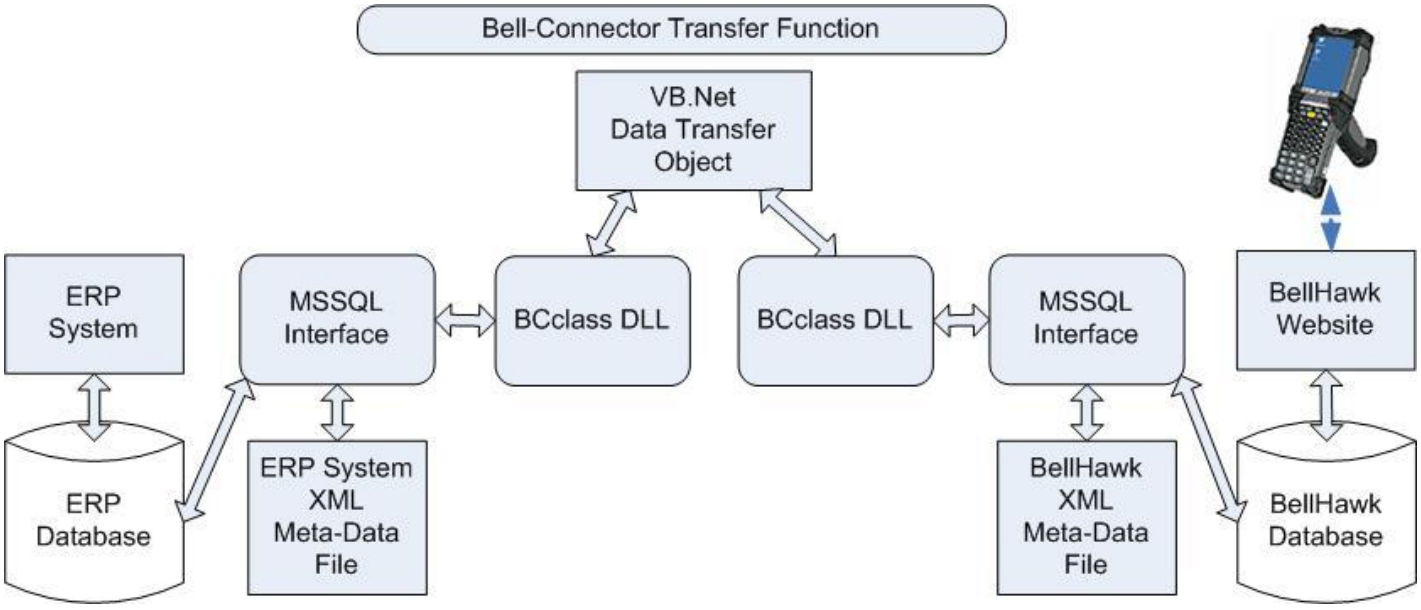
MilramX Programmers Manual

Introduction



This manual describes how to develop a MilramX (BC) transfer function using the Visual Studio VB.Net IDE. In this manual we will use the example of exporting data from a Sage 300 ERP system to BellHawk.

The intended audience for this manual is software developers who know how to develop code in VB.Net using the Microsoft Visual Studio IDE and who are familiar with Microsoft Windows operating systems and SQL Server databases. It also assumes familiarity with the ERP or other system with which data is to be exchanged.

In this manual we use the example of transferring data between an ERP system and the BellHawk database using ODBC connections to their databases. This is a simple example of the use of the MilramX Framework and can lay the foundation for implementing more complex interfaces, such as exchanging data with multiple systems using a mix of communications protocols or the use of the RTX extensions to MilramX to perform real-time monitoring of systems and the generation of Alerts.

We use VB.Net for implementing the DTOs (Data Transfer Objects), of which the BC Transfer Function is comprised. This is because the MilramX is intended for use by IT analyst/programmers and manufacturing engineers who are adequate but not expert .Net programmers. While still complex, the MilramX framework hides all the really complicated interfacing issues, enabling programmers to focus on the business logic of translating High Level Data Objects (HLDOs) from one system to another.

Because of the generality of the .Net platform, the DTOs can probably be programmed in C#, which is typically preferred by expert .Net programmers. Please note, however, that the use of the MilramX class libraries from C# code is not guaranteed to work as this is not supported as

part of our standard MilramX Framework test and development regimen, which is targeted at VB.Net developers.

Prerequisites

This manual assumes that the user has already gone though the following steps.

1. Installed a test version of the BellHawk Website and Database to be the recipient of the transfers. In this case we will be using the BC MSSQL ODBC Interface to exchange data with the BellHawk SQL Server database. Please see the "BellHawk Configuration and Installation Guide" for details. Also please see "Installing both Test and Production copies of BellHawk, TAG and MilramX". These and other manuals referenced here are available in the User Manuals section of www.BellHawk.com under the Support Tab.

2. Installed a test version of Sage 300 (or whatever ERP system you are using). In this case we will be using the BC MSSQL ODBC Interface to exchange data with a Sage 300 SQL Server database. Note that other BC Interface adaptors are available for other databases and systems ranging from QuickBooks to Oracle. Please see the user manual "MilramX Systems Architecture".

3. Created XML Metadata files for both systems for the data to be exchanged, using the BC Metadata Editor and the BC Website's DEXEL functionality as described in the "MilramX High-Level Data Object User Manual". Note that it is normally possible to use same XML Metadata file for BellHawk as is used by the BellHawk website itself (currently BHMeta_v6.60.XML) but sometimes a custom version is necessary to access fields or tables not defined in the standard BellHawk Metadata.

4. Installed a test version of the BC Website and Control database. Also setup the initialization (.ini file) as described in the user manual "MilramX Installation and Configuration Guide". This includes installing the XML Metadata files for both the source and target systems and the corresponding BC Adaptors (in this example both use the BC MSSQL Interface). This is so that the BC Website DEXEL functionality can be used to test that the specified data objects can correctly read and write data objects using the XML Metadata files, which is an important step in the development process.

5. Made sure that your development computer can access the test BC Control database, the test BellHawk database and the test ERP system database over a LAN or VLAN (so it is accessible via ODBC). This may require enabling firewall transmission for port 1433 for SQL server on the server computers and installing ODBC clients on your development computer if you are using other databases than Microsoft SQL Server. I generally run the BC test Website on my local development computer and the test BellHawk and the ERP systems remotely on other computers. This enables me to easily get at the Metadata files and the ini files from the BC Website's folder and to copy these into my transfer function development folder.

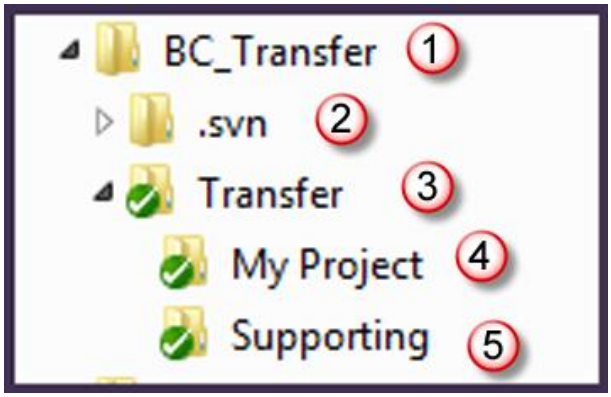
6. Installed a copy of Visual Studio on their development computer. In this example, we are running VS2010 on a Windows 7 Pro computer but MilramX it will work with later VS versions. If developing in VB.Net, it is usually beneficial to set this as the default language for Visual Studio.

Prototype Transfer Function Distribution

Folders and Files

The software supplied with the MilramX Developer's kit includes a prototype solution for implementing a transfer function. This can be used as the basis for implementing an actual transfer function by renaming and expanding the prototype or copying selected parts from this in creating a new Console application solution or project.

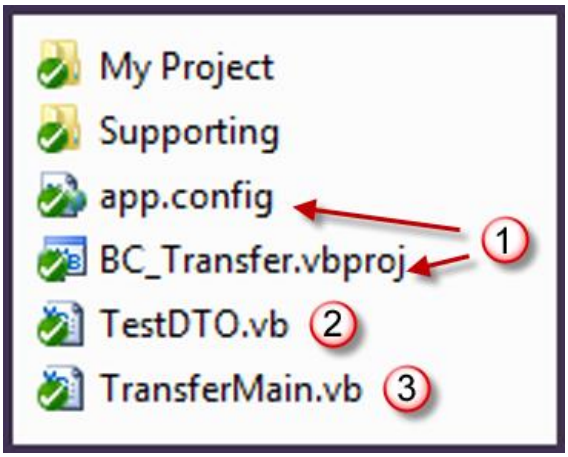
The distribution consists of a number of folders as shown below



These folders are as follows:

1. The distributed solution is named BC-Transfer (1). It contains the folders shown here plus the file BC_Transfer.sln which contains the definition information for the solution.
2. We use Tortoise SVN (highly recommended) as part of our source control and repository mechanism – SVN files (2) are not included in distribution – but screen shots included in this manual show Tortoise SVN symbology and the control files it adds.
3. The project within the solution is named Transfer (3) by default. It contains the VB.Net source files for the project.
4. The My Project folder (4) contains the files created by Visual Studio for the project
5. The Supporting folder (5) contains DLLs (dynamic link libraries).

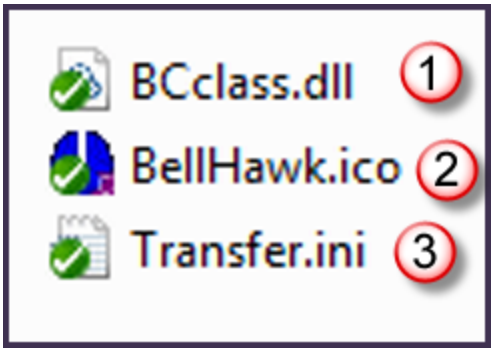
Within the Transfer folder are several files as shown here:



These are

1. The files marked (1) are generated by Visual studio.
2. TestDTO.vb (2) is a sample data transfer object
3. TransferMain.vb (3) is the provided main program for the project.

Within the Supporting folder are the files shown here:



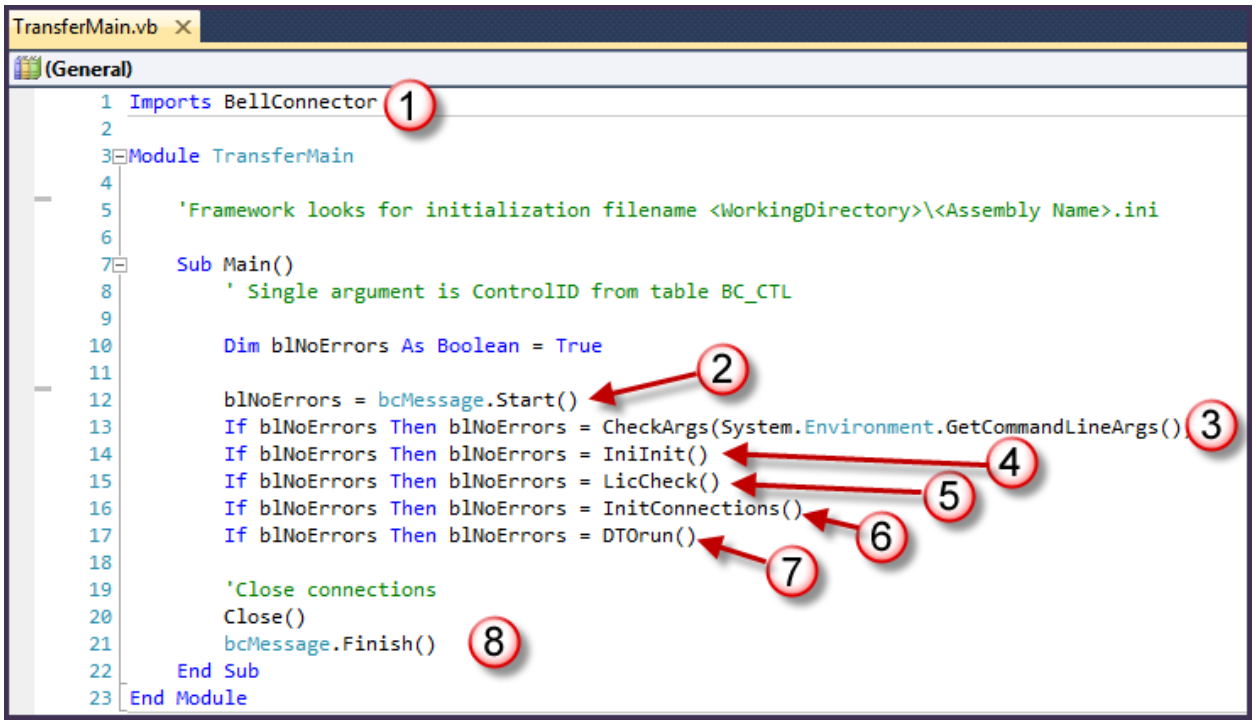
These are:

1. The BCclass DLL (1). This contains all the classes and methods that do all the "heavy lifting" within MilramX.
2. An icon (2) for MilramX
3. A prototype initialization file (3) to be copied into the main project folder and edited to setup the linkages to the databases that the transfer function will use.

The project can be opened in Visual Studio by running the BC_Transfer.sln file.

Main Program

The main program is standard and does not need modifying, except possibly by changing the module name.



Items of note in the Main program:

1. It uses the BellConnector namespace (1) which gives it implicit access to all the BCclass functions.
2. The bcMessage class (2) is used to write status information into the BC Control database so that the DTO status shows as “Running” in the BC website.
3. CheckArgs (3) is used to get the argument with which the transfer function is called and to make sure that it is the ControlID for a legitimate DTO entry in the BC_CTL table in the BC Control database. This is how the BC Framework knows what DTO to run and with which parameters.
4. IniInit (4) is called to get the needed setup data for Adaptors and other setup data from the initialization file. It is expecting to find the ini file in the working directory, which for development is the same folder as the main program. It is expecting the transfer function to have the same name as the Assembly, in this case Transfer.ini.
5. LicCheck (5) checks that the BC software is running on the licensed computer and that the licensed number of adaptors are not exceeded. Gives polite warning messages. Failure to call this will result in system aborting with cryptic error message when the DTOs are run.

6. InitConnections (6) uses the Adaptor information from the ini file to establish connections to other databases and systems.

7. DTORun (7) initiates the DTO specified through the call sequence and runs it. Note DTOs are run by name so the DTO name specified in the BC Web interface needs to correspond to a DTO included as described below in the project.

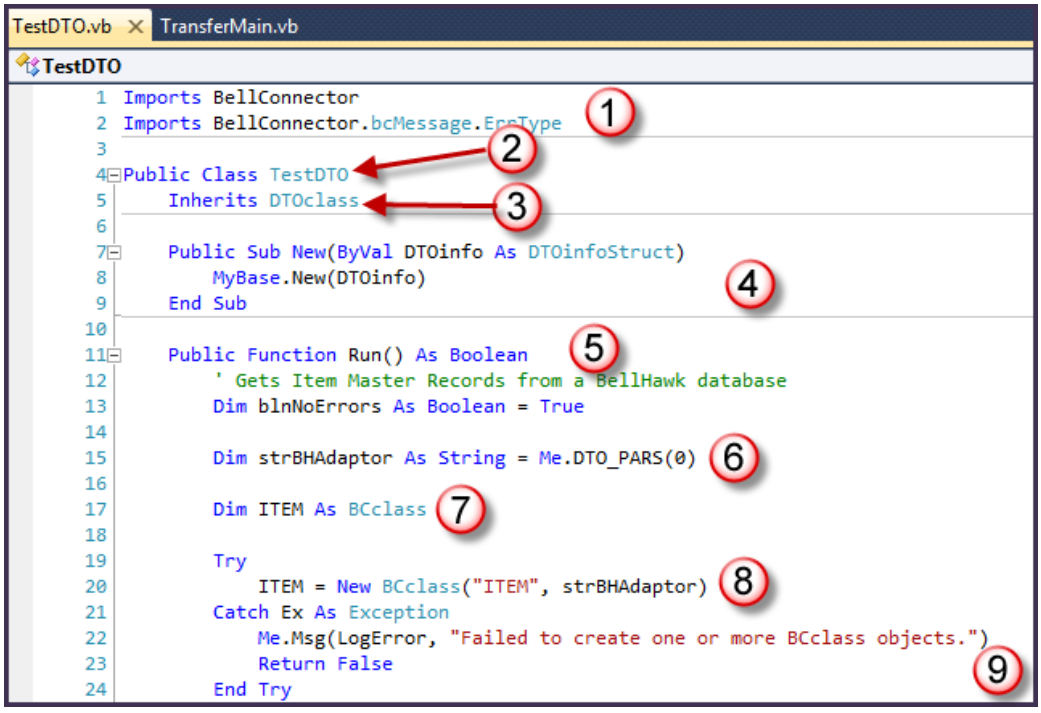
8. bcMessage (8) is used to write data to the BC Control database to indicate that the transfer functions has stopped running so this is visible through the website.

Any of the initialization steps can result in an error due to setup issues. The error messages are written in the daily log file, may be displayed on the Console (depending on the setting of DEBUG in the ini file) and will be written into the BC Control database so they are visible through the BC website interface.

The initializations are done step by step and are aborted if any one step is found to be in error. In early stages of development it is usually beneficial to step through each of these initialization functions using the debugger.

Sample DTO

This sample DTO reads from the BellHawk Item master table and outputs some of the contents as informational messages to the log file and possibly elsewhere depending on the value of DEBUG in the ini file. The initialization portion of a DTO is shown here



Things of note:

1. It imports the BellConnector namespace (1) to give it access to the BCclass methods. It also imports the bcMessage ErrType enumerations of LogError, LogWarning and LogInfo.

2. The name of the class (2) for each DTO, in this case “TestDTO” has to match the name used for the DTO in the BC Web interface. This is how it knows which DTO to run based on the name specified in the ControlID row of the BC_CTL table in the BC Control database. A developer does not have to code any specific linkage to the DTO, they simply include it in their transfer function project and it will be called by refelection.

3. Each DTO inherits methods and properties from DTOclass (3).

4. This is the standard code (4) for creating a New instance of the DTO.

5. The Run method (5) is where the programmer inserts code for the DTO.

6. This is where we retrieve the name of the Adaptor, such as “BellHawk” from the DTO_PARS array. The name of the Adaptor needs to match an entry in the .ini file. DTO_PARS contains the parameters for the DTO setup in the BC Website interface and written into the BC_CTL database table. By convention, the first entry is the name of the source adaptor and the second entry is the name of the destination adaptor, but this is application specific.

7. Here (7) we declare a High Level Data Object (HLDO) named ITEM. HLDOs properties include a keyword, such as “ITEM” and an array of parameter name:value pairs as defined in the XML Metadata for the HLDO.

8. Here (8) is where we form the association between the Keyword “ITEM” and the adaptor, in this case “BellHawk”. This is where the HLDO ITEM is associated with the Item Master database table in the BellHawk database through the XML Metadata and the connection string data in the [BellHawk] adaptor section of the ini file. This enables the programmer to ignore the details of the database table and simply focus on the HLDO.

9. An attempt to instantiate an HLDO can fail and raise an exception, so putting code inside Try-Catch block is a good idea. Here (9) we see the use of the Msg method of the DTOclass to output an error message.

Once we have initialized an HLDO then we can use it, as shown below:

```
25
26          blnNoErrors = True
27          Try
28              Dim strTestOutput As String
29              Dim nrecs As Int16 = ITEM.Fetch("All")  (1)
30
31              If nrecs > 0 Then  (2)
32                  strTestOutput = String.Format("{0} | {1} | {2} | {3}", "ItemNumber", "Category", "UDP", "ItemDescription")  (3)
33                  Me.Msg(LogInfo, strTestOutput)
34              End If
35              For intIndex As Int16 = 0 To (nrecs - 1)  (4)
36                  ITEM.GetRecord(intIndex)
37                  strTestOutput = String.Format("{0} | {1} | {2} | {3}", ITEM("ItemNumber"), ITEM("Category"), ITEM("UDP"), ITEM("ItemDescription"))  (5)
38                  Me.Msg(LogInfo, strTestOutput)
39              Next
40
41          Catch ex As Exception
42              blnNoErrors = False
43              Me.Msg(LogError, String.Format("TestDTO: {0}", ex.Message))
44          End Try
45
46           Return blnNoErrors
47      End Function
48  End Class
```

In this continuation of our TestDTO, we note:

1. Here(1) we are using the BCclass method Fetch( ) to select HLDOs from the Item Master table in BellHawk. Fetch( ), as is described subsequently, can have many different arguments. Here we are simply selecting “All” records.

2. Fetch ( ) returns the number of records available. Here (2) we are testing for any records available.

3. Here (3) we are printing out headers for the columns of informational data.

4. Here (4) we are indexing through the records for the ITEM HLDO, one at a time using the GetRecord method of BCclass. When we call GetRecord it moves the name:value pairs for the HLDO, from the adaptor database, into an in-memory name:value array where they can be accessed.

5. Here (5), we see outputting the parameter values as informational data. The call ITEM(“ItemNumber”) retrieves the string that is the value of the “ItemNumber” parameter for the keyword “ITEM” as defined in the XML metadata specified in the [BellHawk] adaptor in the .ini file.

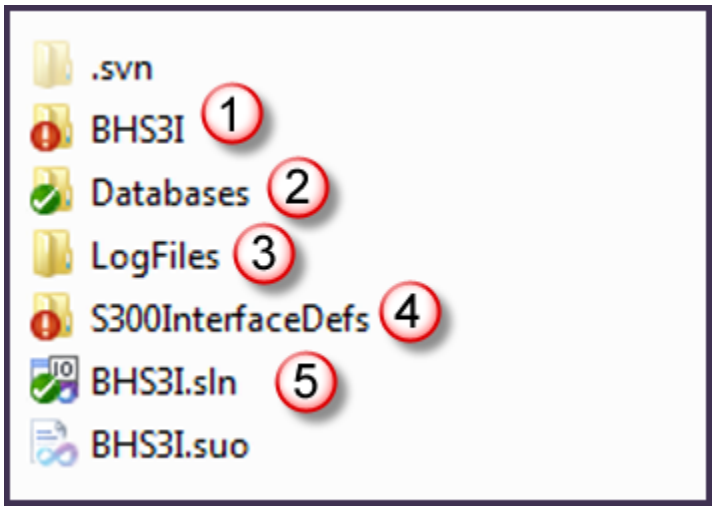
The rest of the code is for handling errors and ending the Run function and the TestDTO class.

A Real Project

The following is an example of a real project to exchange data between BellHawk and a Sage 300 ERP system. It was developed starting from the Distributed Prototype Transfer Function described above.

Folders

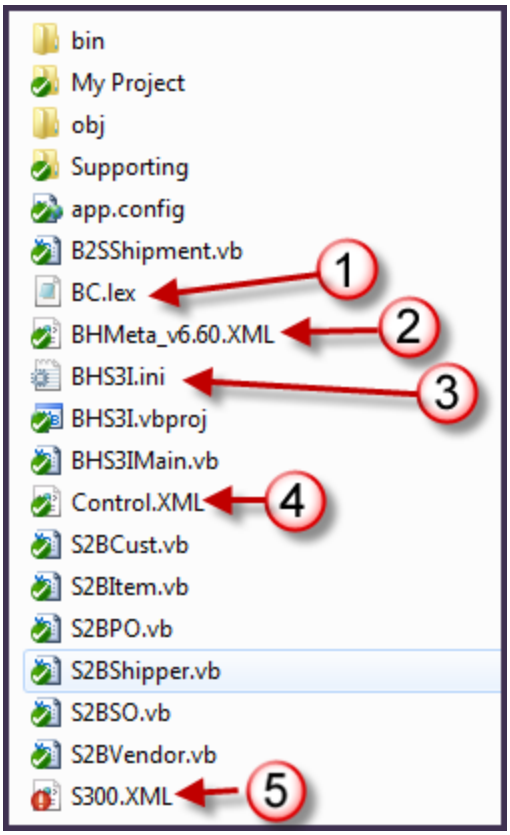
The files in the solution folder are as follows:



Things of note:

1. We have renamed our transfer function BHS3I (1). We follow a convention of naming transfer function with two letters each for the systems followed by I for interface.
2. We typically add a folder called Databases (2) in which to keep backups of databases we use in testing.
3. The transfer bcMessage and Msg methods write into a daily log file that they create if needed. We find it convenient to create a folder (3) for these log files within the development folder.
4. In this case, for convenience, we have created a folder to hold the Excel definition files that are used to create or edit the Sage 300 Metadata. In that way the metadata can be easily edited.
5. The solution has been BHS3I (5) for convenience.

Within the BHS3I project folder, we now find an expanded set of files:



Here we have added a file for each DTO with names like S2BItem.vb to indicate by convention that this DTO transfers Items records from Sage to BellHawk.

In addition we have added:

1. A BC.Lex encrypted licensing file (1) for the MilramX installation, issued by BellHawk Systems. Licensing is typically for a named computer and a given number of adaptors.
2. The Metadata for BellHawk (2) typically copied from the BC Website folder
3. The ini file to be used by the transfer function (3). Typically this is copied from the BC website and modified.
4. The Metadata for the BC Control database itself (4) which is typically copied from the BC website.
5. The Metadata for Sage 300 (5), again copied from the BC Website.

The Initialization File

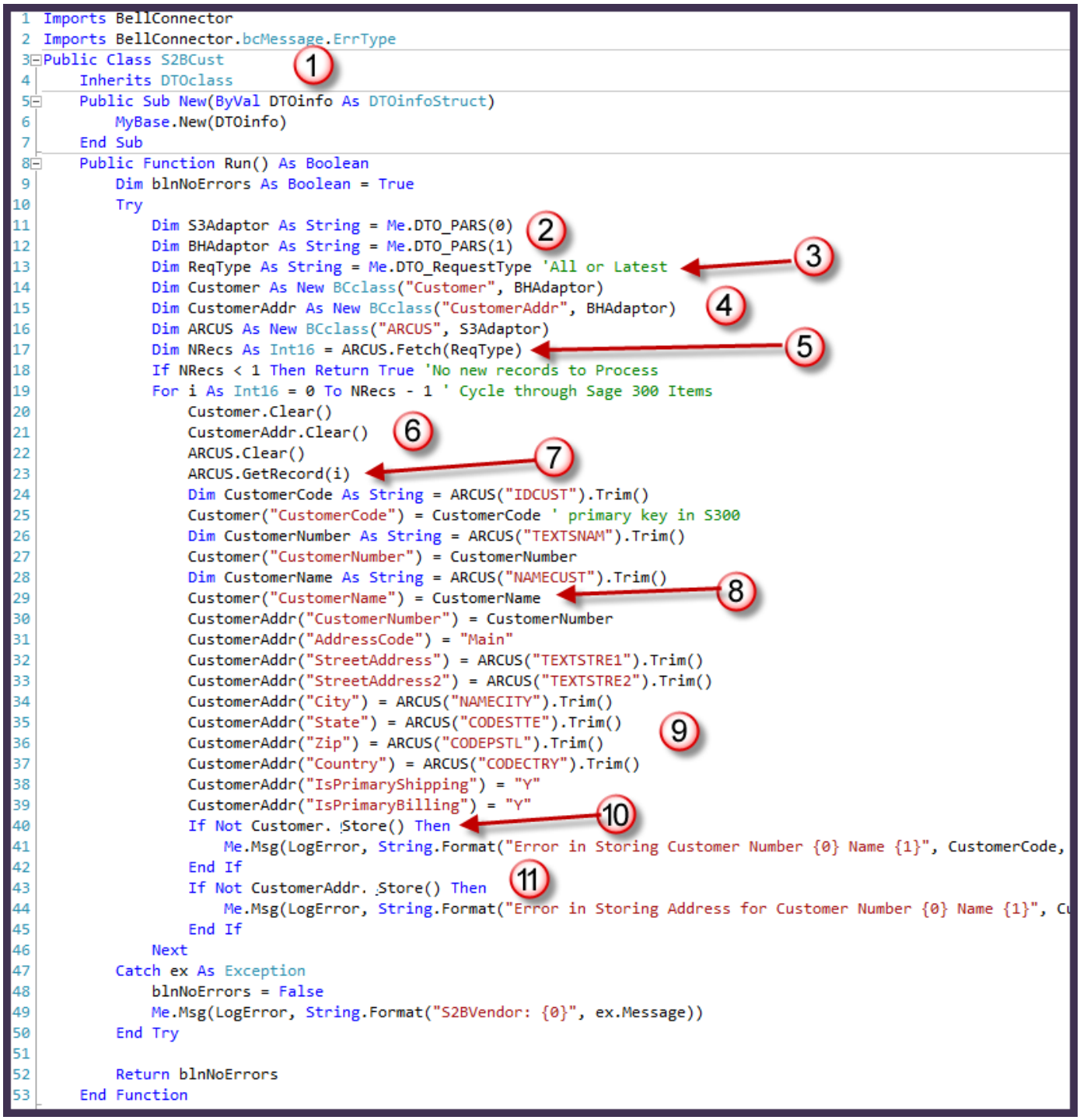
```
[Control]
XMLfile = Control.XML
Type = MSSQL
; Replace the values on the next 4 lines with your BellHawk BHCTL SQL server database/login
settings
Server = PGWIN7\SQLSERVER2008
Database = BellConnectorS300Test
UserID = ********
Password = ******
LogFolderName = F:\DEV\Dev2015\BHS3I\BCWebsite\BCWebsite\Logfiles
DebugLevel = 0

[Adaptors]
; Adaptors should be a comma-separated list of adaptors
Adaptors = S300,BellHawk, Control

[BellHawk]
Type = MSSQL
XMLFile = BHMeta_v6.60.XML
Server = PGWIN7\SQLSERVER2008
Database = V66S300Test
UserID = ********
Password = ********

[S300]
Type = MSSQL
XMLFile = S300.XML
Server = SAGESERVER
Database = SAMINC
UserID = *********
Password = ********
```

(1) (2) (3) (4) (5)

Details of how to setup an initialization file are described in the "MilramX Installation and Configuration Guide. Things of note here are:

1. There needs to be a [Control] section so that the Transfer Function knows how to get at its Control database.
2. If, as in this case, the ini file was copied from the BC Website, it may have an entry for a default Logfile folder (2). This is ignored by the transfer function. Below this is the DebugLevel which sets how errors, warnings, and informational messages generated by the transfer function are displayed.
3. Here (3) we define the Adaptors that are accessible to the transfer function. Note that the Control adaptor is included so that transfer function DTOs can access application specific tables attached to the BC Control database.
4. This is the BellHawk Adaptor. It uses the MSSQL BC Interface and so expects the MSSQL.DLL to be present in the project. It uses the standard BellHawk metatdata definition file.
5. This is the S300 Adaptor. It also uses the MSSQL BC Interface. It uses an S300.XML Metadata file, which was created using the BellHawk MetaData Editor (which is distributed as part of the MilramX developer toolset).

An Example DTO



```
1  Imports BellConnector
2  Imports BellConnector.bcMessage.ErrType
3  Public Class S2BCust                                                    (1)
4      Inherits DTOclass
5      Public Sub New(ByVal DTOinfo As DTOinfoStruct)
6          MyBase.New(DTOinfo)
7      End Sub
8      Public Function Run() As Boolean
9          Dim blnNoErrors As Boolean = True
10         Try
11             Dim S3Adaptor As String = Me.DTO_PARS(0)                    (2)
12             Dim BHAdaptor As String = Me.DTO_PARS(1)
13             Dim ReqType As String = Me.DTO_RequestType 'All or Latest   (3)
14             Dim Customer As New BCclass("Customer", BHAdaptor)
15             Dim CustomerAddr As New BCclass("CustomerAddr", BHAdaptor)  (4)
16             Dim ARCUS As New BCclass("ARCUS", S3Adaptor)
17             Dim NRecs As Int16 = ARCUS.Fetch(ReqType)                   (5)
18             If NRecs < 1 Then Return True 'No new records to Process
19             For i As Int16 = 0 To NRecs - 1 ' Cycle through Sage 300 Items
20                 Customer.Clear()
21                 CustomerAddr.Clear()                                    (6)
22                 ARCUS.Clear()
23                 ARCUS.GetRecord(i)                                      (7)
24                 Dim CustomerCode As String = ARCUS("IDCUST").Trim()
25                 Customer("CustomerCode") = CustomerCode ' primary key in S300
26                 Dim CustomerNumber As String = ARCUS("TEXTSNAM").Trim()
27                 Customer("CustomerNumber") = CustomerNumber
28                 Dim CustomerName As String = ARCUS("NAMECUST").Trim()
29                 Customer("CustomerName") = CustomerName                 (8)
30                 CustomerAddr("CustomerNumber") = CustomerNumber
31                 CustomerAddr("AddressCode") = "Main"
32                 CustomerAddr("StreetAddress") = ARCUS("TEXTSTRE1").Trim()
33                 CustomerAddr("StreetAddress2") = ARCUS("TEXTSTRE2").Trim()
34                 CustomerAddr("City") = ARCUS("NAMECITY").Trim()
35                 CustomerAddr("State") = ARCUS("CODESTTE").Trim()
36                 CustomerAddr("Zip") = ARCUS("CODEPSTL").Trim()          (9)
37                 CustomerAddr("Country") = ARCUS("CODECTRY").Trim()
38                 CustomerAddr("IsPrimaryShipping") = "Y"
39                 CustomerAddr("IsPrimaryBilling") = "Y"
40                 If Not Customer.Store() Then                            (10)
41                     Me.Msg(LogError, String.Format("Error in Storing Customer Number {0} Name {1}", CustomerCode,
42                 End If
43                 If Not CustomerAddr.Store() Then                        (11)
44                     Me.Msg(LogError, String.Format("Error in Storing Address for Customer Number {0} Name {1}", Cu
45                 End If
46             Next
47         Catch ex As Exception
48             blnNoErrors = False
49             Me.Msg(LogError, String.Format("S2BVendor: {0}", ex.Message))
50         End Try
51
52         Return blnNoErrors
53     End Function
```

Please note the following:

1. Here we are creating the code for a DTO named S2BCust (1). This follows our naming convention of letter for source system, “2” for “to”, letter for target system followed by objects being transferred.

2. Here (2) we are getting the source and target adaptor names as specified in the BC web interface.

3. Here we are picking up whether we want to transfer “All’ records or just the “Latest” updates from the DTO property DTO_RequestType, which is set for the DTO in the BC web interface.

4. Here (4) we create the new instances of the HLDOs we want to work with through calls to BCclass.

5. Here (5) we are issuing a request to Fetch ( ) to select All active records or just the Latest updates from the Sage 300 Customer table. We then cycle through these records, as described previously.

6. The first step (6) in the loop is to clear the parameter values in each of the HLDO structures from the prior loop using the BCclass Clear method.

7. We then (7) get the next source record for the data, which in this case contains both the customer name and address.

8. We then use the parameter values in the source record to populate the parameter fields in the HLDOs for Customers (8) and Customer Addresses (9) which are stored in separate tables in the destination BellHawk system. Note that, in this case, we are dealing with source records that are padded with blanks to a fixed field width and so these have to be trimmed before transfer to BellHawk.

9. We then output the resultant HLDOs using a Store method (10) and (11).

10. A DTO can integrate data from multiple tables and systems into one or more resultant HLDOs and then store the results in different tables in different systems if need be.

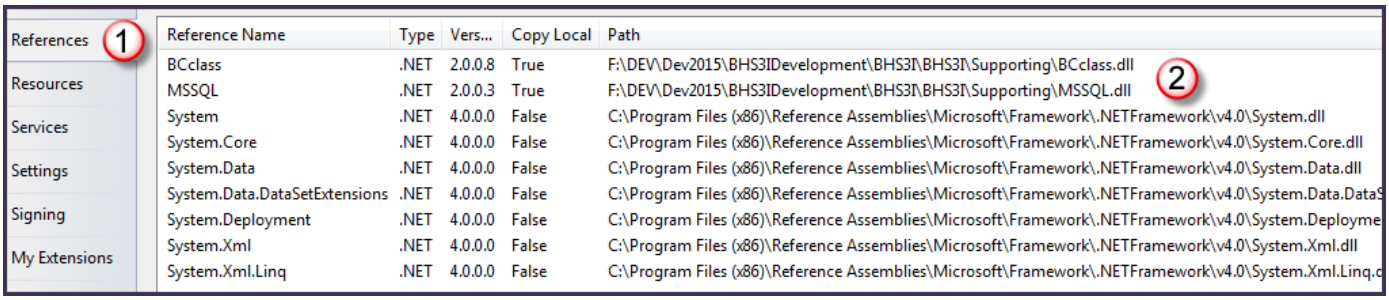
11. There is no need to write any SQL code or to do complex error handling as this is all handled by the BC framework code. This enables business programmer/analysts to focus on mapping the data between source and target systems.

12. The HLDO keyword and parameter names used in the code are all accessible from Excel spreadsheets generated as part of the process of implementing the XML Metadata. These Excel spreadsheets can be exported at any time using the DEXEL functionality of the BC Website.

13. This also enables the XML metadata to be edited as part of the development process by editing the Excel spreadsheet defining an HLDO and re-importing this spreadsheet using DEXEL. Just don’t forget to save the updated Metadata in DEXEL and to move the resultant XML file to your working directory.

Project Setup

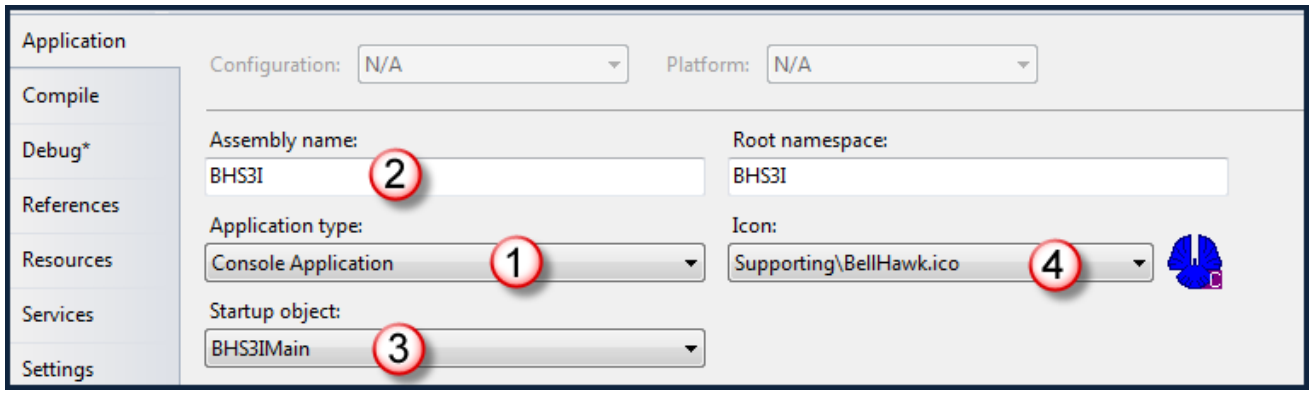
In the Refernces Tab (1) make sure that you have included BCclass and the BC Interfaces (2) that you intend to use



Note that the BC Interface DLL class is called by Name using Reflection, so the BC Interface class name, such as MSSQL, has to match the name given in the adaptor section in the initialization file.

Please note that the BCclass.DLL and any BC Interface DLLs normally reside in the Supporting folder.

The Application setup is shown below



Please note:

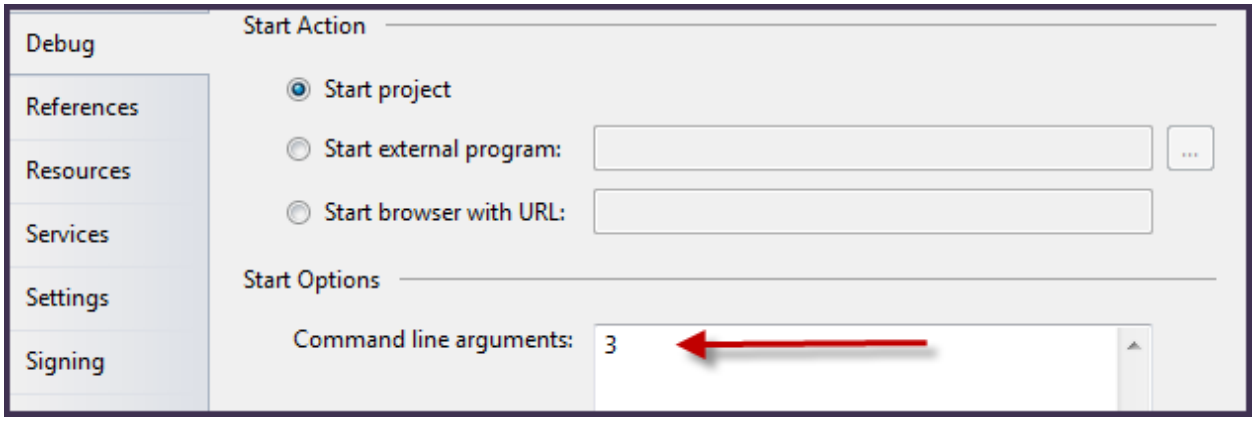
1. This is a console application (1) so it can be run as a “hidden” background process when run by the launcher. Normally the console window is hidden unless an error occurs or unless it is specified to be shown by a setting of DEBUG in the ini file.
2. The Assembly Name (2) needs to match the name of the ini file. So this transfer function is expecting an initialization file named BHS3I.ini.

3. Even though it is code that does not need modifying, you can change the name of the Module containing the main program. This module name needs to be placed in the name of the Startup object (3).

```
1  Imports BellConnector
2  Module BHS3IMain
3      ' BellHawk to Sage 300 Interface
4      Sub Main()
5          Dim blNoErrors As Boolean = True
6          blNoErrors = bcMessage.Start()
7          If blNoErrors Then blNoErrors = CheckArgs(System.Environment.GetCommandLineArgs())
8          If blNoErrors Then blNoErrors = IniInit()
9          If blNoErrors Then blNoErrors = LicCheck()
10         If blNoErrors Then blNoErrors = InitConnections()
11         If blNoErrors Then blNoErrors = DTOrun()
```

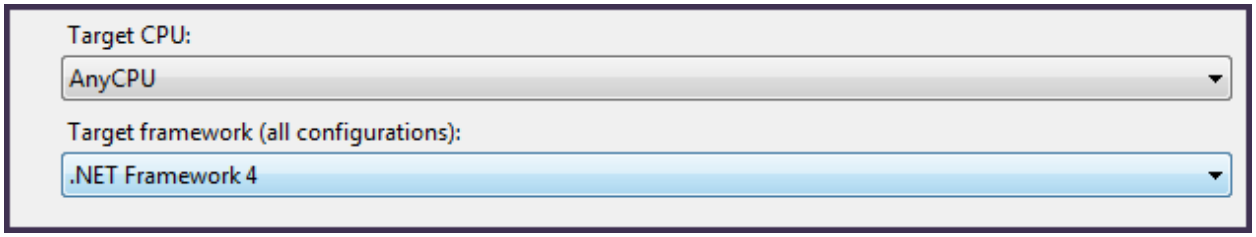
4. You can also provide the icon from the Supporting folder as the icon (4) to be used for this program.

Under the Debug tab, the command line argument needs to be set to the ControlID of the line in the BC_CTL table in the BC Control database in which the setup data for the DTO to be debugged is placed by the BC Website setup.



You can then set breakpoints in the Run code of the referenced DTO and when you run the transfer function code in Debug mode the specified DTO will be executed and you can step through your code with the debugger.

Under the Compiler Options tag you need to set Visual Studio to use .Net 4.0 or later so as to be able to access JSON support. Generally it is a good idea to specify any CPU when debugging.



Accessing and Storing HLDOs

In the following we will use the example of accessing item master part number records in the BellHawk database. The same methods can be used to access, modify and store away any type of HLDO in other databases.

To access a specific class of HLDO, you first need to declare a new instance of a BCclass object class, as in:

```
Dim IMPN As New BCclass("BellHawk", "ITEM")
```

The first argument in this call is the name of the adaptor and the second is the Keyword for the HLDO within the adaptor.

You can then retrieve a specific object instance from the database for the BHSDK object class you just created by (for example)

```
IMPN.LookUp( "ABC123")
```

The one or more arguments to Lookup are the values of the key parameters that uniquely identify the object. Any arguments not supplied are assumed by default to be the empty string `""`. In the case of a part number lookup only one parameter is required. In the case of a PO Line object then the PO Number and the Line Number are required.

Lookup returns true if the object is found and false if not. The data for the object is stored in a temporary store in memory for that class of object.

The temporary memory store for each class of HLDO objects contains the current values for all the parameters for the active instance of that object. These parameter values can be read and written and the current values written to the database.

Your can then reference the value of a specific field for the retrieved data object as (for example)

```
Dim PartNum As String
PartNum =  IMPN("ItemNumber")
```

Here the parameter name, such as `"ItemNumber",` must correspond to the parameter name in the meta-data for the object class. The values are always returned as strings and must be converted to an appropriate data type before being used.

As parameters are retrieved from the temporary memory store, they are checked character-by-character to ensure that they only contain valid characters for the data type specified in the meta-data. This is so they can be converted without further error checking in your program. If they do contain bad characters then an error exception is thrown after it is logged in the log file.

Null values in the database are converted to empty strings for the parameters when they are retrieved from the temporary memory store. Sometimes this can result in a data-checking error exception being thrown, such as when a FLOAT data type field in the database contains a NULL value.

To handle this situation, you can supply an optional second Boolean argument to the get( ) to indicate whether the conversion should allowed if the data is not valid, such as having a NULL value for a FLOAT. An example of this is

```
Dim PriceString As String = IMPN("UnitPrice", True)
```

This will return an empty string if the Unit Price is Null in the database and will otherwise attempt to convert whatever is in the database field to a string. You can then do your own data checking before converting the returned string to a floating point variable value.

Likewise values in the temporary store can be set for parameters of an object:

```
IMPN("UnitPrice") = UnitPrice.ToString
```

These values are set in the temporary in-memory store for the object. You do not need to LookUp( ) the object first and can simply set all the needed parameters for the object to define a new object. Or you can LookUp( ) an existing object and modify its parameters.

The values are checked against the specified meta-data data type for each object to make sure that they do not contain any invalid characters that might inadvertently be entered into the database. If they do, then an error exception is thrown.

Once values have been set or modified for the active object instance in the temporary store then the values can be saved away in the database by, for example:

```
IMPN.Store()
```

Store( ) returns True if the values in the HLDO temporary store for the object is successful and False if an indirect data reference could not be resolved for the value provided (such as an unknown PO number in a PO Line object, which is referenced by both PO and PO Line). Store takes care of deciding whether to Insert a new record in the database or Update an existing record. In doing this, it resolves indirect references and sets database fields to NULL, where required.

Store( ) takes an optional string argument `"C"` or `"M"`. If this is “C” then Store will update all fields in a record being updated, setting those whose current value in the HLDO temporary store for the object is an empty string "" to `NULL`

If the optional parameter is “M” then Store will only modify those fields in a record being updated for which current values in the HLDO temporary store for the object have not been set or are not equal to the empty string `""`.

```
Please note that a parameter value can be set to an empty string by a call
such as:
```

```
IMPN("Material") = ""
```

To retrieve all objects of a particular type, for example part numbers, you will first make a call such as:

```
IMPN.Fetch("All")
```


The records are typically stored in an internal data table, with Fetch returning the number of records read.

Individual records can then be loaded into the temporary memory store for the object class, one at a time by:

```
IMPN.GetRecord(N)
```

Here, N is the record number. Typically these are retrieved in a loop for N from 0 to one less than the number of records returned from Fetch( ).

Once each record is in the temporary store for the object class, it can be manipulated and then stored away again, if needed, as described previously.

Fetch( ) can have a number of parameters, besides “All”:

1. “Latest” gets all records of the specified class that have changed since the last call to Fetch( ). BCclass maintains a date-time at which each HLDO (by Adaptor and Keyword) was last read in a Fetch call.
2. “Since” followed by a second parameter, which is the date and time, expressed as a string in standard date-time format. This fetches all records since the specified time-date.
3. “Range”, which is followed by the starting and ending date-times. All records that were modified between these times are retrieved by Fetch(). If the first parameter is set to “” then this acts as “Before” the second parameter. If the second parameter is omitted, then this is the same as “Since” in action.
4. “Specific”, which can be followed by “lookup” parameters values (in the lookup order specified in the XML meta-data file) that uniquely identify the item being fetched.
5. “Select” which is like “Specific” except some lookup parameters name arguments can be set to an empty string `""`. For example purchase order lines have two lookup parameters, the PO number and the line number. Calling “Select” with just the PO Number as the first parameter will fetch all PO lines for the specified PO number as “Select” assumes that any parameter values not specified contain the empty string by default

Please note that Fetch( ) takes strings as its arguments, so time-date objects need to be converted to strings before calling Fetch.

Once instances of an object have been fetched into an internal table, they can be selected and sorted using the Choose() method, as in

```
IMPN.Choose("ItemNumber LIKE 'ABC%' ", "ItemDescription ASC")
```

This will retrieve all those records with the Item Number beginning with “ABC” and order them as a set of rows in ascending order in memory.

Choose has two call arguments:

1. This argument enables the selection of specific values. It follows the format for a SQL "WHERE" clause without the word "WHERE" and uses parameter names instead of field names for variables. Here we can test for parameters being equal to a value, greater or less than a value or use % wild cards and LIKE clauses. We can also have multiple logic clauses joined by AND and OR with parentheses, if needed. All constants except numeric values need to be enclosed in single quotes.
2. This argument specifies the ordering of the resultant spreadsheet. It follows the same format as a SQL "ORDER BY" clause without the "ORDER BY" words and uses parameter names instead of field names. There can be a succession of parameter names, with each optionally followed by ASC (for ascending) or DESC (for Descending), with successive parameters separated by commas. The sort order is the order of the parameters in this argument.

Choose( ) returns the number of rows selected or -1 if there was an error. Also error messages, such as about bad WHERE or ORDER BY clause formats are written to the log file.

Chosen Rows can then be read into the temporary memory store by calling the GetChosen() method, as in:

```
GetChosen(N)
```

Where the argument N is a 32 bit integer, representing the row number 0, 1, 2 etc.

GetChosen returns false if an invalid row number is elected.

Once the row is in the temporary memory store its parameters can be read and manipulated as before and it can be written out using Store( )

Objects can be deleted from the database by a call such as:

```
IMPN.Delete("P103")
```

This does a lookup of the specific object based on the one to three lookup parameters, specified in the definition for the object keyword, and then sets the deleted field to the deleted value. Please note that it does not physically delete the record from the database.

In BellHawk, objects marked as deleted no longer show up on screens and reports but are retained in the database for referential integrity.

Please note that for the Store( ) and Delete( ) methods, the temporary memory store for the object must contain valid values for the lookup parameters, such as the part number, before they are called.

The availability of these BCclass methods is dependent on what is supported by the BC Interface in use. Store and Lookup are supported by most interfaces but Choose and GetChosen are typically only supported by BC Interfaces that directly access database and not by Interfaces such as the BellHawk Web Services Interface.

In addition the following may be optionally supported by a BC Interface:

- Oldest, as in Item.Oldest (FType). Fetches data from Database into Data Table, returns number of Rows Fetched. 'FType = "All","Latest". '"All" returns rows with timestamp matching the oldest timestamp in database table. "Latest" returns rows with timestamp matching the oldest timestamp since the recorded "Last" timestamp.
- Run, as in MyProc.Run. Runs a stored procedure with the parameter data in the HLDO as its arguments.
- StartGroup, as in POReceipt.StartGroup and EndGroup as in POReceipt.EndGroup. These may be used to bracket Store type transactions that need to be sent as a group to the target system, such as for a receipt consisting of multiple items. The action is specific to the BC Interface and may not be supported.

Other Useful BCclass Methods

Data related to each parameter for each keyword is stored in an array. The numeric index of a specific parameter can be accessed, for example, by:

```
Dim EstProdLifeIndex As Int16 = GetParIndex("EstProdLife")
```

The description associated with a parameter can then be accessed by, for example:

```
IMPN.ParDescription(EstProdLifeIndex)
```

Here the subroutine is called with the parameter index and returns a string containing the description.

The data type for the parameter can also be accessed, for example by:

```
IMPN.ParType(UnitCostIndex)
```

These data types are entries like "TEXT", "FLOAT" that are defined for the parameters in the XML Metadata.

There are `IMPN.NumParam` parameters for an item master part number (for example) and the parameter values can be read from and written to the temporary store for the data object using index numbers as well as parameter names (if it is more convenient to do this in a loop) as in, for example:

```
IMPN(UnitCostIndex) = 350
```

Rather than

```
IMPN("UnitCost") = 350
```

The parameter name corresponding to an index can be referenced, for example, by:

```
IMPN.ParName(UnitCostIndex)
```


which returns a string containing the parameter name.

The values of all the parameters for an object in the temporary store for that object class can be set to the empty string "" by a call to

```
IMPN.Clear()
```

Accessing Views

Some data objects defined in the XML file reference database views rather than the tables themselves. Typically these objects can only be read and not written and you will get an exception error thrown if you attempt to Store( ) values back to these views.

Stored Procedures

The SDK supports the calling of stored procedures as shown in the following example:

```
DIM ADDPO As New BCclass(TA, "ADDPO")
ADDPO("PONumber") = PONumber
ADDPO("VendorNumber") = VendNum
ADDPO("Project") = ProjNum
ADDPO.Run()
```

As you can see the declaration of the BHSDK class and the setting of the object parameter values is identical to that used to access any other object. But instead of using a Store( ) method, we use a Run( ) method, which returns a Boolean True or False indicating whether the stored procedure was executed successfully.

In the XML meta-data, object parameters can be declared as input or output arguments to the stored procedure or as the return value. Output argument values and the return value are returned as parameter values into the temporary memory store for the object instance.

BCclass does not support handling a SQL dataset generated by the call to the stored procedure.

Handling Errors

BCclass has extensive error checking throughout its code. If an error occurs, it is first logged to the specified log file. Then an exception is thrown. So you should always look in the log file when doing a post-mortem on why an exception was thrown.

It is important that all code written using the BCclass, including setting and getting parameter value, be included in a Try-Catch block, to ensure that errors and exceptions are caught and handled appropriately.

When a user attempts to set a parameter value, such as in

```
IMPN("UnitPrice") = "Expensive"
```

BCclass will detect that “Expensive” does not match the specified DOLLAR format for the UnitPrice parameter and throw a FormatException error, with the Parameter Name, Field Type and Parameter Value in the error message. User code can catch these FormatException errors for data format problems separately from Exception errors thrown for more serious problems and use them to appropriately handle the error.

The data read from a database is not verified on a Fetch or a Get, except for handling errors that arise when required parameters that are indirectly referenced cannot be accessed. In this case a regular Exception, with an appropriate error message, is thrown.

The data is validated against the parameter type when a parameter value is retrieved from the temporary store. In this way, errors in the database are ignored if they are not pertinent to the DTO code.

If there is a validation error on getting a parameter value then a FormatException is thrown and the user's code should catch this in a Try-Catch block and handle it appropriately.

Sometimes it is necessary to handle NULL values in the database. These occur when optional fields are retrieved. For example, if we retrieved the UnitPrice parameter value for an Item Master Part object and the Unit Price has not been specified in the database, we would get a format error by default.

If we want to check first get the string, before converting it to a single precision floating point number, we can add a second optional parameter to the get call as shown in the following example:

```
Dim Test As String = IMPN("UnitPrice", True)
If Test = "" Then
      ' Handle NULL value
Else
      Dim UnitPrice as Single = Test.ToSingle
```

The second parameter is the AllowNulls parameter, which is set to False by default. If it is set to True then, if the corresponding database field is NULL, the returned parameter value is set to the empty string `""`. If the database field is not NULL then the data is still checked against the field type specified for the parameter before being returned.

When getting parameter values, a third parameter can also be supplied. This is a substitute field type to use for checking. For example, we may want to get a value from a third party database that has a % embedded in a DECIMAL field. If we tried to get this parameter, we would normally get an error. But by specifying the optional third parameter as `"TEXT"` your code can read the parameter value as if it were a TEXT field and then do its own processing to strip out the % sign.

Note that it is important for user code to place calls to BCclass routines inside a Try-Catch structure to catch and appropriately deal with errors that result from errors due to inconsistencies between the data in the meta-database and data in the underlying BellHawk database.

The BcClass Oldest Method

Sometimes we just need to get the oldest record in a table or the oldest record that was updated since the last Oldest or Fetch method for the HLDO. This is done using the BCclass Oldest() method, which is a Fetch() and GetRecord(0) rolled into one method.

It takes one argument – "All" or "Latest"

- "All" ignores the time-last-accessed timestamp, just gets the oldest record for the keyword.

- "Latest" fetches oldest record since the time-last-accessed timestamp

In either case, the time-last-accessed timestamp is set to the time-last-modified of the fetched record.

It returns an Integer value = number of records fetched with matching time-last-modified timestamp. Ideally (and in most cases) number of records = 1.

If the number of records > 1 (i.e. ambiguous "oldest" record), all of the fetched records are still accessible – none need be "lost". In this case Oldest will return a single record into the temporary array of parameters for the HLDO and subsequent Oldest calls will fetch the others.

Handling BellHawk UDP Objects

BellHawk makes extensive use of JSON encoded User Defined Parameters (UDPs). These are found in the "UDP" parameter of most HLDOs.

To make it easier to manipulate UDPs, there are two methods JEX and JSET in the BClass. Their function is to retrieve a specific parameter from a JSON encoded parameter or to set a value into the parameter.

Example usage:

Length = ITEM.JEX ("UDP", "Length")

ITEM.JSET ("UDP", "Length", Length)

JEX is called with two parameters:

The name of a JSON encoded Then add functions JEX and JSET to the BHSDK Class. Their function is to retrieve a specific parameter from a JSON encoded parameter or to set a value into the parameter.

Example usage:

Length = ITEM.JEX ("UDP", "Length")

ITEM.JSET ("UDP", "Length", Length)

JEX is called with two parameters:

1. The name of a JSON encoded HLDO parameter, which is "UDP" by convention in BellHawk.
2. The name of a JSON UDP Parameter within that JSON encoded parameter's string.

JEX will return a string value, which will be empty if the field is not of type JSON or the parameter is not found within the JSON encoded field.

JSET is called with three parameters:

1. The name of a JSON encoded HLDO parameter, which is "UDP" by convention in BellHawk.

2. The name of a JSON UDP Parameter to be set within the JSON encoded parameter's string.
3. A string value to which the parameter in the JSON encoded string is to be set. If the parameter is not already present in the JSON encoded string then the JSON parameter and its value will be added to the parameters within the JSON encoded string.

JSET will return a Boolean true or false. It will return true if the parameter is correctly set. It will return false if the target string is not of data type JSON.

When setting a JSON UDP parameter then, if the parameter is not already present in the JSON encoded string, the JSON parameter and its value will be added to the parameters within the JSON encoded string. Otherwise the existing UDP parameter will be modified.

Notes on Adaptor names in Metadata

```
204   </xs:schema>
205   <tblAdaptorKeys>
206     <AdaptorName>BellHawk</AdaptorName>
207     <Keyword>Adjust</Keyword>
208     <Description>Adjust Inventory</Description>
209     <TableName>WEBsp_Adjust</TableName>
210     <IDPar1 />
211     <IDPar2 />
212     <IDPar3 />
213     <ReadWrite>Run</ReadWrite>
214   </tblAdaptorKeys>
215   <tblAdaptorKeys>
216     <AdaptorName>BellHawk</AdaptorName>
217     <Keyword>Bases</Keyword>
218     <Description>Conversions Between Packaging</Description>
219     <TableName>tblItemBases</TableName>
220     <SrcSelectionCriteria>IsDeleted = 0</SrcSelectionCriteria>
221     <dtLastUpdateFldName>dtLastMod</dtLastUpdateFldName>
```

XML Metadata for keywords is tied to an Adaptor name, which must be the same as used in the .ini file.

By definition, the XML Metadata file is not limited to a single adaptor definition.
A single XML file can (and may originally have been used to) contain all necessary adaptor definitions used in a transfer.

MilramX and BHSDK use that *AdaptorName* tag in deciding which *KeyStruct* and *ParStruct* definitions belong to the adaptor being instantiated.

To accommodate cases where the same metadata needed to be used for two or more adaptor, we add the *Alias* keyword to the .ini file. This allows "PlantA" and "PlantB" to be Adaptor names in the ini files with *Alias = BellHawk* entries in the adaptor sections, tying them both to the BellHawk adaptor metadata.

Some Useful Notes

The following are some useful tips and reminders based on experience with developing interfaces using MilramX.

1. DTO name must match in Name and Case to DTO Name setup in BC Web interface – such as TestDTO.
2. The BC Framework transfer function code expects that BC.lex and .ini files will be placed in the main transfer function file and the BC Interfaces to be placed in folder Supporting
3. The Logfile folder is set through the BC Website and not the transfer function ini file
4. It is essential to put a Try-Catch block around all keyword-adaptor initializations.
5. Many BCclass methods that use a Paramarray argument, which is special type of argument call that allows arbitrary sequence of parameters.
6. The Me keyword provides a way to refer to the specific instance of a class or structure in which the code is currently executing. Me behaves like either an object variable or a structure variable referring to the current instance. Using Me is particularly useful for passing information about the currently executing instance of a class or structure to a procedure in another class, structure, or module
7. An expired BC.lex causes a warning to be displayed on the website Master page (therefore on each page of the website), but does not prevent login.If BC.lex is more than 30 days past expiration the licensed adaptors cease to work.
8. An expired BC.lex causes a warning to be displayed on the website Master page (therefore on each page of the website), but does not prevent login.
9. If BC.lex is more than 30 days past expiration the licensed adaptors cease to work.
10. The reason that we run the launcher and MilramX transfer function as user processes is to overcome domain privilege and security issues for the transfer function that occur when running the launcher and the transfer function as system processes. Basically when the launcher runs as a service and launches the transfer function, the transfer process is run with System Machine privileges, which means it cannot access anything in most domain setups, for security reasons.
11. For those developers that are adventurous and really understand Active Directory (AD) and want to run the Launcher as a Service, we can make the source code for the Launcher available. In our experience, it is feasible to run the Launcher as a service, launching a transfer function with specific privileges. But this needs to be carefully configured specifically for each AD domain otherwise security can easily be compromised (such as by making System Machine, i.e. the O/S core accessible to Everyone).
12. The reason we use a separate lightweight launcher process is that the execution of the transfer function is tied in closely with ODBC drivers and other functions provided by Microsoft and other third parties. When these have an error they usually do not recover systems resources well (Microsoft is notorious for squandering resources when an error is

detected in the bowels of its software). As a result, we quickly run out of resources, such as the max memory allocated to a process, if we attempt to run the transfer process on a 5 second or so periodic basis (which the launcher runs on) even when it errors out. So, when we end the transfer process run for one DTO (either correctly or in error) we kill the transfer process which releases its memory and other system resources back to the O/S, ready to be allocated again when the transfer process is run again.

13. About half our clients choose to run the launcher and the transfer function, as well as the BHBTI and BarTender, on a Windows 7 Pro machine that is dedicated to this purpose. This off-loads the main server and gives you a lot more remote access control. Some even run the MilramX website on the Windows 7 Pro website on the same Windows 7 Pro machine. Even if you run the MilramX website on your main server you should, in this configuration, put the MilramX control database on the Windows 7 Pro machine for performance reasons.

14. When VS compiles a project, if a .dll is referenced VS copies it from the \Supporting folder reference location to the bin\ folder. In such a case, merely replacing the .dll in bin\ with a newer version won't work – it will be overwritten by the reference copy even if it's older when VS regenerates the executable code.

15. BCclass is a 64 bit DLL and, as such, only likes to be linked with in x64 or Any Class mode and not in x86 mode.

