly()` hook is that the field value can be changed within other hooks of the extension, for example in the `<cdf field>.update()` hook. If a field is read-only, its value cannot be changed.

Example:

```
function extern boolean tcmcs004.cdf_addr.is.derived(long mode)
{
    if tcmcs004.crou(1;1) = "U" then
           return(true)
    endif
    return(false)
}
function extern tcmcs004.cdf_addr.update(long mode)
{
    if tcmcs004.crou(1;1) = "U" then
          tcmcs004.cdf_addr = tcmcs004.cdf_zip & " " & tcmcs004.cdf_city

    endif
}
```

Is Mandatory hook

Use this hook to indicate whether the field is mandatory. If a field is mandatory then it must have a value other than `""`, `0.0`, `0` or blank.

Example:

```
function extern boolean tcmcs004.cdf_city.is.mandatory(long mode)
{
        return(tcmcs004.crou(1;1) = "U")
}
```

Is Read-only hook

Use this hook to indicate whether the field is read-only. If a field is read-only it is made read-only in the UI. The field however, still can have a value.

Example:

```
function extern boolean tdsls400.cdf_blck.is.readonly(long mode)
{
        return(tdsls400.hdst = tdsls.hdst.closed)
}
```

Make Valid hook

Use this hook to adjust the field's value before it is checked. You can use it for example to round a field's value.

Example:

```
function extern long tdsls401.cdf_mprc.make.valid(long mode)
{
        tdsls401.cdf_mprc = round(tdsls401.cdf_mrpc, 2, 1)
        return(0)
}
```

Is Valid hook

Use this hook to perform any checks not already defined in one of the other field hooks.

Example:

```
function extern boolean tcibd001.cdf_colr.is.valid(long mode)
{
        select  txcom002.colr
        from    txcom002
        where   txcom002.colr = :tcibd001.cdf_colr
        as set with 1 rows
        selectdo
                return(true)
        endselect
        dal.set.error.message("txcomt002", tcibd001.cdf_colr)
        |* Color %1$s not found
        return(false)
}
```

Update hook

Use this hook to (re) determine the value of the field based on the current record values. Think of determining defaults and calculating derived values.

Example:

```
function extern tcmcs004.cdf_addr.update(long mode)
{
    if tcmcs004.crou(1;1) = "U" then
           tcmcs004.cdf_addr = tcmcs004.cdf_zip & " " & tcmcs004.cdf_city

    endif
}
```

Standard field logic

With the hooks on standard field level you can influence the behavior of the standard application. This applies to making the field mandatory, update them automatically based on changes of other fields, validations, etc. For the standard fields, also hooks can be present in the standard Data Access Layer. If both hooks are present, in the DAL and in the table extension, the table extension can only restrict the data further. For example, data that cannot be entered because of an `is.valid()` hook in the standard DAL can still not be specified even if the table extension would allow it.

This table shows the hooks that are available for each standard table field:

Name	Signature
Is List Entry Applicable	boolean `<table field>.<constantname>.is.applicable(long mode [,long element])`
Is Derived	boolean `<table field>.is.derived(long mode [,long element])`
Is Mandatory	boolean `<table field>.is.mandatory(long mode [,long element])`
Is Read-only	boolean `<table field>.is.readonly(long mode [,long element])`
Make Valid	long `<table field>.make.valid(long mode [,long element])`
Is Valid	boolean `<table field>.is.valid(long mode [,long element])`
Update	long `<table field>.update(long mode [,long element])`

The input argument for all hooks is the mode. Mode can have these values:

Mode	Description
`DAL_NEW`	A new record is being inserted.
`DAL_UPDATE`	An existing record is being updated.

The argument `element` is available for all array table fields. The element points to the actual occurrence in the array that is being processed.

In all hooks, all field values of the current table record are available.

If standard fields are dependent on other standard fields or CDFs – in other words if in the hooks the values of other fields are used – the hooks are re-executed when the field(s) on which the field depends are changed. Those dependencies are registered automatically.

For more information about the hooks, see the corresponding `-field.<hook>()` of the standard Data Access Layer in the *Infor ES Programmer's Guide*.

Is List Entry Applicable hook

Use this hook to indicate whether a certain list constant is applicable. If the list constant is not applicable it is not displayed in the field's drop down list box, so the end-user cannot select it.

The constant names to be used in the hooks are the constants that are defined for the domain of the table field. The constant names can be found in the **Domains (ttadv4500m000)** session, Enum/Set data.

Example:

```
function extern boolean tdsls400.osta.closed.is.applicable(long mode)
{
        return(txcomdll0001.sales.order.can.be.closed())
}
```

Is Derived hook

Use this hook to indicate whether the field is derived. If a field is derived, then the field is made read-only in the UI. The difference with the `<field>.is.readonly()` hook is that the field value can be changed within other hooks of the extension, for example in the `<field>.update()` hook. If a field is read-only, its value cannot be changed.

Example:

```
function extern boolean tdsls401.pric.is.derived(long mode)
{
        select  txprc100.fixd
        from    txprc100
        where   txprc100.item = :tdsls401.item
        as set with 1 rows
        selectdo
                |* Item price is fixed, user cannot change it
                return(true)
        endselect
        return(false)
}
function extern tdsls401.pric.update(long mode)
{
        select  txprc100.pric
        from    txprc100
        where   txprc100.item = :tdsls401.item
        as set with 1 rows
        selectdo
                tdsls401.pric = txprc100.pric
        endselect
}
```

Is Mandatory hook

Use this hook to indicate whether the field is mandatory. If a field is mandatory then it must have a value other than `""`, `0.0`, `0` or blank.

Example:

```
function extern boolean tcmcs041.dsca.is.mandatory(long mode)
{
        return(true)
}
```

Is Read-only hook

Use this hook to indicate whether the field is read-only. If a field is read-only it is made read-only in the UI. The field however, still can have a value.

Example:

```
function extern boolean tcibd001.dsca.is.readonly(long mode)
{
        if mode = DAL_UPDATE then
                return(true)
        endif
        return(false)
}
```

Make Valid hook

Use this hook to adjust the field's value before it is checked. You can use it for example to round a field's value.

Example:

```
function extern long tcibd001.dsca.make.valid(long mode)
{
        |* Always start with capital
        tcibd001.dsca(1;1) = toupper$(tcibd001.dsca(1;1))
        return(0)
}
```

Is Valid hook

Use this hook to perform any checks not already defined in one of the other field hooks.

Example:

```
function extern boolean tdsls401.item.is.valid(long mode)
{
        if not txexpdll0001.item.allowed(tdsls401.ofbp, tdsls401.item)
then
                dal.set.error.message(
                      "@Item not allowed for this business partner”)
                return(false)
        endif
        return(true)
}
```

Update hook

Use this hook to (re) determine the value of the field based on the current record values. Think of determining defaults and calculating derived values.

Example:

```
function extern tdsls401.pric.update(long mode)
{
        select  txprc100.pric
        from    txprc100
        where   txprc100.item = :tdsls401.item
        as set with 1 rows
        selectdo
                tdsls401.pric = txprc100.pric
        endselect
}
```

Custom index

Use a custom index to create an additional index in a standard table. This index can be used in custom sessions to query the table in a more efficient way. You can use customer defined fields in a custom index.

This table shows the available properties:

Name
Sequence
Label
Description
Duplicates

When you defined a table extension with a custom index, the custom index remains active even if the table extension itself is de-activated. In case of de-activation, the hooks are not executed, but the custom index is not deleted.

Sequence property

The Sequence property is a read-only property that is automatically generated. It starts with 99 and counts back in case of multiple custom indexes. If you remove a custom index, the others are not renumbered. During addition of a new custom index first the open number will be reused.

In the hooks or other script components where to use the custom index, you can refer to it with this property:

`<package><module><table number>._index<sequence>.`

We recommend that you use the individual fields in the queries. In case the standard table definition is modified and the custom index is also available as standard index, you can delete the custom index. Changing the scripts and hooks where you use the index is not required.

Example:

This example shows a validation check to prevent not unique values in a customer defined field.

```
function extern boolean tccom100.cdf_lnam.is.valid(long mode)
{
        domain  tccom.bpid bpid

        select  tccom100.bpid:bpid
        from    tccom100
        where   tccom100._index99 = {:tccom100.cdf_lnam}
        and     tccom100.bpid <> :tccom100.bpid
        as set with 1 rows
        selectdo
                dal.set.error.message(sprintf$("Name used for BP %s",
bpid))
                return(false)
        endselect

 return(true)
}
```

Label property

Use this property if the custom index description must be displayed in different languages. You can select an existing label, or create a new label in the Extensions package. A label can have descriptions in different languages.

For more information see the *Infor LN Studio Application Development Guide*.

The Label property cannot be filled if the Description property is used.

Description property

Use this property if the custom index description is not language dependent.

The Description property cannot be filled if the Label property is used.

Duplicates property

Use this property to indicate whether duplicates are allowed for this custom index.

Note: If you do not select this property you must ensure that the combination of the fields you defined in the custom index have unique values in the table. If there are duplicates and this property is not selected, you will lose data during the table reconfiguration process. The reconfiguration process takes place during the conversion to runtime of the table changes.

Convert to Runtime

Before you can use the custom index, the table extension must be checked in and the table must be converted to runtime.

If you do not use the Extension Modeler to convert the table to runtime. Run the **Create Runtime Data Dictionary (ttadv5210m000)** session to convert the table to runtime. This session must also be used to remove the custom index after you deleted the table extension with the custom index.

Functions

In the hooks of a table extension you can use all trusted functions to do string manipulation, calculations, comparisons, etc.

See Trusted / Untrusted concept

Typical functions to be used in a table extension:

- `with.old.object.values.do()`
- `with.object.set.do()`
- `dal.set.error.message()`
- `disable.table.extension()`
- `enable.table.extension()`
- `ue.get.origin()`

Embedded SQL and the `sql.*` functions are available to read additional data from the LN database. You can also perform database changes with the `db.*` and `dal.*` functions.

Limitations and restrictions

- Transactions

  All updates done in the table extension are part of the transaction that is started in the standard LN application. It is not allowed to call `commit.transaction()`, `abort.transaction()` or `db.retry.point()` from within one of the extension hooks. Doing this may lead to fatal applications errors, or data corruption in the database.
- UI

  A table extension has no access to the UI. You cannot start sessions or reports.

User Exit DLL

Older versions of Infor LN had the concept of User Exit DLLs. User Exit DLLs are similar to the extension scripts for table extensions, but are less rich in functionality. User Exit DLLs are still supported, but do not comply with cloud-ready extensions.

If a table extension is present, the User Exit DLL is ignored. If no table extension is present, the User Exit DLL is executed.

Chapter 6: Report extension point

A report extension is used to enrich the data that is used as input for the report. This applies to reports for which a design is present in Infor Reporting and reports that are personalized with Infor LN Report Designer. A report extension can also be used to redirect the data to an alternative report.

Examples:

- To add the CDFs of the Purchase Order Header to the Purchase Order report.
- To write additional rows with data from one of your own tables. This applies to Infor Reporting only.
- To run an own Sales Order Acknowledgment report. that deviates completely from the standard one.

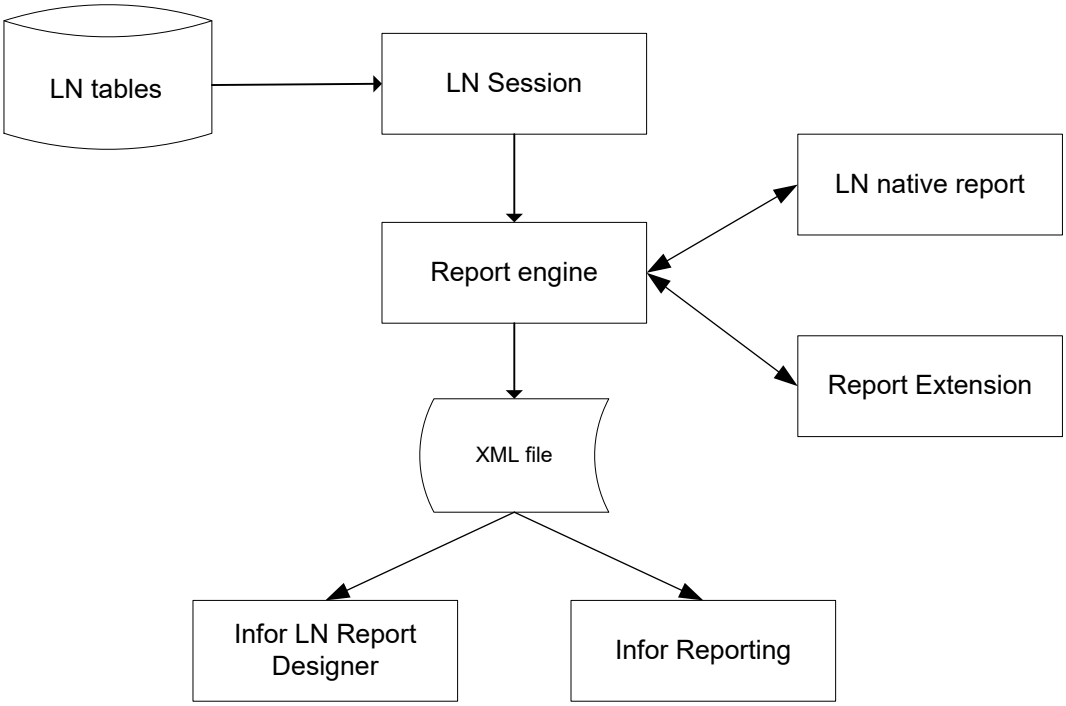
When the report is printed data is sent by the LN session to the native LN report. In case of Infor Reporting, the unformatted data of this native LN report serves as a data source. The report extension adds the fields and the rows to this data source.

In case of Infor LN Report Designer, the added fields by the report extension are available as report input fields to be used in the report layouts.

For the report extension point, three extension types exist:

- Report
- Table selection
- Calculated field

This diagram shows the position of the report extension:



When the Report Extension is created, you must change the report design with Infor Reporting's Report Studio or Infor LN Report Designer to add the new fields to the report.

See these guides:

- *Infor Enterprise Server Connector for Infor Reporting Development Guide*
- *Infor LN Report Designer Development Guide*

Report

The properties and hooks that are defined for the extension type "Report", can intervene in the writing of data rows into the XML file. This XML file is used as input for the reporting tools.

This table shows the available properties:

Name
Include all CDFs

This table shows the available hooks:

Name	Signature
Declarations	
Functions	
Write Row	void write.row()

Name	Signature
Get Alternative Report	string get.alternative.report()

Include all CDFs property

If you check this property, all CDFs of tables, of which already fields are used in the report, are added to the data rows in the XML file. Those tables can be found in the list of tables that is displayed to add a Table Selection for the report.

By default, this property check box is selected when you add a Report extension. Including all CDFs was the default behavior in ES 10.4.2. If you clear this property check box, you can include all CDFs at table level (in the Table Selection) or select individual CDFs or ignore all CDFs.

If the report has linked tables with a lot of CDFs and you do not need most of them, for performance reasons we recommend that you uncheck this property and select the individual CDFs at table level.

Declarations hook

Use this hook to declare tables and variables that must be globally available in all hooks of the extension. Also, the references to include files and DLLs that are used by the extension must be coded in this hook with #include and #pragma.

Tables that are selected in the extension type Table Selection are implicitly declared, so they do not have to be added to this hook.

Example:

```
#include        <bic_text>
        table   txprc100          |* Prices
                string  date.string(14)
                boolean retb
#pragma used dll "otxprcdll0001"
```

Functions hook

Use this hook to code (common) functions to use in the other hooks of the report extension. This helps you in reusing code and to keep the other hooks small and clear.

Example:

```
function string format.date(long i.date)
{
        return(utc.to.iso(i.date, UTC_ISO_DIFF))
}
```

```
function string get.item.description(domain tcitem i.item)
{
        domain  tcdesc     dsca
        select  tcibd001.dsca:dsca
        from    tcibd001
        where   tcibd001.item = :i.item
        as set with 1 rows
        selectdo
                return(dsca)
        endselect
        return("??????????????")
}
```

Write Row hook - Infor Reporting only

You can use this hook in these situations:

- Writing additional rows to the XML file.
- Calculating values for Calculated Fields.

The Write Row hook is executed before the standard row is written to the XML file. If the standard row must be written before the hook is executed, you can force the standard row being written anytime you prefer. This can be achieved by calling the report.super() function.

Example:

```
function extern void write.row()
{
        ext.alternative = tcyesno.no
        report.super()
        ext.alternative = tcyesno.yes
        select  tcibd005.*
        from    tcibd005
        where   tcibd005.item = :tdpur401.item
        selectdo
                rpi.write.additional.row()
        endselect
}
```

In this example there is also an `ext.alternative` Calculated Field. To filter to distinguish the standard rows and the additional rows in the Infor Reporting design, this field is added. It must have the value `no` for the standard rows and value `yes` for the additional rows. The additional rows are written for each record found in `tcibd005` for the current Item.

For `tcibd005` there must be a Table Selection to select the individual fields, but that Table Selection does not require a Table Read hook. The `ext.alternative` Calculated Field does not require a Calculate Value hook, because the value is calculated here.

Get Alternative Report hook

Use this hook to specify an alternative report for the current one. When the current report (the report for which the extension has been created) is opened by a print session, not this report will be opened, but the one returned by this hook. This is useful when the standard LN application opens a report which has not been linked to a session; in case a report is linked to a session, it can be made invisible with the session extension and the alternative one can be added as a custom linked report.

Example:

```
function extern string get.alternative.report()
{
        domain  tcyesno   ext.spec.report

        import("ext.spec.report", ext.spec.report)
        if ext.spec.report = tcyesno.yes then
                return("txibd040114000")
        else
                return("")
        endif
}
```

In this example the alternative report is opened if the session field, custom fields can be added in the session extension, `ext.spec.report` is set to the value “yes”.

Table Selection

With the properties and hooks defined for the extension type Table Selection, you can easily include fields from the selected table in the XML.

When you add a Table Selection, the tables that are already linked to the report are displayed. If the table of which to add fields is not in the list, specify “Other Table” and you can select any table.

This table shows the available properties:

Name
All Customer Defined Fields
All Standard Fields
Field List

This table shows the available hooks:

Name	Signature
Table Read	void <table>.read()

All Customer Defined Fields property

If you select this property check box, all CDFs if the selected table are included in the XML. Note that if you checked the Include All CDFs property on Report level, this property is checked and cannot be changed.

If the table has a lot of CDFs and you do not need them all, for performance reasons we recommend that you clear this property check box and select the individual CDFs in the Field List property.

All Standard Fields property

If you check this property, all standard fields of the selected table are included in the XML.

If the table has a lot of fields and you do not need them all, for performance reasons we recommend that you clear this property check box and select the individual fields in the Field List property.

Field List property

The Field List property can be filled only if not all CDFs and standard fields of the table are already selected with the All Customer Defined Fields and All Standard Fields properties. Click **Details** in the property value cell to get the list of available fields. Select the ones you require in the XML data source.

Table Read hook

Use this hook to write the SQL query to read the data of the table. There are two cases for which it is not required to implement this hook for a Table Selection:

- The table is already linked to the report. However, it can be that not all fields of the table are available; in that case, still a Table Read hook is required to read those additional fields.
- The table data is read in the Write Row hook at report level. This is mandatory if you need data of multiple table records being sent in the XML file.

To select the correct data from the tables, all fields that are sent from the print session to the native report are available. Those fields can be found in the Reports session (ttadv3530m000), option Report Input Fields.

Example:

```
function extern void tccom100.read()
{
        select tccom100.*
        from   tccom100
        where  tccom100.bpid = :tdpur400.otbp
        as set with 1 rows
        selectdo
```

```
        endselect
}
```

Calculated Field

Use a Calculated Field extension type if you need additional fields (non-table fields) in the XML data source.

Examples:

- Aggregations of table fields (average, sum, etc.)
- Results of calculations with standard report fields or fields made available with the Table Selections
- Results of called library functions

This table shows the available properties:

Name
Name
Description
Label
Domain

This table shows the available hooks:

Name	Signature
Calculate Value	void <name>.calculate()

Name property

The Name property is used for the variable name. It is prefixed with "ext.". The maximum length of a variable name is 17, including the prefix. This variable name is the name to be used in the Calculate Value hook or the Write Row hook. This is also the name of the field in the XML data source and available at design time in Infor Reporting's Report Studio and Infor LN Report Designer.

Description property

The Description is sent in the XML data source and available at design time in Infor Reporting's Report Studio. If you must print the reports in different languages (based on user language or the recipient language), do not use the Description property, but link a label to the field with the Label property.

The Description property cannot be filled if the Label property is used.

Label property

Use this property if the report must be printed in different languages. You can select an existing label, or create a new label in the Extensions package. A label can have descriptions in different languages and multiple length variants.

See the *Infor LN Studio Application Development Guide*.

The Label property cannot be filled if the Description property is used.

Domain property

The Domain property is required to define the data type of the Calculated Field. You can select an existing domain or create a new domain in the Extensions package.

See the *Infor LN Studio Application Development Guide*.

Calculate Value hook

Use this hook to calculate the value for the calculated field. The value must be assigned to the variable with the name of the Name property.

In this hook, all fields are available that are sent from the print session to the native report. Those fields can be found in the Reports session (ttadv3530m000), option Report Input Fields. Additionally, all fields are available of the tables read in the Table Read hooks of the Table Selections.

This hook is called by the report engine just before the row of data is written to the XML data source. If you need the calculated field values in the Write Row hook, do not use the Calculate Value hook, but calculate the value in the Write Row hook itself.

Example:

```
function extern void ext.no.po.calculate()
{
        ext.no.po = 0
        select  count(tdpur400.orno):ext.no.po
        from    tdpur400
        where   tdpur400.otbp = :tccom100.bpid
        selectdo
        endselect
}
```

For performance reasons, you can decide to calculate multiple fields in one Calculate Value hook; in that case, you can omit the hooks for the other fields. You can also calculate the values in the Write Row hook. Note that the order in which the Calculate Value hooks are executed is arbitrary.

Functions

In the hooks of a report extension you can use all trusted functions to do string manipulation, calculations, comparisons, etc.

See Trusted / Untrusted concept

Typical function to be used in a report extension:

- rpi.write.additional.row()

This function can be used in the Write Row hook to write additional rows in the XML data source.

Embedded SQL and the sql.* functions are available to read additional data from the LN database.

Limitations and restrictions

- Transactions
  Transactions in a report extension are not supported.
- UI
  A report extension has no access to the UI. You cannot start sessions or (other) reports.
- Native LN reports
  Adding additional fields or making changes in the layouts is not supported for Native LN reports.

Chapter 7: Session extension point

A session extension is used to add fields and commands on the session screen. This applies both to overview screens (grids) and detail screens.

For example to:

- Add the number of Purchase Orders for a Business Partner on the BP overview.
- Show the current weather for a Service Order location.
- Link a new developed print session to an overview session.
- Link a new developed report to a print session.

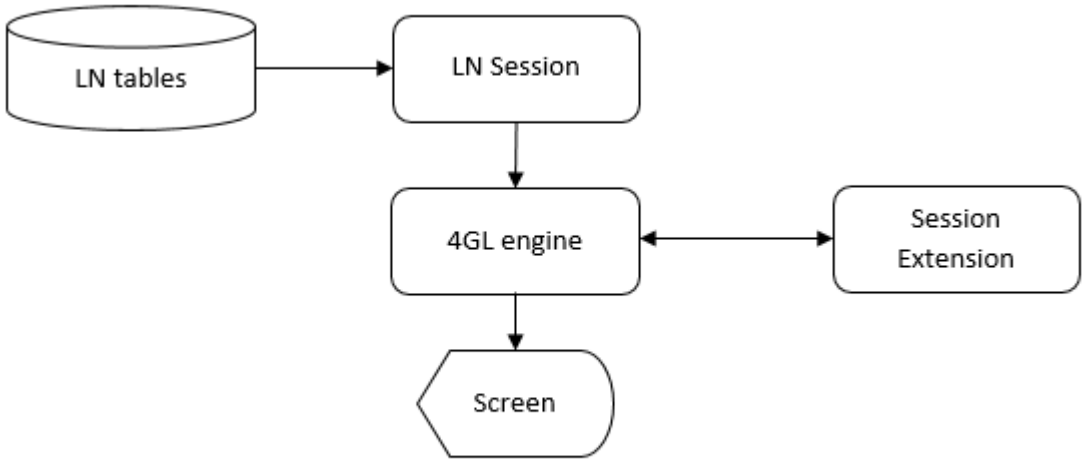
Fields and commands that are added by the session extension are automatically visible the session. With the form personalization options, fields can be moved to the desired location and commands can be added to the toolbar.

For the session extension point these extension types are available:

- Session
- Table selection
- Customer Defined Field
- Standard Field
- Custom Field
- Calculated field
- Standard Linked Report
- Custom Linked Report
- Standard Command
- Standard Form Command
- Custom Form Command

Note: The extension types that are available for a session extension depend on the type of session.

This diagram shows the position of the session extension:



Note on filtering: You can filter on fields that are added by Table Selections. You can also filter on Calculated Fields, except the ones that are calculated with Expression Type “Function”.

Session

With the properties and hooks defined for the extension type Session, you can change the behavior of the session.

This table shows the available properties:

Name
Include CDFs of Used Referenced Tables

This table shows the available hooks:

Name	Signature
Declarations	
Functions	

Include CDFs of Used Referenced Tables property

If you select this property, all CDFs of tables, of which already fields are used in the session screen, are added to the form with the initial hidden state. With Personalize Form you can make those fields visible on the screen.

By default, this property is selected when you add a Session extension. Including CDFs of used referenced tables was the default behavior in Enterprise Server 10.4.1. If you clear the property check box, you can select the required individual CDFs at table level (in the Table Selection).

Declarations hook

Use this hook to declare tables and variables that must be globally available in all hooks of the extension. Also, the references to include files and DLLs that are used by the extension must be coded in this hook with `#include` and `#pragma`.

Tables that are selected in the extension type Table Selection are implicitly declared, so they do not have to be added to this hook.

Example:

```
#include        <bic_text>
        table   txprc100          |* Prices
                string  date.string(14)
                boolean retb
#pragma used dll "otxprcdll0001"
```

Functions hook

Use this hook to code (common) functions to use in the other hooks of the session extension. This helps you in reusing code and to keep the other hooks small and clear.

Example:

```
function string format.date(long i.date)
{
        return(utc.to.iso(i.date, UTC_ISO_DIFF))
}
```

Table Selection

Note: A Table Selection extension type is only possible for sessions of type `Display` and `Maintain`.

With the properties and hooks defined for the extension type Table Selection, you can easily include fields from the selected table in the session's screen.

When you add a Table Selection, you have two options:

- Referenced Table
- Other Table

If you choose Referenced Table, you can select tables that are linked to the main table of the session. This can be a multilevel reference. For example, for a Business Partner, you can make a reference to the Language of the Country of the Address of the Business Partner.

You can select the same Referenced Table multiple times. However, only if the reference path to that table is different. For example, you can reference to a Language from the Business Partner directly, but also by the Country of the Address of the Business Partner. In this case, automatically a new Sequence Number is assigned to the Table Selection.

If you choose Other Table, you can select any other table. In this case, you must specify the query part to join with this table yourself. Note that this is always handled as an inner join. Ensure that the record in the other table exists, otherwise the record of the main table is not displayed. If you cannot be sure that the record in the other table exists, do not use the Table Selection. Use a Calculated Field with the Nested Select option.

See Select property on page 69.

This table shows the available properties:

Name
Field List
Reference Type
Reference Path
Where Clause

Field List property

The Field List property specifies the table fields to be displayed in the session. Click **Details** in the property value cell to get the list of available fields and select the ones to be displayed on the session's screen.

You can leave the Field List blank, to add the Table Selection to be able to use the table fields in the expression for a Calculated Field.

See Table property on page 68.

Reference Type property

This is a read-only property that indicates the reference type that is generated in the query of the session. For a Reference Table the property is `Refers`, for an Other Table it is `Where`.

Reference Path property

This is a read-only property that shows the reference path to the table in the Table Selection of type Referenced Table. The starting point is the main table of the session; the end point is the table in the Table Selection. Click **Details** to see the detailed information of the reference path.

Where Clause property

The Where Clause property can only be filled for Table Selections of type Other Table. This where clause is added to the query of the session to join the data. Click **Details** to specify the where clause in the window.

Example:

```
        txprc001.pcod = tcibd001.cdf_pcod
```

This example shows how data from another table is joined to the main table `tcibd001`. As described earlier, ensure that the data in the joined table exists, otherwise the record of the main table is not visible in the session.

Note: In the Where Clause you cannot use form fields.

Customer Defined Field

Note: A Customer Defined Field extension type is only possible for sessions of type Display and Maintain.

Use a Customer Defined Field extension type to link a zoom session to the CDF.

Example:

- You have a custom table with Colors and want to zoom to the Colors session for a CDF in the Items table.

This table shows the available hooks:

Name	Signature
Get Zoom Session	`string <CDF name>.get.zoom.session()`

Name	Signature
Get Zoom Return Field	`string <CDF name>.get.zoom.return.field()`
Selection Filter	`void <CDF name>.selection.filter()`
Before Zoom	`void <CDF name>.before.zoom()`
After Zoom	`void <CDF name>.after.zoom()`

Get Zoom Session hook

Use this hook to specify the zoom session for the customer defined field. To use a zoom session for a CDF, you must implement the Get Zoom Return Field hook.

Example:

```
function extern string tcibd001.cdf_colr.get.zoom.session()
{
        return("txcom1500m000")
}
```

Note: This hook is called after the form is loaded in the session. If the zoom session depends on the data in the session, use this hook only to set the default zoom session. The specific zoom session based on the data must be set in the Before Zoom hook.

Get Zoom Return Field hook

Specify the field that must be returned of the selected record in the zoom session and specified in the customer defined field. It must be one of the fields that are shown in the zoom session.

Example:

```
function extern string tcibd001.cdf_colr.get.zoom.return.field()
{
        return("txcom100.colr")
}
```

Note: This hook is called after the form is loaded in the session. If the zoom return field depends on the data in the session, use this hook only to set the default zoom return field. The specific zoom return field that is based on the data must be set in the Before Zoom hook.

Selection Filter hook

Use this hook to specify the query extension to filter the records that are shown in the zoom session.

Example:

```
function extern void tcibd001.cdf_colr.selection.filter()
{
        on case tcibd001.citg
        case "001":
               query.extend.where.in.zoom("txcom100.ctyp = txctyp.hard")

               break
        case "002":
               query.extend.where.in.zoom("txcom100.ctyp = txctyp.soft")

               break
        default:
               |* no filter
        endcase
}
```

Before Zoom hook

Use this hook to instruct the zoom session to adapt its behavior. For example, the index the session should use or the record that should be shown as the first one. In the latter case, you must specify the key of that record. Note that standard sessions may not always show the desired behavior because of specific implementations.

Examples:

```
function extern void tcibd001.cdf_prbp.before.zoom()
{
        attr.zoomindex = 2
}
function extern void tcibd001.cdf_prbp.before.zoom()
{
        tccom100.bpid = tcibd001.cdf_prbp
}
function extern void tcibd001.cdf_prbp.before.zoom()
{
    on case tcibd001.cdf_bunt
    case txbunt.cons:
         attr.zoomsession$ = "txcpr0501m000"
         attr.zoomreturn$ = "txcpr001.prid"
         break
    case txbunt.b2b:
         attr.zoomsession$ = "txcpr0502m000"
         attr.zoomreturn$ = "txcpr002.prid"
         break
    default:
         attr.zoomcode = 0
    endcase
}
```

After Zoom hook

Use this hook to react on the chosen entry in the zoom session.

Example:

```
function extern void tirou102.cdf_aitm.after.zoom()
{
        if isspace(tirou102.cdf_aitm(1;9)) then
                tirou102.cdf_aitm = tirou102.cdf_aitm(10)
        endif
}
```

Before Input hook

Use this hook to influence the standard behavior for input fields by setting predefined variables.

Examples:

```
function extern void tcibd001.cdf_prbp.before.input()
{
        attr.dorp = DORP.DEFAULT
}
```

Standard Field

Note: A Standard Field extension type is only valid for sessions of type `Print`, `Update` and `Update_print`.

Use a Standard Field extension type if additional validations of updating of related fields are required in the session's screen.

Examples:

- If an option is selected, another option must be cleared automatically.
- To prevent a session running for a range.

This table shows the available hooks:

Name	Signature
When Field Changes	`void <field name>.when.field.changes()`
Check Input	`void <field name>.check.input()`

When Field Changes hook

Use this hook to react on a change in a field. You can use and set the values of the form fields.

Note: To have the standard fields available for the hooks ensure you have at least Enterprise Server version 10.5.2 with KB 1923135 applied.

Example:

```
function extern void currency.when.field.changes()
{
        if currency = "EUR" then
                prnt.sellpr = tcyesno.yes
        else
                prnt.sellpr = tcyesno.no
        endif
        display("prnt.sellpr")
}
```

Check Input hook

Use this hook to validate a field. You can use and set the values of the form fields. You can only use this hook to set additional restrictions on a field. The standard validations are also applied.

Example:

```
function extern void currency.check.input()
{
        if txcomdll0001.is.currency.blocked.for.printing(currency) then
                set.input.error("@" &
                  sprint$("You cannot select currency %s; it is blocked",

                           currency))
        endif
}
```

Custom Field

A Custom Field extension type is only valid for sessions of these types:

- Print
- Update
- Update_print.

Use a Custom Field extension type if you must specify additional input on the session's screen.

The Extension Modeler can show the option to add a custom field based on a process extension. This is for future use.

In the current version, custom fields are applicable in combination with a report extension. For example, add range fields for a CDF in the table, to print only those records for which the CDF value is within that range.

This table shows the available hooks:

Name	Signature
Calculate Initial Value	`void <field name>.calculate.default.value()`
When Field Changes	`void <field name>.when.field.changes()`
Check Input	`void <field name>.check.input()`
Is Read-only	`boolean <field name>.is.readonly()`
Get Zoom Session	`string <field name>.get.zoom.session()`
Get Zoom Return Field	`string <field name>.get.zoom.return.field()`
Selection Filter	`void <field name>.selection.filter()`
Before Zoom	`void <field name>.before.zoom()`
After Zoom	`void <field name>.after.zoom()`

Calculate Initial Value hook

Use this hook to set the initial value of the custom field. Note that this value is only applied if the form has no saved defaults for the current user.

Example:

```
function extern void ext.colr.t.calculate.default.value()
{
        ext.colr.t = "ZZZZZZZZZZ"
}
```

When Field Changes hook

Use this hook to react on a change in a field. You can use and set the values of the form fields.

Example:

```
function extern void ext.colr.f.when.field.changes()
{
```

```
        ext.colr.t = ext.colr.f
        display("ext.colr.t")
}
```

Check Input hook

Use this hook to validate a custom field. You can use the values of the standard form fields and other custom fields.

Example:

```
function extern void ext.currency.check.input()
{
        if txcomdll0001.is.currency.blocked.for.printing(ext.currency)
then
                set.input.error("@" &
                  sprint$("You cannot select currency %s; it is blocked",

                           ext.currency))
        endif
}
```

Is Read-only hook

Use this hook to set a custom field to read-only. This hook is executed only once at form initialization. To set a field read-only if other fields are changed, use the `When Field Changes` hook to set the custom field to read-only. You can use the `disable.fields()` function.

Example:

```
function extern boolean ext.currency.is.read.only()
{
        return(txcomdll0001.is.currency.fixed(ext.currency))

}
```

Get Zoom Session hook

Use this hook to specify the zoom session for the custom field. To use a zoom session, you must also implement the Get Zoom Return Field hook.

Example:

```
function extern string ext.colr.f.get.zoom.session()
{
        return("txcpr0501m000") }
```

This hook is called after the form is loaded in the session. If the zoom session depends on the data in the session, use this hook to set the default zoom session. The specific zoom session that is based on the data must be set in the Before Zoom hook.

Get Zoom Return Field hook

Use this hook to specify the field that must be returned of the selected record in the zoom session and specified in the custom field. It must be one of the fields that are shown in the zoom session.

Example:

```
function extern string ext.colr.f.get.zoom.return.field()
{
        return("txcpr001.colr") }
```

This hook is called after the form is loaded in the session. If the zoom return field depends on the data in the session, use this hook to set the default zoom return field. The specific zoom return field that is based on the data must be set in the Before Zoom hook.

Selection Filter hook

Use this hook to specify the query extension to filter the records that are shown in the zoom session.

Example:

```
function extern void ext.currency.selection.filter()
{
        query.extend.where.in.zoom("tcmcs008.rapr = tcyesno.yes")
}
```

Before Zoom hook

Use this hook to instruct the zoom session to adapt its behavior. For example, the index the session must use or the record that must be shown as the first one. In the latter case, you must specify the key of that record. Standard sessions do not always show the desired behavior because of specific implementations.

Examples:

```
function extern void ext.currency.before.zoom()
{
        attr.zoomindex = 2
}
```

After Zoom hook

Use this hook to react on the chosen entry in the zoom session.

Example:

```
function extern void ext.item.after.zoom()
{
        if isspace(ext.item(1;9)) then
                ext.item = ext.item(10)
        endif
}
```

Calculated Field

A Calculated Field extension type is only valid for sessions of type `Display` and `Maintain`.

Use a Calculated Field extension type if additional fields are required in session's screen.

Examples:

- Aggregations of table fields (average, sum, etc.)
- Fields of tables that cannot be joined, because the record to join might not exist
- Results of calculations with standard main table fields or fields made available with the Table Selections
- Results of called library functions

This table shows the available properties:

Name
Name
Description
Label
Domain
Display Length
Table

Name
Expression Type
Simple Expression
Select
From
Where

This table shows the available hooks:

Name	Signature
Calculate Value	`void <name>.calculate()`

The Calculate Value hook is only available for Calculated Fields with Expression Type `Function`:

Name property

The Name property is used for the variable name. It is always prefixed with `ext.`. The maximum length of a variable name is 17, including the prefix. This variable name is the name to be used in the Calculate Value hook.

Description property

The Description is displayed as column header in an overview session or before the field in a details session. If you require the descriptions in different languages (based on user language), do not use the Description property. Link a label to the field with the Label property.

The Description property cannot be specified the Label property is used.

Label property

Use this property if the field description must be displayed in different languages. You can select an existing label, or create a new label in the Extensions package. A label can have descriptions in different languages and multiple length variants.

See the *Infor LN Studio Application Development Guide*.

The Label property cannot be filled if the Description property is used.

Domain property

The Domain property is required to define the data type of the Calculated Field. You can select an existing domain or create a new domain in the Extensions package.

See the *Infor LN Studio Application Development Guide*.

Display Length property

Use this property to limit the display length of the field. If you do not specify this property, the field is created on the screen with the length of the domain.

Table property

This property is a link to a Table Selection of which to use fields in a Simple Expression.

Expression Type property

Select an Expression Type to indicate how the value of the Calculated Field must be determined:

This table shows the available Expression Types:

Expression Type	Description
Simple Expression	Use this Expression Type if the table data is already available. The data is available if the used table fields in the Simple Expression are part of: • The main table of the session. • A table that is linked with the Table property For a Simple Expression, you must fill the Simple Expression property.
Query Extension	Use this Expression Type if the table data is not yet available, but can be added to the session's query as an inner join. The complete query extension (`Select`, `From` and `Where` properties) must be specified to determine the value of the Calculated Field. Use this Expression Type if you are sure the data read by the query extension exists. If you cannot be sure that the data exists, use the `Nested Select` Expression Type. You cannot use this Expression Type if you need aggregated values (`count`, `average`, etc.). For aggregations, you must use the `Nested Select` Expression Type.
Nested Select	Use this Expression Type if the data is not yet available and an inner join is not possible, because the data may not be present. This Expression Type is also required for aggregations of table data. The complete query (`Select`, `From` and `Where`) must be coded in the `Select` property.

Expression Type	Description
Function	Use this Expression Type if the calculation of the field cannot be expressed in a query. For example, a complex calculation or a web service call. Note that Calculated Fields with Expression Type “Function” are not enabled for (easy) filtering.

Simple Expression property

A Simple Expression computes a value with table fields that are available in the main table of the session or are part of a table. This is added to the session by a Table Selection.

Examples:

```
tccom100.bpid(1;3) & "-" & tccom100.clan
case tccom100.clan
     when "ARA" then "Arabic"
     when "NLD" then "Dutch"
     else "Other"
end
```

See the SQL chapter in the *Infor ES Programmer's Guide*.

Note: In the Simple Expression, you cannot use form fields.

Select property

Use the `Select` property for Expression Type `Query Extension` or `Nested Select` to define the fields to add to the standard session query.

Expression Type Query Extension

The `Select` property must contain one single field or an expression that results in one value.

Examples:

```
tcmcs046.dsca
case tcmcs046.clan
     when "ARA" then "Arabic"
     when "NLD" then "Dutch"
     else "Other"
end
```

Expression Type Nested Select

The `Select` property must contain a complete query to determine the value of the calculated Field.

Examples:

```
select  count(*)
from    tdpur400
where   tdpur400.otbp = tccom100.bpid
select  txprc001.pric
from    txprc001
where   txprc001.item = tcibd001.item
```

Note: In the `Where` Clause of the `Nested Select` you cannot use form fields.

From property

Use the `From` property to specify the tables to be used for an Expression Type `Query Extension`.

Where property

Use the `Where` property to join the tables for an Expression Type `Query Extension`.

Note: In the `Where` Clause you cannot use form fields.

Calculate Value hook

Use this hook to calculate the value for the calculated field. The value must be assigned to the variable with the name of the `Name` property. You must implement this hook for Expression Type `Function`.

In this hook all fields of the main table are available. Those fields can be found in the **Table Definitions session (ttadv4520m000)**. Additionally, all selected fields from the Table Selections and the Calculated Fields with other Expression Types than `Function` are available. You cannot use other Calculated Fields with Expression Type `Function`, because the order in which the Calculate Value hooks are executed is arbitrary.

Example:

```
function extern void ext.price.calculate()
{
        txprcdll0001.calculate.price(
                                tcibd001.item,
                                tcmcs023.catg,
                                utc.num(),
                                ext.price)
}
```

Standard Linked Report

Use a Standard Linked Report extension type to code remove (conditionally) the report from the session.

This table shows the available hooks:

Name	Signature
Is Visible	`boolean <report>.is.visible()`

Is visible hook

Use this hook to remove standard linked reports from the session.

This hook can use actual values of the form fields. The availability of a certain report depends on the options chosen in the form.

Example:

```
function extern boolean tcibd040111000.is.visible()
{
        if ext.details = tcyesno.yes then
                return(true)
        else
                return(false)
        endif
}
```

Custom Linked Report

Use a Custom Linked Report extension type to add a report to a session.

For example, to add a report, you created in the `tx` package to print item data in a completely different layout.

These are the available properties:

- Report Group
- Sequence

This table shows the available hooks:

Name	Signature
Is Visible	boolean <report>.is.visible()

Report Group property

This property is used to set the report group to which the custom linked report must belong. At runtime the user must select which report from the report group must be printed. If the session does not have different report groups,you must add the report to the existing group. You can check this in the session definition to which you link the custom report. You cannot add report groups in the extension.

Sequence property

This property is used to set the sequence of the report within the report group.

Is Visible hook

This hook is used to conditionally add the custom linked report. This hook can use the actual values of the form fields. The availability of a report depends on the options selected in the form.

Example:

```
function extern boolean txcpr040111000.is.visible()
{
        if ext.details = tcyesno.yes then
                return(true)
        else
                return(false)
        endif
}
```

Standard Command

Use a Standard Command extension type to code additional logic around a session's standard command, such as `mark. delete`, `print`, `edit. text`, etc.

Examples:

- To prevent Excel Import in a session.
- To run an own session after a standard command is executed.

This table shows the available hooks:

Name	Signature
Is Visible	`boolean <command>.is.visible()`
Is Enabled	`boolean <command.is.enabled()`

Name	Signature
Before Command	`void <command>.before.command()`
After Command	`Void <command>.after.command()`

Do not use the hooks of the session extension to influence the behavior of updating tables. We recommend that you use the table extension, for example the `Method Is Allowed` and `Before Save` hooks.

In an overview session with multiple records, you must be aware that multiple records are selected. The values available in the selection are the ones of the last (un)selected record. If the command disabling/enabling is dependent on all selected records, you must iterate over the selected records.

Is Visible hook

Use this hook to remove standard commands from the session. Depending on the standard command, the command can remain visible but disabled. This is the case when removing the command would change the standard toolbar.

This hook should not use actual values of the form fields. The code in the hook is processed before actual data is read from the database or the form. To control the availability of the command based on data on the screen, you can use the `Is Enabled` hook. You can use data that is not related to actual contents of the screen, for example parameter data.

Example:

```
function extern boolean cmd.ssi.import.is.visible()
{
|* Don't allow excel import
        return(false)
}
```

Is Enabled hook

Use this hook to disable standard commands in the session. This hook can use actual values of the form fields.

Note: This hook only applies to commands that use the actual data. Commands that are independent of actual data can only be disabled by the `Is Visible` hook. For example; Close (`abort.program`), New (`add.set`) in a type-2 form, an overview without view fields, etc.

Example:

```
function extern boolean dupl.occur.is.enabled()
{
|* Don't allow copying purchased items
```

```
        return(tcibd001.kitm <> tckitm.purchase)
}
```

Before Command hook

Use this hook to perform additional actions before the command is executed. This hook can use actual values of the form fields. You can cancel the execution of the command by calling the `choice.again()` function.

Example:

```
function extern dupl.occur.before.command()
{
|* Don't allow copying purchased items
        if tcibd001.kitm = tckitm.purchase then
                message("It is not allowed to copy purchased items; " &
                  "add a new item to ensure actual defaults are applied.")

                choice.again()
        endif
}
```

This is an alternative for the Is `Enabled` hook. You can keep the command enabled and this hook gives a message why a record cannot be copied.

After Command hook

Use this hook to perform additional actions after the command is executed. This hook can use actual values of the form fields.

Example:

This example shows that the delete action is aborted if one of the selected records cannot be deleted.

```
function extern void mark.delete.after.command()
{
|* Not allowed to delete if purchased item is in selection
        g.pur.selected = false
        do.selection(false, check.pur)
        if g.pur.selected then
                message("Not allowed to delete purchased item")
                choice.again()
        endif
}
```

In the `Function` hook:

```
function check.pur()
{
```

```
        if tcibd001.kitm = tckitm.purchase then
                g.pur.selected = true
        endif
}
```

Standard Form Command

Use a Standard Form Command extension type to code additional logic around a session's standard form command.

Examples:

- To remove the standard form command.
- To run an own session after a standard form command is executed.
- To disable a standard form command in case a certain condition applies.

This table shows the available properties:

Name
Overwrite Description
Description Label
Short Description
Long Description

This table shows the available hooks:

Name	Signature
Is Visible	`boolean <command>.is.visible()`
Is Enabled	`boolean <command.is.enabled()`
Before Command	`void <command>.before.command()`
After Command	`Void <command>.after.command()`

In an overview session with multiple records, you must be aware that multiple records are selected. The values available in the selection are the ones of the last (un)selected record. If the form command disabling/enabling is dependent on all selected records, you must iterate over the selected records.

Overwrite Description property

If you select this property check box, you can overwrite the standard description of the standard form command. In this case, you must either specify the Description Label property or the Description property.

Description Label property

Use this property to have different descriptions for users that are working in different languages. You can select an existing label, or create a new label in the Extensions package. The label used must have the context `General use`. A label can have descriptions in different languages and multiple length variants; for the form command, you can specify two length variants. The one for the short description (used on text buttons) cannot be longer than 17 characters.

See the *Infor LN Studio Application Development Guide*.

The `Description Label` property cannot be filled if the Short/Long Description property is used.

Short Description property

Use this property if your standard form command description is not language dependent. This is the description that is used if the form command is available as a button.

The `Short Description` is read-only in case the `Overwrite Description` property is not checked or the `Description Label` property is filled. In those cases, the `Short Description` shows the description that is used when the form command is displayed at runtime.

Long Description property

Use this property if your standard form command description is not language dependent. This is the description that is used in the menus of the toolbar (Views, References, Actions or a session specific one).

The `Long Description` is read-only in case the `Overwrite Description` property is not checked or the `Description Label` property is filled. In those cases, the `Long Description` shows the description that is used when the form command is displayed at runtime.

Is Visible hook

Use this hook to remove standard form commands from the session.

This hook should not use actual values of the form fields. The code in the hook is processed before actual data is read from the database or the form. If you must control the availability of the form command based on data on the screen, you can use the `Is Enabled` hook. You can use data that is not related to actual contents of the screen, for example parameter data.

Example:

```
function extern boolean function.create.bp.easy.is.visible()
{
```

```
|* Quick creation of business partners not allowed for users
|* of department 300
        select  tccom001.cwoc
        from    tccom001
        where   tccom001.loco = :logname$
        as set with 1 rows
        selectdo
                if strip$(tccom001.cwoc) = "300" then
                        return(false)
                endif
        endselect
        return(true)
}
```

Is Enabled hook

Use this hook to disable standard form commands in the session. This hook can use actual values of the form fields.

Example:

```
function extern boolean function.approve.order.line.is.enabled()
{
|* Check approval against company rules
        if txpurdll0001.can.approve.line(tdpur401.orno, tdpur401.pono)
then
                return(true)
        endif
        return(false)
}
```

Note that this hook applies to standard form commands only that are enabled by the standard application if exactly one record is selected. In that case the extension can disable the command. In case the standard form command allows multiple records being selected, the command remains enabled. In that case, you must use the `Before Command` hook to skip the processing if required.

Before Command hook

Use this hook to perform additional actions before the form command is executed. This hook can use actual values of the form fields. You can cancel the execution of the command by calling the `choice.again()` function.

Example:

```
function extern function.approve.order.line.before.command()
{
        string l.mess(200) mb
|* Check approval against company rules
        if not txpurdll0001.can.approve.line.with.mess(
```

```
                        tdpur401.orno, tdpur401.pono, l.mess) then
                message(l.mess)
                choice.again()
        endif
}
```

Note: This is an alternative for the `Is Enabled` hook as described earlier. You can keep the command enabled and this hook gives a message that a line cannot be approved.

After Command hook

Use this hook to perform additional actions after the form command is executed. This hook can use actual values of the form fields.

Example:

```
function extern void function.approve.order.line.after.command()
{
        txpurdll0001.publish.approval(tdpur401.orno, tdpur401.pono)
}
```

Custom Form Command

Use a Custom Form Command extension type to add a form command to a session.

Examples:

- To add a form command to start an own session with the selection made in the standard session.
- To execute an own function to calculate a field value.

This table shows the available properties:

Name
Activation Type
Command Type
Field
Name
Description Label
Short Description
Long Description
Advanced Properties

This table shows the available hooks:

Name	Signature
Is Visible	`boolean <command>.is.visible()`
Is Enabled	`boolean <command.is.enabled()`
Before Command	`void <command>.before.command()`
Command Executed	`void <command>.command.execute()`
After Command	`Void <command>.after.command()`

In an overview session with multiple records, you must be aware that multiple records are selected. The values available in the selection are the ones of the last (un)selected record. If the form command disabling/enabling is dependent on all selected records, you must iterate over the selected records.

Activation Type property

The `Activation Type` is a read-only property that depends on the `Command Type`.

This table shows the possible Activation Types:

Activation Type	Description
session	This type applies to Command Type `Print`. In this case a print session is started.
function	This type applies to Command Type `Form` or `Field`. In this case the Command Execute hook is executed.

Command Type property

Use this property to indicate the type of the custom form command.

This table shows the possible Command Types:

Command Type	Description
Form	Use this command type for custom form commands that must be added in the **Views**, **References** or **Actions** menu.
Field	Use this command type for custom form commands that must be linked to a specific field on the form. The Field property must be specified as well.

Field property

Use this property to specify the form field to which the custom form command with Command Type `Field` must be linked. Both standard form fields and fields that are added by the extension can be selected.

This property can only be specified for custom form commands with Command Type `Field`.

Name property

Use this property to specify the function name for custom form commands with Activation Type `function`. For custom for commands with Activated Type `session` (for Command Type `Print`), the Name property holds the session code of the print session.

Description Label property

Use this property to have different descriptions for users that are working in different languages. You can select an existing label, or create a new label in the Extensions package. The label used must have the context ‘General use’. A label can have descriptions in different languages and multiple length variants. For the form command, you can specify two length variants. The one for the short description (used on text buttons) must not be longer than 17 characters.

See the *Infor LN Studio Application Development Guide*.

The `Description Label` property cannot be filled if the Short/Long Description property is used.

Short Description property

Use this property if your standard form command description is not language dependent. This is the description that is used if the form command is available as a button.

The Short Description is read-only in case the `Description Label` property is filled. In this case, the `Short Description` shows the description that is used when the form command is displayed at runtime.

Long Description property

Use this property if your standard form command description does is not language dependent. This is the description that is used in the menus of the toolbar (**Views**, **References**, **Actions** or a session specific one).

The `Long Description` is read-only in case the **Overwrite Description** check box is cleared or the **Description Label** property is specified. In those cases, the `Long Description` shows the description that is used when the form command is displayed at runtime.

Advanced properties

Use the advanced properties to influence the appearance and behavior of the custom form command. The dialog box displays several properties that control the display, availability and execution of the custom form command.

See the *Infor LN Studio Application Development Guide*.

Is Visible hook

Use this hook to remove custom form commands from the session.

This hook should not use actual values of the form fields. The code in the hook is processed before actual data is read from the database or the form. To control the availability of the form command based on data on the screen, you can use the `Is Enabled` hook. You can use data that is not related to actual contents of the screen, for example parameter data.

Example:

```
function extern boolean function.publish.item.is.visible()
{
|* Publishing only available for production companies
        return(get.compnr() >= 0100 and get.compnr() < 1000)
}
```

Is Enabled hook

Use this hook to disable custom form commands in the session. This hook can use actual values of the form fields.

Example:

```
function extern boolean function.publish.item.is.enabled()
{
|* Publishing only enabled for manufactured items
        if tcibd001.kitm = tckitm.manufacture then
                return(true)
        endif
        return(false)
}
```

This hook only applies if the custom form command is defined with the `One Record Selected` option. In case the custom form command allows selecting multiple records, the command remains enabled. In that case, you must use the `Before Command` hook to skip the processing if required.

Before Command hook

Use this hook to perform additional actions before the custom form command is executed. This hook can use actual values of the form fields. You can cancel the execution of the command by calling the `choice.again()` function. If multiple records can have been selected to be processed.

Example:

```
function extern function.publish.before.command()
{
|* Publishing only for manufactured items
        if tcibd001.kitm <> tckitm.manufacture then
                choice.again()
        endif
}
```

Note: This is an alternative for the `Is Enabled` hook described earlier. You can keep the command enabled and this hook gives a message if an item should not be published.

Command Execute hook

Use this hook to perform the real actions for the custom form command. This hook can use actual values of the form fields.

Example:

```
function extern function.publish.command.execute()
{
        string l.mess(200) mb
|* Publish item
        if not txdll0007.item.publish(tcibd001.item, l.mess)
                message(l.mess)
                choice.again()
        endif
}
```

After Command hook

Use this hook to perform additional actions after the custom form command is executed. This hook can use actual values of the form fields.

Example:

```
function extern void function.publish.after.command()
{
        message(sprint$("Item %s published", strip$(tcibd001.item))
}
```

Functions

In the hooks of a session extension you can use all trusted functions to do string manipulation, calculations, comparisons, etc.

See Trusted / Untrusted concept on page 126

Embedded SQL and the `sql.*` functions are available to read additional data from the LN database.

Calling (own) DLL functions is also possible.

Limitations and restrictions

- Transactions

  Transactions in a session extension are not supported.
- UI

  A session extension can add fields to the UI. However, doing other UI actions, such as starting other sessions, is not supported in the hooks that calculate the values. Starting other sessions is supported in the hooks that are available for the session commands. In cloud-ready extensions you can only start own developed sessions in the Extensions package.

  See Governance on page 126

  A session extension cannot add fields to the view part of a session.

Chapter 8: BOD extension point

A BOD extension is used to publish additional fields with a BOD. You can also process additional fields that are part of an inbound BOD.

Examples:

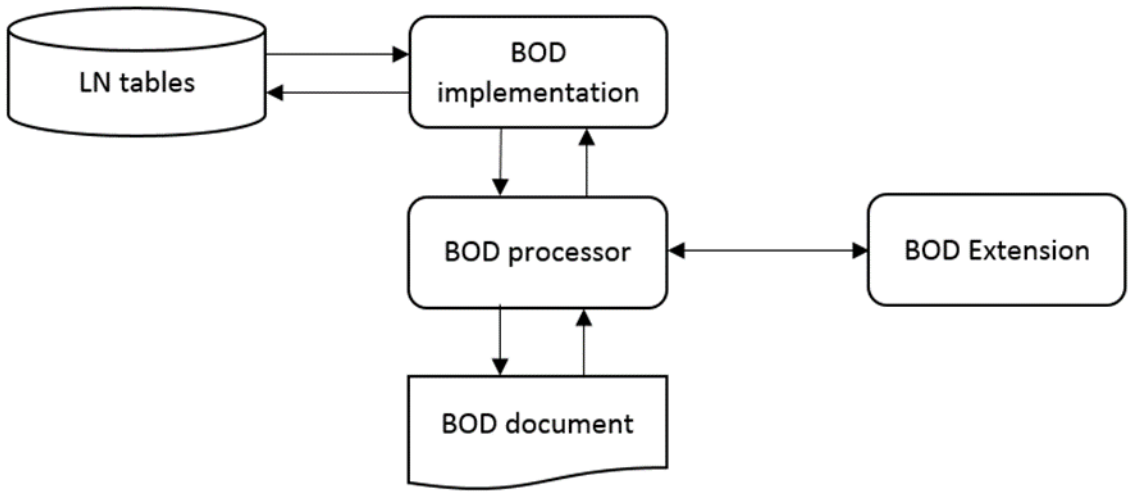
- To include all CDFs of the Purchase Order in the PurchaseOrderBOD.
- To include some standard Business Partner fields in the PurchaseOrderBOD.
- To update an own table with fields in an incoming BOD.

Those fields are added to or processed from the UserArea of the component in the BOD.

For the BOD extension point you have two extension types:

- BOD
- Component Extension

This diagram shows the position of the BOD extension:



BOD

The hooks you can define on BOD level are supporting hooks for the hooks on Component level.

This table shows the available hooks:

Name	Signature
Declarations	
Functions	

Declarations hook

Use this hook to declare tables and variables that must be globally available in all hooks of the extension. Also, the references to include files and DLLs that are used by the extension must be coded in this hook with `#include` and `#pragma`.

Tables that are used in the field mappings of the BOD Components, are implicitly declared. Adding them to this hook is not required.

Example:

```
#include        <bic_text>
        table   txprc100          |* Prices
                string  date.string(14)
                boolean retb
#pragma used dll "otxprcdll0001"
```

Functions hook

Use this hook to code (common) functions to use in the other hooks of the BOD extension. This helps you in reusing code and to keep the other hooks small and clear.

Example:

```
function string format.date(long i.date)
{
        return(utc.to.iso(i.date, UTC_ISO_DIFF))
}
```

Component Extension

With the properties and hooks defined for the extension type Component Extension, you can include fields from the linked tables and other fields to the UserArea of the Component. Note that you can only add fields to the UserArea and not to other structures of the BOD XML. Components that have no UserArea cannot be extended.

When you add a Component Extension, you can select one of the Components that have a UserArea.

This table shows the available properties:

Name
All Customer Defined Fields
Field List

This table shows the available hooks:

Name	Signature
Add Calculated Fields	Not applicable. The lines of code you add in this hook are included in a function that is generated in the Extension Script.
Process Inbound User Area	Not applicable. The lines of code you add in this hook are included in a function that is generated in the Extension Script.

All Customer Defined Fields property

If you select this property, all CDFs of the tables, that are linked to the Component, are included in the UserArea of the Component in the BOD. Click **Details** in the `Field List` property to see which tables are linked to the Component. If you do not select this property, you can select individual CDFs in the `Field List` property.

If you select all CDFs by selecting this property, the CDF is added to the UserArea with the technical field name as element name. For example `tdpur400.cdf_name`. For CDFs of type `List`, the constant name is used as value. You can deviate from those defaults. For example, by choosing a different element name or publishing the enum value instead of the constant. In this case, do not select all CDFs, but select the CDFs individually in the `Field List`. Within the `Field List` you can specify the deviations.

Field List property

In the `Field List` property, you can select all fields to add to the UserArea. Click **Details** in the property value cell to show the list of available fields. Select the ones to add to the UserArea. If you did not select the **All Customer Defined Fields** check box, you also can select the CDFs in the list.

For each selected field two additional properties are available:

Name
Element Name
Use Constant Name

Element Name property

You can specify the Element Name to be used for the field in the BOD XML. If you do not specify the Element name, the technical field name is used.

Use Constant Name property

This property is available for enumerated fields only. If you select this property, the constant name of the enum is published in the BOD XML. For example, for the **Sales Order status** field the string `closed` is published. If you do not select this property, the numeric value is published.

Add Calculated Fields hook

Use this hook to add additional fields to the UserArea. Examples:

- A concatenation of table fields
- Table fields that are not part of the table(s) which is/are linked to the component
- A result that is returned by calling a DLL function
- An XML tree built up with data from any source

The lines of code in this hook are included in the function that the runtime BOD processor calls to fill the UserArea. The structure of this generated function is:

```
function extern long get.additional.elements(
                const   stringi.component,
                ref     long        o.xml)
{
        ...
        on case i.component
        case "component1":
             |* Generated code for selected fields for component1
             |* Hook code for Add Calculated Fields for component1
             break
        case "component2":
             |* Generated code for selected fields for component2
             |* Hook code for Add Calculated Fields for component2
             break
        default:
             break
```

```
        endcase
        ...
        return(0)
}
```

In the Add Calculated Fields hook you can use these macros to add fields to the UserArea:

- addValue
- addAmountValue
- addCodeValue
- addMasterDataReferenceValue
- addQuantityValue
- addDescription
- addEffectiveTimePeriod
- addXML

Note: Those macros only work in the hook itself. You cannot use them in a function you call from the hook.

In the Add Calculated Fields hook you cannot use directly the table fields of the linked table(s) of the Component. The actual values of the table fields are undefined. With some additional macros, you have access to the identifying attributes of the current Component and with those attributes you can query the database to get additional values. These macros are available:

- `getTableIdentifiers.<Component>` (for each component with a UserArea)
- `getIdentifierValueFromIdentifierStructure`
- `getIdentifierDataTypeFromIdentifierStructure`

addValue macro

Use this macro to add a simple value to the UserArea. A simple value has a Name, a Value and a Data Type.

```
addValue(string name, string value, string datatype)
```

This table shows the arguments:

Argument	Description
Name	The element name the field needs to get in the BOD XML
Value	The value of the field; this is always a string
Data Type	The table below shows the supported data types:

Data Type	Remark
String	Single byte or multibyte string
Integer	Integer number (long)
Numeric	Numeric number (float or double)

Data Type	Remark
Date	Date in the format "yyyy-mm-ddThh:mm:ssZ" (GMT) "yyyy-mm-ddThh:mm:ss+hh:mm" (local time later than GMT) "yyyy-mm-ddThh:mm:ss-hh:mm" (local time earlier than GMT) For more information on those date formats, see function utc.to.iso() in the Infor ES Programmers Manual.
Checkbox	“true” or “false”

Example:

```
        addValue("StringElement", "stringValue", "String")
        addValue("IntegerElement", "1", "Integer")
        addValue("NumericElement", "123.45", "Numeric")
       addValue("DateElement", utc.to.iso(utc.num(), UTC_ISO_Z), "Date")

        addValue("CheckboxElement", "true", "Checkbox")
```

addAmountValue macro

Use this macro to add an amount value to the UserArea. An amount value has a Name, a Value and a currency. The data type “amount” is implied.

```
addAmountValue(string name, string value, string currency)
```

This table shows the arguments:

Argument	Description
Name	The element name the field requires to get in the BOD XML
Value	The value of the field; the amount value must be converted to string
Currency	The currency in which the amount is represented

Example:

```
        addAmountValue("PurchaseOrderAmount",
                       str$(tdpur400.amnt), tdpur400.ccur)
```

addCodeValue macro

Use this macro to add a code value to the UserArea. A code value has a Name, a Value, a List Identification which the code is part of and an Accounting Entity. The data type `code` is implied.

```
addCodeValue(string name, string value, string listId,
             string accountingEntity)
```

This table shows the arguments:

Argument	Description
Name	The element name the field needs to get in the BOD XML
Value	The value of the field; this is always a string
ListId	The list to which the code belongs
AccountingEntity	The accounting entity of the list

Example:

```
        addCodeValue("Country", tccom130.ccnt, "CountryCodes",
                     str$(get.compnr()))
```

addMasterDataReferenceValue macro

Use this macro to add a master data reference value to the UserArea. A master data reference value value has a Name, a Value, a Noun and an Accounting Entity. The data type `masterDataReference` is implied.

```
addMasterDataReferenceValue(string name, string value, string noun,
             string accountingEntity)
```

This table shows the arguments:

Argument	Description
Name	The element name the field needs to get in the BOD XML
Value	The value of the field; this is always a string
Noun	The noun of the master data entity
AccountingEntity	The accounting entity of the noun

Example:

```
        addMasterDataReferenceValue("Item", tcibd001.item, "ItemMaster",
                     str$(get.compnr()))
```

addQuantityValue macro

Use this macro to add a quantity value to the UserArea. A quantity value has a Name, a Value and a unit. The data type “quantity” is implied.

```
addQuantity(string name, string value, string unit)
```

This table shows the arguments:

Argument	Description
Name	The element name the field needs to get in the BOD XML
Value	The value of the field; the quantity value must be converted to string
Currency	The unit in which the amount is represented

Example:

```
        addQuantityValue("StockQuantity",
                         str$(total.stock), "pcs")
```

addDescription macro

Use this macro to add a description value to an element that was added before to the UserArea. The call of this macro must immediately follow the call of the macro to add the value because the description is added to the latest element that was added.

```
addDescription(string description)
```

This table shows the argument:

Argument	Description
Description	The description that must be added to the element added previously

Example:

```
        addQuantityValue("StockQuantity",
                         str$(total.stock), "pcs")
        addDescription("The total stock of the item")
```

addEffectiveTimePeriod macro

Use this macro to add an effective time periode to an element that was added before to the UserArea. The call of this macro must immediately follow the call of the macro to add the value because the timeperiod is added to the latest element that was added.

```
addEffectiveTimePeriod(string startDateTime, string endDateTime)
```

This table shows the arguments:

Argument	Description
Start Date Time	The start date time that must be added to the element added previously. This field must be in the format: • "yyyy-mm-ddThh:mm:ssZ" (GMT) • "yyyy-mm-ddThh:mm:ss+hh:mm" (local time later than GMT) • "yyyy-mm-ddThh:mm:ss-hh:mm" (local time earlier than GMT) See function `utc.to.iso()` in the *Infor ES Programmer's Guide*.
End Date Time	The end date time that must be added to the element added previously. Format see Start Date Time.

Example:

```
        addQuantityValue("StockQuantity",
                         str$(total.stock), "pcs")
        addEffectiveTimePeriod(utc.to.iso(utc.num(), UTC_ISO_Z),
                               utc.to.iso(utc.num()+24*60*60, UTC_ISO_Z))
```

addXML macro

Use this macro to add an XML node to the UserArea. The UserArea can only contain `Property` elements, so an XML node to be added must represent a `Property` element.

```
addXml(long xmlnode)
```

This table shows the arguments:

Argument	Description

XML node	The XML node that contains the XML tree to be added to the BOD XML. This XML tree must have the following structure (all non-italic words must be added as shown):
	`<Property>`
	`  <NameValue name="anyName" type="AnyType">`
	`  <myElement>`
	`  <mySubElement>value</mySubElement>`
	`  </myElement>`
	`  </NameValue>`
	`</Property>`

Example:

```
    long property.node
    long name.node
    long xml.node
    long child.node
    property.node = xmlNewNode("Property")
    name.node = xmlNewNode("NameValue", XML_ELEMENT, property.node)
```

```
    xmlSetAttribute(name.node, "name", "BusinessPartnerData")
    xmlSetAttribute(name.node, "type", "AnyType")
    xml.node = xmlNewNode("BPElements", XML_ELEMENT, name.node)
           child.node = xmlNewDataElement("LongName", tccom100.cdf_lnam,

                                      xml.node)
    child.node = xmlNewDataElement("Name", tccom100.nama, xml.node)
    addXML(property.node)
```

getTableIdentifiers.<Component> macro

Use this macro to retrieve the identifying attributes of the current Component that is being processed.

```
long getTableIdentifiers.<Component>(ref long xmlnode)
```

This table shows the arguments:

Argument	Description
XML node	XML node that contains the table identifiers after the call

Example:

```
        long                    ret
        long                    header.xml
        domain  tcorno          orno
        domain  tccom.bpid      otbp
        ret = getTableIdentifiers.PurchaseOrderBOD(header.xml)
        orno = getIdentifierValueFromIdentifierStructure(
                                        header.xml, "tdpur400", "orno")

        select  tdpur400.otbp:otbp
        from    tdpur400
        where   tdpur400.orno = :orno
        selectdo
                select  tccom100.*
                from    tccom100
                where   tccom100.bpid = :tdpur400.otbp
                selectdo
                      addValue("LongBpName", tccom100.cdf_lnam, "String")

                endselect
        endselect
```

In this example the identifying attribute of the current Component `PurchaseOrderBOD` are stored in `header.xml`. With the macro `getIdentifierValueFromIdentifierStructure` the individual table field values can be retrieved. Those values can be used in subsequent queries or function calls.

getIdentifierValueFromIdentifierStructure macro

Use this macro to retrieve the values of the individual identifying attributes.

```
string getIdentifierValueFromIdentifierStructure(
                long xmlnode, string table, string field)
```

This table shows the arguments:

Argument	Description
XML node	XML node that contains the table identifiers (retrieved with macro `getTableIdentifiers.<Component>()`)
Table	The table to retrieve the identifying attribute of
Field	The field name of the identifying attribute to retrieve

Example:

```
        ret = getTableIdentifiers.PurchaseOrderBOD(header.xml)
        orno = getIdentifierValueFromIdentifierStructure(
                                        header.xml, "tdpur400", "orno")
```

getIdentifierDataTypeFromIdentifierStructure

Use this macro to retrieve the data types of the individual identifying attributes.

```
string getIdentifierDataTypeFromIdentifierStructure(
                long xmlnode, string table, string field)
```

This table shows the arguments:

Argument	Description
XML node	XML node that contains the table identifiers (retrieved with macro `getTableIdentifiers.<Component>()`)
Table	The table to retrieve the data type of the identifying attribute
Field	The field name of the identifying attribute to retrieve the data type

Example:

```
        ret = getTableIdentifiers.PurchaseOrderBOD(header.xml)
        datatype = getIdentifierDataTypeFromIdentifierStructure(
                                        header.xml, "tdpur400", "orno")
```

BOD UserArea example

The table shows an example of the BOD UserArea. The left column shows the XML structure of the BOD UserArea, which is created by the BOD extension. The column on the right shows the Add Calculated Fields hook that built this User Area.

BOD UserArea	Add Calculated Fields hook

BOD UserArea

```
<UserArea>
 <Property><NameValue name="NegotiationDate"
  type="DateTimeType">2013-05-
16T07:46:37Z</NameValue>
 </Property>
 <Property><NameValue name="NegotiationLevel"
  type="EnumerationType">hard</NameValue>
 </Property>
 <Property><NameValue name="StringElement"
  type="StringType">stringValue</NameValue>
 </Property>
 <Property><NameValue name="IntegerElement"
  type="IntegerNumericType">1</NameValue>
 </Property>
 <Property><NameValue name="NumericElement"
  type="NumericType">123.45</NameValue>
 </Property>
 <Property><NameValue name="DateElement"
  type="DateTimeType">2016-07-
08T07:38:28Z</NameValue>
 </Property>
 <Property><NameValue name="CheckboxElement"
  type="IndicatorType">true</NameValue>
 </Property>
.<Property><NameValue name="LongBpName"
  type="StringType">LONG BP NAME</NameValue>
 </Property>
 <Property><NameValue name="Name"
  type="StringType">BP Name</NameValue>
 </Property>
 <Property><NameValue name="DatatypeOfOrno"
  type="StringType">DB.STRING</NameValue>
 </Property>
 <MyOwnUserAreaExtension>
  <LongName>LONG BP NAME</LongName>
  <Name>BP Name</Name>
 </MyOwnUserAreaExtension>
```

Add Calculated Fields hook

```
Negotiation Date and Negotiation Level are
selected in the Field List of the
PurchaseOrderBOD Component
  long                    ret, header.xml
  long                    xmlnode
  long                    childnode
  domain  tcorno          orno
  domain  tccom.bpid      otbp
  addValue("StringElement", "stringValue",
"String")
  addValue("IntegerElement", "1", "Integer")
  addValue("NumericElement", "123.45", "Numeric")

  addValue("DateElement",
        utc.to.iso(utc.num(), UTC_ISO_Z), "Date")

  addValue("CheckboxElement", "true", "Checkbox")

  ret = getTableIdentifiers.PurchaseOrderBOD(
         header.xml)
  orno =
  getIdentifierValueFromIdentifierStructure(
         header.xml, "tdpur400", "orno")
  select  tdpur400.otbp:otbp
  from    tdpur400
  where   tdpur400.orno = :orno
  selectdo
    select  tccom100.*
    from    tccom100
    where   tccom100.bpid = :tdpur400.otbp
    selectdo
      addValue("LongBpName", tccom100.cdf_lnam,
               "String")
      addValue("Name", tccom100.nama, "String")
    endselect
  endselect
 addValue("DatatypeOfOrno",
```

BOD UserArea	Add Calculated Fields hook
</UserArea>	getIdentifierDataTypeFromIdentifierStructure(header.xml, "tdpur400", "orno"), "String") xmlnode = xmlNewNode("MyOwnUserAreaExtension") childnode = xmlNewDataElement("LongName", tccom100.cdf_lnam, xmlnode) childnode = xmlNewDataElement("Name", tccom100.nama, xmlnode) addXML(xmlnode)

Process Inbound User Area hook

Use this hook to execute additional actions when the BOD is processed. Examples:

- Update another table based on fields in the UserArea
- Update the linked tables with results of expressions
- The inbound processing is based on the presence of the UserArea but is not limited to fields in the UserArea.
- This processing of the UserArea is executed when the BOD component is completely processed by the BOD handler. The standard fields and the fields of the UserArea that are mapped to fields of the linked tables are processed already. The database records are updated, although not committed yet. To update the linked tables, select the current record again before updating the fields.

The lines of code in this hook are included in the function that the runtime BOD processor calls to process the UserArea. The structure of this generated function is:

```
function extern long process.inbound.user.area(
                const   string   i.component,
                        long     i.xml)
{
        ...
        on case i.component
        case "component1":
             |* Hook code for Process Inbound User Area for component1
             break
        case "component2":
             |* Hook code for Process Inbound User Area for component2
             break
        default:
             break
        endcase
        ...
        return(0)
}
```

You can report errors by calling function `dal.set.error.message()` and do an early return(DALHOOKERROR).

In the **Process Inbound User Area** hook you can use these macros to find the fields and their values in the UserArea:

- getFirstProperty
- getNextProperty
- getNrProperties
- getPropertyNr
- getNamedProperty
- getPropertyName
- getPropertyType
- getPropertyValue
- getAccountingEntity
- getCurrencyID
- getListID
- getNounName

- getUnitCode
- getDescription
- getStartDateTime
- getEndDateTime
- getUserAreaParent

Note: Those macros only work in the hook itself. You cannot use them in a function you call from the hook. There are basically three ways to process the properties in the UserArea:

- Get the required properties by their names with `getNamedProperty()`
- Get the number of properties with `getNrProperties()` and process them one by one with `getPropertyNr()`
- Get the first property with `getFirstProperty()` and the next ones with `getNextProperty()`

The examples used in the descriptions of the macros assume a UserArea in the PersonInBOD with this content:

```
<UserArea>
  <Property>
      <NameValue name="salary" type="AmountType"
                 currencyID="USD">43000</NameValue>
      <EffectiveTimePeriod>
          <StartDateTime>2017-07-21T06:05:13Z</StartDateTime>
          <EndDateTime>2017-08-20T23:59:59Z</EndDateTime>
      </EffectiveTimePeriod>
  </Property>
  <Property>
      <NameValue name="oldWorkCenter"
                 type="MasterDataReferenceType"
                 nounName="WorkCenter"
                 accountingEntity="AE1000">WC1</NameValue>
  </Property>
  <Property>
      <NameValue name="country" type="CodeType"
                 listID="Countries"
                 accountingEntity="AE1000">US</NameValue>
      <Description>United States</Description>
  </Property>
  <Property>
      <NameValue name="skill_01" type="StringType">S00</NameValue>
  </Property>
  <Property>
      <NameValue name="skill_02" type="StringType">S01</NameValue>
  </Property>
  <Property>
      <NameValue name="skill_03" type="StringType">S02</NameValue>
  </Property>
  <Property>
      <NameValue name="contract" type="QuantityType"
                 unitCode="hrs">40</NameValue>
  </Property>
 </UserArea>
```

getFirstProperty macro

Use this macro to get the first Property the UserArea.

```
long getFirstProperty()
```

Return value: the xml node of the first property in the UserArea or 0 if the UserArea is empty.

Example:

```
domain  tcskll  skill.code
long    property.node
string  property.name(50)
long    ret
property.node = getFirstProperty()
while property.node <> 0
    property.name = getPropertyName(property.node)
    if property.name(1;5) = "skill" then
        skill.code = getPropertyValue(property.node)
        if not isspace(skill.code) then
            |* Assign skill to employee
            select  tcppl020.skll
            from    tcppl020
            where   tcppl020.emno = :bpmdm001.emno
            and     tcppl020.skll = :skill.code
            as set with 1 rows
            selectdo
            selectempty
                ret = dal.new.object("tcppl020")
                dal.set.field( "tcppl020.emno" , bpmdm001.emno )
                dal.set.field( "tcppl020.skll" , skill.code )
                ret = dal.save.object("tcppl020")
                if ret <> 0 then
                    dal.set.error.message(sprintf$(
                         "@Cannot add skill %s to employee %s",
                         skill.code, bpmdm001.emno))
                    return(DALHOOKERROR)
                endif
            endselect
        endif
    endif
    property.node = getNextProperty(property.node)
endwhile
```

getNextProperty macro

Use this macro to get the next Property the UserArea.

```
long getNextProperty(long property.node)
```

This table shows the argument:

Argument	Description
Property Node	The node of the previously retrieved property. Note that if the given node is not a property, the result is unpredictable; just the next right sibling of the given node will be returned.

Return value: the xml node of the next property in the UserArea or 0 if the given property node is already the last one.

Example: See the example given for `getFirstProperty()`.

getNrProperties macro

Use this macro to get the number of properties in the UserArea.

```
long getNrProperties()
```

Return value: the number of properties in the UserArea.

Example:

```
domain  tcskll  skill.code
long    property.node
string  property.name(50)
long    ret, i
for i = 1 to getNrProperties()
    property.node = getPropertyNr(i)
    property.name = getPropertyName(property.node)
    if property.name(1;5) = "skill" then
        skill.code = getPropertyValue(property.node)
        if not isspace(skill.code) then
            |* Assign skill to employee
            select  tcppl020.skll
            from    tcppl020
            where   tcppl020.emno = :bpmdm001.emno
            and     tcppl020.skll = :skill.code
            as set with 1 rows
            selectdo
            selectempty
                ret = dal.new.object("tcppl020")
                dal.set.field( "tcppl020.emno" , bpmdm001.emno )
                dal.set.field( "tcppl020.skll" , skill.code )
                ret = dal.save.object("tcppl020")
                if ret <> 0 then
                    dal.set.error.message(sprintf$(
                         "@Cannot add skill %s to employee %s",
                         skill.code, bpmdm001.emno))
                    return(DALHOOKERROR)
                endif
            endselect
        endif
    endif
endfor
```

getPropertyNr macro

Use this macro to get a Property from the UserArea by its index.

```
long getPropertyNr(long index)
```

This table shows the argument:

Argument	Description
Index	The index of the property.

Return value: the xml node of the property with the given index in the UserArea or 0 if the given index is outside the range of properties.

Example: See the example given for `getNrProperties()`.

getNamedProperty macro

Use this macro to get a Property the UserArea by its name.

```
long getNamedProperty(string name)
```

Return value: the xml node of the property in the UserArea or 0 if the property with the given name is not present.

Example:

```
domain  tcamnt  salary
long    property.node
long    ret
ret = 0
property.node = getNamedProperty("salary")
if property.node <> 0 then
    salary = val(getPropertyValue(property.node))
    |* Create/update salary record
    select  txppl001.*
    from    txppl001 for update
    where   txppl001.emno = :bpmdm001.emno
    as set with 1 rows
    selectdo
        ret = dal.change.object("txppl001")
        dal.set.field("txppl001.sala", salary)
    selectempty
        ret = dal.new.object("txppl001")
        dal.set.field("txppl001.emno", bpmdm001.emno )
        dal.set.field("txppl001.sala", salary)
    endselect
    ret = dal.save.object("txppl001")
    if ret <> 0 then
        dal.set.error.message(sprintf$(
             "@Cannot store salary for employee %s; error %d",
             bpmdm001.emno, ret))
```

```
        return(DALHOOKERROR)
    endif
endif
```

getPropertyName macro

Use this macro to get the name of a Property the UserArea.

```
string getPropertyName(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, the result is unpredictable.

Return value: The name of the given property.

Example: See the example given for `getFirstProperty()`.

getPropertyType macro

Use this macro to get the type of a Property the UserArea.

```
string getPropertyType(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, the result is unpredictable.

Return value: The type of the given property.

Example:

```
long    property.node
string  property.type(50)
property.node = getNamedProperty("salary")
if property.node <> 0 then
    property.type = getPropertyType(property.node)
    if property.type <> "AmountType" then
        dal.set.error.message(sprintf$(
             "@Property type %s is invalid for salary property",
             property.type))
        return(DALHOOKERROR)
    endif
    |* Handling for salary ...
endif
```

getPropertyValue macro

Use this macro to get the value of a Property the UserArea.

```
string getPropertyName(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, the result is unpredictable.

Return value: The value of the given property. Note that this value is always a string. To assign it to table fields the correct casting or conversion must be done.

Example: See the example given for `getNamedProperty()`.

getAccountingEntity macro

Use this macro to get the accounting entity of a Property the UserArea. This only applies to Properties of type `CodeType` or `MasterDataReferenceType`

```
string getAccountingEntity(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, or of a type for which the accounting entity attribute is not applicable, the result is unpredictable.

Return value: The accounting entity of the given property.

Example:

```
domain  tccwoc old.wc
long    property.node
string  accounting.entity(20)
property.node = getNamedProperty("oldWorkCenter")
if property.node <> 0 then
    old.wc = getPropertyValue(property.node)
    accounting.entity = getAccountingEntity(property.node)
    txppldll0001.process.old.wc(old.wc, accounting.entity)
endif
```

getCurrencyID macro

Use this macro to get the currency ID of a Property the UserArea. This only applies to Properties of type `AmountType`

```
string getCurrencyID(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, or of a type for which the currency ID attribute is not applicable, the result is unpredictable.

Return value: The currency ID of the given property.

Example:

```
domain  tcccur currency
domain  tcamnt salary
long    property.node
property.node = getNamedProperty("salary")
if property.node <> 0 then
    salary = val(getPropertyValue(property.node))
    currency = getCurrencyID(property.node)
    if currency <> "USD" then
        txppldll0002.convert.salary(currency, salary)
    endif
    |* Handling for salary ...
endif
```

getListID macro

Use this macro to get the list ID of a Property the UserArea. This only applies to Properties of type `CodeType`

```
string getListID(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, or of a type for which the list ID attribute is not applicable, the result is unpredictable.

Return value: The list ID of the given property.

Example:

```
long    property.node
string  property.type(50)
string  list.id(20)
string  list.value(100) mb
string  description(100) mb
property.node = getFirstProperty()
while property.node <> 0
    property.type = getPropertyType(property.node)
    if property.type = "CodeType" then
        list.id = getlistID(property.node)
        list.value = getPropertyValue(property.node)
        description = getDescription(property.node)
```

```
        txppldll0003.update.code.lists(list.id, list.value, description)
    endif
    property.node = getNextProperty(property.node)
endwhile
```

getNounName macro

Use this macro to get the noun name of a Property the UserArea. This only applies to Properties of type `MasterDataReferenceType`

```
string getNounName(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, or of a type for which the noun name attribute is not applicable, the result is unpredictable.

Return value: The noun name of the given property.

Example:

```
long    property.node
string  property.type(50)
string  noun.name(50)
string  property.value(100) mb
string  l.message(200) mb
property.node = getFirstProperty()
while property.node <> 0
    property.type = getPropertyType(property.node)
    if property.type = "MasterDataReferenceType" then
        noun.name = getNounName(property.node)
        property.value = getPropertyValue(property.node)
        if txppldll0003.check.reference(noun.name,
                    property.value, l.message) <> 0 then
            dal.set.error.message("@" & l.message)
            return(DALHOOKERROR)
        endif
    endif
    property.node = getNextProperty(property.node)
endwhile
```

getUnitCode macro

Use this macro to get the unit code of a Property the UserArea. This only applies to Properties of type `QuantityType`

```
string getUnitCode(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, or of a type for which the unit code attribute is not applicable, the result is unpredictable.

Return value: The unit code of the given property.

Example:

```
long    contract
long    property.node
long    ret
string  unit.code(10)
property.node = getNamedProperty("contract")
if property.node <> 0 then
    contract = lval(getPropertyValue(property.node))
    unit.code = getUnitCode(property.node)
    |* Update salary record
    select  txppl001.*
    from    txppl001 for update
    where   txppl001.emno = :bpmdm001.emno
    as set with 1 rows
    selectdo
        ret = dal.change.object("txppl001")
        dal.set.field("txppl001.cont", contract)
        dal.set.field("txppl001.unit", unit.code)
        ret = dal.save.object("txppl001")
        if ret <> 0 then
            dal.set.error.message(sprintf$(
             "@Cannot update contract for employee %s; error %d",
             bpmdm001.emno, ret))
            return(DALHOOKERROR)
        endif
    endselect
endif
```

getDescription macro

Use this macro to get the description of a Property the UserArea.

```
string getDescription(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, the result is unpredictable.

Return value: The description of the given property.

Example: See the example given for `getListID()`.

getStartDateTime macro

Use this macro to get the start date of a Property the UserArea.

```
string getStartDateTime(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, the result is unpredictable.

Return value: The start date of the given property. This is a string in ISO format. If it must be stored in a table field with a datetime domain, it should be converted with function iso.to.utc().

Example:

```
long    property.node
long    ret
property.node = getNamedProperty("salary")
if property.node <> 0 then
    |* Update salary record
    select  txppl001.*
    from    txppl001 for update
    where   txppl001.emno = :bpmdm001.emno
    as set with 1 rows
    selectdo
        ret = dal.change.object("txppl001")
        dal.set.field("txppl001.sala",
                val(getPropertyValue(property.node)))
        dal.set.field("txppl001.stdt",
                iso.to.utc(getStartDateTime(property.node)))
        dal.set.field("txppl001.endt",
                iso.to.utc(getEndDateTime(property.node)))
        ret = dal.save.object("txppl001")
        if ret <> 0 then
            dal.set.error.message(sprintf$(
             "@Cannot update salary for employee %s; error %d",
             bpmdm001.emno, ret))
            return(DALHOOKERROR)
        endif
    endselect
endif
```

getEndDateTime macro

Use this macro to get the end date of a Property the UserArea.

```
string getEndDateTime(long property.node)
```

Argument	Description
Property Node	The node of a previously retrieved property. Note that if the given node is not a property, the result is unpredictable.

Return value: The end date of the given property. This is a string in ISO format. If it must be stored in a table field with a `datetime` domain, it must be converted with function `iso.to.utc()`.

Example: See the example given for `getStartDateTime()`.

getUserAreaParent macro

Note: This macro is only available after KB 1924843 is applied.

Use this macro to get the parent node the UserArea. Through this node you have access to other data in the BOD.

```
long getUserAreaParent()
```

Return value: The node of the parent of the User Area.

Example:

```
string  name(100) mb
|* Get the name of the employee from the PersonInBOD
name = xmlData$(xmlFindFirst("Name", getUserAreaParent()))
```

Functions

In the hooks of a BOD extension you can use all trusted functions to do string manipulation, calculations, comparisons, etc.

See the Trusted / Untrusted concept on page 126.

Embedded SQL and the `sql.*` functions are available to read additional data from the LN database.

Calling (own) DLL functions is also possible.

Limitations and restrictions

- Transactions

  Transactions in a BOD extension are not supported. Updates done for the new fields in the UserArea are done in the context of the transaction that is already started for the BOD itself. It is not allowed to call `commit.transaction(), abort.transaction()` or `db.retry.point()` from within

one of the extension hooks. Doing this can lead to fatal applications errors, or data corruption in the database.

- UI

  A BOD extension has no access to the UI. You cannot start sessions or reports.

CC-library

Older versions of Infor LN had the concept of CC-libraries for BODs. CC-libraries are similar to the extension scripts for BODs, but more complex to construct. CC-libraries are still supported, but do not comply with cloud-ready extensions.

If a BOD extension is present, the CC-library is ignored. If no BOD extension is present, the CC-library is executed.

Chapter 9: Menu extension point

A Menu extension is used to add additional menu items or to hide standard menu items.

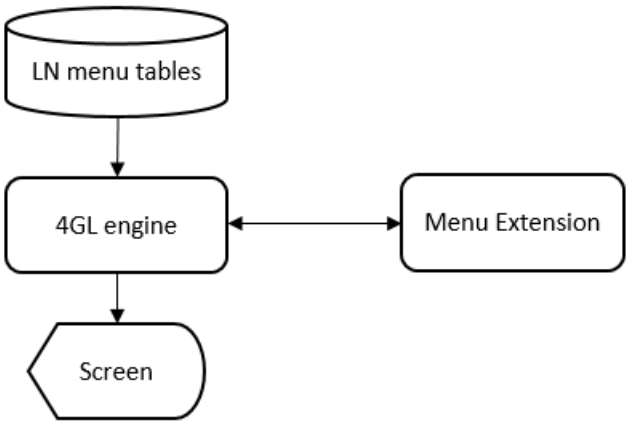
Examples:

- Have a sub menu with all own developed sessions in the Extensions package on the main menu.
- Hide some sessions you do not use.
- Overrule standard menu item descriptions.

For the Menu extension point there are three extension types:

- Menu
- Standard Menu Item
- Custom Menu Item

This diagram shows the position of the Menu extension:



Menu

The hooks you can define on Menu level are supporting hooks for the hooks on Component level.

This table shows the available hooks:

Name	Signature
Declarations	
Functions	

Declarations hook

Use this hook to declare tables and variables that must be globally available in all hooks of the extension. Also, the references to include files and DLLs that are used by the extension must be coded in this hook with `#include` and `#pragma`.

Example:

```
#include         <bic_tt>
         table   txprc000          |* Price Parameters
#pragma used dll "otxprcdll0000"
```

Functions hook

Use this hook to code (common) functions to use in the other hooks of the Menu extension. This helps you in reusing code and to keep the other hooks small and clear.

Example:

```
function boolean own.pricing.implemented()
{
         txprcdll0000.read.parameter()
         return(txprc000.impl = tcyesno.yes)
}
```

Standard Menu Item

With the properties and hooks defined for the extension type Standard Menu Item, you can overrule the standard menu item description or make it (conditionally) invisible for the end user.

This table shows the available properties:

Name
Overwrite Description
Description Label
Description

This table shows the available hooks:

Name	Signature	
Is Visible	`boolean`	`<type>.<name>.is.visible()`

Overwrite Description property

If you select this property, you can overwrite the standard description of the menu item, which is the sub menu description, the session description or the query description. In this case, you must either specify the `Description Label` property or the `Description` property.

Description Label property

Use this property to have different descriptions for users that are working in different languages. You can select an existing label, or create a new label in the Extensions package. The label used must have the context `General use`. A label can have descriptions in different languages and multiple length variants; for the menu item the longest one is shown at runtime.

See the *Infor LN Studio Application Development Guide*.

You cannot specify the `Description Label` property if the `Description` property is used.

Description property

Use this property if your menu item description is not language dependent.

The Description is read-only in case the `Overwrite Description` property is not checked or the `Description Label` property is filled. In those cases, the Description shows the description that is used when the menu is displayed at runtime.

Is Visible hook

Use this hook to remove the standard menu item from the menu.

Example:

```
function boolean menu.tcemm00005001.is.visible()
{
|* Use this hook to remove the menu item from the
|* menu. You can do that based on conditions. To
|* remove it, let the function return the value
```

```
|* false.
        select  txcom001.*
        from    txcom001
        where   txcom001.user = :logname$
        and  tccom001.shem = tcyesno.yes
        as set with 1 rows
        selectdo
                return(true)
        endselect
        return(false)
}
```

Custom Menu Item

With the properties and hooks defined for the extension type Custom Menu Item, you can (conditionally) add menu items to standard menus.

This table shows the available properties:

Name
Type
Code
Overwrite Description
Description Label
Description
Process Info

This table shows the available hooks:

Name	Signature	
Is Visible	`boolean`	`<type>.<name>.is.visible()`

Type property

Choose the Type of the Custom Menu Item.

This table shows the available Types:

Type	Description
Session	The Custom Menu Item is a session.
Menu	The Custom Menu Item is a sub menu.

Type	Description
Query	The Custom Menu Item is an SQL Query,

Code property

The code of the Session, Menu of Query. Sessions and menus can be standard sessions or menus, or own developed sessions or menus in the Extension package. Queries are always own developed queries.

Overwrite Description property

If you select this property, you can overwrite the standard description of the menu item, which is the sub menu description, the session description or the query description. In this case, you must either specify the `Description Label` property or the `Description` property.

Description Label property

Use this property to have different descriptions for users that are working in different languages. You can select an existing label, or create a new label in the Extensions package. The label used must have the context ‘General use’. A label can have descriptions in different languages and multiple length variants; for the menu item the longest one is shown at runtime.

See the *Infor LN Studio Application Development Guide*.

You cannot specify the `Description Label` property if the `Description` property is used.

Description property

Use this property if your menu item description is not language dependent.

The Description is read-only in case the `Overwrite Description` property is not checked or the `Description Label` property is filled. In those cases, the Description shows the description that is used when the menu is displayed at runtime.

Process Info property

Use this property to pass (static) information from the menu to the session. This only makes sense for own developed sessions, because the standard sessions do not retrieve this information.

Is Visible hook

Use this hook to add the custom menu item conditionally to the menu.

Example:

```
function boolean session.txprc5500m000.is.visible()
{
|* Use this hook to remove the menu item from the
|* menu. You can do that based on conditions. To
|* remove it, let the function return the value
|* false.
        return(own.pricing.implemented())
        |* This function is available in the Functions hook
}
```

Functions

In the hooks of a Menu extension you can use all trusted functions to do string manipulation, calculations, comparisons, etc.

See Trusted / Untrusted concept on page 126.

Embedded SQL and the `sql.*` functions are available to read data from the LN database.

Calling (own) DLL functions is also possible.

Limitations and restrictions

- Transactions

  Transactions in a Menu extension are not supported.
- UI

A Menu extension has no access to the UI. You cannot start sessions or reports, or display messages.

- Top menu

The top menu which is displayed in LN UI can be extended. However, if you add a session directly to this top menu, a sub menu is automatically added. The Xi-style does not allow individual sessions in the top menu.

- Testing menu extensions

  Menu extensions can be tested after the Activity Context is set. If a menu is opened already, you must restart LN UI, set the Activity Context and open the menu. For the top menu, you must commit the extension before you see the changes.

  After KB 1884185 is installed, you can refresh the menu (including the top menu) from the Extension Modeler.

Chapter 10: Extension debugging

For debugging the extensions in Infor LN you can use:

- Debug Workbench, which runs within LN UI. Use this debugger for simple extensions debugging or when you do not have Infor LN Studio installed.
- LN Studio debugger. Use this debugger for more complex extensions where also new components (developed in LN Studio) are involved.

  For more information about the setup of the connection to the LN server and debugging in LN Studio, see the *Infor LN Studio Application Development Guide*.

  The information about the setup of LN Studio in combination with extension development is described in the same guide.

Debug Workbench

The Debug Workbench is mainly meant for debugging the generated extension scripts.

See Extension scripts on page 24

Other script components, such as UI scripts of sessions, DALs and other libraries can also be debugged with de Debug Workbench.

Starting the Debug Workbench

The Debug Workbench can be started in these ways:

- Starting from the **Extensions (ttext1500m000)** session.
- Starting from Debug and Profile 4GL.

Starting from the Extensions (ttext1500m000) session

1. Select **Tools > Application Extensibility > Extensions**.
2. Select the extension to debug.

Note that the extension is debugged in the context of the current selected Activity. If no Activity is selected, the committed version of the extension is debugged.

3 Click **Start Debugger** under **Actions**.

4 Specify this information:

Starting from Debug and Profile 4GL

1 Select **Debug and Profile 4GL** in the **Options** menu.

2 Select the **Debug Mode** option.

3 Select **Debug Workbench** as the Debug UI.

4 **Application Name** and **Activity Name** should show the Application and Activity, that you have current in LN Studio. If Application and/or Activity are not specified with the current Activity, change the fields.

5 Click **OK**.

Selection of sources

If you started the Debug Workbench with the **Extensions (ttext1500m000)** session, the generated script for the selected extension is already loaded in the Debug Workbench.

To load (additional) scripts into the Debug Workbench:

1 Select **Select Components** (magnifier glass).

2 Specify the selection string (package, module and remainder of code) in the **Selection** field. For example, specifying **`tx`** displays all script components in the Extension package. Specifying **`txess`** displays the generated extension scripts for session extensions.

3 Select one or more components and click **OK**.

Breakpoints and watchpoints

Before you start the session to debug the script, you must set a breakpoint or watchpoint, otherwise the session process is not suspended. A condition watchpoint suspends the process if the variable changes to a defined value. A modification watchpoint suspends the process if the variable changes.

Breakpoints and watchpoints are visible in the Breakpoints view. In this view the breakpoints and watchpoints can be deleted or temporarily disabled.

Setting or deleting breakpoints

1 Go to the line to set a breakpoint on, or for which to delete the breakpoint.

2 Double-click the line in the area before the line number.

Setting a watchpoint

1 Select a variable to create the watchpoint.

2 Right-click **Select Condition Watchpoint** or **Modification Watchpoint**.
3 For a condition watchpoint, specify the value to suspend on.
4 Click **OK**.

Run the session

After you prepared the breakpoints and watchpoints, run the session that executes the script to debug:

- For a table extension, this can be a session that uses this table as a main table. But it can also be a session for another table, which does a dependent update in your table. For example, if you have a table extension for the inventory allocations table, you can start the sales order lines session to debug your extension.
- For a report extension, you must start the print session that produces the report.
- For a session extension, you must start the session you have extended. Note that it may be necessary to start with another session if your session cannot be started directly from the menu.
- For a BOD extension, you must start a session that publishes the BOD. This can be a session in the normal process flow, but you can also use the session that simulates the publishing of your BOD. Those sessions can be found in the **Common** menu under **BOD Messaging > Publish BODs**.
- For a menu extension, just expand the menu.

Variables and Expressions

The Variables view shows the values of the variables when the process suspends. Which variables are shown depends on the filter. You can change the filter by clicking the arrow down button. Because of the huge list of variables that can be displayed, table fields are not shown in the Variables view. To inspect the values of table fields you can hover over them in the script view or create an expression for it in the Expressions view.

In the Variables view you can also change the value of variables during the debugging process.

Call Stack

The Call stack shows all processes that are started and the state of those processes. It can be cleaned up by right-clicking the Launched Infor LN Sessions and selecting **Remove All Terminated**.

Toolbar

On the toolbar, these commands are available:

Save and Exit:
the state of the current Debug Workbench is saved, although without the specific process information. The open sources, breakpoints, watchpoints and expressions are saved and the next time you start the Debug Workbench, those are available. If you close the Debug Workbench with the “X” in the title bar, the state is not saved.

Search:
see Selection of sources.

Resume:
continue with the suspended process.

Suspend:
the selected process in the Call stack is suspended.

Terminate:
the selected process in the Call stack is killed.

Step Into:
current line is executed, or if the current line contains a function call, the first line of the function is executed.

Step Over:
current line is executed; if the current line is a function call, this function is executed completely and the debug pointer goes to next line.

Step Return:
current function is executed to the end and the debug pointer goes back to the calling function.

Run to Line:
debug pointer is set to the current selected line and the process continues from there.

Skip All Breakpoints:
quick way to disable temporarily all breakpoints.

LN Studio

Debugging with LN Studio is preferred when complex extensions are developed with new tables, sessions, etc. LN Studio handles also other components than scripts. Information of those components can be required during debugging as well.

Preparations

To prepare for debugging:

1. Open Infor LN Studio.
2. If you have already a current activity in the Extension Modeler, go to step 4. Otherwise click **Create a new Activity**.
3. In the **Create a new Activity** dialog box, select your **Project Name**, which is typically “EXT” followed by your package combination.
4. Specify a **Name**, **Description** and **Type** and click **Finish**.
5. Click **Open an Infor LN Studio Activity** in the Activity Explorer view.

6 Select your **Project Name** and click **Next**.

If you are prompted to configure an Administrator Connection, click **Yes**.

7 Configure the Connection Point as described in the *Infor LN Studio Application Development Guide* or click **Help** to get more information. Repeat those steps, if required, for the Development and Runtime connections.

8 Select your **Activity Name** and click **Finish**.

Debugging

To debug the extension scripts:

1 After the last step of the previous paragraph the Activity Explorer can contain already some components. This is the case when the activity is also used in the Extension Modeler. If the extension script to debug is not in the Activity Explorer, you must retrieve it from the LN server. To retrieve an extension script, expand the **tx package** in the Component Explorer and expand **Libraries**. Choose the module which holds the extension script for your extension point and expand it. Select the extension script, right-click it and click **Get**.

Alternative: Click **Select a Software Component (Alt+Q)**, specify **`txes`** in **Component Code** and click **Search Components (Ctrl+Space)**. Select the extension script to debug and click **OK**. A message whether to open the editor for the new software component is displayed. Click **Yes**.

2 Click **Source** at the bottom of the component editor.

3 Set a breakpoint or watchpoint in the source.

4 Switch to LN UI.

5 Select **Debug and Profile 4GL** in the **Options** menu.

6 Check the **Debug Mode** option.

7 Select **LN Studio** as the Debug UI.

8 Application Name and Activity Name must show the Application and Activity, that you current have in LN Studio. If Application and/or Activity are not filled with the current Activity, change the fields.

9 Click **OK**.

10 Start a session that executes the extension script:

a For a table extension, this can be a session that uses this table as a main table. But it can also be a session for another table, which does a dependent update in your table. For example, if you have a table extension for the inventory allocations table, you can start the sales order lines session to debug your extension.

b For a report extension, you must start the print session that produces the report.

c For a session extension, you must start the session you have extended. Note that it can be required to start with another session if your session cannot be started directly from the menu.

d For a BOD extension, you must start a session that publishes the BOD. This can be a session in the normal process flow, but you can also use the session that simulates the publishing of your BOD. Those sessions can be found in the **Common** menu under **BOD Messaging > Publish BODs**.

11 Use the available options of the debug Perspective in LN Studio to debug your extensions.

Chapter 11: New Component Development with Infor LN Studio

You can create new components and new modules within the Infor LN application. The development of new components is done in Infor LN Studio. See these guides:

- *Infor LN Studio Application Development Guide*
- *Infor LN Studio Integration Development Guide*

The topics that are described are relevant for developing extensions and specific configurations.

Before you start to use LN Studio for new component development in combination with Extensibility, read Configuration specifics on page 125.

Infor LN Studio

Infor LN Studio is the Eclipse based development environment for Infor LN. Within the Extensions (tx) package you can create new components such as tables, sessions, messages, etc. The development of components in the tx packages does not differ from the normal Infor LN development. For extensions to be ready for the cloud, some restrictions apply.

This table shows the component types that can be developed:

Component Type	Remark
Session	Including the UI-script that handles the screen events.
Report	LN native reports can be developed, but no layouts can be defined. This report is a container of data: the report input fields define the fields that are available to be sent to Infor Reporting. The design of the report design is made in Infor Reporting's Report Studio .
Table	This is including the DAL that handles the table events. For the table fields standard domains can be used, but also new domains can be created.
Domain	
Library	
Function	
Menu	

Component Type	Remark
Label	
Message	
Question	
Additional File	
Business Object	With an Integration Project.

For more information about Infor LN Studio and component development see these guides:

- *Infor LN Studio Application Development Guide*
- *Infor LN Studio Integration Development Guide*

Configuration specifics

If you use LN Studio for the development of new components to be used in your extensions, the configuration should be done by the Extension Modeler. This applies to the configuration of the Base VRC, Development Environment, Application and Project. When you create the first activity in the Extension Modeler, the setup of a Base VRC, Development Environment, Application and Project are automatically done. Those are the ones you must also use in LN Studio.

This table shows the names of the various configuration items that are generated:

Configuration	Name	Remark
Base VRC	B61O_a_ext	This is the default Base VRC generated in PMC. If you specify another VRC code during Initialize Extensibility, this VRC code is used as Base VRC.
Development Environment	EXT	This value cannot be changed.
Application	EXT<package combination>	This value cannot be changed.
Project	EXT<package combination>	This value cannot be changed.

This configuration applies to the Extensions package (tx) only. To combine classic customizations development (customization VRCs for the standard Infor LN packages) and extensibility. Specify a different VRC code for the Extensions package with a separate Base VRC. The classic customizations are in a separate Application and Project. By setting Activity Context you can link the activity for the Extensions and the activity for the classic customization.

In LN Studio define Related Software Projects.

In LN UI go to **Options > Debug and Profile 4GL**

Chapter 12: Governance

Build the extensions in a way that they are ready for the cloud. This means that upgradability is guaranteed, no infrastructure data is revealed and other customers are not impacted by your extensions.

In Infor LN, you can use several mechanisms to govern your extensions whether they are ready for the cloud:

- Trusted / Untrusted concept
- Performance governors
- File system governors
- Best practices

If you do not develop with the **Extensions ready for Cloud** parameter switched on, the governors are not activated.

See Cloud readiness on page 21 why we do not recommend this.

Trusted / Untrusted concept

With the introduction of trusted functions, the LN infrastructure can restrict the extensions to break the general rules for cloud readiness. Other software added by customers to the LN environment, such as Exchange scripts can also be restricted. Extensions are only allowed to call trusted functions. This applies to the 3GL and 4GL functions of LN's programming language, which are described in the *Infor ES Programmer's Guide*. It also applies to application functions in DLLs, which can be called by the extensions to retrieve and store data with LN's application logic.

These functions are untrusted, and cannot be used within extensions:

- Functions that can harm the infrastructure if they are used in the incorrect way
  Example: `run.prog()`
- Functions that reveal information about the infrastructure
  Example: `hostname$()`
- Functions that are deprecated
  Example: `cf$()`
- Functions that may disturb the flow of the standard application
  Example: `dal.get.error.message()`

- Functions that may use standard components and the interface of the standard components may break

  Example: `wait.and.activate()`

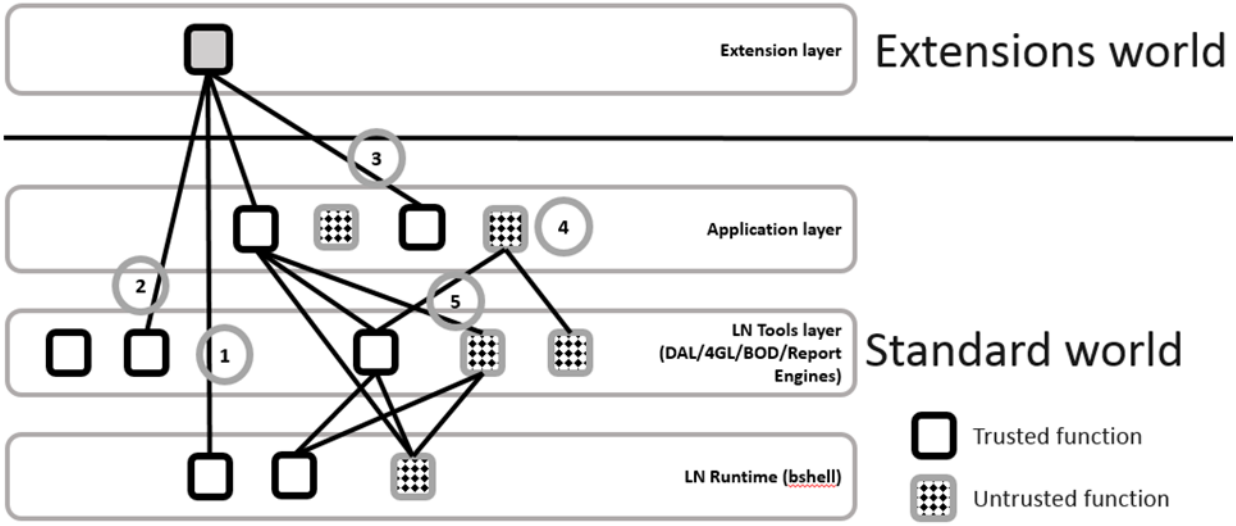
Functions in application DLLs, even if declared as `extern`, are untrusted by default. A new specific trusted layer is available with functions that can be used by extensions. The new functions are called LN APIs.

Note: During LN's 10.5 release this layer was not present yet. It is made available after the release through PMC solutions.

Infor LN Studio shows the available trusted application functions in the help pages.

During compilation of an extension script or any other script in the Extensions (tx) package, messages are raised when untrusted functions are called.

This diagram shows the different layers with trusted and untrusted functions:



1. An extension can call trusted functions in the LN Runtime layer, bshell functions, that are documented as trusted in the *Infor ES Programmer's Guide*.
2. The extension can also call a trusted function in the LN Tools layer; those are also documented as trusted in the *Infor ES Programmer's Guide*.
3. The extension can call trusted functions (LN APIs) in the application layer, that are documented in the help pages of LN Studio and the Extension Modeler.
4. Untrusted standard functions cannot be called from the extensions. All functions in the extensions are untrusted, but those can be called by the extension itself.
5. With the standard software, the distinction between trusted and untrusted is not considered.

Performance governors

The goal of the performance governors is to restrict the impact your extensions can have on the infrastructure. This especially applies to the resource consumption.

Extensions are restricted in:

- Time spent (elapsed time)
- Amount of data written to the file system
- Future versions may have more restrictions

The counter starts each time the extension starts execution. It is reset when the extension stops execution. This implies that for example each hook in a table extension has its own scope regarding the governors.

The exact limits are set by the Infor Cloud team. If you develop extensions in an on-premises environment with the **Extensions Ready for Cloud** option selected, you can change the values in the `$BSE/lib/extensibility/config.<package combination>` file:

Resource	Default	Remark
`governor_elapsed_time`	5000	Elapsed time in milliseconds.
`governor_write_file_quotum`	5000000	5 Mb

Increasing those resources to higher values than required by the Infor Cloud team results in extensions that are not ready for the cloud and may not run after they are moved to the cloud.

File system governors

In cloud environments, the file system access is restricted. Cloud-ready extensions must comply with those restrictions.

Extensions are restricted to certain folders in the BSE. Outside those folders data cannot be read or written. The LN standard software can read outside those folders, but can only write in a restricted number of folders. End users cannot choose all locations to put their files that are output of their sessions.

Extensions are also restricted in writing files with certain file extensions. For example, writing a file with a `.exe` file extension is not allowed.

If you develop extensions in an on-premises environment with the **Extensions Ready for Cloud** option switched on, you can change the values in the `$BSE/lib/extensibility/config.<package combination>` file:

Resource	Default	Remark
`user_writable_dirs`	appdata, tmp	Within `$BSE`.

Resource	Default	Remark
not_trusted_object_accessible	appdata, tmp	Within $BSE.
forbidden_filename_extensions	exe,vb*,com	

Adding folders or extensions to those resources results in extensions that are not ready for the cloud and may not run after they are moved to the cloud.

Best practices

To reduce the risk that your extensions are not compatible with newer versions of LN, we recommend that you comply with the rules.

Database

Queries

The LN development team has the responsibility to keep the data model compatible. Sometimes it is required to change indexes. We recommend that you do not refer to indexes, but to the fields directly. See these code examples:

- This syntax is incorrect because it refers to an index:

```
function extern void tdsls401.read()
{
        select tdsls401.*
        from   tdsls401
        where  tdsls401._index1 = {:rep.orno, :rep.pono}
        selectdo
        endselect
}
```

- Instead, use this syntax, which refers to the fields directly:

```
function extern void tdsls401.read()
{
        select tdsls401.*
        from   tdsls401
        where  tdsls401.orno = :rep.orno
        and    tdsls401.pono = :rep.pono
        selectdo
        endselect
}
```

If trusted application functions are available to read data from the database, use those instead of querying the database directly. See these code examples:

- This syntax is incorrect because it queries the database directly:

```
function extern void ext.item.desc.calculate()
{
        select tcibd001.dsca:ext.item.desc
        from   tcibd001
        where  tcibd001.item = :rep.item
        selectdo
        endselect
}
```

- Instead, use this syntax, which uses a trusted application function:

```
function extern void ext.item.desc.calculate()
{
        ext.item.desc = tcibd.dll0001.read.item.description(rep.item)
}
```

Table definitions

If you create own tables in the Extensions (tx) package, use standard domains if you store copies of standard data in your tables. This ensures that your tables also are reconfigured if the standard tables are reconfigured after a domain change. For enumerated domains, new values can be added. Prepare your extension for possible new values.

Standard table updates

If you update standard tables, use the DAL. Always check the return values of the functions such as `dal.save.object()` and react accordingly.

Standard components

Do not use standard components. Except for the LN APIs, their interfaces can change.

Chapter 13: Extension Deployment

After extensions are developed, they can be exported from one environment and imported in another environment.

The same procedure must be used when extensions must be copied from one package combination to another package combination.

The Product Maintenance and Control (PMC) module must be used to create PMC solutions with the extensions. Use also PMC to install those solutions in the other environment.

For more information on the distributor (export) side of PMC, see the *Infor LN - Development Tools Development Guide*.

For more information on the recipient (import) side of PMC, see the *Infor Enterprise Server - Administration Guide*.

Exporting extensions

1. If not present, create a Base VRC with the **Base VRC's (ttpmc0110m000)** session that has your VRC for the extensions as Export VRC.
2. Create a PMC solution with the **Solutions (ttpmc1100m000)** session.
3. Add your component(s) to the PMC solution. If you add an extension script as a component, the extension data is added.
4. Generate dependencies.
5. Validate the solution.
6. View the report to see whether error messages are printed. If required, take corrective actions and repeat the previous step. Note that extensions that are being modified in an activity are reported as warnings. The committed versions of those extensions are exported.
7. Follow the standard PMC process to export and release the solution.

Importing extensions

1. If not present, define an Update VRC with the **Update VRC's (ttpmc2140m000)** session for the Extensions package (tx).

2. Scan the PMC dump that was created with the export procedure.
3. Run the **Check to Install from Process Solutions (ttpmc2101m000)** session.
4. Check to install reports errors in case the extensions that are to be installed are being modified in the recipient environment. Warnings are reported for extensions that are changed in the recipient environment since the previous PMC install.
5. Complete the installation with the normal PMC process and execute the post-installation instructions.