

UPGRADE

Microsoft Dynamics[®] AX

How to Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

April 5, 2011

White Paper

This document describes how to use the Microsoft Dynamics AX Data Upgrade Framework and to write data upgrade scripts for customer data upgrade data models (Microsoft Dynamics AX tables).

Disclaimer: This is pre-release documentation of a preliminary nature, and is subject to change any time, without notice. Microsoft cannot guarantee the accuracy of any information provided herein.

<http://www.microsoft.com/dynamics/ax>



Microsoft Dynamics is a line of integrated, adaptable business management solutions that enables you and your people to make business decisions with greater confidence. Microsoft Dynamics works like and with familiar Microsoft software, automating and streamlining financial, customer relationship, and supply chain processes in a way that helps you drive business success.

U.S. and Canada Toll-Free 1-888-477-7989

Worldwide +1-701-281-6500

www.microsoft.com/dynamics

This document supports a preliminary release of a software product that may be changed substantially prior to final commercial release. This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results from the use of this document remains with the user. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Copyright © 2011 Microsoft. All rights reserved.

Microsoft, BizTalk, Excel, Microsoft Dynamics, the Microsoft Dynamics logo, Outlook, SharePoint, SQL Server, and Visual Studio are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Contents

Introduction	6
When is a Data Upgrade Script Needed?	7
1. Terms and Abbreviations	7
How to Upgrade Data for a Major Release or Service Pack	10
The Preprocess Upgrade Checklist (source environment)	12
Add Items to the Upgrade Checklist	16
The Data Upgrade Framework for the Target Environment	18
SYS Versions and Data Upgrade of Interim SYS releases	21
Data Upgrade for Service Packs	22
Data Upgrade for Customization.....	22
Create a single upgrade script that combines changes across multiple product versions.....	23
Using Configuration Key to Remove Obsolete Objects after Upgrade	23
Data Upgrade Scripts in the target environment.....	24
Writing Data Upgrade Scripts for Target Environment	26
Writing Data Upgrade Scripts for ISV solution in the Target Environment.....	28
Upgrade script configuration keys	29
Script Dependencies	31
The Data Upgrade Framework for the Source Environment	35
Defining transformation and preprocessing scripts	35
i) Create transformation.....	36
ii) Define source table	36
iii) Define source fields	36
i) Per Company To Global Table.....	37
ii) Schedule preprocess scripts	37
iii) Save transformation	38
iv) Define dependencies between transformations	38
v) Supporting Pause/Resume Functionality and Committing Data in Batches.....	39
i) Logging Data Issues in Validation Scripts.....	44
j) Modify a subset of records in a table during upgrade	45
Writing Data Upgrade Scripts for Source Environment	47

Writing Live preprocessing scripts	47
Writing Delta and single user mode preprocessing scripts	48
Best Practices for Writing Data Upgrade Scripts.....	50
Best Practice Checks.....	50
In Microsoft Dynamics AX 2010, there are now several best practice checks that help to verify the completeness of upgrade script attributes.	50
Transaction and Idempotency	51
Coding Best Practices	53
Indicating Progress	53
Documenting Scripts	53
Deleting a Table or Field from the Data Model	53
Unique Indexes	54
Consideration for date effective table in upgrade script.....	55
Preventing Copying of Table Data.....	55
Specifying the Shadow/Source Table JOIN Type	56
Upgrading a Table with name and fieldid or tableid changed	56
Upgrading a Table with Table ID or Field ID Changed	56
Deleting Configuration Keys.....	57
Referencing Number Sequences within upgrade scripts	57
Performance Guidelines.....	58
Performance Improvement Options.....	59
Using the Set-based Operators Delete_From, Update_RecordSet and Insert_SecordSet	59
Calling skipDataMethods and skipDatabaseLog Before Calling Update_RecordSet or Delete_From	59
Using RecordInsertList Class to Batch Multiple Inserts.....	60
Optimizing X++ logic	60
Appendix 1: Guidelines for Writing Direct SQL in Upgrade Scripts	62
Using Set-Based Updates in X++	62
Executing Direct SQL from X++	63
How to Execute Direct SQL for X++	63
Best Practices Warning when Executing Direct SQL.....	63
Using Utility Functions to Execute Direct SQL	64
Documenting Direct SQL	64
Using Table Names in Direct SQL	64
Adding Literals in Direct SQL.....	64
Specifying DataArealid in Where-Clauses	65
Determining Whether a Table or Field Exists in the Database	66
Defining String Lengths	67
How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012	

Applying LTrim for String Comparisons in the WHERE Clause	67
Implementing Complex Inserts and Updates in Direct SQL	68
Creating Stored Procedures and Functions	68
Implementing Set-Based Updates with Joins.....	69
Using Direct SQL for Set-Based Updates.....	70
Using a Set-Based Insert Operation.....	71
System Sequence Considerations	72
RECID in Microsoft Dynamics AX 2012	72
Assigning RECID on INSERT	73
Looking Up Table ID and Field IDs	74
Assigning Business Sequences on Insert.....	74
Calling FN_FMT_NUMBERSEQUENCE	77
Appendix 2: INTERNAL ONLY.....	78
Adding Upgrade Scripts to the Upgrade Preprocessing Framework in Source Depot.....	78
Appendix 3: Debugging batch jobs.	79

Introduction

This document describes how to use the Microsoft Dynamics™ AX Data Upgrade Framework and how to write data upgrade scripts for customer data upgrade data models (Microsoft Dynamics AX tables). The data upgrade framework can be used to perform data correction or data transformation.

The intended audience for this document is Microsoft Dynamics AX application developers.

This document is based on *Leveraging the Microsoft Dynamics AX 2012 Data Upgrade Framework*, a Microsoft Dynamics AX 2012 Technical Information document, and on the Microsoft Dynamics AX 2012 Data Upgrade Framework. It has been updated regarding the new data upgrade framework and best practices for performance.

Microsoft Dynamics AX 2012 supports upgrading data in the following ways:

- Microsoft Dynamics AX 4.0 to Microsoft Dynamics AX 2012
- Microsoft Dynamics AX 2009 to Microsoft Dynamics AX 2012

When is a Data Upgrade Script Needed?

The following list describes changes that require an upgrade script:

- 1) Change the name of a field, when field ID is different
- 2) Change the name of a table, when table ID is different
- 3) Delete a table and save data
- 4) Delete a field and save data
- 5) Add or change unique indexes
- 6) Change a non-unique index into a unique index
- 7) Restructure where data is stored. For example, move data from one field to another
- 8) Correct old data inconsistencies
- 9) Populate new tables with existing data
- 10) Populate new fields with existing data or a default value that is different from the default value for the data type

There are changes that can be made in the data model without the need for an upgrade script. The following list describes changes that can be made without an upgrade script:

- 1) Change the name of a field, when field ID is the same
- 2) Change the name of a table, when table ID is the same
- 3) Add a field to a table with a default value for every field
- 4) Add or change relations
- 5) Add or change non-unique indexes
- 6) Add or change delete actions
- 7) Add or change/delete temporary table

1. Terms and Abbreviations

The following table provides terms and definitions that relate to the data upgrade process.

Term/Abbreviation	Definition
Source environment or source system	<p>The Microsoft Dynamics AX system which is being upgraded.</p> <p>The supported source systems for direct upgrade to Microsoft Dynamics AX 2012 are:</p> <ul style="list-style-type: none">• Microsoft Dynamics AX 4.0• Microsoft Dynamics AX 2009 <p>The upgrade starts while the source environment is still live, but there is minimal impact to the live system.</p>

Term/Abbreviation	Definition
Transformation	Data is transformed between source and target environment by using field mapping and joins between necessary tables. Examples include the Address table normalization and the Dimension table normalization.
Source affected table	Tables and fields within source environment that have to be updated with transformations. Examples are Dimension fields and Address fields. No update or changes are made to the source table directly, changes are made to shadow tables.
Source non-affected table	Tables that have to be copied to the target system as they are with the exception of mapping transformation.
Shadow table	Auxiliary table created on the source environment. The shadow tables will contain all fields from the source tables which have to be updated.
Dictionary table	New target Microsoft Dynamics AX 2012 tables. These tables will be imported into the source environment for the application pre-upgrade checklist, and upgrade script execution where needed. The definition of the table must match the target system, the dictionary tables will be copied to the target as they are.
Target affected table	Target tables affected by transformations. The table columns will be partially copied from the source tables, and partially from the shadow tables.
Target non-affected table	Tables copied unchanged from the source database. These tables already have the Microsoft Dynamics AX 2012 schema. Mapping between Microsoft Dynamics AX 4.0, Microsoft Dynamics AX 2009, and Microsoft Dynamics AX 2012 schemas must be created on copy (similar to SYNC).
Preprocess upgrade script	Preprocess upgrade script executed by the upgrade framework for a particular table in the source environment based on the template provided by an application team.

Preprocess upgrade script template	<p>Template of a preprocess script created by an application team and registered with the upgrade framework API. Template provides the following for the upgrade framework:</p> <ul style="list-style-type: none"> • Table and fields that will be involved in the application pre-upgrade checklist. • Business logic for the source data that occurs for the application pre-upgrade checklist.
Application pre-upgrade checklist tasks	<p>Application pre-upgrade checklist tasks that require user intervention to clean-up before the upgrade, for example Address normalization.</p> <p>Many of the addresses are duplicate. As a result, the application pre-upgrade checklist form will provide users with the ability to decide how the master address entity should look.</p>
Delta upgrade script or delta job	Upgrade scripts implemented by application teams to find the changes within table records since the last run of the related preprocessing upgrade scripts.
Single-user mode	No active user transaction is running on the source system. Logic will check if only one administrator user is connected to the source Microsoft Dynamics AX system.
Exception tables	System tables that are specific to the installation that will be excluded from the copy operation. For example, License tables.
Validation script	Special upgrade scripts defined to identify issues with upgrade or data upgrade. These kinds of upgrade scripts should read data from live Microsoft Dynamics AX tables and display messages to take corrective action as needed.
Upgrade throttling	Method used to pause scripts, resume scripts, or change the amount of resources assigned to execute a script.
Throttling APIs	APIs provided by the upgrade framework that can be used by upgrade scripts to support pause and resume functionality.

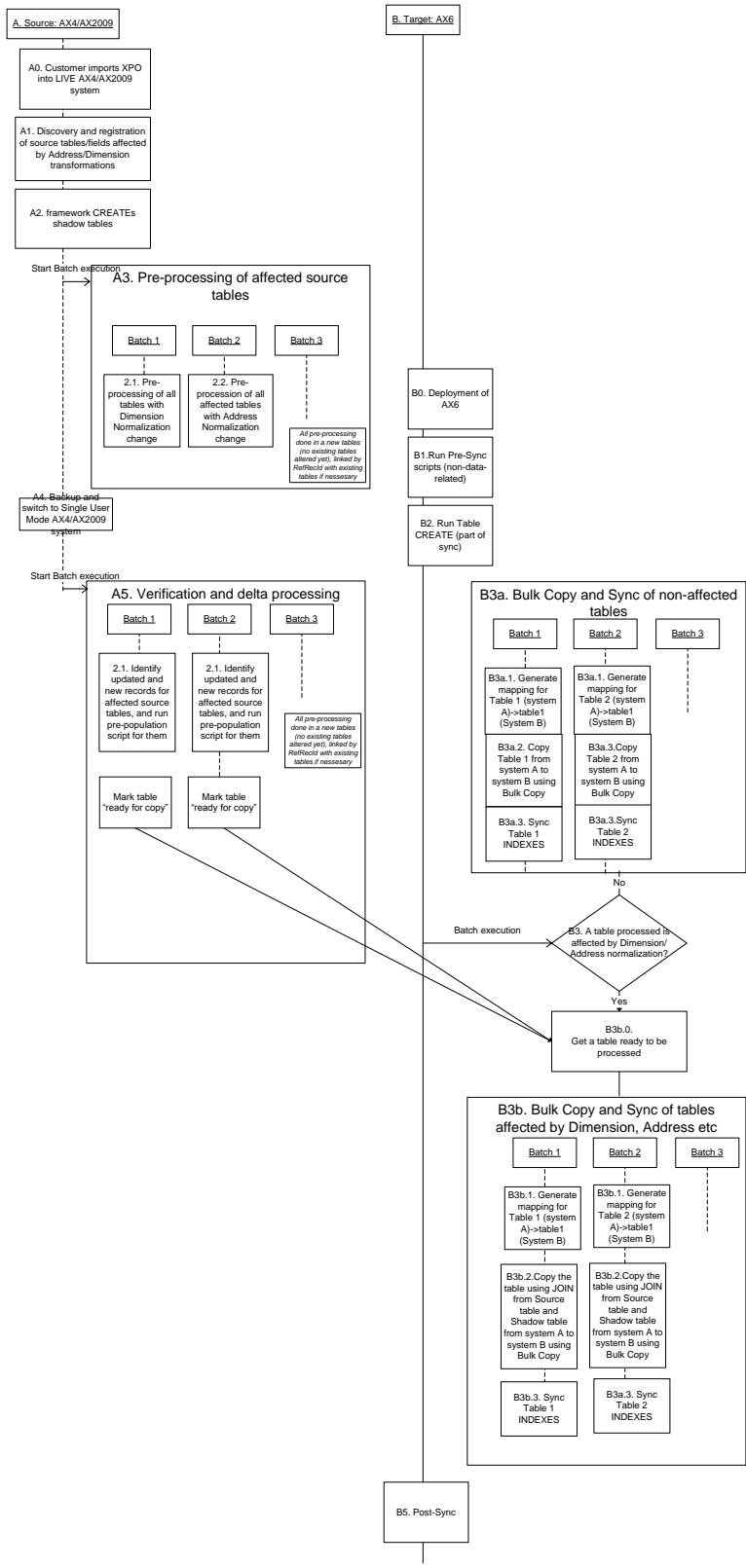
How to Upgrade Data for a Major Release or Service Pack

The data upgrade framework drives the data upgrade scripts that transform an older version of the Microsoft Dynamics AX database to the new version. These steps are described in later sections.

The upgrade process consists of two parts:

- Preprocess in the source environment
- Upgrade in the target environment

The following diagram illustrates the data upgrade process.



The Preprocess Upgrade Checklist (source environment)

The preprocess upgrade checklist is a navigation pane that guides you through the preprocess data upgrade steps in the source environment. Use the following steps to access the preprocess upgrade checklist.

- 1) Import the preprocess XPO into the source environment. The XPOs are located in the retail\CD\DatabaseUpgrade directory on the setup CDROM.
- 2) Open the checklist manually. Navigate to the SysCheckList_PreUpgrade40/50 menu item.

Data upgrade is performed using the preprocess upgrade checklist in the following order:

- 1) Prepare for upgrade.
- 2) Prepare application data for preprocessing
- 3) Preprocess data on live system
- 4) Validate pre-upgrade
- 5) Finalize pre-upgrade

The following diagram illustrates the preprocess upgrade checklist.

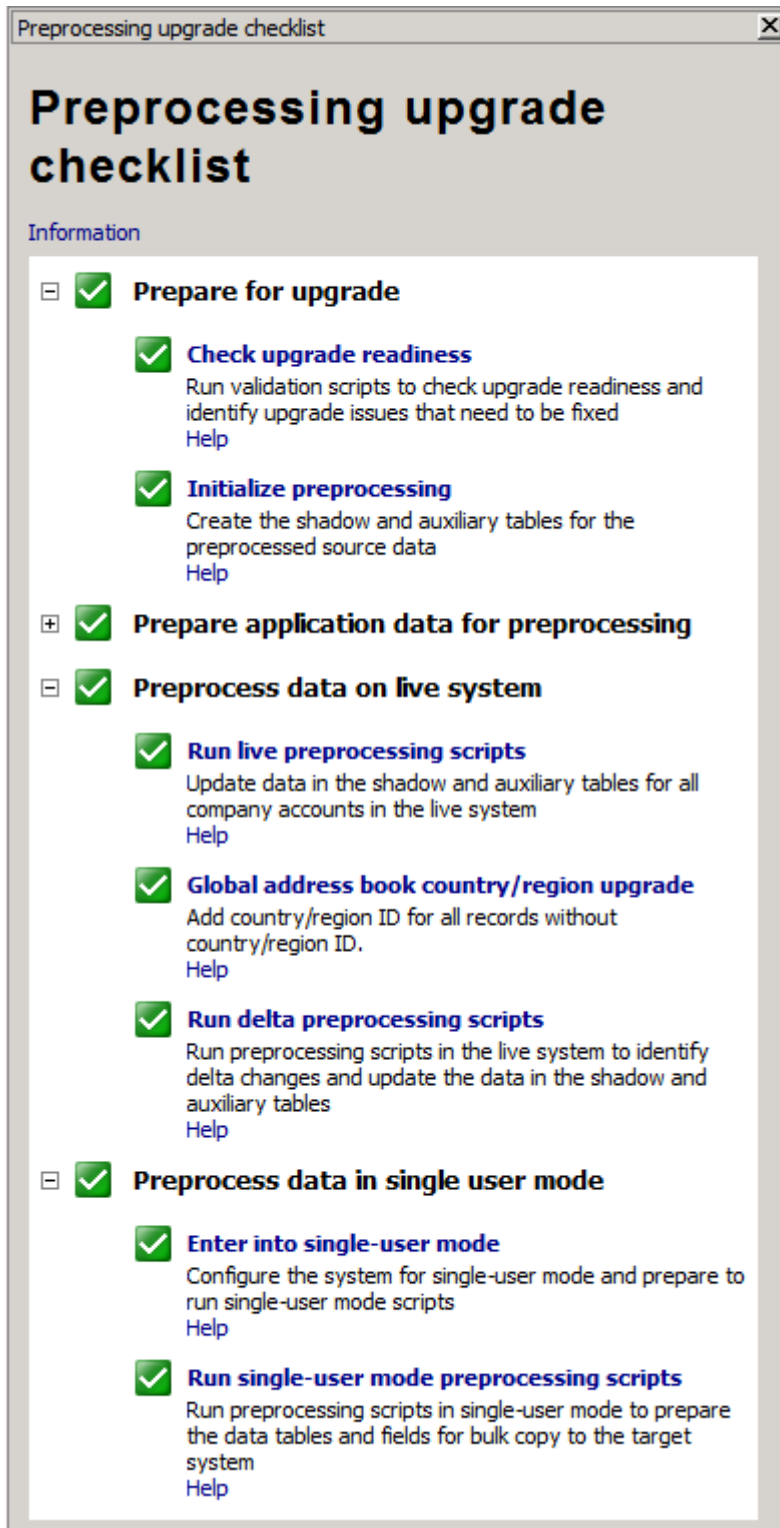


Figure 1. The preprocessing upgrade checklist

The upgrade checklist is a navigation pane that guides you through the data upgrade steps in the target environment. It is invoked automatically when Microsoft Dynamics AX starts after a service pack or major release is installed. Data upgrade is performed using the Upgrade Checklist in the following order:

- 1) Presynchronize
- 2) Postsynchronize
- 3) Upgrade additional features

The following diagram illustrates the upgrade checklist.

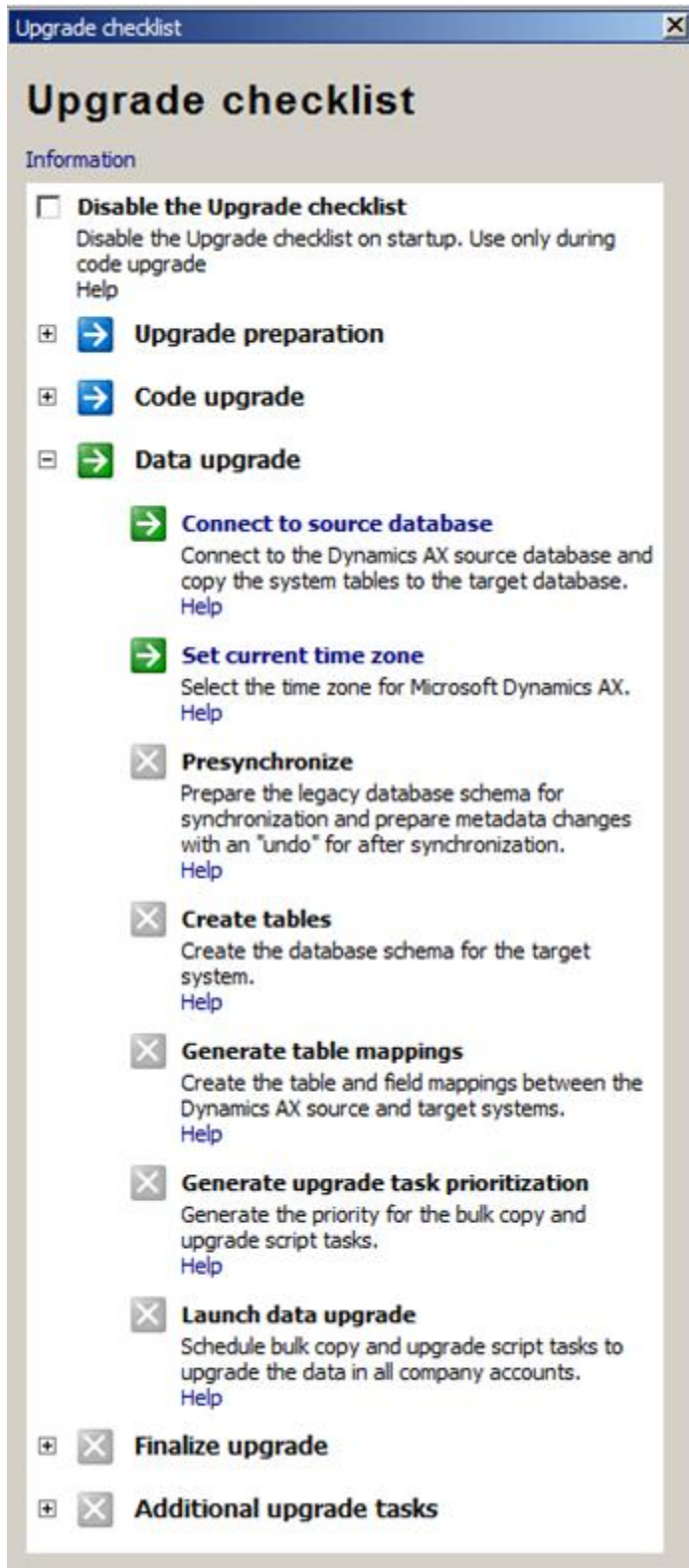


Figure 2. The upgrade checklist

Add Items to the Upgrade Checklist

This section describes how to add new items to the upgrade checklist. You will create a class that extends the `SysCheckListItem` class. Use the following steps to add an item to the upgrade checklist.

1. Create a new class that extends the **SysCheckListItem** class and implements the **SysCheckListItemUpgrade** class. Name the class with the **SysCheckListItem_<name>** format where <name> is a unique name.
2. Override the **getCheckListGroup** method and return the name of the group that the checklist item should appear in.

For example, Upgrade preparation. You may use an existing group name or create a new group.

Nested groups can be created by using the `GroupName\SubGroupName` format.

Note: All checklist items must belong to a group.

3. Create a new action menu item and name the menu action item the same as the class created in Step 1. Set the following properties:
 - a) **Label**: The text that should appear in the checklist for the item
 - b) **HelpText**: The help text that should appear below the checklist item text
 - c) **ObjectType**: Class
 - d) **Object**: Select the name of the class created in Step 1
 - e) **SecurityKey**: AdminSetup
4. In the class, override the **getHelpLink** method and provide a link to a .chm topic.

The following code illustrates an example of providing a link to a .chm topic:

```
#define.TopicId('AxShared.chm:~/html/7b533e0b-f64d-410e-99ae-0296ace-50900.htm')
return SysCheckListItem::sharedGuide(#TopicId);
```

5. Override the **getMenuItemName** method and return the name of the action menu item created in Step 3.
6. Override the **getMenuItemType** method and return the type of the action menu item created in Step 3.

The following code illustrates an example of returning an action menu item:

```
return MenuItemType::Action;
```

7. If necessary, override the **isRunnable** method. Determine what conditions the checklist item should appear in the checklist. Return **True** to show the checklist item, or return **False** to hide it.
8. Override the **new** method. Specify where your item should appear in the checklist and what other checklist items your checklist item depends on.
 - a) Call this **placeAfter** to specify the order of your checklist item.

The following code example illustrates the checklist item placed after the **Detect code upgrade conflicts** checklist item.

```
this.placeAfter(classnum(SysCheckListItem_SysUpgradeDetectCon));
```

- b) Call this **addDependency** to specify which checklist items your checklist item depends on.

The following code example illustrates the checklist item depends on the **Set current time zone**

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

checklist item. The **Set current time zone** checklist item must be completed before this checklist item is enabled.

```
this.addDependency(classnum(SysCheckListItem_BaseTimezoneUpgrade));
```

9. Override the **main** method. This is the method that is executed when the checklist item is clicked in the checklist.
10. In class SysCheckList, add an entry corresponding to your checklist item to the list in method **checkListItems**. The list contains all the possible checklist items. Add your entry in the correct order in which it should appear in the list.

The Data Upgrade Framework for the Target Environment

The data upgrade framework gives developers the infrastructure to insert data upgrade scripts written in X++. The data upgrade framework manages the dependencies of the scripts, schedules them to be run in parallel by batch clients, and provides progress reports on the running scripts. The data upgrade framework has a built-in error recovery mechanism that helps to ensure system integrity when the upgrade has to be resumed after an error.

With the exception of the base `ReleaseUpdateDB` class, the `ReleaseUpdateDB*` classes contain implementations of data upgrade scripts. The scripts provide abstract methods and utility functions for data upgrade classes.

Note that classes with preprocessing upgrade scripts are derived from the class `ReleaseUpdateTransformDB`, and have different types of scripts and ways of scheduling them.

The class diagram of the upgrade script classes is shown in Figure 2.

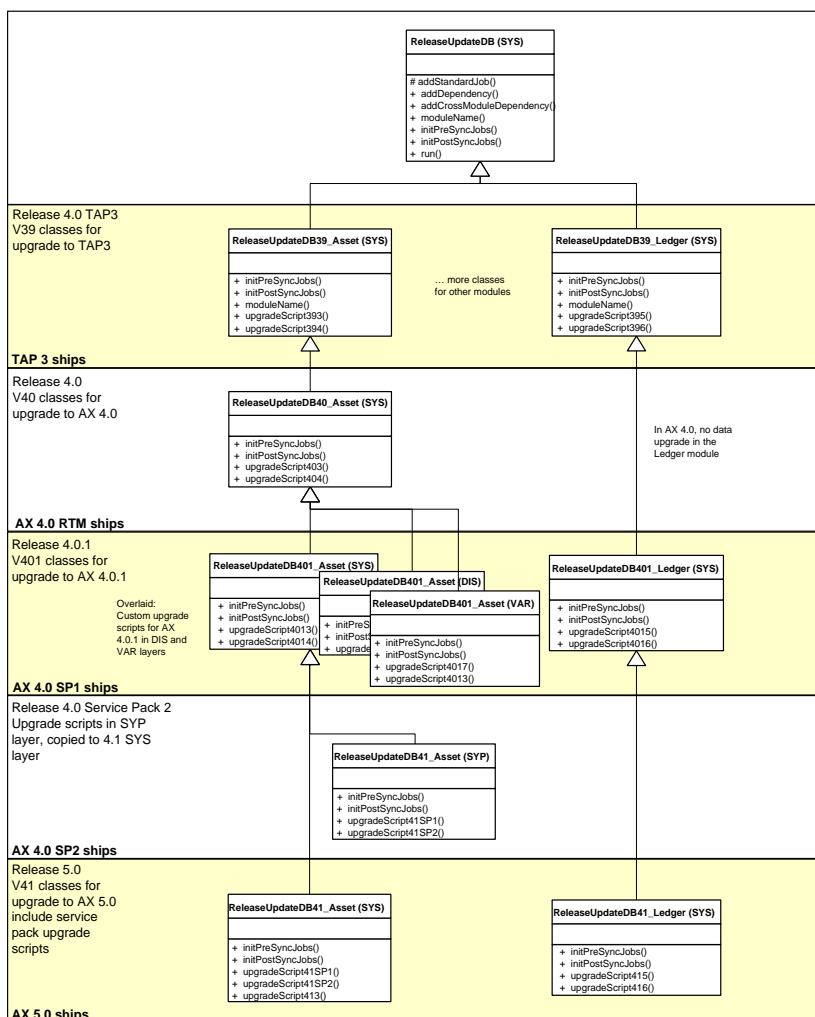


Figure 2. Data Upgrade Script Classes

Data Upgrade Scripts by Module

Data upgrade scripts are inserted into the data upgrade as methods of a ReleaseUpdateDB<NN>_<module> class, where <NN> is the version of Microsoft Dynamics AX being upgraded to, and <module> is the module name the script belongs to. These classes are derived from the base class ReleaseUpdateDB and are connected to the data upgrade framework.

When you create upgrade scripts for your version of Microsoft Dynamics AX, you can use any of the new classes in the following table according to your script's application module and the version you are developing.

401	41	60
ReleaseUpdateDB401_Administration	ReleaseUpdateDB41_Administration	ReleaseUpdateDB60_Administration
ReleaseUpdateDB401_Bank	ReleaseUpdateDB41_Asset	ReleaseUpdateDB60_Asset
ReleaseUpdateDB401_COS	ReleaseUpdateDB41_Bank	ReleaseUpdateDB60_Bank
ReleaseUpdateDB401_Cust	ReleaseUpdateDB41_Basic	ReleaseUpdateDB60_Basic
ReleaseUpdateDB401_Ledger	ReleaseUpdateDB41_COS	ReleaseUpdateDB60_Cat
ReleaseUpdateDB401_Proj	ReleaseUpdateDB41_Cust	ReleaseUpdateDB60_Client
ReleaseUpdateDB401_Vend	ReleaseUpdateDB41_HRM	ReleaseUpdateDB60_COS
	ReleaseUpdateDB41_Invent	ReleaseUpdateDB60_Cust
	ReleaseUpdateDB41_Jmg	ReleaseUpdateDB60_EcoRes
	ReleaseUpdateDB41_KM	ReleaseUpdateDB60_EMS
	ReleaseUpdateDB41_Ledger	ReleaseUpdateDB60_HRM
	ReleaseUpdateDB41_Prod	ReleaseUpdateDB60_Invent
	ReleaseUpdateDB41_Proj	ReleaseUpdateDB60_Jmg
	ReleaseUpdateDB41_Req	ReleaseUpdateDB60_KM
	ReleaseUpdateDB41_SMA	ReleaseUpdateDB60_Lean
	ReleaseUpdateDB41_smm	ReleaseUpdateDB60_Ledger
	ReleaseUpdateDB41_Trv	ReleaseUpdateDB60_PBA
	ReleaseUpdateDB41_Vend	ReleaseUpdateDB60_Prod
		ReleaseUpdateDB60_Proj
		ReleaseUpdateDB60_PurchReq
		ReleaseUpdateDB60_Req
		ReleaseUpdateDB60_SMA
		ReleaseUpdateDB60_smm
		ReleaseUpdateDB60_Sourcing
		ReleaseUpdateDB60_Trv
		ReleaseUpdateDB60_Vend

Name ReleaseUpdateDB60 means upgrade to Microsoft Dynamics AX 2012. Pre-synchronization, Post-synchronization and Additional features upgrade methods coexist in these classes.

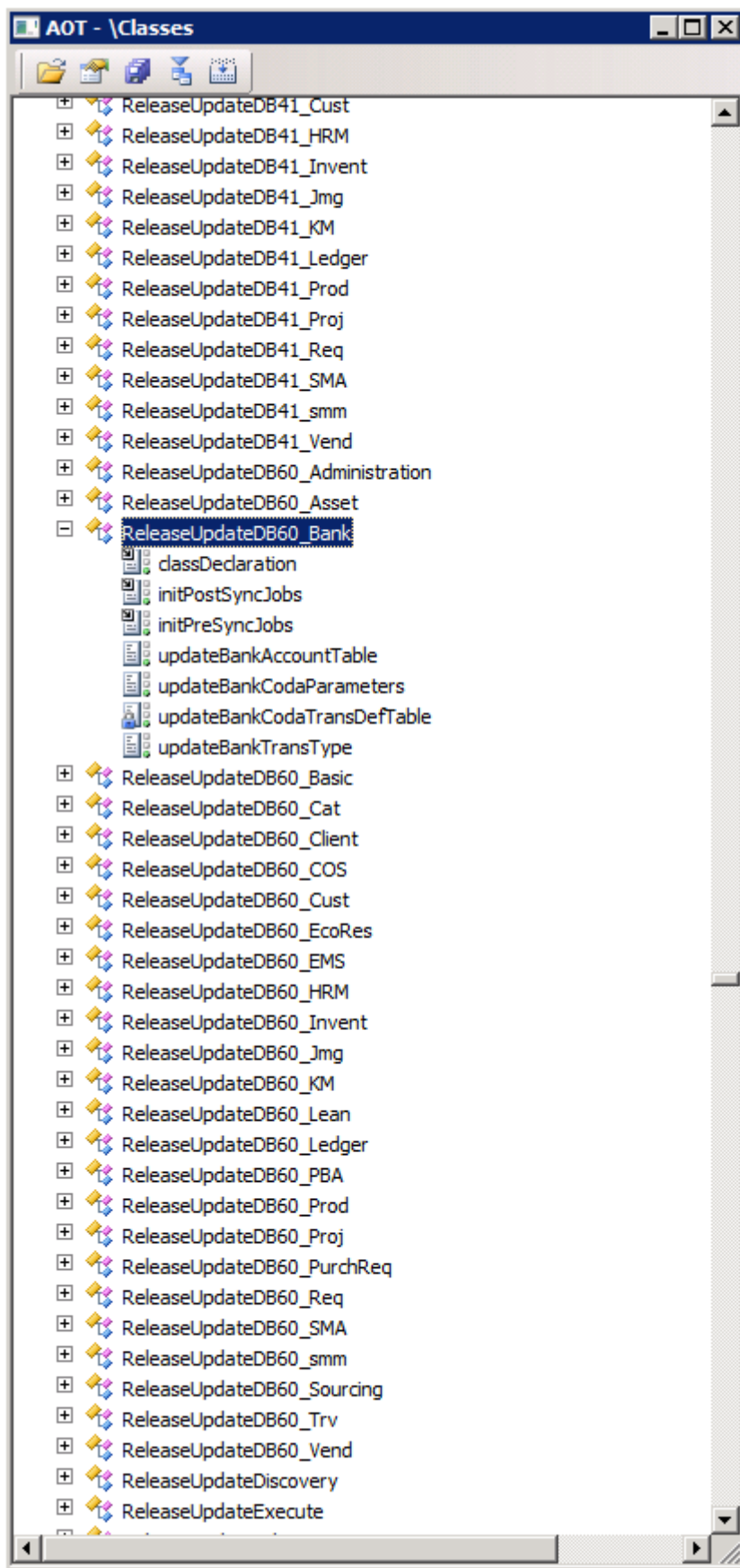
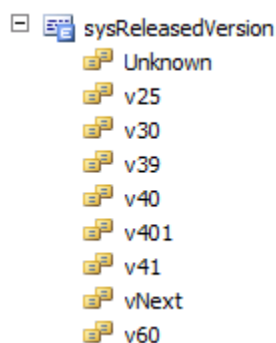


Figure 3. Upgrade Classes in the Applications Object Tree

SYS Versions and Data Upgrade of Interim SYS releases

The SYS layer contains the core functionality of Microsoft Dynamics AX. A modification to this layer is shipped to partners and customers in beta versions (for example, Microsoft Dynamics AX 4.0 TAP3), final release version (for example, Microsoft Dynamics AX 4.0), and refresh versions of major releases (for example, Microsoft Dynamics AX 4.0.1), referred to here as interim SYS releases. The data upgrade framework supports upgrades that span multiple SYS releases by providing the infrastructure to incrementally upgrade from one SYS release to another, later release.



SYS versions are defined in the Base Enum SysReleaseVersion

Each ReleaseUpdateDB* class (except for the base ReleaseUpdateDB class and preprocessing upgrade classes) is associated with a SYS version and named accordingly. The class hosts the data upgrade scripts that upgrade the SYS data model from the previous SYS version to the current SYS version.

Upgrade scripts can span more than one SYS release. Therefore, each data upgrade script class inherits upgrade scripts from the class of the same module in the most recent previous release. When you need upgrade scripts for a new interim release, and when the upgrade script class for the corresponding module does not yet exist, you create the class that uses the right naming convention and ensure this class inherits upgrade scripts from the previous version of the upgrade script class of the same module.

For example, in Figure 2, the Ledger module has upgrade scripts for version 4.0 TAP3 (39) and 4.0.1 (401), but does not have an upgrade script for release version 4.0 (40). Therefore, the class ReleaseUpdateDB401_Ledger inherits directly from ReleaseUpdateDB39_Ledger. While for the Asset module, there are upgrade scripts for versions 39, 40, 401, 41 (Microsoft AX 2009). Therefore the class ReleaseUpdateDB401_Asset must inherit from ReleaseUpdateDB40_Asset, which in turn inherits from ReleaseUpdateDB39_Asset.

```
Public class ReleaseUpdateDB401_Ledger extends ReleaseUpdateDB39_Ledger
{
}
```

```
Public class ReleaseUpdateDB40_Asset extends ReleaseUpdateDB39_Asset
{
}
```

```
Public class ReleaseUpdateDB401_Asset extends ReleaseUpdateDB40_Asset
{
}
```

```
Public class ReleaseUpdateDB41_Asset extends ReleaseUpdateDB401_Asset
{
}
```

In order to incrementally upgrade from a SYS release that is two or more versions earlier, the `initPreSyncJobs`, `initPostSyncJobs` and `initAdditionalJobs` methods must be overridden and you must call “`#initSyncJobsPrefix`” to include the previous upgrade. The `initPreSyncJobs`, `initPostSyncJobs` and `initAdditionalJobs` jobs detect the earlier (“from”) version of the upgrade and skips if necessary.

```
void initPostSyncJobs ()
{
    #initSyncJobsPrefix
}
```

Finally, the purpose of an individual script is to upgrade a table's data from `SysVer -1` to `SysVer`. Each script is used to upgrade the data to the current version.

Data Upgrade for Service Packs

Service packs are shipped in the SYP layer of each major Microsoft Dynamics AX release. Service pack fixes are rolled forward into the next version of the SYS release. Therefore, they can be viewed as a pre-release of the next major release, and, to perform a data upgrade, the upgrade scripts are added to the upgrade script classes of the next major release in the SYP layer.

For example, in Figure 2, a data upgrade script in the Fixed Asset module for the Service Pack for Microsoft Dynamics AX 4.0.1 is implemented in the `ReleaseUpdateDB41_Asset` class in the SYP layer. This script will be merged with the data upgrade scripts for SYS release Microsoft Dynamics AX 4.1 into `ReleaseUpdateDB41_Asset` in the SYS layer. The data upgrade framework handles service pack releases by detecting at individual script level what has been run already in a service pack of the previous SYS release and skips the upgrade script.

```
void initPostSyncJobs ()
{
    #initSyncJobsPrefix
}
```

Data Upgrade for Customization

Customizations are performed in layers higher than the SYS (and SYP) layers. If the customization requires a data upgrade, the same layer would be used to update the data upgrade scripts.

Customization of a data upgrade is performed by overlaying the SYS level data upgrade scripts classes in the same layer as the customization. This can be achieved by either overriding a SYS layer upgrade script or by adding a new upgrade script. This is illustrated in Figure 2. There are two overlaid `ReleaseUpdateDB401_Asset` classes in the DIS and VAR layer.

Note: Service pack releases and customizations (including local features, option pack providers, and partner customizations) have different purposes. Therefore, implementation of the data upgrade scripts for a Service Pack and for customization data upgrade will be different.

When customizing a data upgrade by overlaying data upgrade script classes, the `initPreSyncJobs`, `initPostSyncJobs` and `initAdditionalJobs` methods must be overlaid and the jobs from lower layers must be included in the current layer. For example, the `ReleaseUpdateDB401_Asset::InitPreSyncJob` in the VAR layer in Figure 2 should resemble the following sample:

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

```

void initPostSyncJobs()
{
    #initSyncJobsPrefix

    // Add SYS upgrade scripts, including overlaid upgrade scripts
    ...
    // Add new DIS upgrade scripts, not DIS overlaid upgrade scripts
    ...
    // Add new VAR upgrade scripts, not VAR overlaid upgrade scripts
}

```

Create a single upgrade script that combines changes across multiple product versions

When upgrading to version n (target) from version $n-2$ (source), you can sometimes provide an algorithm that upgrades data directly from the source to the target version without upgrading to the interim version. We call these algorithms combined upgrade scripts. In cases for which you can create a combined upgrade script, follow the best practices below:

- 1) Place the algorithm in the upgrade class for the source version, replacing the original algorithm. For example, if you are upgrading from version 3.0 to 4.0 SP1, put the combined algorithm in the `ReleaseUpdateDB39` class.
- 2) Put a condition in a script in the upgrade class for the target version, setting it to execute only if you are not upgrading from the source version. For example, change the script in the 4.0 SP1 version to

```

3) public void updateCustTrans()
4) {
5)     if (ReleaseUpdateDB::getFromVersion() != sysReleasedVersion::v30)
6)     {
7)         Original script logic for upgrade from 4.0 to 4.0 SP1
8)     }
9) }

```

Using Configuration Key to Remove Obsolete Objects after Upgrade

Note that after the upgrade is finished, you can disable the configuration keys “Keep update objects” ([SysDeletedObjects40](#), [SysDeletedObjects41](#) and [SysDeletedObjects60](#) for Microsoft Dynamics AX 2012). After database synchronization is complete, all obsolete components of the data model will be removed and performance will be improved. The components that are removed are those needed to perform the data upgrade, but provide no value when the process is completed.

Data Upgrade Scripts in the target environment

Data upgrade scripts comprise the majority of the data upgrade framework. For each version, a set of classes exists - one upgrade class per module. Currently, there are 26 application modules for upgrade scripts. They are named `ReleaseUpdateDB<version>_<module>`, for example `ReleaseUpdateDB60_Bank`.

Each of these classes contains scripts for pre-synchronization, post-synchronization and additional upgrades.

The scripts are scheduled by their attributes specified at the beginning of every script method.

The methods `initPreSyncJobs`, `initPostSyncJobs` and `initAdditionalJobs` are still kept for version checking.

Each class can handle your upgrade script in one of four different ways - Start, Shared, Standard, and Final. Note that it is important to choose the right one so that the script runs at the correct time and in the correct manner:

Pre-synchronization	Post-synchronization	Additional upgrade
Start (allow duplicates)	-	
Shared/Normal	Shared/Normal	Shared/normal
-	Final (undo allow duplicates)	

1) Presynchronize Start scripts

(Executed first)

Start scripts are used to change indexes that have become unique in order to allow duplicates. This is a modification of meta data and must be undone in a post-synchronization final script (see below). Start scripts are run once versus once per company as with normal scripts.

2) Presynchronize Shared scripts

(Executed once in parallel with pre-synchronization normal scripts)

Shared scripts are used mainly for cleanup jobs such as deleting duplicate records for tables that have changed an index from allowing duplicates to being unique. Shared scripts are run at the same time as normal scripts. The only way to ensure that a shared script is run before another shared script or a normal script is to set up a dependency between the scripts. To perform this operation, see [Writing Data Upgrade Scripts](#) below. Shared scripts are run only once, as compared to normal scripts, which are run once per company

3) Presynchronize Standard scripts

(Executed for each company account in parallel with pre-synchronization shared scripts)

Normal scripts are run once per company and are used for company-specific clean up jobs, rebuilding indexes, or deleting company-specific data that will be regenerated later.

4) Presynchronize Final scripts

Used very rarely. Pre-synchronization start, shared and normal scripts manage dependencies better.

5) Postsynchronize Start scripts

Used very rarely. Post-synchronization shared, normal and final scripts manage dependencies better.

6) Postsynchronize Shared scripts

(Executed once in parallel with post-synchronization normal scripts)

Shared scripts are run once and used to update non company-specific tables.

7) Postsynchronize Standard scripts

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

(Executed for each company account in parallel with post-synchronization shared scripts.)

Standard scripts are run once per company and are used to update company specific tables. (~90% of all scripts are of this type)

8) Postsynchronize Final scripts

(Executed last)

Final scripts are used to undo changes to indexes that were made to allow duplicates using the pre-synchronization start script. Final scripts are run only once, as compared to normal scripts, which are run once per company.

9) Upgrade additional features scripts

Upgrade additional features scripts are used to upgrade of the non-core functionality after the functional data upgrade

Writing Data Upgrade Scripts for Target Environment

To create a script you need to create a method on the appropriate class. For example, for Microsoft Dynamics AX 2012 the class is ReleaseUpdateDB60_<module>. You must also inform the framework how to handle the script. This is done by providing attributes for their upgrade script methods like following:

```
[UpgradeJobTypeAttribute(Standard),
UpgradeJobTitleAttribute ("@SYS97795"),
UpgradeJobConfiguratuionKeyAttribute (configurationKeyStr(ProjBasic)),
UpgradeJobConfiguratuionKeyAttribute (configurationKeyStr(HRMBasic)),
UpgradeJobDependsOnJobAttribute (ReleaseUpdateDB60_Proj, updateProjOnAccountPosting)]
void updateProjCategory()
{
    ProjCategory projCategory;
    ttsbegin;
    update_recordset projCategory
        setting Active = NoYes::Yes;
    ttscommit;
}
```

Here is the list of available attributes:

- Shared/ Standard / Start/ Final
- PreSync/PostSync/Additional
- Configuration keys associated with the script
- Script description(label)
- Does the script requires its own transaction
- (optional) Dependencies on another scripts or tables
- Table names
- Type of access to each table : Create, Read, Update, Delete

UpgradeScriptDescriptionAttribute : one attribute per method allowed

Value: String

UpgradeScriptStageAttribute: one attribute per method allowed

Values: enum values PreSync, PostSync, Additional

UpgradeScriptTypeAttribute: one attribute per method allowed

Values: enum values Standard, Shared, Start, Final

UpgradeScriptTransactionAttribute: one attribute per method

Value: enum Yes/No

UpgradeScriptTableAttribute: several attributes allowed

Value: String (table name),

enum: Yes/No (Create),

enum: Yes/No (Read),

enum: Yes/No (Update),

enum: Yes/No (Delete)

for example,

```
[UpgradeScriptTableAttribute(tableStr(LedgerTable), false, true, true, false)]
```

Also, for legacy compatibility, Microsoft Dynamics still support another way of adding upgrade scripts, by adding line in the `initPreSyncJobs` or `initPostSyncJobs` or `initAdditionalJobs` method on the class. Each of these `ReleaseUpdateDBxx_xxx` classes contains three separate methods you can modify to schedule your jobs – `initPreSyncJobs`, `initPostSyncJobs` and `initAdditionalJobs`. If you would like your job to run in pre-synchronize phase, add it to the `initPreSyncJobs` method, otherwise add it to the `initPostSyncJobs` method or to the `initAdditionalJobs` method for the additional feature upgrade.

Note that this method is not recommended and is mainly preserved for legacy support. However, here are the following are script templates you can use:

```
this.addStartJob(methodStr(<ClassName>, <MethodName>), "description",  
[configurationkeynum(ConfigurationKey1), ..., configurationkeynum(ConfigurationKey1) ] );  
  
this.addSharedJob(methodStr(<ClassName>, <MethodName>), "description",  
[configurationkeynum(ConfigurationKey1), ..., configurationkeynum(ConfigurationKey1) ] );  
  
this.addStandardJob(methodStr(<ClassName>, <MethodName>),  
"description", [configurationkeynum(ConfigurationKey1), ..., configurationkeynum(ConfigurationKey1) ] );  
  
this.addFinalJob(methodStr(<ClassName>, <MethodName>),  
"description", [configurationkeynum(ConfigurationKey1), ..., configurationkeynum(ConfigurationKey1) ] );
```

Writing Data Upgrade Scripts for ISV solution in the Target Environment

ISV has two options to write upgrade scripts:

Option1: Use the upgrade framework to write upgrade scripts in the same way as the core upgrade scripts have been created in the SYS layer. Choosing this option, ISV solution should be installed before running the major version upgrade, all ISV scripts will be loaded and scheduled along with the core upgrade scripts during the major upgrade.

Option2: Use the upgrade framework to write upgrade scripts for a minor upgrade after the major version upgrade has been completed. Choosing this option, ISV scripts can be implemented in the same way as option 1 with some differences:

- preSync script must use the enum `ReleaseUpdateScriptStage::PreSyncUpdate`
- postSync script must use the enum `ReleaseUpdateScriptStage::PostSyncUpdate`
- Additional script must use the enum `ReleaseUpdateScriptStage::AdditionalUpdate`
- Script class should implement `initPreSyncUpdateJobs()`, `initPostSyncUpdateJobs()` and `initAdditionalUpdateJobs()`, all of these methods should contain a single call to the local macro, for example:

```
void initPostSyncUpdateJobs()
{
    #initSyncJobsPrefix
}
```

- After installing the scripts, ISV installer should call the `ReleaseUpdateDB::RegisterForMinorUpgradeScript` method in Microsoft Dynamics AX to register the scripts. The following code example registers three scripts. Use `ttsbegin` and `ttscommit` to make sure all or none of the script is registered.

```
ttsbegin;

ReleaseUpdateDB::RegisterForMinorUpgradeScript(
    classStr(ReleaseUpdateDB60_ISV01), methodStr(ReleaseUpdateDB60_ISV01, Script01));

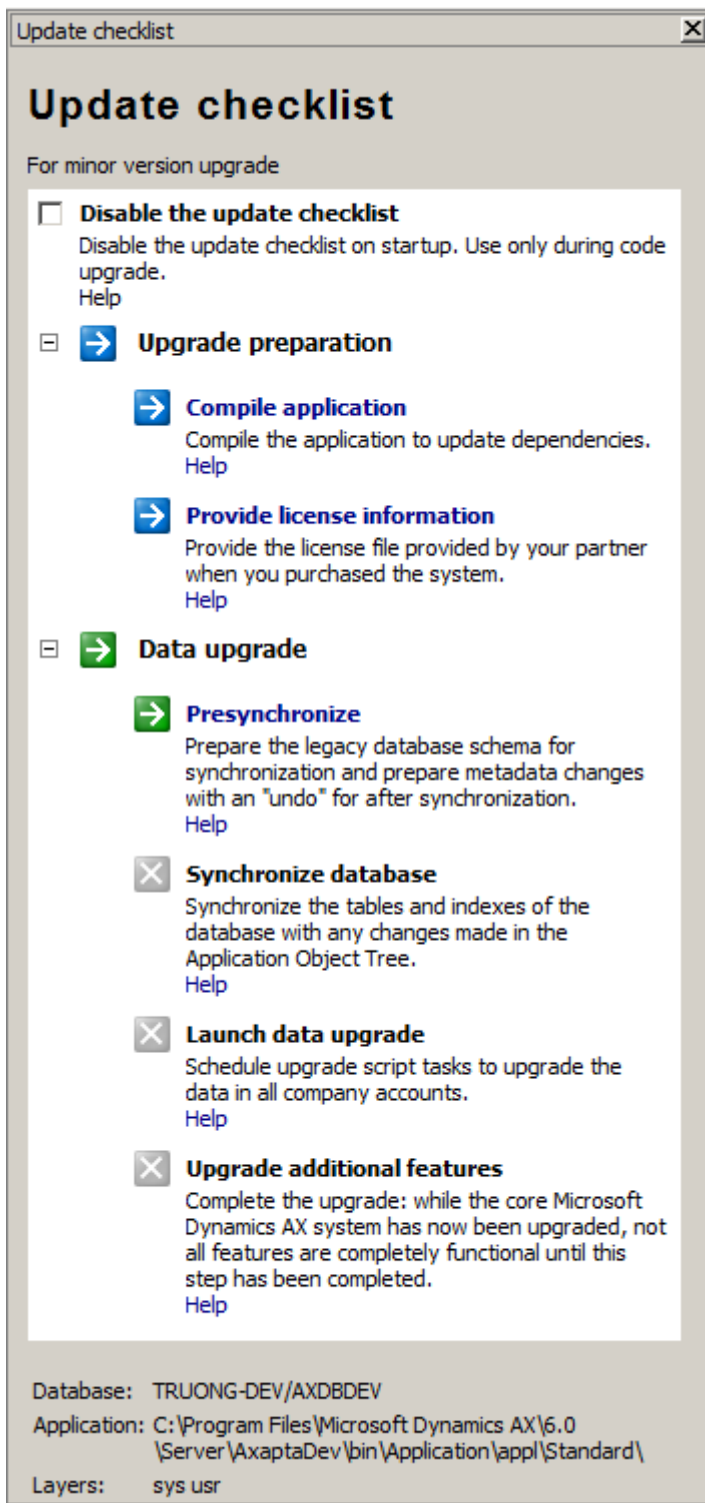
ReleaseUpdateDB::RegisterForMinorUpgradeScript(
    classStr(ReleaseUpdateDB60_ISV01), methodStr(ReleaseUpdateDB60_ISV01, Script02));

ReleaseUpdateDB::RegisterForMinorUpgradeScript(
    classStr(ReleaseUpdateDB60_ISV01), methodStr(ReleaseUpdateDB60_ISV01, Script03));

ttscommit;
```

- After the scripts have been registered, the next time Microsoft Dynamics AX starts, the Update checklist will display to allow users to schedule and run the ISV scripts, the checklist can also be opened manually using the display menuitem `SysCheckList_Update`. The following illustration shows the upgrade checklist.

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012



Upgrade script configuration keys

Developers can provide an optional set of configuration keys associated with an upgrade script - [*configurationkeynum* (<*config key name1*,*config key name2*, ... , *config key name n*>)]. The script will be scheduled to run if at least one configuration key associated with script is enabled during upgrade.

You can do this by specifying attribute for your upgrade script:

UpgradeConfigurationKeyAttribute : several attributes allowed

(attribute will be joined with OR; for complex OR and AND between configuration keys – upgrade script should handle it with if/else condition within upgrade script)

Value: Configuration key

For compatibility purposes, legacy way of specifying configuration key is still supported, but not recommended:

```
this.addFinalJob(methodstr(ReleaseUpdateDB60_Administration, allowDupSysExpImpTableGroupIdx),
"@SYS97945", [configurationkeynum(Asset), configurationkeynum(Bank) ]);
```

Also, you can specify a set of configuration keys on the module level by using the [setModuleConfigKey](#) function. The module configuration key set is joined with each upgrade script configuration key set for that module.

```
this.setModuleConfigKey([configurationkeynum(ConfigurationKey1), ..., configurationkeynum(ConfigurationKey1) ])
```

Note that if you are using `setModuleConfigKey`, it should be called from `InitPreSyncJobs`, `initPostSyncJobs` and `InitAdditionalJobs` method separately.

Script Dependencies

You can also add dependencies between your scripts. This can be useful to avoid locking and for enforcing a logical flow of your scripts. To add a dependency, you can add appropriate attribute before your upgrade script (recommended) or include call to `addDependency` method in the appropriate `InitXXXJobs` method:

- a) If you have a dependency **between the scripts inside a module**, use the **`UpgradeDependsOnTaskAttribute`** or `addDependency` method.

```
this.addDependency(methodStr(<ClassName>, <MethodName>),  
                  methodStr(<ClassName>, <MethodName>));
```

where the first method must be executed before the second method executes.

- b) If the script is **dependant on another module's script**, you can use the **`UpgradeDependsOnModuleAttribute`** or `addCrossModuleDependency` method to ensure a correct execution sequence between scripts placed in the different classes:

```
this.addCrossModuleDependency(classnum(<ClassName>), methodStr(<ClassName>, <MethodName>),  
                              classnum <ClassName>, methodStr(<ClassName>, <MethodName>));
```

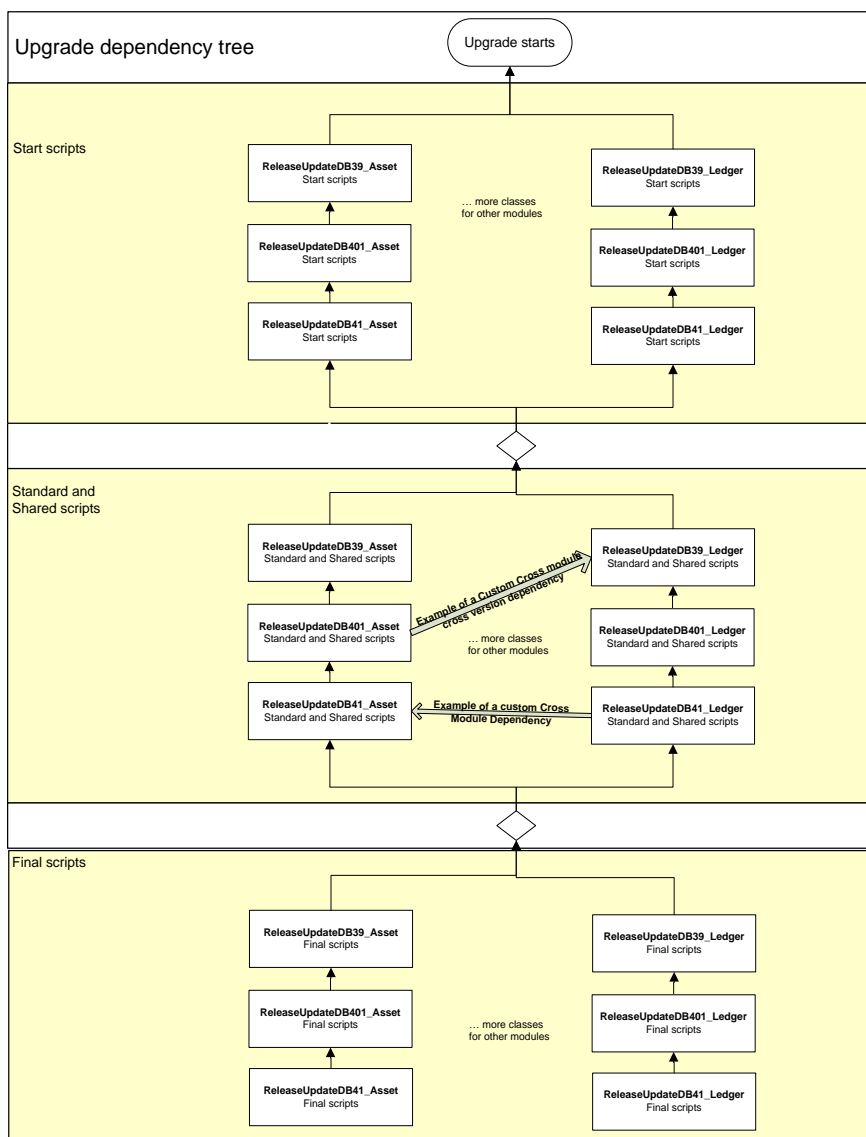
- c) If the script is **dependant on another module's script from a previous version**, you can use the **`UpgradeDependsOnVersionAttribute`** or `addCrossVersionModuleDependency` method to ensure that the correct execution sequence between scripts placed in the different versions and modules:

```
this.addCrossVersionModuleDependency(  
    classnum(<ClassName>),  
    methodStr(<ClassName>, <MethodName>),  
    SysReleaseVersion::<version>,  
    classnum <ClassName>,  
    methodStr(<ClassName>, <MethodName>),  
    SysReleasedVersion::<version>);
```

- d) If there are scripts depending on your script and you want to set the dependency but do not want to change the other scripts, you can use the **`UpgradeTaskDependsOnMeAttribute`** or **`UpgradeModuleDependsOnMeAttribute`**.

4. If a script is **dependent on another script from a previous version but located in the same module**, then you do not need a dependency, the upgrade framework will provide an implicit dependency in this case.

Here is an example of the dependency tree:



Precautions When You Write Data Scripts Before Synchronization

Pre-synchronization data upgrade scripts are executed before the new version of Microsoft Dynamics AX Object Data (AOD) is synchronized to the Microsoft Dynamics AX database and before the data is copied over from the source system to target system. This means that the executed code will use a new version of metadata, but the database will still be empty.

Also, please note that several special tables are synchronized during AOS startup even before the upgrade checklist starts:

```
SysSetupLog
SysSetupCompanyLog
SysRecordTemplateTable
SysRecordTemplateSystemTable
SysTraceTable
SysTraceTableSQL
SysTraceTableSQLExecPlan
```


SysTraceTableSQLTabRef
SysUserLog
SysUserInfo
SysUtilElementsLog
SysCompanyUserInfo
SysInetCSS
SysInetThemeTable
SysImageTable
SysPersonalization
LanguageTable
SysSignatureSetup
SysDataBaseLog
SysExceptionTable
DEL_Batch
DEL_BatchGroup
BatchJob
BatchConstraints
Batch
BatchGroup
SysServerConfig
SysClusterConfig
BatchServerConfig
BatchJobAlerts
BatchServerGroup
BatchJobHistory
BatchHistory
BatchConstraintsHistory
BatchGlobal
EPStateStore
EPStateStoreSettings
EPServerStateCleanupSettings
EPWebSiteParameters
EPGlobalParameters
SysBCProxyUserAccount
SRSServers
AifWebsites
Currency
SysSQMSettings
SysProgress
SysBPPParameters
SysRemoveFields
SysRemoveTables
SysRemoveConfig
SysRemoveLicense
SysLicenseCodeSort
DocuParameters
SysSecurityFormTable
SysSecurityFormControlTable
SysEvent
KMConnectionType
SalesParmUpdate
SalesParmSubTable
PurchParmUpdate
PurchParmSubTable
SysVersionControlParameters
ReleaseUpdateScripts
ReleaseUpdateScriptDependency
ReleaseUpdateJobStatus
ReleaseUpdateScriptsUsedTables
ReleaseUpdateBulkTableInfo
ReleaseUpdateSysDeleted
DocuOpenFile
TimezoneInfo
DEL_SysUpgradeTimeZone

```

DEL_SysUpgradeBaseTimeZone
WorkflowWorkItemDelegationParameters
WorkflowWorkItemCommentTable
SysUpgradeTreeNodeConflictInfo
SysUpgradeParameters
SysUpgradeOverriddenEstimates
ReleaseUpdateBulkCopyParameters
ReleaseUpdateBulkFlags
ReleaseUpdateBulkCopyTable
ReleaseUpdateBulkCopyField
DEL_SysSetupLog
ReleaseUpdateTransformTable
DEL_ReleaseUpdateTransformTable
ReleaseUpdateTransformSourceField
ReleaseUpdateTransformTargetField
ReleaseUpdateBulkCopyTableExceptions
ReleaseUpdateDiscoveryStatus
DEL_ReleaseUpdateDictionaries
DEL_SqlDictionary
DEL_SystemSequences
DEL_SysLastValue
ReleaseUpdateBulkCopyLog
ReleaseUpdateSpecialTableMapping
ReleaseUpdateSpecialFieldMapping
ReleaseUpdateBulkRefRecIdPatch
ReleaseUpdateExtendedDataTypes
ReleaseUpdateDataAreaOffsets
ReleaseUpdateConfiguration
DataArea
DEL_CompanyDomainList
ReleaseUpdateConfigKey
NumberSequenceDatatype
NumberSequenceDatatypeParameterType
SysXppAssembly
SecurityRoleAllTasksView

```

For these special tables, you cannot use pre-synchronization Start scripts. So, if you change field ID on one of these tables, code changes must be made directly in the `\Classes\Application\syncApplTables()` method, for example:

```

if (!this.isRunningMode())
{
    ttsbegin;
    if (isConfigurationkeyEnabled(configurationkeynum(CRSEGermany)))
    {
        ReleaseUpdateDB::changeFieldByName('TaxRepresentative', 41, 0, 75);
    }
    ttscommit;
}
syncTable(tablenum(CompanyInfo));

```

Note that changes in Application class are risky and should be made with caution.

The Data Upgrade Framework for the Source Environment

The data upgrade framework for the source environment gives developers the infrastructure to insert data upgrade scripts written in X++ to be executed in the source environment before the actual upgrade in Microsoft Dynamics AX 2010 starts

Framework and application scripts are shipped as a separate XPO file which customers have to manually import into their live AX4 or AX5 environment before the upgrade process starts.

All source preprocessing scripts must be derived from the `ReleaseUpdateTransformDB` class.

Defining transformation and preprocessing scripts

Before writing upgrade scripts, you have to define **transformation**. Usually transformation and preprocessing on the source environment are used an upgrade impact of a change is major, and using usual post-synchronize upgrade script in the target environment is not an option for performance reasons.

By defining transformation, you provide the framework the tables you are going to use, which additional tables the framework needs to create in the source system, and which scripts to run in the source environment. Later, that information will be used when the upgrade framework copies the data to the target system.

Note that you cannot change any existing tables or any data in the existing source system. The only tables in the source environment which you can insert records into are “shadow tables” created by the framework or the “dictionary tables” backported from Microsoft Dynamics AX 2010.

The following lists the required steps to define transformation for each version, Microsoft Dynamics AX 4 and Microsoft Dynamics AX 2009:

- a) Import the preprocessing upgrade framework XPO file (from the installation CD) into the Microsoft Dynamics AX 4 and Microsoft Dynamics AX 2009 development environments.
- b) Create a class derived from `ReleaseUpdateTransformDB`, or modify the existing class. For each version, a set of classes exists - one upgrade class per module. They are named `ReleaseUpdateTransformDB<version>_<module>`, for example `ReleaseUpdateTransformDB41_Bank`.
- c) Create or modify the `initTransformationJobs` method. Add the definition of the new transformation.
- d) Export your new class and all necessary tables into the standalone XPO file.

To define a valid transformation, you have to do the following:

i) Create transformation

ReleaseUpdateTransformDB.New()

```
ReleaseUpdateTransformDB transformation = new ReleaseUpdateTransformDB();
```

Instantiate a new Table Transformation class *object*

ii) Define source table

Application teams can:

- Hardcode the source table names (which may require end customers to update that code to fit their customizations) or
- Create automated discovery based on their business rules, for example find all fields with certain EDT or relationships.

void ReleaseUpdateTransformDB.SetSourceTable(TableName _sourceTableName)

Description: Register transformation class for specific source affected table.

Example:

```
transformation.SetSourceTable(tablestr(LedgerTable));
```

iii) Define source fields

void ReleaseUpdateTransformDB.addSourceField(identifiername _fieldName)

Description: Registers a field on the source affected table to be used for transformation. By registering field a source, user tells upgrade framework NOT to copy that field from the source database, but copy a fields registered as target field from the shadow table instead

Example:

```
transformation.addSourceField(fieldstr(LedgerTable, AccountNum));
```

iv) Define Shadow table

Shadow tables should be included in the preprocessing XPO files so they can be created during the XPO import.

v) Define target fields

void ReleaseUpdateTransformDB.addTargetField(extendedtypeId typeId, identifiername _fieldName)

Description: Registers a field on the target affected table to be used for transformation. That field will be automatically created in the shadow table with TypeID and FieldName provided by the user. Later, when

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

processing BULK COPY of the data from source environment to target, this field will be automatically copied to the target affected table from the shadow table

Example:

```
transformation.addTargetField(typeid(Description), 'NewAccountNum');
```

i) Per Company To Global Table

If your table is global in Microsoft Dynamics AX 2012 but per company in the previous version, you need to follow these steps:

- Call the API `SetPerCompanyToGlobal`, for ex., in the `ReleaseUpdateTransformDB50_Basic.initTransformationJobs():transformation_DirPartyTable.SetPerCompanyToGlobal(NoYes::Yes);SaveDataPerCompany` is set to No in the shadow table
- Create a field name **Shadow_DataAreaID** in the shadow table
- Implement the logic to copy data from the source table to the shadow table as appropriate

ii) Schedule preprocess scripts

Data upgrade scripts in the source environment are executed in AX4 and AX5 environments. You have to create a separate version of your script for AX4 and AX5 upgrades.

Application teams can provide 4 kinds of scripts for process their transformation, which have to be scheduled in the `initTransformationJobs` method:

- Scripts called when the source system is live = live preprocessing. These scripts must be written using row-by-row operations
- Scripts which will be called when system is in the single user mode, but also optionally can be run by a user when the system is live = Delta processing
- Scripts called when the source system is only in Single User mode = Single User Preprocessing. These scripts must be written as set-based operations.
- Scripts called in a test environment to validate to identify upgrade readiness and data validation issues for data upgrade process. These scripts can be run in a Test Run mode which can be scheduled by clicking on the first item in the preprocessing checklist.

```
void ReleaseUpdateTransformDB.SetLivePreProcessingScript(ClassId _scriptClassId, identifiername  
_scriptName, ReleaseUpdateScriptType _scriptType = ReleaseUpdateScriptType::SharedScript, container  
_configKeys = connull(), boolean _requiresXact = true)
```

Description: Register preprocessing script template (for the LIVE environment) to be used for the transformation.

```
transformation.SetLivePreProcessingScript(classidget(this), methodstr(ReleaseUpdateTransformDB50_Invent,  
checkAndPreparePreUpgrade));
```

```
void ReleaseUpdateTransformDB.SetDeltaPreProcessingScript(ClassId _scriptClassId, identifiername  
_scriptName, ReleaseUpdateScriptType _scriptType = ReleaseUpdateScriptType::SharedScript, container  
_configKeys = connull(), boolean _requiresXact = true)
```

Description: Register preprocessing script template (for the Single User Mode and optionally for LIVE environment) to be used for the transformation.

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

```
transformation.SetDeltaPreProcessingScript(classidget(this), methodstr(ReleaseUpdateTransformDB50_Invent,
checkPreUpgrade));
```

*void ReleaseUpdateTransformDB.SetSingleUserPreProcessingScript(ClassId _scriptId, identifiername
_scriptName, ReleaseUpdateScriptType _scriptType = ReleaseUpdateScriptType::SharedScript, container
_configKeys = connull(), boolean _requiresXact = true)*

Description: Register preprocessing script template (for the Singe User Mode environment) to be used for the transformation.

```
transformation.SetSingleUserPreProcessingScript(classidget(this), methodstr(ReleaseUpdateTransformDB50_Invent,
checkPreUpgrade));
```

*void ReleaseUpdateTransformDB.SetValidationPreProcessingScript(ClassId _scriptId, identifiername
_scriptName, ReleaseUpdateScriptType _scriptType = ReleaseUpdateScriptType::SharedScript, container
_configKeys = connull(), boolean _requiresXact = true)*

Description: Register preprocessing script template (for the Live validation environment) to be used for validation. Validation scripts are only run in a test environment and not in a production environment. See the section *Logging Data Issues in Validation Scripts* for more information about this script type.

```
transformation.SetValidationPreProcessingScript(classidget(this), methodstr(ReleaseUpdateTransformDB50_Invent,
validatePreUpgrade));
```

iii) Save transformation

Save()

Description: Saves the transformation

Example:

```
TableTransformation.save ();
```

iv) Define dependencies between transformations

static void addTransformDependency(RefRecId _firstId, RefRecId _thenId)

Description: Adds dependency between transformations.

Example:

```
ReleaseUpdateTransformDB::addTransformDependency(Transformation1. getTransformationId(),Transformation2.
getTransformationId());
```

static void addTransformDependencyByTable(tableId _tableId, RefRecId _thenId)

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

Description: Adds dependency between one transformation and all other transformations that affect the specified table, including transformations from other modules; i.e. 'cross module'. These dependencies are resolved in `ReleaseUpdateTransformDB::initJobs`.

Example:

```
ReleaseUpdateTransformDB::addTransformDependencyByTable(tablenum(<table>),Transformation2. getTransformationId());
```

Note: The first parameter is the source table of the transformation, not the shadow table.

static void addTransformDependencyByMethod(classId _classId, IdentifierName _methodName, RefRecId _thenId)

Description: Adds dependency between one transformation and another transformation by class/method. This allows 'cross module' transformation dependencies to be defined. These dependencies are resolved in `ReleaseUpdateTransformDB::initJobs`.

Example:

```
ReleaseUpdateTransformDB::addTransformDependencyByMethod(classnum(<class>), <methodName>,Transformation2. getTransformationId());
```

v) Supporting Pause/Resume Functionality and Committing Data in Batches

During an upgrade you may want to pause an upgrade script and later resume the script from where it left off. In addition, you may want to improve performance by committing processed data in batches. The Upgrade Preprocessing Framework has API support for these scenarios. This is called script throttling. In most cases, live preprocessing scripts should be implemented with throttling.

The `ReleaseUpdateTransactionManager` class implements the following methods that support script throttling:

NOTE: When calling the `Set*` methods in `initTransformationJobs` to register your upgrade scripts, be sure to set the `requiresXact` flag to `false`. Example:

```
SetLivePreProcessingScript(_classnum(ReleaseUpdateTransformDB_Admin),  
methodstr(ReleaseUpdateTransformDB_Admin, myScriptName,  
ReleaseUpdateScriptType::SharedScript, connull(), false);
```

*ReleaseUpdateTransactionManager::newTransactionManager(classId _upgradeScriptClassId,
identifierName _upgradeScriptMethodName, identifierName _helperMethodName = '', freeText_blockId =
'')*

Description: Creates a new instance of the transaction manager and associates it with your upgrade script. An optional helper method name and block ID can be specified for multiple code blocks. The `_blockId` can be any value that is appropriate for your situation.

NOTE: When you use multiple instances of `ReleaseUpdateTransactionManager` in your upgrade script, if one instance receives a pause command be sure to properly exit the upgrade script. Do not allow the script to continue. `ReleaseUpdateTransactionManager::newTransactionManager` will throw an exception if it detects another instance was paused. See the example below.

bool initialize()

Description: Initializes the transaction manager. Returns 'true' if the script is in resume mode.

validateTransaction(container _controlData)

Description: Call this method at the beginning of the while select loop in the upgrade script. The method determines if a ttsbegin is needed. Note: For Microsoft Dynamics 2009, the ttsbegin and matching ttscommit must be within the same code block. This method accounts for that limitation.

container geLastIterationState

Description: Returns information that the script uses to determine at what point it should resume. In most cases, this can be the last RecId processed by the script, but can be any set of values that make sense for your situation (control data).

boolean doIterationAndContinue(container _controlData)

Description: Call at the end of the while select loop in the upgrade script. The method keeps track of how many iterations the while select has processed and automatically performs a ttscommit and ttsbegin when the iteration limit has been reached. For example, if a script should process batches in rows of 10,000, this method will count the rows, then commit when it reaches 10,000. The method returns `false` if the script has been paused by the user. Your script should 'break' at this point.

Here is an example of using the throttling API in a Live script. Note that the pattern for Delta scripts is different.

```
public void throttlingAPITest()
{
    testTableSource      srcTable;
    testTable2Source     srcTable2;
    testTableShadow      shadowTable;
    RefRecId             lastRecId;
    boolean               isResuming;
    ReleaseUpdateTransactionManager transactionManager1;
    ReleaseUpdateTransactionManager transactionManager2;
    ;
    // Do not delete_from the shadow table here
    // Create a new instance of the transaction manager
    transactionManager1 =
ReleaseUpdateTransactionManager::newTransactionManager(classidget(this),
        methodstr(ReleaseUpdateTransformDBTest, throttlingAPITest), ',', '1');
    // Determine if this script is resuming from a previous pause
    isResuming = transactionManager1.initialize();
    // Retrieve the last RecId processed
    [lastRecId] = transactionManager1.getLastIterationState();

    while select *
```



```

    from srcTable
    order by RecId, RecVersion
    where (other conditions) && (srcTable.RecId > lastRecId || !isResuming) //
Understand your script logic and implement accordingly - don not copy and paste as
this example uses RecId to track the last processed row. Your scenario may vary.
{
    // validateTransaction will start the transaction, if needed
    transactionManager1.validateTransaction([srcTable.RecId]);

    shadowTable.clear();
    shadowTable.Name = srcTable.Name;
    shadowTable.insert();

    // doIterationAndContinue checks if a commit should be performed
    if (!transactionManager1.doIterationAndContinue([srcTable.RecId]))
    {
        // If doIterationAndContinue returns false, exit
        return;
    }
}
// Complete the final transaction
transactionManager1.endTransaction([srcTable.RecId]);

// Create a new instance of the transaction manager for loop 2
transactionManager2 =
ReleaseUpdateTransactionManager::newTransactionManager(classidget(this),
    methodstr(ReleaseUpdateTransformDBTest, throttlingAPITest), ',', '2');
// Determine if this script is resuming from a previous pause
isResuming = transactionManager2.initialize();
// Retrieve the last RecId processed
[lastRecId] = transactionManager2.getLastIterationState();

while select *
    from srcTable2
    order by RecId, RecVersion
    where (srcTable.RecId > lastRecId) || !isResuming
{
    // validateTransaction will start the transaction, if needed

```

```

transactionManager2.validateTransaction([srcTable2.RecId]);

shadowTable.clear();
shadowTable.Name2 = srcTable2.Name;
shadowTable.insert();

// doIterationAndContinue checks if a commit should be performed
if (!transactionManager2.doIterationAndContinue([srcTable2.RecId]))
{
    // If doIterationAndContinue returns false, exit
    return;
}
}
// Complete the final transaction
transactionManager2.endTransaction([srcTable2.RecId]);
}

```

Note: In the above example, we have to add the RecVersion to the *Order By* clause since *Order By* does not work with RecId alone due to a bug in Microsoft Dynamics AX.

Note: The RecID might not be unique, select a combination of fields to track the processed rows in your query. For example, with non-shared tables, DataAreaID might be used in your query condition:

```

while select crosscompany AddressCounty order by DataAreaId, RecId, RecVersion
where (other conditions) && ((AddressCounty.DataAreaId > lastDataAreaId ||
(AddressCounty.RecId > lastRecId && AddressCounty.DataAreaId == lastDataAreaId)) ||
!isResuming)

```

In this case, DataAreaId should be an additional input/output when calling the throttling APIs, for example:

```

...
[lastRecId, lastDataAreaId] = transactionManager.getLastIterationState();
...
if (!transactionManager.doIterationAndContinue([AddressCounty.RecId,
AddressCounty.DataAreaId]))
...

```

Note: DataAreaId is used for a shared script only. For a standard script, which runs per company, DataAreaId is redundant in the above query.

Example of API use in a Delta script:

```
isResuming = transactionManager.initialize();//isResuming might not be used but
initialize() should be called.
```

```
delete_from shadow_InventCostTrans
notexists join RecId, recVersion from InventCostTrans
where   InventCostTrans.RecId           == shadow_InventCostTrans.RefRecId &&
        InventCostTrans.recVersion     == shadow_InventCostTrans.RecVersionId;

while select RecId, recVersion, Dimension from InventCostTrans
order by InventCostTrans.RecId, InventCostTrans.recVersion
notexists join shadow_InventCostTrans
where   InventCostTrans.RecId           == shadow_InventCostTrans.RefRecId &&
        InventCostTrans.recVersion     == shadow_InventCostTrans.RecVersionId &&
        ((InventCostTrans.RecId > lastRecId) || !isResuming)// ← Do not add a
where clause for checking last recid processed here. Deltas may have occurred in
rows prior to this. The not exists join takes care of this.
{
    // validateTransaction will start the transaction, if needed
    transactionManager.validateTransaction([InventCostTrans.RecId]);

    shadow_InventCostTrans.RefRecId      = InventCostTrans.RecId;
    shadow_InventCostTrans.RecVersionId  = InventCostTrans.recVersion;
    shadow_InventCostTrans.DefaultDimension =
DimensionConversionHelper::getNativeDefaultDimension(InventCostTrans.Dimension);

    shadow_InventCostTrans.insert();

    // doIterationAndContinue checks if a commit should be performed
    if (!transactionManager.doIterationAndContinue([InventCostTrans.RecId]))
    {
        // If doIterationAndContinue returns false, exit
        return;
    }
}

// Complete the final transaction
transactionManager.endTransaction([InventCostTrans.RecId]);
```

i) Logging Data Issues in Validation Scripts

Validation scripts in preprocessing have a unique feature that allows them to identify data issues discovered and alert the user. The script can rely on the framework to present these issues to the user, and can optionally provide a custom UI for either displaying or fixing these issues. To provide these features, validation scripts can use the following API:

ReleaseUpdateValidationLogger

ReleaseUpdateValidationMessages (macros)

```
log(ReleaseUpdateMessageId _messageId, ReleaseUpdateLogResultType _logResult, freeText _messageText,
freeText _resolution, SysInfoAction _action = null, ReleaseUpdateActionType _actionType =
ReleaseUpdateActionType::None)
```

```
logDetail(ReleaseUpdateMessageId _messageId, freeText _message)
```

ReleaseUpdateValidationLogger::newValidationLogger(classId _upgradeScriptClassId, identifierName _upgradeScriptMethodName)

Description: Creates a new instance of the validation logger and associates it with your upgrade script.

log(ReleaseUpdateMessageId _messageId, ReleaseUpdateLogResultType _logResult, freeText _messageText, freeText _resolution, SysInfoAction _action = null, ReleaseUpdateActionType _actionType = ReleaseUpdateActionType::None)

Description: Logs a unique message. The message is tracked by _messageId (entered in ReleaseUpdateValidationMessages) and only logged once per script, even if the log message is called multiple times with the same messageId.

logDetail(ReleaseUpdateMessageId _messageId, freeText _message)

Description: Logs a detail message associated with a previously logged messageId (log).

The following is an example of a log entry and its corresponding details:

Log: There are invalid states in table x

Detail: Row with RecId 1 has invalid state 'Wash'

Row with RecId 2 has invalid state 'Mass'

Row with RecId 100 has invalid state 'Fla'

Example script using the validation APIs:

```
#ReleaseUpdateMessageIds

MyAddresses                myAddresses;
SysInfoAction_ValidationForm  action;
ReleaseUpdateLoggerInterface  logger;
```

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

```

;

action = new
SysInfoAction_ValidationForm::newFormname(formstr(MyAddressesResolveState));

logger = ReleaseUpdateValidationLogger::newValidationLogger(
    classidget(this),

methodstr(ReleaseUpdateTransformDB41_Admin,
    validateMyAddresses));

while select *
    from myAddresses
{
    if (!this.IsValidState(myAddresses.State))
    {
        logger.log(
            #myAddressesStateInvalid,           // MessageId
            ReleaseUpdateLogResult::Error,      // Type
            "The following rows have invalid states:", // Message Text
            "Update these rows to use valid states.", // Resolution
            action,                             // SysInfoAction
            ReleaseUpdateActionType::Fix)       //
        ReleaseUpdateActionType

        logger.logDetail(
            #myAddressesStateInvalid,
            strFmt("RecId: %1 (%2)",
                myAddresses.RecId,
                myAddresses.State))
    }
}

```

j) Modify a subset of records in a table during upgrade

Using the upgrade framework, you can upgrade a subset of records in a table while keeping the rest unchanged.

For example, the MarkupTrans table has the Keep field which should only be changed for a subset of records.

The recommended solution is to write the upgrade scripts as if the field should be upgraded for the entire table.

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

The difference is that the **select** statement in the upgrade scripts should have a **where** clause to indicate that only records that should be upgraded are selected. Shadow records are then created for the records that should be upgraded.

When registering the upgrade script in the transformation, it should set the shadow table join type to be left outer join:

```
transformation.setShadowTableJoinType(ReleaseUpdateJoin::LeftOuterJoin);
```

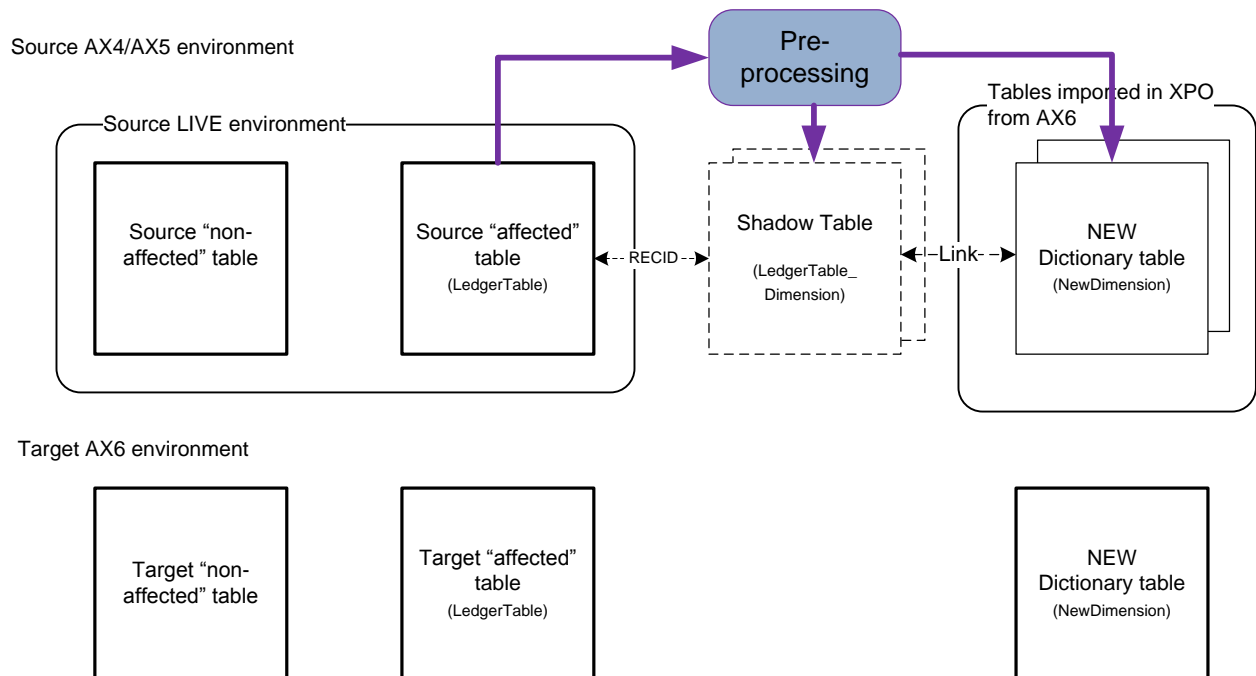
During bulk copy, the framework will then automatically set the upgraded field to the upgraded value if one exists otherwise it will keep the original value for the field.

This approach assumes that the fields on a record which determine whether it should have a record in the shadow table cannot change in the source environment. Otherwise, it is a challenge to identify modified records.

Writing Data Upgrade Scripts for Source Environment

Writing Live preprocessing scripts

Preprocessing scripts can be Direct SQL or X++ application scripts executed from within AX by the framework:



Scripts must correctly accept source affected table names and field names, and respective shadow table name, and perform necessary transformation of data from source tables/fields:

- (1) **Records in new tables:** should be created directly in the new "dictionary" table (f.ex. NewDimension table) which was imported from target system into the source system
- (2) **Modified fields of the existing "source" table:** saved in a "shadow" table, linked with the source table using RecId and RecVersionId link

```
preUpgradeScript(SourceTableName, SourceFieldNames[], ShadowTableName, ShadowFieldNames[])
{
// application logic goes here

//insert into new dictionary tables
WHILE SELECT FROM SourceTableName
{
    // Check if record exist in Dictionary Table1
    SELECT FROM Dictionary Table1 where <some condition based on SourceTableName >
```

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

```

// IF it does not exist, insert it
INSERT INTO DictionaryTable1 <some fields from SourceTableName>

// Check if record exist in Dictionary Table2
SELECT FROM Dictionary Table2 where <some condition based on SourceTableName >

// IF it does not exist, insert it
INSERT INTO DictionaryTable2 <some fields from SourceTableName>

// insert into "shadow table" modified source fields link to the Dictionary tables and
// RecId/RecVersionId link to the source table

S = INSERT INTO ShadowTableName
(REFRECID, REFDATAAREAID, RECVERSIONID, shadowFieldName[1], shadowFieldName[2])
SourceTableName.RECID,
SourceTableName.DATAAREAID,
SourceTableName.RECVERSIONID
DictionaryTable1.targetField1,
DictionaryTable2.targetField2
executeStatement(S);
}
}

```

Note that if direct SQL is used, then 2 versions of the script has to be implemented by application teams – Oracle and MS SQL

Writing Delta and single user mode preprocessing scripts

Purpose of these scripts is to check the state of the source tables since preprocessing script was run on the LIVE system, and resolve all discrepancies occurred since.

These will be Direct SQL or application scripts executed from within AX by the framework. It'll be script's responsibility to identify new or modified records in the source table and take appropriate actions (for example, rerun the business logic for new records). However, framework will provide some guidance on this, for example:

There are 3 types of changes which may have happened to a record

- a) How to identify new records
Script can identify new records by using shadow table RefRecIds.

For instance,

```
SELECT SourceTable.RecIDs NOT EXIST ShadowTable.RefRecId
```

If such record was found, we can execute preprocessing script on it

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

b) Record updated

Script can find updated records using RecVersion field. A search like provided below can be used:

```
SELECT SourceTable, ShadowTable where SourceTable.RECID = TargetTable.RECID and SourceTable.RecVersion != ShadowTable.RecVersion
```

If such record was found, we will delete the record from shadow table and re-run preUpgradeScript() on it. However, that approach may leave some orphan records in the “dictionary” tables. Also, RecVersion may not be very reliable due to third party components writing directly to the database

c) Record deleted

Script can identify new records by using shadow table RefRecIds, but usually there is no need as later we will be joining Shadow table and Source table on RecID (see section 4). However, orphan records may be left in the “dictionary” tables

As soon as table’s delta processing completed for all transformations, the table must be marked as **ready to be copied**

Best Practices for Writing Data Upgrade Scripts

Best Practice Checks

In Microsoft Dynamics AX 2010, there are now several best practice checks that help to verify the completeness of upgrade script attributes.

How An Upgrade Script Is Identified

Any method that is decorated with any of the following attributes is considered an upgrade script:

- UpgradeScriptDescriptionAttribute
- UpgradeScriptTypeAttribute
- UpgradeScriptStageAttribute
- UpgradeScriptTableAttribute

Note: There are conditions that further identify a method as an upgrade script that are not considered for performance reasons when you check for best practices. During an actual upgrade, the additional conditions are considered and validated.

The following lists BPErrors codes when a violation occurs and resolutions.

BPErrors code and condition/message	Resolution
BPErrorsMethodIsStatic An upgrade script cannot be a static method.	Remove the static modifier.
BPErrorsMethodHasParams An upgrade script cannot accept parameters.	Remove the parameters.
BPErrorsMissingReqDesc For scripts that upgrade to Microsoft Dynamics AX 2012, the UpgradeScriptDescriptionAttribute attribute is required.	Add the required attribute.
BPErrorsMissingReqType For scripts that upgrade to Microsoft Dynamics AX 2012, the UpgradeScriptTypeAttribute attribute is required.	Add the required attribute.
BPErrorsMissingReqStage For scripts that upgrade to Microsoft Dynamics AX 2012, the UpgradeScriptStageAttribute attribute is required.	Add the required attribute.
BPErrorsMissingReqTable One or more UpgradeScriptTableAttribute attributes are required for all upgrade scripts. In addition, any table referenced in the method must have a corresponding	Add the required attribute for any tables referenced by the upgrade script. The following upgrade scripts are exempt from this requirement: <ul style="list-style-type: none">• deleteDuplicatesUsingIds

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

UpgradeScriptTableAttribute attribute.	<ul style="list-style-type: none"> • deleteDuplicatesUsingNames • indexAllowDup • indexAllowNoDup
<p>BPErrortableNotFound</p> <p>The table name specified in the UpgradeScriptTableAttributes attribute is not valid. Use the tableStr(<table>) method to catch this issue at compile time.</p>	Ensure the table specified exists and that the name is typed correctly.
<p>BPErrortableConfigurationKeyNotFound</p> <p>The configuration key specified in UpgradeScriptConfigKeyAttribute attribute is not valid. Use the configurationKeyStr(<configurationKey>) method to catch this issue at compile time.</p>	Ensure the configuration key exists and that the name is typed correctly.
<p>BPErrortableClassNameNotFound</p> <p>The class name specified in UpgradeDependsOnModuleAttribute attribute or UpgradeDependsOnVersionAttribute attribute is not valid. Use the classId(<class>) method to catch this issue at compile time.</p>	Ensure the class exists and that the name is typed correctly.
<p>BPErrortableMethodNameNotFound</p> <p>The method name specified in UpgradeDependsOnModuleAttribute attribute, UpgradeDependsOnTaskAttribute attribute, or UpgradeDependsOnVersionAttribute attribute is not valid. Use the methodStr(<class>, <method>) method to catch this issue at compile time.</p>	Ensure the method exists and that the name is typed correctly.
<p>BPErrortableInvalidScriptVersion</p> <p>The module for an upgrade script must have a version macro defined in the class declaration:</p> <pre>#define.version(sysReleasedVersion::v60)</pre>	Ensure the version macro is defined and that it is not Unknown or vNext.

Transaction and Idempotency

It is an important requirement that each data upgrade script be *idempotent*. Idempotent means that if the execution fails, it must be able to execute successfully with the desired results upon reexecution.

The data upgrade framework guarantees idempotency by enclosing each script within a transaction, ensuring that the script is only executed once. Although this is a simple and robust way to ensure idempotency it results in a performance decline when an upgrade script has complex logic in a loop on a large table. In Microsoft Dynamics AX 4.0, this mechanism is optional such that an individual script can be run without the transaction at the highest level. When this option is chosen, the individual script must implement its own idempotency logic.

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

Another important consideration for implementing idempotency is that you can upgrade from many different versions. For example, if you write an upgrade script for SP2 of version N, when version N+1 is shipped, your customers are upgrading from both Version N SP1 and Version N SP2. This means that some customers already are upgraded and others are not. If your upgrade script is idempotent, you can just reuse it for the upgrade to version N+1.

Note that if an upgrade script contains an error, it is easier to resolve the problem if the script is idempotent.

Coding Best Practices

Indicating Progress

To supply progress status, you can use a simplified version operation progress by calling:

```
this.tableProgress(<tableId>);
```

and including the table-ID for the table you have just updated. This should only be called once in each outermost loop (even if you are updating several tables in the inner loops).

Documenting Scripts

You should include meaningful comments in each data upgrade script to explain the functionality of the script.

Deleting a Table or Field from the Data Model

It is not possible to simply delete data from the data model as this would be the equivalent of deleting customer data. This also applies to fields that were never used or fields that appear in the UI (unless they are temporary). Removing a field or table requires careful planning and execution as follows:

- 1) Prefix the name of the item to be removed with "DEL_" and move it to the upgrade model, using the following steps:
 - a) For fields and indexes:
 - i) Rename them in the AOT using F2 or the PropertySheet. The field/index will automatically be moved into the upgrade model in the next build.
 - ii) In the rare case where you want the DEL_ fields/indexes to remain in the Foundation, you can add the table to the exclusion list here:
//depot/main/source/application/CombineXPOs/NoDelExclusionList.txt.

This should only be used when production code needs the tables in a post-upgrade scenario.

- b) For tables, enums and extended data types:
 - i) Turn-off version control.
 - ii) Rename the element in the AOT using F2 or the PropertySheet.
 - iii) Right-click the element and click **Move to model**. Click **Foundation Upgrade**, click **Ok**.
 - iv) In CoreXT type:

```
CD "source\application\sys\data dictionary"
```

For tables:

```
SD integrate "tables\<tablename>.xpo" "..\..\sysupgrade\data dictionary\tables\DEL_<tablename>.xpo"
```

```
SD delete "tables\<tablename>.xpo"
```

For enums:

```
SD integrate "base enums\<enumname>.xpo" "..\..\sysupgrade\data dictionary\base enums\DEL_<enumname>.xpo"
```

```
SD delete "base enums\<enumname>.xpo"
```

For extended data types:

```
SD integrate "extended data types\<typename>.xpo"  
"..\..\sysupgrade\data dictionary\extended data  
types\DEL_<typename>.xpo"
```

```
SD delete "extended data types\<typename>.xpo"
```

v) Turn-on version control again

- 2) Set the configuration property to: "[SysDeletedObjectsXX](#)" where XX is the next version, for example "60" (for Microsoft Dynamics AX 2012)
- 3) Implement the upgrade script that will transform the data into the new data model. Verify the current model is **Foundation Upgrade** when you create the class.
- 4) Test the upgrade script
- 5) Benchmark the upgrade script
- 6) (*New for Interim Upgrade*): **When deleting a field from a table**, do not delete the table or field permanently from the AOT. They need to stay in the source until the release, where they are deleted is no longer supported by upgrade. For example, if a Microsoft Dynamics AX 4.0 field is renamed in Microsoft Dynamics AX 4.1 as a DEL_field, it needs to stay in the source until Microsoft Dynamics AX 2012.
- 7) When deleting a table, after you rename the table to DEL_TableName then the upgrade process will copy the data from the previous version to this Del_table. It is not a good practice to have two copies of the same table. The old table & field can be deleted after creating the DEL_table & fields. These DEL_tables should be tied to SysDeleted configuration keys.

Unique Indexes

It is important that the database can synchronize without errors when the customer upgrades. Three scenarios require special attention when dealing with index changes:

- 1) Removing a field from a unique index
- 2) Adding a new unique index
- 3) Making a non-unique index unique, (setting the [AllowDuplicates](#) property to false)

All these scenarios make an index more restrictive and will cause the synchronization to fail if not handled properly.

The easiest solution is to delete the data that collides with the index. This should only be done in situations where it doesn't make sense to keep the duplicate records. This is performed using the following steps:

- 1) Create a start pre-synchronization upgrade script. This will change the index to allow duplicates:

```
DictIndex dictIndex = new  
    DictIndex (TableNum (<TableName>), indexNum (<TableName>, <IndexName>));  
;  
ReleaseUpdateDB::indexAllowDup (dictIndex);
```

- 2) Create a normal upgrade script. This will move the data according to the new data model.
- 3) Create a final post-synchronization upgrade script. This will change the index to not allow duplicates:

```
DictIndex dictIndex = new  
    DictIndex (TableNum (<TableName>), indexNum (<TableName>, <IndexName>));  
;  
ReleaseUpdateDB::indexAllowNoDup (dictIndex);
```

Consideration for date effective table in upgrade script

Some Microsoft Dynamics AX 2010 tables with date effective indexes that are disabled then re-enabled after synchronization (see section '[Unique Indexes](#)') will also require that the property 'ValidTimeStateKey' is also set to 'Yes' along with re-enabling of the index. To set the 'ValidTimeStateKey' property on the index, follow this pattern:

```
public void allowNoDupAssetParmDeprRtsDEDateEffcIdx()
{
    DictIndex dictIndex = new DictIndex(tablenum(AssetParametersDeprRates_DE),
        indexnum(AssetParametersDeprRates_DE, DateEffcIdx));
    dictIndex.modify(true, false, true);
    dictIndex.setAlternateKey(true, true);
    // Set the ValidTimeStateMode to the appropriate value for your index;
    // i.e. what the value was before the index was disabled.
    dictIndex.setValidTimeStateKey(true, ValidTimeStateMode::Gap, true);
    appl.dbSynchronize(dictIndex.tableid(), false);
}
```

Preventing Copying of Table Data

Situations may occur where you may not want to copy the contents of a table from the source environment to the target environment, for example, if the table will be used in the new version (but not if the table has become obsolete). This can be useful when the contents of the table are auto-generated. To do this, create a pre-synchronization shared script using the pattern:

```
ReleaseUpdateDB::addBulkCopyTableException(tableStr(<table>),
ReleaseUpdateBulkCopyTableExceptionType::DoNotCopy);
```

Alternatively, you can create a pre-synchronization standard script using the delete_from construct.

Applying Field Options

Situations may occur when you need to perform operations on specific table fields during copy & sync. To do this, create a pre-synchronization shared script using the pattern:

```
ReleaseUpdateDB::addBulkCopyFieldOption(tableStr(<table>), fieldStr(<table>, <field>),
ReleaseUpdateBulkCopyFieldOption::LTrimTarget);
```

ReleaseUpdateBulkCopyFieldOption supports the following elements:

LTrimTarget

Applies the Sql LTRIM operator to the source value as it is copied to the target, allowing previously right aligned data to be left-aligned in the target environment.

DoNotCopy

Removes the specified field from the source table's SELECT statement during the bulk copy phase, preventing the field's data from being copied to the target system.

Truncating a Table During Copy & Sync

Situations may occur when you need to truncate the contents of a table in the target environment during copy & sync. To do this, create a pre-synchronization shared script using the pattern:

```
ReleaseUpdateDB::addBulkCopyTableException(tableStr(<table>),
ReleaseUpdateBulkCopyTableExceptionType::TruncateOnCopy);
```

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

This will allow data to be copied to the target table even if it is not empty.

Specifying the Shadow/Source Table JOIN Type

By default, shadow tables are joined to the source table during the bulk copy operation using a simple 'JOIN'. Specify a different join type using the following pattern in the preprocessing script:

```
ReleaseUpdateTransformDB.setShadowTableJoinType (_shadowJoinType);
```

`_shadowJoinType` is of type `ReleaseUpdateJoin` and can be one of the following values:

```
ReleaseUpdateJoin::Join           // 'JOIN' (default)
ReleaseUpdateJoin::FullOuterJoin  // 'FULL JOIN'
ReleaseUpdateJoin::LeftOuterJoin  // 'LEFT JOIN'
ReleaseUpdateJoin::RightOuterJoin // 'RIGHT JOIN'
```

Upgrading a Table with name and fieldid or tableid changed

When a table or field's name is changed, and there is a possible conflict between new and existing table or field names, in order to preserve the table and its data, you must call the following methods in a pre-synchronization Start script:

```
static void void ReleaseUpdateBulkCopyMap.addTableNameMapping(tableName _oldTableName, tableName
_newTableName, NoYes _system = NoYes::No)
```

```
static void ReleaseUpdateBulkCopyMap.addFieldNameMapping(tableName _oldTableName, fieldName
_oldFieldName, tableName _newTableName, fieldName _newFieldName, NoYes _system = NoYes::No)
```

Note: FIELDTYPE of the source and target mapping should match. If there is a mismatch in fieldtype the framework do not copy the data.

Upgrading a Table with Table ID or Field ID Changed

When a table or field's id is changed, no upgrade scripts are needed.

However, for legacy compatibility, we support the following functions to accommodate ID of a table or field change:

```
ReleaseUpdateDB::ChangeTableID (for table ID changes)
```

```
ReleaseUpdateDB::ChangeFieldID (for field ID changes)
```

You can also use the following methods to address tables and fields by name:

```
ReleaseUpdateDB::ChangeTableByName (for table ID changes)
```

```
ReleaseUpdateDB::ChangeFieldByName (for field ID changes)
```

Note that for few special tables listed in the "Precautions When You Write Data Scripts Before Synchronization" section you cannot use pre-synchronization Start script. Please refer to that section for more details and code samples.

Deleting Configuration Keys

Note: Configuration keys should **not** be deleted. Configuration key changes are not handled by code upgrade, therefore, changes will not be detected at code upgrade time. If a customization has been set up to use a Microsoft Corporation shipped configuration key in custom tables, and if the configuration key is deleted, the table will be lost during synchronization.

Referencing Number Sequences within upgrade scripts

If a number sequence has to be referenced within a X++ upgrade script, it is recommended to code that reference as a separate method instead of hardcoding it within the script itself, which will make the process of changing it easier for a user running the upgrade

```
private NumberSequenceReference numberSequenceReference_SQ()  
{  
    return NumberSeqReference::findReference(extendedTypeEnum(SQ));  
}
```

Later in the upgrade script, you can use that method to get the actual number sequence

```
num = NumberSeq::newGetNum(this.numberSequenceReference_SQ(), false);  
salesQuotationTable.QuotationId = num.num();
```

Note that if the number sequence you are using is for an extended data type that is new to Dynamics AX 2012, or if there is the possibility that it has not been previously instantiated within the current scope, you must first instantiate the number sequence prior to using it. Although this is normally an administrator task, which allows the administrator to override some default settings, this can be done programmatically from the upgrade script by calling:

```
NumberSequenceTable::autoCreate(extendedTypeEnum(SQ), [scope]);
```

Performance Guidelines

Performance is a critical piece of the upgrade process and requires that you think about each line in your script. Most companies will perform this task over a weekend, so the entire upgrade process must be able to be completed within 48 hours. The actual update will typically be performed between Friday night and Monday morning. In addition, prior to running the upgrade process on a live system, the upgrade process is tested several times on a test system.

In addition to the following considerations, please read [Performance Improvement Options](#) to determine which apply to your upgrade scripts:

- 1) Monitor and minimize the number of client/server calls.
- 2) Use record set functions whenever possible.
- 3) Break down your scripts into smaller pieces. For example, do not upgrade two independent tables in the same script even if there is a pattern in the way the scripts work. This is because:
 - 1) Each script, by default, runs in one transaction (=one rollback segment). If the segment becomes too large, the database server will start swapping memory to disk, and the script will slowly come to a halt.
 - 2) Each script can be executed in parallel with other scripts.
- 1) Partial commits can only be used out of the box in one situation; this is when the table to upgrade is large and contains a discriminator that can be used to split the script into several scripts. For example, update all "Open" in one script and all "Closed" in another. The scripts should be set up to be dependant on each other to avoid locking problems. (see point below regarding database lock contention)
- 2) Take care when you sequence the scripts. For example, do not update data first and then delete it afterwards.
- 3) Be careful when calling normal business logic in your script. Normal business logic is not usually optimized for upgrade performance. For example, the same parameter record may be fetched for each record you need to upgrade. The parameter record is cached, but just calling the Find method takes an unacceptable amount of time. For example, the kernel overhead for each function call in Microsoft Dynamics AX is 5 ms. Usually 10-15 ms will elapse before the Find method returns (when the record is cached). If there are a million rows, two hours will be spent getting information you already have. The solution is to cache whatever is possible in local variables.
- 4) Run benchmarking on your script using large datasets to verify your performance is acceptable.
- 5) If database lock contention prevents the data upgrade process from scaling up with multiple batch clients running in parallel, consider disabling the transaction in the framework and ensuring idempotency by one of the following:
 - Using an existing field/condition that can check if the table/record has been updated
 - Adding new fields to track upgrade status
 - Using the primary key as ordering columns and recording the last row that was updated
- 1) Use index tunint. Create indexes to speed up the upgrade and possibly remove them after the upgrade. Setting up a configuration key to [SysDeletedObjects<version>](#) can help you ensure that the index is deleted after the upgrade is finished.
- 1) If there is no business logic in the script, rewrite the script to issue a direct query to bulk update the data. To write Direct SQL queries, see [Appendix 2: Guidelines for Writing Direct SQL in Upgrade Scripts](#).

Performance Improvement Options

Using the Set-based Operators Delete_From, Update_RecordSet and Insert_SecordSet

If the script performs inserts, updates, or deletes within a loop, you should consider changing the logic to use one of the set-based statements. If possible, use these set options to perform a single set-based operation.

Note when using set-based operations:

- 1) With Insert_RecordSet you cannot use a literal or function call in the field list. This operation does not handle configuration keys so special care is required.
- 2) With Update_RecordSet you cannot perform inner or left outer joins.
- 3) Set based statements do not support memo fields.

Please refer to Speeding Up SQL Operations and Maintain Fast SQL Operations in the SDK documentation for list and syntax of set based operations available in Microsoft Dynamics AX 2012.

Example:

Before performance improvement:

```
while select inventTable
  where inventTable.ItemType == ItemType::Service
{
  this.tableProgress(tablenum(InventTable));
  delete_from inventSum where inventSum.ItemId == inventTable.ItemId;
}
```

After performance improvement:

```
delete_from inventSum
exists join inventTable
where inventTable.ItemId == inventSum.ItemId
&& inventTable.ItemType == ItemType::Service
```

Calling skipDataMethods and skipDatabaseLog Before Calling Update_RecordSet or Delete_From

If your script runs delete_from or update_from on a large table where the delete() or update() methods of the target table have been overwritten, the bulk database operation will fall back to record-by-record processing. To prevent this, call the skipDataMethods(true) method to cause the update() and delete() methods to be skipped. Also, you can call the skipDatabaseLog(true) method to improve performance.

Example:

```
taxExchRateAdjustment.skipDataMethods(true);
taxExchRateAdjustment.skipDatabaseLog(true);

update_recordset taxExchRateAdjustment
  setting GovernmentExchRate = taxExchRateAdjustment.UseGovtBankRate
  where taxExchRateAdjustment.UseGovtBankRate == NoYes::Yes;
```

Using RecordInsertList Class to Batch Multiple Inserts

If the business scenario cannot be written as `insert_recordset`, consider using the `RecordInsertList` class to batch multiple inserts to reduce network calls. This operation is not as fast as `insert_recordset`, but is faster than individual inserts in a loop.

Example:

```
rilAssetTransMerge = new RecordInsertList(tablenum(assetTransMerge));
while select assetTrans
{
    if (!AssetTransMerge::exist(AssetBookType::ValueModel, assetTrans.RecId))
    {
        assetTransMerge.AssetId = assetTrans.AssetId;
        assetTransMerge.AssetGroup = assetTrans.AssetGroup;
        ...
        rilAssetTransMerge.add(assetTransMerge);
    }
}
rilAssetTransMerge.insertDatabase();
```

Optimizing X++ logic

To optimize X++ logic, apply the following rules:

- 1) Minimize the amount of time spent in the X++ interpreter
- 2) For database related code, ensure SQL is fully utilized by including where conditions, for example, to check for null values, using joins across tables
- 3) Use set-based updates and inserts instead of record-based updates and inserts

Examples of the wrong way to code:

```
while select forupdate projForecastCost
where ! projForecastCost.TransId
{
    if (! projForecastCost.TransId)
    {
        numberSeq = NumberSeq::newGetNum(ProjParameters::numRefProjTransIdBase());
        .....
    }
}
```

The `where !projForecastCost.TransId` is already checked by SQL. There is no need to check the value again. The entire statement `if (! projForecastCost.TransId)` should be removed.

```
void someFunc()
{
    while select custTable
    {
        if (custNum != 0)
        {
            dosomething()
        }
    }
}
```

Again, this is not good coding practice. SQL can perform this operation for you.

Rewrite the above function as:

```
void someFunc()
```

```

{
  while select custTable where custNum != 0
  {
    dosomething()
  }
}

```

Below is another example of wasting CPU cycles in the X++ interpreter:

```

private ledgerSRUCode somefunc(AccountNum _accountNum)
{
  .....
  if (auxAccountNum >= '1910' &&
      auxAccountNum <= '1979')
  {
    ledgerSRUCode = '200';
  }

  if ((auxAccountNum >= '1810' && auxAccountNum <= '1819') ||
      (auxAccountNum >= '1880' && auxAccountNum <= '1889'))
  {
    ledgerSRUCode = '202';
  }
  ..... and so on

  return ledgerSRUCode;
}

```

This function only gets the ledgerSRU. So, when this is done, you should exit the function and not execute the if statements. Also, if you are aware of the most likely results, test for these most likely options early in your code.

Below is a corrected version:

```

private ledgerSRUCode someFunc(AccountNum _accountNum)
{
  .....

  if (auxAccountNum >= '1910' &&
      auxAccountNum <= '1979')
  {
    return '200';
  }

  if ((auxAccountNum >= '1810' && auxAccountNum <= '1819') ||
      (auxAccountNum >= '1880' && auxAccountNum <= '1889'))
  {
    return '202';
  }
  ..... and so on

}

```

Appendix I: Guidelines for Writing Direct SQL in Upgrade Scripts

Using Set-Based Updates in X++

Whenever possible, set-based updates should be used in place of row-based updates. Set-based updates have a partial implementation in X++ as `insert_recordset`, `update_recordset`, and `delete_from`. You can implement set-based operations in X++ when:

- 1) An update involves data or references to a single table only. In other words, the data to be updated in a table is not derived from another column. For example:

```
while select forupdate some_table where some_table.some_column == some_value
{
    some_table.some_column = new_value;
    some_table.doUpdate();
}
```

Can be rewritten in X++ as:

```
Some_table st;
Update_recordset st
Setting some_column == new_value
Where st.some_column = some_value;
```

If the update method is overridden, the `update_recordset` will change into a row-by-row update, executing the update code for each row. You can prevent this by using the `skipDataMethod` operator. Refer to [Calling skipDataMethods and skipDatabaseLog Before Calling Update RecordSet or Delete From](#) for more details.

- 1) An `update_recordset` or `delete_from` that includes in its selection criteria a check for existence or absence of data in the same or different table. In X++ these can be implemented directly using the EXISTS Join or NOT EXISTS Join.

For example:

```
while select SalesBasketId from salesBasket
where salesBasket.CustAccount == guestAccount
{
    delete_from salesBasketLine
    where salesBasketLine.SalesBasketId ==
        salesBasket.SalesBasketId;
}
```

Can be rewritten as:

```
delete_from salesBasketLine
exists join salesBasket
where salesBasket.SalesBasketId == salesBasketLine.SalesBasketId
&& salesBasket.CustAccount == guestAccount;
```

Executing Direct SQL from X++

How to Execute Direct SQL for X++

- 1) If Direct SQL code is executed using X++, it requires checking for Code Access Security.as follows:

In the variable definition section, add:

```
SqlStatementExecutePermission permission;  
;
```

In the code section, add:

```
stmtString = < SQL Statement >;  
stmt = con.createStatement();  
permission = new SqlStatementExecutePermission( stmtString );  
permission.assert();  
stmt.executeUpdate(stmtString);  
// the permissions needs to be reverted back to original condition.  
CodeAccessPermission::revertAssert();
```

- 1) Direct SQL stored procedures are executed using X++ as shown in the following example:

```
str sql;  
str dataAreaId;  
Connection conn;  
SqlStatementExecutePermission permission;  
;  
dataAreaId = curExt();  
sql = = 'execute <StoredProcName> \'' + dataAreaId + '\\\' \\' + numSeq + '\\';  
permission = new SqlStatementExecutePermission(sql);  
conn = new Connection();  
permission = new SqlStatementExecutePermission(sql);  
permission.assert();  
conn.createStatement().executeUpdate(sql);  
// the permissions needs to be reverted back to original condition.  
CodeAccessPermission::revertAssert();
```

Best Practices Warning when Executing Direct SQL

Executing Direct SQL is a deviation from Best Practices recommendations, so, whenever Direct SQL is executed, the X++ compiler will flag it as a best practice error. To suppress this warning, before the `stmt.executeUpdate(stmtString)` statement you will need to place the following comment indicating that this is a known deviation from best practices:

```
//BP Deviation Documented
```

The code will be changed to:

```
stmtString = < SQL Statement >;  
stmt = con.createStatement();  
permission = new SqlStatementExecutePermission( stmtString );  
permission.assert();  
// BP Deviation Documented  
stmt.executeUpdate(stmtString);  
CodeAccessPermission::revertAssert();
```

Using Utility Functions to Execute Direct SQL

Two new methods, `statementExeUpdate()` and `statementExeQuery()`, have been added to the `ReleaseUpdateDB` class. They can be used to run any Direct SQL statements in `ReleaseUpdateDB` based classes. Note that, for security reasons, these functions do not have `CAS assert()` or `revertAssert()` methods, these should be called by the caller. See the code example in [Stored Procedure and function Guidelines](#) for `ReleaseUpdateDB::statementExeUpdate` and `ReleaseUpdateDB::statementExeQuery` use.

Documenting Direct SQL

For debugging and maintenance purposes, always put the resulting direct SQL statement as a comment before the code that performs the string construction.

Using Table Names in Direct SQL

Use `ReleaseUpdateDB::backendFieldName` and `ReleaseUpdateDB::backendTableName` to look up the actual table name in the database. These methods use the correct look up procedure:

```
new DictTable(TableNum(<sometable>)).name(DbBackend::Sql)
new DictField(TableNum(<sometable>),FieldNum(<someTable>,<somefield>)).name(DbBackend::Sql)
```

Adding Literals in Direct SQL

It is important for security, amongst other advantages, to pass parameters into the Direct SQL statement. For example, when creating Direct SQL code there are several scenarios where you will need to add literal values to the SQL statement. The most common examples are data area identification and empty date strings. These scenarios are handled by the following examples:

```
/* UPDATE PROJTRANSPosting
   SET EMPLITEMID = PET.EMPLID,
       CATEGORYID = PET.CATEGORYID,
       PROJTYPE = PT.TYPE,
       QTY = PET.QTY
   FROM PROJTRANSPosting PTP, PROJEMPLTRANS PET, PROJTABLE PT
   WHERE PTP.TRANSID = PET.TRANSID
        AND PTP.PROJTRANSTYPE = 2
        AND PET.PROJID = PT.PROJID
        AND PTP.DATAAREAID = N'xyz' AND PET.DATAAREAID = N'xyz' AND PT.DATAAREAID = N'xyz' */

sqlStmt = strfmt('UPDATE %1', #T(ProjTransPosting));
sqlStmt += strfmt(' SET %1 = %2, %3 = %4, %5 = %6, %7 = %8',
    #F(ProjTransPosting, EmplItemId), #AF(ProjEmplTrans, EmplId),
    #F(ProjTransPosting, CategoryId), #AF(ProjEmplTrans, CategoryId),
    #F(ProjTransPosting, ProjType), #AF(ProjTable, Type),
    #F(ProjTransPosting, Qty), #AF(ProjEmplTrans, Qty));
sqlStmt += strfmt(' FROM %1 %2, %3 %4, %5 %6',
    #T(ProjTransPosting), #A(ProjTransPosting),
    #T(ProjEmplTrans), #A(ProjEmplTrans),
    #T(ProjTable), #A(ProjTable));
sqlStmt += strfmt(' WHERE %1 = %2 AND %3 = %4 AND %5 = %6 AND %7 = %8 AND %9 = %10 AND %11 =
%12',
    #AF(ProjTransPosting, TransId), #AF(ProjEmplTrans, TransId),
    #AF(ProjTransPosting, ProjTransType), int2str(enum2int(ProjTransType::Hour)),
    #AF(ProjEmplTrans, ProjId), #AF(ProjTable, ProjId),
    #AF(ProjTransPosting, DataAreaId), sqlSystem.sqlLiteral(ProjTransPosting.DataAreaId),
    #AF(ProjEmplTrans, DataAreaId), sqlSystem.sqlLiteral(ProjEmplTrans.DataAreaId),
    #AF(ProjTable, DataAreaId), sqlSystem.sqlLiteral(ProjTable.DataAreaId));

/*
UPDATE SALESLINE
SET SHIPPINGDATEREQUESTED =
( SELECT MAX(DATEEXPECTED) FROM INVENTTRANS
WHERE INVENTTRANS.DATAAREAID = INVENTTRANS.DATAAREAID
```



```

AND SALESLINE.INVENTTRANSID = INVENTTRANS.INVENTTRANSID
AND INVENTTRANS.DATEEXPECTED <> '1900-01-01')
WHERE SHIPPINGDATEREQUESTED = '1900-01-01'
AND DATAAREAID = SALESLINE.DATAAREAID
AND EXISTS
( SELECT DATEEXPECTED
FROM INVENTTRANS
WHERE INVENTTRANS.DATAAREAID = N'ext'
AND SALESLINE.INVENTTRANSID = INVENTTRANS.INVENTTRANSID
AND INVENTTRANS.DATEEXPECTED <> '1900-01-01')
*/
    sqlStmt = 'UPDATE ' + dictTable SalesLine.name(DbBackend::Sql);
    sqlStmt += ' SET ' +
dictTable_SalesLine.fieldName(fieldnum(SalesLine, ShippingDateRequested), DbBackend::Sql);
    sqlStmt += ' = ( SELECT MAX(' +
dictTable_InventTrans.fieldName(fieldnum(InventTrans, DateExpected), DbBackend::Sql);
    sqlStmt += ') FROM ' + dictTable_InventTrans.name(DbBackend::Sql);
    sqlStmt += ' WHERE ' + dictTable_InventTrans.name(DbBackend::Sql);
    sqlStmt += '.' +
dictTable_InventTrans.fieldName(fieldnum(InventTrans, DataAreaId), DbBackend::Sql);
    sqlStmt += ' = ' + sqlSystem.sqlLiteral(inventTrans.DataAreaId);
    sqlStmt += ' AND ' + dictTable_SalesLine.name(DbBackend::Sql);
    sqlStmt += '.' +
dictTable_SalesLine.fieldName(fieldnum(SalesLine, InventTransId), DbBackend::Sql);
    sqlStmt += ' = ' + dictTable_InventTrans.name(DbBackend::Sql);
    sqlStmt += '.' +
dictTable_InventTrans.fieldName(fieldnum(InventTrans, InventTransId), DbBackend::Sql);
    sqlStmt += ' AND ' + dictTable_InventTrans.name(DbBackend::Sql);
    sqlStmt += '.' +
dictTable_InventTrans.fieldName(fieldnum(InventTrans, DateExpected), DbBackend::Sql);
    sqlStmt += ' <> ' + sqlSystem.sqlLiteral('1900-01-01') + ')';
    sqlStmt += ' WHERE ' +
dictTable_SalesLine.fieldName(fieldnum(SalesLine, ShippingDateRequested), DbBackend::Sql);
    sqlStmt += ' = ' + sqlSystem.sqlLiteral('1900-01-01');
    sqlStmt += ' AND ' +
dictTable_SalesLine.fieldName(fieldnum(SalesLine, DataAreaId), DbBackend::Sql);
    sqlStmt += ' = ' + sqlSystem.sqlLiteral(salesLine.DataAreaId);
    sqlStmt += ' AND EXISTS';
    sqlStmt += ' (SELECT ' +
dictTable_InventTrans.fieldName(fieldnum(InventTrans, DateExpected), DbBackend::Sql);
    sqlStmt += ' FROM ' + dictTable_InventTrans.name(DbBackend::Sql);
    sqlStmt += ' WHERE ' + dictTable_InventTrans.name(DbBackend::Sql);
    sqlStmt += '.' +
dictTable_InventTrans.fieldName(fieldnum(InventTrans, DataAreaId), DbBackend::Sql);
    sqlStmt += ' = ' + sqlSystem.sqlLiteral(inventTrans.DataAreaId);
    sqlStmt += ' AND ' + dictTable_SalesLine.name(DbBackend::Sql);
    sqlStmt += '.' +
dictTable_SalesLine.fieldName(fieldnum(SalesLine, InventTransId), DbBackend::Sql);
    sqlStmt += ' = ' + dictTable_InventTrans.name(DbBackend::Sql);
    sqlStmt += '.' +
dictTable_InventTrans.fieldName(fieldnum(InventTrans, InventTransId), DbBackend::Sql);
    sqlStmt += ' AND ' + dictTable_InventTrans.name(DbBackend::Sql);
    sqlStmt += '.' +
dictTable_InventTrans.fieldName(fieldnum(InventTrans, DateExpected), DbBackend::Sql);
    sqlStmt += ' <> ' + sqlSystem.sqlLiteral('1900-01-01') + ')';

```

Specifying DataAreaId in Where-Clauses

The DataAreaId to be used in a where-clause may not be equal to the current company code returned by curExt(). Therefore, curExt() should not be used to build the query string.

Because of the virtual company feature, it cannot be guaranteed that two tables in any join statement will fetch its data using the same DataAreaId. In this instance a Where clause should not use the following predicate: A.DATAAREAID = B.DATAAREAID.

The DataAreaId field should always be compared to a literal or a placeholder.

The following statement may not always work correctly:

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

```
DELETE FROM INVENTSUM
WHERE DATAAREAID=N'dmo' AND
EXISTS (SELECT 'x' FROM INVENTTABLE B
WHERE B.DATAAREAID=INVENTSUM.DATAAREAID
AND B.ITEMID=INVENTSUM.ITEMID AND B.ITEMTYPE=2)
```

The statement should always be written as follows:

```
DELETE FROM INVENTSUM
WHERE DATAAREAID=N'dmo' AND
EXISTS (SELECT 'x' FROM INVENTTABLE B
WHERE B.DATAAREAID=N'dmo'
AND B.ITEMID=INVENTSUM.ITEMID AND B.ITEMTYPE=2)
```

In the event that the InventTable is shared among several companies in the 'dmo' company, then the statement should be as follows, where the virtual company is assumed to be named 'vir':

```
DELETE FROM INVENTSUM
WHERE DATAAREAID=N'dmo' AND
EXISTS (SELECT 'x' FROM INVENTTABLE B
WHERE B.DATAAREAID=N'vir'
AND B.ITEMID=INVENTSUM.ITEMID AND B.ITEMTYPE=2)
```

To get the correct DataAreaId, declare a table buffer of the specific table type and use the value of the DataAreaId field in the table buffer.

To get the correct formatting with the '-s and the preceding N, parse the DataAreaId to the SqlSystem.sqlLiterals method and use the return value.

The following shows the use of DataAreaId and sqlLiteral:

```
static void UseDataAreaId(Args _args)
{
    InventSum    inventSum;
    InventTable  inventTable;
    str          sqlStr;
    SqlSystem    sqlSystem = new SqlSystem();
    ;
    sqlStr = sprintf(@"DELETE FROM INVENTSUM
                    WHERE DATAAREAID=%1 AND
                    EXISTS (SELECT 'x' FROM INVENTTABLE B
                    WHERE B.DATAAREAID=%2
                    AND B.ITEMID=INVENTSUM.ITEMID AND B.ITEMTYPE=2)",
                    sqlSystem.sqlLiteral(inventSum.dataAreaId),
                    sqlSystem.sqlLiteral(inventTable.dataAreaId));
}
```

Notes:

- 1) The example above assumes that DataAreaId is left justified, which is a valid assumption as it is a system field where the justification cannot be changed by the customers or partners.
- 2) The example is only used for demonstrating the use of DataAreaId. The table names and fields should be retrieved from the dict classes and the statement should be built using name(DbBackend::Sql).

Determining Whether a Table or Field Exists in the Database

You can test whether a table exists in the database by using the isTmp() method on the table buffer as follows:

```
static void TestTable(Args _args)
{
```

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

```

SalesTable salesTable;
;
if (!salesTable.isTmp()) // remember the NOT operator
{
    // table exists in the database.
    // isTmp will return true if the table is
    // specifically marked as temporary or if it is
    // disabled by the configuration key.
}
}

```

You can test whether a field exists in the database by testing its configuration key as follows:

```

static void TestField(Args _args)
{
    DictField dictField;
;
    dictField = new DictField(tableNum(SalesTable),
                             fieldNum(SalesTable, PriceGroupId));
    if (isConfigurationKeyEnabled(dictField.configurationKeyId()))
    {
        // Field exists in the database
    }
}

```

There is no need to test every field. If you know the field is always in the database because the table is in the database, then there is no need to test each field individually. You only need to test fields that have a different configuration key to the table.

Defining String Lengths

When writing Direct SQL or stored procedures, it cannot be assumed that a given string field has the currently defined length as it may have been changed by the user before the execution of the upgrade job.

A variable designed to hold an ItemId cannot be defined as NVARCHAR(20) even though the current maximum length for the ItemId data type is 20. It may have been changed to a higher value, and, consequently, the variable cannot hold the entire value for all items. The length of the variable should therefore be defined taking the length of the type at execution time into consideration.

The current maximum length of a field should be retrieved through the [ReleaseUpdateDB::fieldStringSize](#) method.

Applying LTrim for String Comparisons in the WHERE Clause

In X++, left and right justification is managed by the kernel using string comparisons in the WHERE clause. Microsoft Dynamics AX 4.0 is left justified when installed, so there is no need to handle compares within Microsoft shipped upgrade scripts. If customers use mixed-mode, then, in Direct SQL, the code needs to check the justification of the two sides of the comparison and apply LTRIM on the right justified side if the two sides have different justification properties.

The new static method fields2WhereClause() is created in [ReleaseUpdateDB](#) class. It returns a string to be used in a Direct SQL WHERE clause.

Implementing Complex Inserts and Updates in Direct SQL

Complex updates cannot be implemented directly in X++. When these conditions are encountered, the update operations must be rewritten in Direct SQL.

If the method being examined involves one or a small number of update operations, the SQL can be constructed as a string and executed as described in [Executing Direct SQL from X++](#) in this document.

For more complex methods that operate on multiple tables, it is advisable that the method be rewritten as a stored procedure. The stored procedure can be executed via X++ as described in [Stored Procedure and Function Guidelines](#) in this document.

Creating Stored Procedures and Functions

If stored procedures are needed in order to implement direct Transact-SQL logic, it may be created during execution time, executed, and then dropped after the upgrade script has run.

The AOS account has the privilege to create a stored procedure but it does not have execute permission on all stored procedures or functions. In order for your upgrade script to have the permission to execute the stored procedure or function you created, you need to prefix the object with the schema that the AOS account owns, and always use the two part name:

[schema name].[object name]

in the create, execute, and drop statements.

To get the correct schema name, use the utility function:

ReleaseUpdateDB::getSchemaName().

Example:

```
void createDimHistory_PurchInvoice_DSQl()
{
    InventReportDimHistory dimHistory;
    VendInvoiceTrans vendInvoiceTrans;
    InventTrans inventTrans;
    SqlSystem sqlSystem = new SqlSystem();
    SqlStatementExecutePermission sqlStatementExecutePermission;
    str str_ExecSproc;
    str str_SQLEXEC = 'EXEC [%1].%2 %3';

    void runOraCode()
    {
        while select vendInvoiceTrans
        exists join inventTrans
            where inventTrans.InventTransId == vendInvoiceTrans.InventTransId
            && inventTrans.InvoiceId == vendInvoiceTrans.InvoiceId
        notexists join dimHistory
            where dimHistory.InventTransId == vendInvoiceTrans.InventTransId
            && dimHistory.TransRefId == vendInvoiceTrans.InvoiceId
            && dimHistory.TransactionLogType ==
    InventReportDimHistoryLogType::PurchInvoice
    {
        InventReportDimHistory::addFromVendInvoiceTrans(vendInvoiceTrans);
    }
    }

    if (dimHistory.isTmp() || inventTrans.isTmp() || vendInvoiceTrans.isTmp())
        return;

    select firstonly RecId from vendInvoiceTrans;
```

```

if (!vendInvoiceTrans.RecId)
    return;

switch (SqlSystem::databaseBackendId())
{
    case DatabaseId::Oracle:
        runOraCode();
        break;
    case DatabaseId::MS_Sql_Server:
        str_ExecSproc = strfmt(str_SQLEXEC, ReleaseUpdateDB::getSchemaName(),
            #CREATEDIMHISTORY_PURCHINVOICE

, sqlSystem.sqlLiteral(vendInvoiceTrans.DataAreaId));
        sqlStatementExecutePermission = new
SqlStatementExecutePermission(str_ExecSproc);
        sqlStatementExecutePermission.assert();
        ReleaseUpdateDB::statementExeUpdate(str_ExecSproc);
        CodeAccessPermission::revertAssert();
}

```

When writing stored procedures that replace X++ methods or functions in the upgrade class, use the following guidelines:

- 1) The stored procedure name should be the same as the method or function that it is replacing.
- 2) The stored procedure should include the original X++ statements as comments to provide context during testing and troubleshooting.
- 3) Transactional control statements (BEGIN TRANSACTION, COMMIT) should not be coded in the stored procedure. Transaction management is implemented in X++.
- 4) The stored procedure must accept a required parameter of DATAAREAID as data type NVARCHAR(3).
- 5) If the stored procedure will be populating a table with a formatted business sequence column (described in [Assigning Business Sequences on Insert](#) section of this document), the procedure must accept the following parameters:
 - 1) @NUMBERSEQUENCE NVARCHAR(20). This will be used as a key to the NUMBERSEQUENCE table to retrieve the next key value and format requirements.
 - 2) @RJUSTIFY CHAR(1). If "Y", this indicates the column is to be right justified.

Implementing Set-Based Updates with Joins

Update operations that involve true joins (in contrast to exists joins) cannot be directly implemented in X++ and represent one case where a Transact-SQL rewrite is needed. The following code is an example of an update that derives data from another table:

```

while select forupdate salesLine
where salesLine.ShippingDateRequested == dateNull()
join firstly maxof(DateExpected) from inventTrans
group by InventTransId
where inventTrans.InventTransId == salesLine.InventTransId &&
    inventTrans.DateExpected != dateNull()
{
    salesLine2 =
    SalesLine::findInventTransId(inventTrans.InventTransId, true);
    salesLine2.ShippingDateRequested = inventTrans.DateExpected;
    if (salesLine2)
        salesLine2.doUpdate();
}

```

The corresponding Transact-SQL update is written as follows:

```
UPDATE SALESLINE
SET SHIPPINGDATEREQUESTED =
(
  SELECT MAX(B1.DATEEXPECTED) FROM INVENTTRANS B1
  WHERE A.DATAAREAID = B1.DATAAREAID
  AND
    A.DATAAREAID = @dataareaid
  AND
    A.INVENTTRANSID = B1.INVENTTRANSID
  AND
    B1.DATEEXPECTED <> '1900-01-01'
  AND
    A.SHIPPINGDATEREQUESTED = '1900-01-01')
FROM SALESLINE A, INVENTTRANS B0
WHERE A.SHIPPINGDATEREQUESTED = '1900-01-01'
AND A.DATAAREAID = @dataareaid
AND A.INVENTTRANSID = B0.INVENTTRANSID
AND B0.DATEEXPECTED <> '1900-01-01'
```

Using Direct SQL for Set-Based Updates

The following code is an example of performing a set-based update using the `updateSalesAndTransLineDlvAddress`:

```
while select salesTable
{
  update_recordset salesLine
    setting deliveryAddress      = salesTable.DeliveryAddress,
           deliveryName        = salesTable.DeliveryName,
           deliveryStreet      = salesTable.DeliveryStreet,
           deliveryZipCode     = salesTable.DeliveryZipCode,
           deliveryCity        = salesTable.DeliveryCity,
           deliveryCounty      = salesTable.DeliveryCounty,
           deliveryState       = salesTable.DeliveryState,
           deliveryCountryRegionId = salesTable.DeliveryCountryRegionId
  where salesLine.SalesId      == salesTable.SalesId
     && salesLine.DeliveryAddress == '';

  //The journal lines must be updated for intrastat to function
  update_recordset custInvoiceTrans
    setting DlvCountryRegionId = salesTable.DeliveryCountryRegionId,
           DlvCounty          = salesTable.DeliveryCounty,
           DlvState           = salesTable.DeliveryState
  where custInvoiceTrans.SalesId      == salesTable.SalesId
     && custInvoiceTrans.DlvCountryRegionId == '';

  update_recordset custPackingSlipTrans
    setting DlvCountryRegionId = salesTable.DeliveryCountryRegionId,
           DlvCounty          = salesTable.DeliveryCounty,
           DlvState           = salesTable.DeliveryState
  where custPackingSlipTrans.SalesId      == salesTable.SalesId
     && custPackingSlipTrans.DlvCountryRegionId == '';
}
```

In this example, the code loops through every `SalesTable` Entry and:

- 1) Updates `SalesLine` with the relevant address information for the salesid.
- 2) Updates `CustInvoicetrans` with the address information for salesid.
- 3) Updates `custPackingSlipTrans` with the address information for salesid.

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

Direct SQL needs to be rewritten in this case because of the need to:

- 1) Perform one mass update where possible.
- 2) Reduce looping on a large transactional table such as salesline.

The following is the Transact-SQL code that you should generate from X++:

```
UPDATE      SALESLINE
SET  DELIVERYADDRESS      = T.DELIVERYADDRESS,
     DELIVERYNAME         = T.DELIVERYNAME,
     DELIVERYSTREET       = T.DELIVERYSTREET,
     DELIVERYZIPCODE      = T.DELIVERYZIPCODE,
     DELIVERYCITY         = T.DELIVERYCITY,
     DELIVERYCOUNTY     = T.DELIVERYCOUNTY,
     DELIVERYSTATE        = T.DELIVERYSTATE,
     DELIVERYCOUNTRYREGIONID = T.DELIVERYCOUNTRYREGIONID
FROM SALESLINE L,
     SALESTABLE T
WHERE      T.DATAAREAID = @DATAAREAID
AND L.DATAAREAID = T.DATAAREAID
AND L.SALESID = T.SALESID
AND L.DELIVERYADDRESS = ''

UPDATE      CUSTINVOICETRANS
SET  DLVCOUNTRYREGIONID   = T.DELIVERYCOUNTRYREGIONID,
     DLVCOUNTY            = T.DELIVERYCOUNTY,
     DLVSTATE             = T.DELIVERYSTATE
FROM CUSTINVOICETRANS C,
     SALESTABLE T
WHERE      T.DATAAREAID = @DATAAREAID
AND C.DATAAREAID = T.DATAAREAID
AND C.SALESID = T.SALESID
AND C.DLVCOUNTRYREGIONID = ''

UPDATE      CUSTPACKINGSLIPTRANS
SET  DLVCOUNTRYREGIONID = T.DELIVERYCOUNTRYREGIONID,
     DLVCOUNTY          = T.DELIVERYCOUNTY,
     DLVSTATE           = T.DELIVERYSTATE
FROM CUSTPACKINGSLIPTRANS C,
     SALESTABLE T
WHERE      T.DATAAREAID = @DATAAREAID
AND C.DATAAREAID = T.DATAAREAID
AND C.SALESID = T.SALESID
AND C.DLVCOUNTRYREGIONID = ''
```

The performance improvement achieved in this example is significant. On a database, Baseline ran for 24 minutes. With SET BASED CHANGE, it ran in 16 seconds.

This type of update, which does not require sequencing conditional to each record, can be written in X++ as a sequence of Direct SQL statements.

Using a Set-Based Insert Operation

There are a number of cases in the upgrade process where tables that are new in Microsoft Dynamics AX 4.0 must be populated from one or more tables. If the volume of data to be processed in these tables is large, and if INSERT_RECORDEDSET does not achieve the desired performance, then using a set-based insert operation is required.

Example Transact-SQL set-based inserts are written as:

```
INSERT INTO SOME_NEW_TABLE (column-list)
SELECT column-list FROM SOME_OLD_TABLE WHERE criteria
```

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

System Sequence Considerations

A complicating factor when we use a Direct SQL set-based insert into a table in the Microsoft Dynamics AX database is that tables have one or more sequentially assigned numbers which are derived from the SYSTEMSEQUENCES and NUMBERSEQUENCETABLE tables.

A two-step process of initially populating a temporary table that uses a DBMS-specific sequence mechanism (IDENTITY for Transact-SQL, ROW NUMBER for Oracle) and then copying the temporary table's rows to the final permanent table is required.

The two sections that follow provide Transact-SQL examples of populating both a system sequence (RECID) and business sequence.

RECID in Microsoft Dynamics AX 2012

The RECID allocation algorithm has undergone significant changes in Microsoft Dynamics AX 2012. A RECID can be allocated in two different ways:

- 1) Kernel automatically allocates the RECID during insert and INSERT_RECORDSET
- 2) User manually chooses to allocate the RECID

In the case of upgrade, we are concerned about #2. This section will document the allocation APIs, the usage and some patterns. The document does not dwell in the allocation algorithm itself.

Manually allocating RECID

There are cases where you want to allocate the RECID manually in your script. The following are some of the scenarios:

- 1) You are trying to do a bulk insert manually. There are cases where row by row insert is not sufficient and you want to do a bulk insert. Import/Export code is an example of this usage pattern. In such a case, you need to allocate the RECID manually.
- 2) An upgrade script uses direct SQL to insert data. In this usage pattern, you need to allocate RECID manually.
- 3) Upgrade script was optimized to use RecordInsertList instead of row by row insert. But, cross references need to be set up on another table (for example REFRECID). In such a case, allocate the RECID upfront for the record so that cross references can be patched up.

In all the above scenarios, the allocation is done the same way, using the RECID allocation APIs. There are three APIs that you need to know about:

RECID suspension - suspendRecids

RECID reservation - reserveValues

RECID releasing suspension - removeRecidSuspension

The APIs are members of the SystemSequence class.

The following is a code snippet of how to use the allocation APIs.

```
static void Job2(Args _args)
{
    SystemSequence s;
    AAMyTable t;
    int64 startValue;
```

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012


```

int i;
;
s = new SystemSequence();
s.suspendRecIds( tablenum (AAMyTable) );
startValue = s.reserveValues( 10, tablenum( AAMyTable ) );

for ( i = 0; i <10; i++ )
{
    t.IntFld = i;
    t.RecId = startValue + i;
    t.insert();
}

s.removeRecIdSuspension( tablenum ( AAMyTable ) );
}

```

Create a new instance of the systemSequence class

Suspend the RECID allocation by the kernel

Reserve the RECID by passing in the number of id's to reserve. The return value is the starting value of the range you reserved. The API gaurantees that the allocated id's are contiguous.

Assign the RECID to the RECID column

Remove the suspension

Tips on using the RECID allocation API:

- 1) Once you suspend the RECID allocation for that table, the kernel will not dispense any more RECID's for that table on that session.
- 2) The ReserveValues API will guarantee contiguity of the RECID range that is being reserved.
- 3) If you try to insert an id that has not been reserved, then kernel will raise an exception.
- 4) If you are trying to assign a RECID without suspending, kernel will raise an exception.
- 5) If you do not remove the suspension after using the reservation API's, the suspension remains until the end of your session.

Assigning RECID on INSERT

RECID is a continuously ascending key value for each table in the Microsoft Dynamics AX schema. It is derived from table SYSTEMSEQUENCES which keeps the next available key value (NEXTVAL) for each table by that table's Table ID.

Note that the SystemSequences table may be empty if the table is new and no records have been inserted. Please refer to the ReleaseUpdateDB39_Cust.createDimHistorySprocs(), which provides an example of the solution for that problem: it checks if a RECID existed and if not, inserting and deleting a record to get the RECID's started.

In Microsoft Dynamics AX 4.0, RECID is a 64-bit integer column; this data type is implemented in SQL Server as BIGINT.

The abbreviated example below illustrates using SYSTEMSEQUENCES and a temporary table using IDENTITY for sequential numbers:

```

CREATE      PROCEDURE initFromSMMQuotationTable
            @DATAAREAID NVARCHAR(3)
AS
DECLARE    @NEXTVAL      BIGINT,
            @ROWCOUNT    BIGINT

SELECT     ..... ,
            RECID        = IDENTITY(BIGINT,1,1) AS QUOTATIONID
INTO       #TEMP
FROM       DEL_SMMQUOTATIONTABLE
WHERE      QUOTATIONSTATUS = 0 -- SMMQUOTATIONSTATUS::INPROCESS

SELECT @NEXTVAL=NEXTVAL
FROM SYSTEMSEQUENCES (UPDLOCK, HOLDLOCK)
WHERE ID = -1
AND TABID = 1967

INSERT INTO SALESQUOTATIONTABLE
(column-list)
SELECT     ..... ,
            RECID        = QUOTATIONID+@NEXTVAL
FROM       #TEMP

SELECT @ROWCOUNT = COUNT(*) FROM #TEMP

UPDATE SYSTEMSEQUENCES
SET NEXTVAL=NEXTVAL + @ROWCOUNT
WHERE ID = -1
AND TABID = 1967
GO

```

Assign an IDENTITY column with a starting value of 0 incremented by 1

Retrieve the next value for RECID for this table (by TABID)

When we insert into the permanent table, we add the temporary table's IDENTITY column to the next value retrieved from SYSTEMSEQUENCES

We update SYSTEMSEQUENCES to reflect the number of rows that we have added to this table

Looking Up Table ID and Field IDs

If you are getting TABID in the stored procedure, you should perform the fetch from the SQL Dictionary.

Assigning Business Sequences on Insert

Business sequences are a more complex problem to solve with Direct SQL; not only is the number sequentially assigned from a table (NUMBERSEQUENCETABLE), but you also have to consider the following factors:

- 1) The specific number sequence to be used for a specific column.
- 2) Whether the column is to be left or right justified.
- 3) The customer's specific formatting requirements (FORMAT) for the column.

The first two factors are accessible in X++ and, as described in the stored procedure guidelines above, must be passed as parameters to any stored procedure which must populate a formatted business sequence number.

Once the specific numbersequence to be used is known, the formatting requirement must be retrieved from FORMAT column of the NUMBERSEQUENCETABLE table.

Notes:

- 1) The stored procedure is passed an indicator that specifies if right justification is to take place. A value of "Y" means right-justify the column. The default is to left-justify the column.

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

- 2) Because formatted sequence columns are of different maximum lengths, you must look up the length of the column that is to be formatted and record the length in your procedure. The instructions that follow will describe how you pass the column's length, along with the formatting requirements, to a user-defined SQL function that will format the column correctly.

The example below illustrates the use of a user-defined function FN_FMT_NUMBERSEQUENCE which accomplishes the formatting and justification requirements of a business sequence column:

```

CREATE      PROCEDURE initFromSMMQuotationTable
            @DATAAREAID NVARCHAR(3),
            @NUMBERSEQUENCE NVARCHAR(20),
            @RJUSTIFY CHAR(1)

AS
DECLARE    @NEXTREC      BIGINT,
            @FORMAT      NVARCHAR(40),
            @ROWCOUNT    BIGINT
            @RJUSTIFY_LENGTH INT

IF RJUSTIFY = 'Y'
    SET @RJUSTIFY_LENGTH = 40
ELSE
    SET @RJUSTIFY_LENGTH = 0

SELECT QUOTATIONID = IDENTITY(BIGINT,1,1),
       .....
INTO #TEMP
FROM DEL_SMMQUOTATIONTABLE
WHERE   QUOTATIONSTATUS = 0 -- SMMQUOTATIONSTATUS::INPROCESS

SELECT @NEXTREC = NEXTREC, @FORMAT=FORMAT
FROM NUMBERSEQUENCETABLE (UPDLOCK, HOLDLOCK)
WHERE   DATAAREAID = @DATAAREAID
AND     NUMBERSEQUENCE = @NUMBERSEQUENCE

INSERT INTO SALESQUOTATIONTABLE
(column-list )
SELECT
    DBO.FN_FMT_NUMBERSEQUENCE(@FORMAT,QUOTATIONID,@NEXTREC, @RJUSTIFY_LENGTH) ,
    .....

FROM #TEMP

SELECT @ROWCOUNT = COUNT(*) FROM #TEMP

UPDATE NUMBERSEQUENCETABLE
SET NEXTREC = NEXTREC+@ROWCOUNT
WHERE   DATAAREAID = @DATAAREAID@NUMBERSEQUENCE
AND     NUMBERSEQUENCE = @NUMBERSEQUENCE

```

You must determine the column's length if it is to be right justified and set a variable so we can pass that to the formatting function

As in the previous example, we create an IDENTITY column in the temporary table with initial value of 0

Retrieve the next value from NUMBERSEQUENCETABLE using the NUMBERSEQUENCE key supplied

Details on calling this function follow

Update NUMBERSEQUENCE to reflect the number of rows added to the table

In many cases it will be necessary to assign a sequential number both for RECID and a business sequence column. However, SQL Server only permits one IDENTITY column per table.

The following example demonstrates how to use the single IDENTITY column for both purposes. This example is also useful as a template for creating new procedures to upgrade data into new tables in the Microsoft Dynamics AX 4.0 schema:

How to: Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

```

CREATE      PROCEDURE initFromSMMQuotationTable
            @DATAAREAID NVARCHAR(3),
            @NUMBERSEQUENCE NVARCHAR(20),
            @RJUSTIFY CHAR(1) ='N'

AS

DECLARE    @NEXTREC          BIGINT,
            @NEXTVAL         BIGINT,
            @FORMAT          NVARCHAR(40),
            @ROWCOUNT       BIGINT
            @RJUSTIFY_LENGTH INT

-- Set the length of the column that is to be right-justified
-- Confirm length in table definition
IF RJUSTIFY = 'Y'
    SET @RJUSTIFY_LENGTH = 40
ELSE
    SET @RJUSTIFY_LENGTH = 0

-- The SELECT INTO creates a temp table
-- RECID is assigned during the insert and given
-- a sequentially ascending number starting with 0
SELECT QUOTATIONID = ''
        .....
        RECID = IDENTITY(BIGINT,1,1),
INTO #TEMP
FROM DEL_SMMQUOTATIONTABLE
WHERE     QUOTATIONSTATUS = 0 -- SMMQUOTATIONSTATUS::INPROCESS

-- Retrieve next key value for RECID
-- Note TABID; you need to determine the
-- value here from table SQLDICTIONARY
SELECT @NEXTVAL=NEXTVAL
FROM SYSTEMSEQUENCES (UPDLOCK, HOLDLOCK)
WHERE ID = -1 AND TABID = 1967

-- Retrieve next key value for business sequence (QUOTATIONID)
-- NUMBERSEQUENCE is supplied in X++ and passed in @NUMBERSEQUENCE
SELECT @NEXTREC = NEXTREC, @FORMAT=FORMAT
FROM NUMBERSEQUENCETABLE (UPDLOCK, HOLDLOCK)
WHERE     DATAAREAID = @DATAAREAID AND NUMBERSEQUENCE = @NUMBERSEQUENCE

-- Insert from temp table to final table. Note that temp table RECID
--is sued to supply values to both QUOTATIONID and REID in final table
INSERT INTO SALESQUOTATIONTABLE
(column-list )
SELECT
    DBO.FN_FMT_NUMBERSEQUENCE(@FORMAT, RECID,@NEXTREC, @RJUSTIFY_LENGTH) ,
    ..... ,
    RECID+@NEXTVAL
FROM #TEMP

-- Row count of temp table then used to update both NUMBERSEQUENCETABLE
-- and SYSTEMSEQUENCES tables
SELECT @ROWCOUNT = COUNT(*) FROM #TEMP

UPDATE NUMBERSEQUENCETABLE      SET NEXTREC = NEXTREC+@ROWCOUNT
WHERE     DATAAREAID = @DATAAREAIDAND AND NUMBERSEQUENCE = @NUMBERSEQUENCE

UPDATE SYSTEMSEQUENCES      SET NEXTVAL=NEXTVAL + @ROWCOUNT
WHERE ID = -1 AND TABID = 1967

```

Calling FN_FMT_NUMBERSEQUENCE

A user defined function FN_FMT_NUMBERSEQUENCE is provided to assist with the formatting requirements of a business sequence column. This function enables the following operations to be performed:

- 1) Adds the value of the IDENTITY column to the NEXTREC value retrieved from NUMBERSEQUENCETABLE.
- 2) Formats the result according to the FORMAT column retrieved from NUMBERSEQUENCETABLE.
- 3) Right justifies the formatted column to the length specified. If the function encounters a value of 0, no justification occurs and the formatted value remains left justified by default.

The parameters that are supplied to FN_FMT_NUMBERSEQUENCE are:

- 4) The FORMAT column value from NUMBERSEQUENCETABLE.
- 5) The integer value to be formatted.
- 6) The value from NEXTREC in NUMBERSEQUENCETABLE. If this is not supplied, it is set to 0 by default.
 - The length of the column to be right justified. If this is not supplied it is set to 0 by default. If 0 is specified or becomes the default, then no justification occurs.

The ReleaseUpdateDB38_Basic::createFnFmtNumberSequence method creates the FN_FMT_NUBMERSEQUENCE function. If your script needs to call the function, you should make the script depend on the ReleaseUpdateDB38_Basic::createFnFmtNumberSequence script and then you can reference the function in your Direct SQL code.

Appendix 2: INTERNAL ONLY.

Adding Upgrade Scripts to the Upgrade Preprocessing Framework in Source Depot

The Upgrade Preprocessing framework consists of two main parts: the Framework itself, which provides the base functionality for upgrade; and application team business logic that performs preprocessing specific to a business module. During build, these elements are combined into a single XPO and dropped on the CD. Below are the steps to follow when an application team needs to add scripts to the Upgrade Preprocessing Framework (note that these steps are the same for AX4, substituting appropriate file names):

1. In an AX5 environment, import the base preprocessing framework XPO from %SDROOT%\source\Application\UpgradePreProcessingAX5**SharedProject_Ax50PreUpgradeFramework.xpo** and **SharedProject_Ax50PreUpgradeFrameworkShare.xpo**. **If your scripts will depend on other projects then you need to import them as well.**
2. Create upgrade business logic as described in the section ‘*The Data Upgrade Framework for the Source Environment*’ and add to a new private project.
3. Once coding is complete, export the USR layer without IDs of the private XPO created in Step 2 and add it to Source Depot in %SDROOT%\source\Application\UpgradePreProcessingAX5.
4. Edit the file %SDROOT%\source\Application\UpgradePreProcessingAX5**XPOFileList.txt**, adding the private XPO to the list.
5. Within a CoreXT window, navigate to %SDROOT%\source\Application\UpgradePreProcessingAX5 and type ‘BUILD RETAIL ...’ to test that the XPO is properly combined.
 - 5.1. WARNING: Executing ‘BUILD RETAIL ...’ in any other folder may uninstall the AX client on the machine.
6. Verify that the resulting XPO appears on the CD and contains the newly created business logic:
 - 6.1. %SDROOT%\drop\retail\CD\DatabaseUpgrade**UpgradeAX5.xpo**.
7. Finally, verify that the resulting XPO will successfully import and compile by importing it into a **clean** AX5 environment.

Appendix 3: Debugging batch jobs.

Debugging upgrade batch jobs in Microsoft Dynamics AX 2012:

Batch jobs in Microsoft Dynamics AX 2012 run as IL code by default, please follow this link to see how to debug IL code in Microsoft Dynamics AX 2012.

Note : When you add or change an upgrade script in the batch, you must do an X++ IL incremental build. Right-click on the root node of the AOT, point to Add-Ins, and then click on X++ IL incremental build.

For some reason, you need to turn of IL so scripts can run as X++ code then follow these steps:

- Run regedit.exe and create a **string** value **ilbatch** under
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\Dynamics Server\6.0\01\Original
(installed configuration)
- Set the value to 0.
- Restart the AOS.

Deugging upgrade batch jobs in AX4/5:

- Edit the method runImpersonated of your batch job class to always return True. For upgrade scripts, the class is ReleaseUpdateExecute.
- In AX5 this fix may be needed: <http://hotfix/search.aspx?search=962952>