Navigating BCP with .NET

Peter Sparks, University of Michigan

1. Abstract

The Blaise Component Pack is a gateway to expanding the capabilities of Blaise by working with metadata, data, and other parts of the Blaise system. However finding just the right information can be difficult, and the current documented examples are in Visual Basic 6 and C++. This paper looks at the different components available, how to navigate them from the various entry points using the .Net 2005 programming environment in both VB and C# and BCP 2.0, and some tips to make coding easier. Equivalent examples will be given in both Visual Basic.Net and C#.Net.

2. Introduction

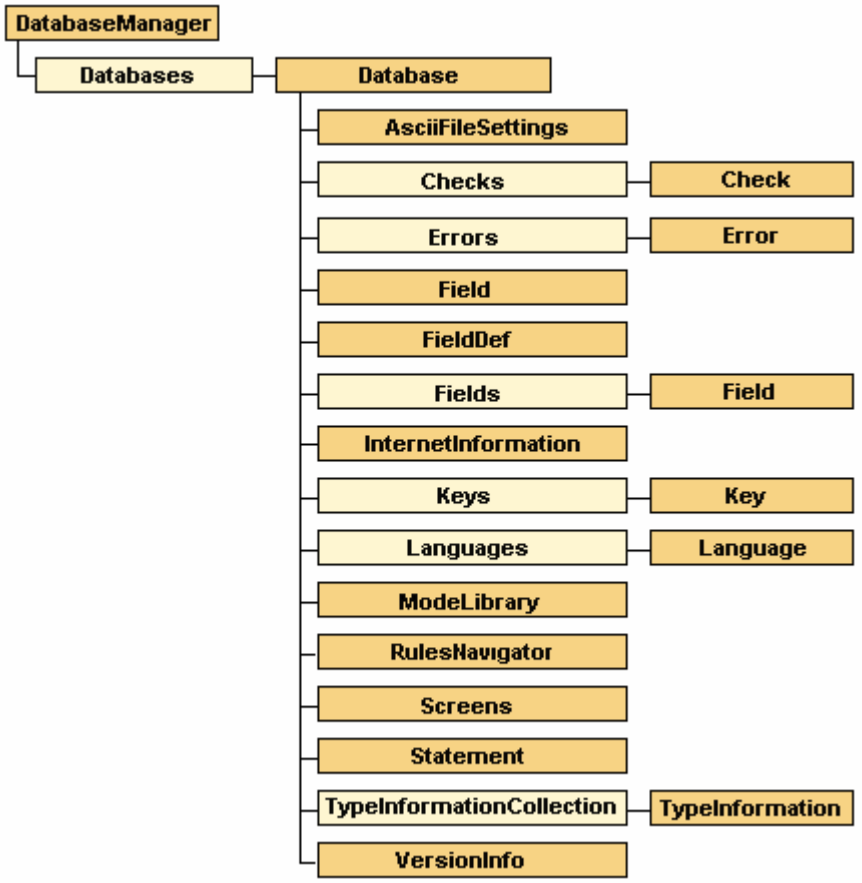
The BCP is a powerful tool for accessing information stored in the Blaise datamodel and database. It has been available for a number of years and has been used extensively by The University of Michigan in many of their utilities. As new programming environments have changed, the utilities have also been updated.

The change from Visual Basic 6 to C# in .Net 2.0 was not as straightforward as could be hoped. Some procedures and methods changed names, the way of retrieving indexed information in .Net was different, and the environment was changed. The documentation provided in the Blaise system uses VB6 and C++ and is a good starting point.

The purpose of this paper is to show several examples of common tasks to retrieve and modify information stored in the datamodel and database within the .Net environment. Appendices are provided to give more detail on certain topics, such as a VB6 to .Net translation guide, charts of method names between VB and C#, and some general programming tips.

This discussion uses the BCP (BLAPI4A2.dll) from Blaise 4.7 in this discussion. At the time of this writing a new Blaise 4.8 BlAPI3A.dll is now available.

3. Overview of the BCP Database Manager



The above illustration is from the Blaise 4.7 help "Blaise API Object Model."

The Blaise Database Manager is the root object used to connect to the datamodel. As shown in the illustration there are many different entry points depending upon the task from the Database Manager. However, most of the functions needed are at the Database level.

The "Database" is somewhat misleading in that it is the way to access both the datamodel and the data stored with the datamodel.

There have been some additions and changes between VB6 and C#/VB.Net (see Appendix A). The syntax in using some of these methods has also changed.

One of the greater difficulties when first being exposed to the BCP is realizing that some commands are irrelevant if data is not being used. The discussion thus will start with strictly the datamodel, and then look at the data retrieved from the datamodel.

It is assumed the reader has familiarity with .Net peculiarities, such as being zero-based for its indexing. Appendix B has a brief discussion of the differences from VB6 to .Net.

The remarkable strength of working with the BCP is the ability to process and analyze data from the datamodel in a variety of ways. However, choose existing tools such as Cameleon when all that is needed is a quick list of variables or data dictionaries without needing a user interface. Create .Net programs when complex processing and interfaces are needed.

Although there are many possible tasks that can be accomplished using BCP only a few will be addressed in this paper.

3.1 Things you can do with BCP (datamodel)

1. Retrieve any field, auxfield, or local definition generally or specifically and associated information regardless where it is used
2. Step through all statements in the same order as defined in the rules, including blocks and procedures.
3. Retrieve any specific statement
4. Read through all the modelib settings
5. Read through the screen layouts defined for the DEP (main interview, parallel blocks)
6. Retrieve defined primary and secondary keys

3.2 Things you can do with BCP (database)

1. Read through all external lookups data files
2. Retrieve cases via specific keys, update fields, check case status, reevaluate the rules
3. Retrieve text with fills based upon the current case data

3.3 Other areas to explore (not in this paper)

1. Interact with the Data Entry Program (DEP) via Alien Routers and Alien Procedures
2. Retrieve Internet information
3. Navigate through the datamodel using data and the RulesNavigator
4. Explore all the type definitions, both system and user-defined
5. Process all errors in a form
6. Examine all checks and signals within the datamodel
7. Retrieve version information for a datamodel and a database
8. Make use of built-in database handling functions, such as copying or deleting Blaise databases.
9. Look at the language definitions for a datamodel

Given the lists above, it is surprising there are not some things you can do using the API. There are at least two areas that functionality has not been provided.

3.4 Things you cannot do with BCP

1. You cannot update or create datamodels. Blaise only allows the programmer to develop source code. The b4cpars.exe program can be used to prepare source code outside of the development environment.
2. You cannot work with compiled Manipula scripts, such as opening an .MSU and treat it as a datamodel.

4. Accessing BCP/API

4.1 Companion Source Code

A companion program and source code is available with this paper. All examples in this paper have been taken from the working program.

4.2 Initial steps – Accessing the datamodel

All work with the BCP begins first with Database Manager. Be sure you have made a reference to the BLAPI4A2.dll (Blaise 4.7) in your project references in order to work with BCP.

It is recommended that only one instance of the Database Manager is ever created. This is the entry point for both the Meta information and the data.

```
C#
BlAPI4A2.DatabaseManager curDatabaseManager = new BlAPI4A2.DatabaseManager();

VB
Dim curDatabaseManager As BlAPI4A2.DatabaseManager
```

In order to retrieve information from a datamodel and the associated database create a Database object.

```
C#
BlAPI4A2.Database curDatabase;

VB
Dim curDatabase As BlAPI4A2.Database
```

This defines the Database object but you have to use the OpenDatabase method of the DatabaseManager to get an instance. This still is not associated with either a datamodel or database.

```
C#
curDatabase = curDatabaseManager.OpenDatabase("");

VB
curDatabase = curDatabaseManager.OpenDatabase("")
```

To actually open the datamodel assign the full path and name of the datamodel to the DictionaryFileName property.

```
C#
curDatabase.DictionaryFileName = @"c:\temp\SomeDatamodel.bmi";

VB.Net
curDatabase.DictionaryFileName = "c:\temp\SomeDatamodel.bmi";
```

Now that the connection to the datamodel has been made, all meta information about the datamodel is available. This includes all fields, auxfields, locals, parameters, procedures, blocks, statements, screen layoutouts, modelib fonts, external datamodel definitions, categories, and more. The sole task is being able to get the information and process it.

4.2 The difference between Fields and Defined Fields

The main difference is the FieldDef object will give you the base information to the field as it is defined, while the Field object will give you an instance for every Field. For example, suppose there is a rostered field called Age. In Blaise it may be defined as

```
Age : ARRAY[1..20] OF 25..80
```

The FieldDef object will show this as one entity: Age[].

The Field object will give twenty entities: Age[1], Age[2], …, Age[20].

There are additional methods and properties available only through the FieldDef object that are not in the Field object, such as the format of the external lookup files.

5. BCP/API Examples

Each of the following examples shows how to accomplish the task. The examples are a little shorter than the working programming code (removed procedure headings, Try/Catch statements, extra interface interactions) to keep the code short.

5.1 Retrieve any field, auxfield, or local definition generally or specifically and associated information regardless where it is used

There are two major ways to retrieve information needed from the datamodel: strictly all fields retrieved at one time, or recursively.

All at the same time has the advantage of simpler code but at the overhead cost of storing all those fields in memory at the same time.

So if working with datamodel with a large number of fields consider recursion, but at the cost of longer execution time.

There are properties that are specific to types of objects. In the Field/FieldDef examples below, the property .IndexedName has a non-null/non-Nothing value if the FieldKind property is not a generated parameter (BlFieldKind.blfkGenParameter).

5.1.1 FieldDef: All fields non-recursive

Please note that using recursion strictly on a FieldDef will not work because the FieldDef does not have the DefinedFields()/ get_DefinedFields() method. If you need to use recursion and the FieldDef then use the Field recursive example below, retrieve the FieldDef for each field using the .FieldDef method, and finally store the results and remove duplicates.

get_DefinedFields returns a set of Fields and not FieldDef may be expected.

```
C#
BlAPI4A2.Fields curFields =
curDatabase.get_DefinedFields((int)BlAPI4A2.BlFieldKind.blfkAll,
(int)BlAPI4A2.BlFieldType.blftAll);

for (int i = 0; i < curFields.Count; i++)
{
        if (curFields[i + 1].FieldKind != BlAPI4A2.BlFieldKind.blfkGenParameter)
                lbResults.Items.Add(curFields[i + 1].FieldKind.ToString() + " "
                + curFields[i + 1].IndexedName);
        else
                lbResults.Items.Add(curFields[i + 1].FieldKind.ToString() + " "
                + curFields[i + 1].Parent.IndexedName + " GP");
}

VB
Dim curFields As BlAPI4A2.Fields =
curDatabase.DefinedFields(BlAPI4A2.BlFieldKind.blfkAll,
BlAPI4A2.BlFieldType.blftAll)

Dim i As Integer
For i = 0 To curFields.Count - 1
        If (curFields(i + 1).FieldKind <> BlAPI4A2.BlFieldKind.blfkGenParameter)
Then
                lbResults.Items.Add(curFields(i + 1).FieldKind.ToString() & " " _
                & curFields(i + 1).IndexedName)
        Else
                lbResults.Items.Add(curFields(i + 1).FieldKind.ToString() & " " _
                & curFields(i + 1).Parent.IndexedName & " GP")
        End If
Next i
```

5.1.2 Field: All fields Non-recursive

This routine is exactly like the non-recursive defined fields method with a different method, get_Fields(). It also returns a set of fields, but every field is made unique by the addition of the index number for sets and arrays. As a result many more fields are returned compared to the get_DefinedFields() method.

```
C#
BlAPI4A2.Fields curFields =
curDatabase.get_Fields((int)BlAPI4A2.BlFieldKind.blfkAll,
(int)BlAPI4A2.BlFieldType.blftAll);

for (int i = 0; i < curFields.Count; i++)
{
if (curFields[i + 1].FieldKind != BlAPI4A2.BlFieldKind.blfkGenParameter)
        lbResults.Items.Add(curFields[i + 1].FieldKind.ToString() + " "
        + curFields[i + 1].IndexedName);
else
        lbResults.Items.Add(curFields[i + 1].FieldKind.ToString() + " "
        + curFields[i + 1].Parent.IndexedName + " GP");
}

VB
Dim curFields As BlAPI4A2.Fields = curDatabase.Fields(BlAPI4A2.BlFieldKind.blfkAll,
BlAPI4A2.BlFieldType.blftAll)

Dim i As Integer
For i = 0 To curFields.Count - 1
```

```
        If (curFields(i + 1).FieldKind <> BlAPI4A2.BlFieldKind.blfkGenParameter)
Then
                lbResults.Items.Add(curFields(i + 1).FieldKind.ToString() & " " _
                & curFields(i + 1).IndexedName)
        Else
                lbResults.Items.Add(curFields(i + 1).FieldKind.ToString() & " " _
                & curFields(i + 1).Parent.IndexedName & " GP")
        End If
Next i
```

5.1.3 Field: All fields Recursive

This uses the same method as before for retrieving all fields, but with a difference. The routine only recurses on those fields with a DataType of blftBlock. As a result each set of fields retrieved via the get_Fields()/Fields() method gets fields specific to that block, unlike the prior example where all fields of the datamodel were retrieved.

There is additional overhead and setup to work with recursion, and the general recommendation is to use it only if needed. It tends to have a greater toll on memory and processing time than other methods.

To start a recursive routine a starting value is always needed. In this example the root name of the datamodel is needed, and this is retrieved from the dictionary name curDatabase.DictionaryAsField.Name. The dictionary can be considered the uppermost block of the datamodel.

C#

```
int totfields = 0, lastPos = lbResults.Items.Count;
retrieveRecursive(curDatabase.get_Field(curDatabase.DictionaryAsField.Name), ref
totfields);


/// <summary>
/// Retrieve all fields in the datamodel using filters recursively
/// </summary>
/// <param name="curDatabase">Pass in the Blaise database object</param>
/// <param name="totfields">Variable to store the total number of
statements</param>
void retrieveRecursive(BlAPI4A2.Field curField, ref int totfields)
{
    if (curField.DataType == BlAPI4A2.BlFieldType.blftBlock)
    {
       BlAPI4A2.Fields curFields =
curField.get_Fields((int)BlAPI4A2.BlFieldKind.blfkAll,

                                            (int)BlAPI4A2.BlFieldType.blftAll);
        totfields += curFields.Count;

       for (int i = 0; i < curFields.Count; i++)
       {
        if (curFields[i + 1].FieldKind != BlAPI4A2.BlFieldKind.blfkGenParameter)
                lbResults.Items.Add(curFields[i + 1].FieldKind.ToString() + " "
                + curFields[i + 1].IndexedName);
        else
                lbResults.Items.Add(curFields[i + 1].FieldKind.ToString() + " "
                + curFields[i + 1].Parent.IndexedName + " GP");

        if (curFields[i + 1].DataType == BlAPI4A2.BlFieldType.blftBlock)
                retrieveRecursive(curFields[i + 1], ref totfields);
        }

    }
}
```

VB

```
Dim totfields As Integer = 0, lastPos = lbResults.Items.Count
retrieveRecursive(curDatabase.Field(curDatabase.DictionaryAsField.Name), totfields)


''' <summary>
''' Retrieve all fields in the datamodel using filters recursively
```

```
''' </summary>
''' <param name="curField">The field to use as the starting point</param>
''' <param name="totfields">Variable to store the total number of
statements</param>
Private Sub retrieveRecursive(ByVal curField As BlAPI4A2.Field, ByRef totfields As
Integer)

    If (curField.DataType = BlAPI4A2.BlFieldType.blftBlock) Then
       Dim curFields As BlAPI4A2.Fields =
curField.Fields(BlAPI4A2.BlFieldKind.blfkAll,

                                                    BlAPI4A2.BlFieldType.blftAll)
       totfields += curFields.Count

        Dim i As Integer
        For i = 0 To curFields.Count - 1
            If (curFields(i + 1).FieldKind <>
       BlAPI4A2.BlFieldKind.blfkGenParameter) Then
                lbResults.Items.Add(curFields(i + 1).FieldKind.ToString() +
" " _
                + curFields(i + 1).IndexedName)
            Else
                lbResults.Items.Add(curFields(i + 1).FieldKind.ToString() +
            " " _
                + curFields(i + 1).Parent.IndexedName + " GP")
            End If

            If (curFields(i + 1).DataType = BlAPI4A2.BlFieldType.blftBlock)
Then
                retrieveRecursive(curFields(i + 1), totfields)
            End If
        End If
       Next i

    End If
End Sub
```

5.1.4 Field: Filtered fields non-recursive

To get specific categories of fields change the filter parameters on the get_Fields()/Fields() method. For example, to get fields that store data in the database use the FieldKind with blfkDataField.

```
C#
BlAPI4A2.Fields curFields =
curDatabase.get_Fields((int)BlAPI4A2.BlFieldKind.blfkDataField,

                                          (int)BlAPI4A2.BlFieldType.blftAll);

VB
Dim curFields As BlAPI4A2.Fields =
curDatabase.Fields(BlAPI4A2.BlFieldKind.blfkDataField,
                                                    BlAPI4A2.BlFieldType.blftAll)
```

The different FieldKind and FieldTyes are shown below. Note that C# requires type casting to (int) of the parameters for the method, whereas VB does this automatically.

blFieldKind	
blfkParameter	Field parameters are used within a block or procedure.
blfkGenParameter	The field is a generated parameter. This happens whenever a field reference from within a block is defined outside that block
blfkDataField	The field is stored in a Blaise database
blfkAuxField	The field is not stored in a Blaise database
blfkGenAuxField	Like a generated parameter, but based upon an Auxfield.
blfkExternal	A field reference to an external datamodel (lookup)
blfkLocal	Typically used for FOR loop indexes, counters, ...
blfkAll	Retrieve all field kinds

blFieldType	
blftUnknown	This should be an error condition if it occurs
blftString	String type
blftInteger	Numeric data as INTEGER or range x..y
blftFloat	Real data as REAL, REAL[n], REAL[n,m] or range x.d..y.c
blftDate	Datetype
blftTime	Timetype
blftMemo	Open type
blftEnumeration	Field contains categories
blftClassification	Classification type
blftExternal	The field is an external type
blftBlock	The field is a block (contains other subfields)
blftAll	All field types

5.1.5 Examining DK/RF, Sets, Arrays

Note that there are not filters available for other field attributes, such as whether the field may allow DK/RF, or whether the field is a set or an array. To look at DK/RF look at the .DontKnowAllowed and .RefusalAllowed properties on the field, and the .Is*xxx* properties on the FieldDef method. curField in the below example is defined as BlAPI4A2.Field.

```
curField.DontKnowAllowed
curField.RefusalAllowed

curField.FieldDef.IsArray
curField.FieldDef.IsEmbeddedBlock
curField.FieldDef.IsSet
curField.FieldDef.IsTable
```

5.2 Step through all statements in the same order as defined in the rules, including blocks and procedures.

A major requirement for many of the utilities used at The University of Michigan is to extract the field information in the same order of execution as the rules. This can be accomplished using the RulesNavigator, and can also be done via the Statements collection. Both methods use recursion.

The RulesNavigator is designed to be used with the data of a case and is documented within the Blaise help file. It makes use of a navigator pointed at particular statements, and can move back and forth throughout the instrument according to the data contained in the case. The MoveTo*xxx* method is common to the navigator and causes the navigator pointer to change to the next appropriate statement. A true value is returned if the action was succesful.

The Statement object is similar in concept to the RulesNavigator. It can be used with/without data, and also can follow the statements. The advantage of the Statement object is the ability to jump around the rules via the StatementIdent, whereas the RulesNavigator always starts at the first statement and then proceeds forwards from there.

The example below makes use of the Statement object. If the StatementIdent values are stored and indexed, then it is possible to process the rules of a datamodel in many different ways.

The routines uses of the .ThenStatement, .ElseStatement, and an implied .ChildStatement to navigate. The first statement of the datamodel is found at the DictionaryAsStatement property on the database. Only For loops, Blocks, and Procedures will have a statement collection associated with them that includes a count. All other types, such as assignment (Let), are considered as a standalone statement.

Navigating the datamodel via the rules can be like driving through a maze of cul-de-sacs (dead ends). The main roads are easy to navigate (block level), but there are subdivisions (sub blocks), houses (fields), and eventually the cul-de-sac at the end of the street. At the end of a block, the end of a procedure, the end of the IF statetement, the Else statement, … there will be no further statements. The method .NextStatement will return null/Nothing even though the next logical statement is just the next street over. You're not allowed to drive through the backyard to get to the next logical street. Hence, you have to back up to the last turn you made and go down the next street. This is the recursive element to navigating the datamodel.

The routine can be summarized as follows.

- Step through all statements in the current block (and process them as needed)
  - If the current statement groups other statements (loop, block, procedure), then step through all those substatements before continuing.
  - If the current statement is an IF, navigate the THEN portion, then navigate the ELSE portion.
- Once at the end of the current block, pop back up to the prior block where you last left off and continue processing.

C#

```
int totfields = 0;
retrieveRecursiveStatement(curDatabase.DictionaryAsStatement, ref totfields);


/// <summary>
/// Retrieve all statements including procedures
/// </summary>
/// <param name="curStatement">Starting statement</param>
/// <param name="totStatements">Variable to store the total number of
statements</param>
void retrieveRecursiveStatement(BlAPI4A2.Statement curStatement, ref int
totStatements)
{
totStatements += curStatement.Statements.Count;
if (curStatement.Statements.Count > 0)
    for (int i = 1; i <= curStatement.Statements.Count; i++)
    {
    lbResults.Items.Add(curStatement.Statements[i].StatementType + "\t"
                      + curStatement.Statements[i].StatementIdent + "\t"
                      + curStatement.Statements[i].StatementText);

    switch (curStatement.Statements[i].StatementType)
    {
        case (BlAPI4A2.BlStatementType.blstCondition):
            retrieveRecursiveStatement(curStatement.Statements[i].ThenStatement,
                                       ref totStatements);
            if (curStatement.Statements[i].ElseStatement != null)
                retrieveRecursiveStatement(
                                       curStatement.Statements[i].ElseStatement,
                                        ref totStatements);
```

```
                break;

            case (BlAPI4A2.BlStatementType.blstForLoop):
                retrieveRecursiveStatement(curStatement.Statements[i],
                                           ref totStatements);
                break;

            case (BlAPI4A2.BlStatementType.blstBlockQuest):
                retrieveRecursiveStatement(curStatement.Statements[i],
                                           ref totStatements);
                break;

            case (BlAPI4A2.BlStatementType.blstProcedure):
                retrieveRecursiveStatement(curStatement.Statements[i],
                                           ref totStatements);
                break;

            }
    }

}
```

VB

```
Dim totfields As Integer = 0
retrieveRecursiveStatement(curDatabase.DictionaryAsStatement, totfields)

''' <summary>
''' Retrieve all statements including procedures
''' </summary>
''' <param name="curStatement">Starting statement</param>
''' <param name="totStatements">Variable to store the total number of
statements</param>
Private Sub retrieveRecursiveStatement(ByVal curStatement As BlAPI4A2.Statement,
                                       ByRef totStatements As Integer)

totStatements += curStatement.Statements.Count

Dim i As Integer
    For i = 1 To curStatement.Statements.Count
        lbResults.Items.Add(curStatement.Statements(i).StatementType & vbTab _
        & curStatement.Statements(i).StatementIdent & vbTab _
        & curStatement.Statements(i).StatementText)

         Select Case (curStatement.Statements(i).StatementType)
           Case (BlAPI4A2.BlStatementType.blstCondition)
             retrieveRecursiveStatement( _
                          curStatement.Statements(i).ThenStatement,  totStatements)
              If Not (curStatement.Statements(i).ElseStatement Is Nothing) Then
                 retrieveRecursiveStatement( _
                          curStatement.Statements(i).ElseStatement,  totStatements)
              End If

           Case (BlAPI4A2.BlStatementType.blstForLoop)
             retrieveRecursiveStatement(curStatement.Statements(i), totStatements)

           Case (BlAPI4A2.BlStatementType.blstBlockQuest)
             retrieveRecursiveStatement(curStatement.Statements(i), totStatements)

           Case (BlAPI4A2.BlStatementType.blstProcedure)
             retrieveRecursiveStatement(curStatement.Statements(i), totStatements)

        End Select
    Next i

End Sub
```

5.3 Retrieve any specific statement

As mentioned above, if the StatementIdent associated with a particular statement is stored, then any statement in the datamodel can be retrieved easily. Each number in the dot notation refers to a depth in the statement structure. For example, “12.35.7” refers to the 12^{th} statement at the main rules of the datamodel, the 35^{th} statement in the subblock at that 12^{th} statement, and then finally the 7^{th} statement at the 35^{th} subblock’s statement.

```
C#
BlAPI4A2.Statement curStatement = curDatabase.get_Statement("12.35.7");

VB
Dim curStatement As BlAPI4A2.Statement = curDatabase.Statement("12.35.7")
```

5.4 Read through all the modelib settings

The modelib contains all the interface information that will be used with the DEP, such as screen layouts, screen ordering, contents of questions per page, custom and system fonts, and so forth. This is accessible via the modelib object.

Note that the information retrieved is via the modelib settings stored in the datamodel. This doesn't read the .BML file, nor does this read the configuration .DIW file.

5.4.1 Font information

The user-defined fonts are stored in the .ModeLibrary.Style.Fonts object. Use the get_CustomFont()/CustomFont() methods to retrieve information about the fonts. Only a few of the font options are shown below. The CustomFont expects a letter denoting the font being retrieved.

```
C#
string fontName = curDatabase.ModeLibrary.Style.Fonts.get_CustomFont("A").Name;
int fontSize = curDatabase.ModeLibrary.Style.Fonts.get_CustomFont("A").Size;

VB
Dim fontName As String = curDatabase.ModeLibrary.Style.Fonts.CustomFont("A").Name
Dim fontSize As Integer = curDatabase.ModeLibrary.Style.Fonts.CustomFont("A").Size
```

5.4.2 Read through the screen layouts defined for the DEP (main interview, parallel blocks)

The screen layouts are accessible via the Screens collection at the database level. They consist of Layout, Parallel, Page, and Quest collections and tend to work together.

5.4.2.1 Layout, Parallel, Page, and Quest collections

These four collections give access to when each field/auxfield is presented to the interviewer. By reading through these collections and comparing it against all fields/auxfields in the datamodel the defined but never-asked fields/auxfields can be located.

These collections represent a hierarchy, and can be shown below

Layout – such as Interviewing, Editing, or self-defined
    Parallel – such as the main parallel block, the primary key parallel, listing …
        Page – each page contains a set of fields/auxfields that will appear on it
            Question – the actual field/auxfield can be referenced from here

```
C#
BlAPI4A2.LayoutSet Layout;
BlAPI4A2.Parallel Parallel;
BlAPI4A2.StoredPage Page;
BlAPI4A2.Question Quest;

int iLayout, iParallel, iPage, iQuest;
```

```
for (iLayout = 1; iLayout <= curDatabase.Screens.LayoutSetCollection.Count;
iLayout++)
{
  Layout = curDatabase.Screens.LayoutSetCollection[iLayout];
  for (iParallel = 1; iParallel <= Layout.ParallelCollection.Count; iParallel++)
    {
    Parallel = Layout.ParallelCollection[iParallel];
    for (iPage = 1; iPage <= Parallel.StoredPageCollection.Count; iPage++)
      {
      Page = Parallel.StoredPageCollection[iPage];
      for (iQuest = 1; iQuest <= Page.QuestionCollection.Count; iQuest++)
        {
        Quest = Page.QuestionCollection[iQuest];

        lbResults.Items.Add("Layout: " + iLayout.ToString()
                        + ", Parallel: " + iParallel.ToString()
                        + ", Page: " + iPage.ToString()
                        + ", Question: " + iQuest.ToString()
                        + ": " + Quest.Field.IndexedName);
        }
      }
    }
} /*for*/
```

```
VB
Dim Layout As BlAPI4A2.LayoutSet
Dim Parallel As BlAPI4A2.Parallel
Dim Page As BlAPI4A2.StoredPage
Dim Quest As BlAPI4A2.Question

Dim iLayout As Integer, iParallel As Integer, iPage As Integer, iQuest As Integer

For iLayout = 1 To curDatabase.Screens.LayoutSetCollection.Count
  Layout = curDatabase.Screens.LayoutSetCollection(iLayout)
  lbResults.Items.Add(vbTab & "Layout: " & Layout.Name)
  For iParallel = 1 To Layout.ParallelCollection.Count
    Parallel = Layout.ParallelCollection(iParallel)
    For iPage = 1 To Parallel.StoredPageCollection.Count
      Page = Parallel.StoredPageCollection(iPage)
      For iQuest = 1 To Page.QuestionCollection.Count
        Quest = Page.QuestionCollection(iQuest)
        lbResults.Items.Add("Layout: " & iLayout.ToString() _
                        & ", Parallel: " & iParallel.ToString() _
                        & ", Page: " & iPage.ToString() _
                        & ", Question: " & iQuest.ToString() _
                        & ": " + Quest.Field.IndexedName)
      Next iQuest
    Next iPage
  Next iParallel
Next iLayout
```

5.5 Retrieve defined primary and secondary keys

Each key for a Blaise datamodel can consist of multiple parts. The number of keys via the .Keys property refer to both primary and secondary keys. To get the pieces that make up a key step through the InvolvedFields collection. The Keys.Kind method will be one of blkkPrimary, blkkSecondary, or blkkTrigram.

```
C#
for (int i = 0; i < curDatabase.Keys.Count; i++)
{
        string keyResult = "Name: " + curDatabase.Keys[i + 1].Name
                        + ", Kind: " + curDatabase.Keys[i + 1].Kind + ", using ";

        for (int j = 0; j < curDatabase.Keys[i + 1].InvolvedFields.Count; j++)
        {
                if (j > 0)
                        keyResult += ", ";

                keyResult += curDatabase.Keys[i + 1].InvolvedFields[j +
        1].IndexedName;
        }

        lbResults.Items.Add(keyResult);
}
```

```
VB
Dim i As Integer
For i = 0 To curDatabase.Keys.Count - 1
        Dim keyResult As String = "Name: " & curDatabase.Keys(i + 1).Name _
                                & ", Kind: " & curDatabase.Keys(i + 1).Kind & ",
using "

        Dim j As Integer
         For j = 0 To curDatabase.Keys(i + 1).InvolvedFields.Count - 1
                If (j > 0) Then
                        keyResult += ", "
                End If

                keyResult += curDatabase.Keys(i + 1).InvolvedFields(j +
        1).IndexedName
        Next j

        lbResults.Items.Add(keyResult)
Next i
```

5.6 Read through all external lookups data files

Reading an external datamodel is much like reading the main datamodel. The external references are found via the BlAPI4A2.BlFieldKind.blfkExternal type on the get_DefinedFields ()/DefinedFields() method.

Once that is found, it is just a matter of stepping through each of those references, making a connection to the appropriate database, and then stepping through the fields and data contained therein.

The steps to read a data file are as follows.

1. Create a database connection via the OpenDatabase() method
2. For Ascii and AsciiRelational data files set the AsciiFileSettings.Separator and .Delimiter values
3. Assign the datamodel via the DictionaryFileName
4. Connect to the data via the .Connect property set to True
5. Use .ReadNextRecord() method to read the case data into memory
6. When finished, disconnect from the data files (.Connect = False), and dispose of the database connection.

The entire datarecord in text format is available via the ActiveRecord.Value property. Specific individual fields can be retrieved via get_Field()/Field() method, passing in full dot name of the field and looking at the .Text or .Value properties.

```
C#
BlAPI4A2.Fields oFields =
curDatabase.get_DefinedFields((int)BlAPI4A2.BlFieldKind.blfkExternal,
(int)BlAPI4A2.BlFieldType.blftAll);

for (int i = 0; i < oFields.Count; i++)
{
  BlAPI4A2.Database DBExt = curDatabaseManager.OpenDatabase("");


  if (oFields[i+1].FieldDef.ExternalInformation.StorageFormat ==
BlStorageFormat.blsfAscii ||
   oFields[i+1].FieldDef.ExternalInformation.StorageFormat ==
BlStorageFormat.blsfAsciiRelational)
    {
    DBExt.StorageFormat = oFields[i +
1].FieldDef.ExternalInformation.StorageFormat;
```

```
    DBExt.AsciiFileSettings.Separator =

oFields[i+1].FieldDef.ExternalInformation.AsciiFileSeparator;
    DBExt.AsciiFileSettings.Delimiter =
oFields[i+1].FieldDef.ExternalInformation.AsciiFileDelimiter;
    }

  DBExt.DictionaryFileName =
oFields[i+1].FieldDef.ExternalInformation.DictionaryFileName;
  DBExt.DataFileName = oFields[i+1].FieldDef.ExternalInformation.DataFileName;

  // Make the connection to the Blaise database
  DBExt.Connected = true;

  string s = null;
  for (int j = 0; j < DBExt.ActiveRecord.Count; j++)
  {
    if (j > 0)
      s += ", ";
    s += DBExt.ActiveRecord[j + 1].Name;
  }

  lbResults.Items.Add("Database: " + DBExt.DataFileName + ", Columns = " + s);

  // Start reading through the records and display the results
  for (int j = 0; j < DBExt.RecordCount; j++)
  {
    DBExt.ReadNextRecord();
    s = null;
    for (int k = 0; k < DBExt.ActiveRecord.Count; k++)
    {
      if (k > 0)
        s += ", ";
      s += DBExt.ActiveRecord[k + 1].Value;
     }
     lbResults.Items.Add(s);
   }

   // close the database
   DBExt.Connected = false;

   // release the object for GC
   DBExt = null;

 } /*for*/
```

VB

```
Dim oFields As BlAPI4A2.Fields =
curDatabase.DefinedFields(BlAPI4A2.BlFieldKind.blfkExternal,

BlAPI4A2.BlFieldType.blftAll)

Dim i As Integer
For i = 0 To oFields.Count - 1
  Dim DBExt As BlAPI4A2.Database = curDatabaseManager.OpenDatabase("")


  If (oFields(i + 1).FieldDef.ExternalInformation.StorageFormat =
                                                    BlAPI4A2.BlStorageFormat.blsfAscii
Or _
     oFields(i + 1).FieldDef.ExternalInformation.StorageFormat =
                                      BlAPI4A2.BlStorageFormat.blsfAsciiRelational)
Then
      DBExt.StorageFormat =
oFields(i+1).FieldDef.ExternalInformation.StorageFormat
      DBExt.AsciiFileSettings.Separator =

oFields(i+1).FieldDef.ExternalInformation.AsciiFileSeparator
      DBExt.AsciiFileSettings.Delimiter =

oFields(i+1).FieldDef.ExternalInformation.AsciiFileDelimiter
  End If

  DBExt.DictionaryFileName =
oFields(i+1).FieldDef.ExternalInformation.DictionaryFileName
  DBExt.DataFileName = oFields(i+1).FieldDef.ExternalInformation.DataFileName

  '' Make the connection to the Blaise database
  DBExt.Connected = True
```

```
Dim s As String = Nothing
Dim j As Integer
For j = 0 To DBExt.ActiveRecord.Count - 1
  If (j > 0) Then
    s += ", "
  End If
  s += DBExt.ActiveRecord(j + 1).Name
Next j

lbResults.Items.Add("Database: " & DBExt.DataFileName & ", Columns = " + s)

' Start reading through the records and display the results
For j = 0 To DBExt.RecordCount - 1
  DBExt.ReadNextRecord()
  s = Nothing
  Dim k As Integer
  For k = 0 To DBExt.ActiveRecord.Count - 1
    If (k > 0) Then
      s += ", "
    End If
    s += DBExt.ActiveRecord(k + 1).Value
    Next k
    lbResults.Items.Add(s)
  Next j

  ' close the database
  DBExt.Connected = False

  ' release the object for GC
  DBExt = Nothing

Next i
```

5.7 Retrieve cases via specific keys, update fields, check case status, and reevaluate the rules

A specific field for a case in a database can be updated, or the same field across all cases, and so forth can be updated using the BCP/API. The method is similar to that for external lookups. Make the connection to the database. Retrieve the specific case, then update the particular field. Write the results back to the database.

Note: the form status may change depending upon the mode that the update is being made. The mode, DataEntryBehaviour, can be changed to one of bldbUncheckedEditing, bldbFreeInterviewing, bldbStrictInterviewing, and bldbCheckedEditing.

The value for a field can only be changed via the get_Field().Text/Field().Text property. The .Value property is read-only and will be updated according to the value stored in the .Text property. The .Text property can accept either string representations of numeric values, string data appropriately formatted for the data type of the field, or category names for enumerated fields.

C#

```
curDatabase.DataFileName = databaseName;
curDatabase.Connected = true;

// Try to retrieve the specific case
curDatabase.KeyValue = key;
curDatabase.ReadRecord();

// guarantee that any field can be adjusted
curDatabase.DataEntryBehaviour =
BlAPI4A2.BlDataEntryBehaviour.bldbUncheckedEditing;

// Note: the .Value is read-only and cannot be assigned

// and set the new value via the code name or the code value
curDatabase.get_Field(field).Text = data;

// cause any checks/signals to be noted
```

```
curDatabase.CheckRecord();

// save the results
curDatabase.UpdateRecord();

//
curDatabase.Connected = false;
curDatabase.DataFileName = null;

VB
curDatabase.DataFileName = databaseName
curDatabase.Connected = True

' Try to retrieve the specific case
curDatabase.KeyValue = key
curDatabase.ReadRecord()

' guarantee that any field can be adjusted
curDatabase.DataEntryBehaviour =
BlAPI4A2.BlDataEntryBehaviour.bldbUncheckedEditing

' Note: the .Value is read-only and cannot be assigned

' and set the new value via the code name or the code value
curDatabase.Field(field).Text = data

' cause any checks/signals to be noted
curDatabase.CheckRecord()

' save the results
curDatabase.UpdateRecord()

curDatabase.Connected = False
curDatabase.DataFileName = Nothing
```

5.8 Retrieve text with fills based upon the current case data

The datamodel has all the question text, but it is possible to retrieve the text with the fills per DEP if a dat has been associated with the datamodel. In this example assume the datamodel has been opened, and the database has been connected. The specific case to be retrieved is noted via the **key** variable.

The key point for this to work is make sure the BlDataEntryBehaviour has been set to interviewing. If it was left in data editing, the default mode, then the rules will not be executed and the fills will essentially come back as empty.

The example below steps through all the fields of the case and examines each question text. If it contains a '^' character signifying a fill it then gets the original text for the question and the filled text based upon the case data.

The first parameter of get_QuestionText()/QuestionText() is the language index, and the second is a boolean to indicate if fills should be replaced.

```
C#
curDatabase.DataEntryBehaviour =
BlAPI4A2.BlDataEntryBehaviour.bldbStrictInterviewing;

// Try to retrieve the specific case
curDatabase.KeyValue = key;
curDatabase.ReadRecord();

BlAPI4A2.Fields curFields =
curDatabase.get_Fields((int)BlAPI4A2.BlFieldKind.blfkDataField,

(int)BlAPI4A2.BlFieldType.blftAll);

for (int i = 1; i <= curFields.Count; i++)
  if (!string.IsNullOrEmpty(curFields[i].get_DefinedQuestionText(1)))
    if (curFields[i].get_DefinedQuestionText(1).Contains("^"))
```

```
      lbResults.Items.Add("Record #" + curDatabase.ActiveRecord.FormID + ",
Field="
                          + curFields[i].IndexedName + ", Original Text="
                          + curFields[i].get_QuestionText(1, false) + ", Filled
Text="
                          + curFields[i].get_QuestionText(1, true));

curDatabase.Connected = false;
curDatabase.DataFileName = null;
```

VB

```
curDatabase.DataEntryBehaviour =
BlAPI4A2.BlDataEntryBehaviour.bldbStrictInterviewing

' Try to retrieve the specific case
curDatabase.KeyValue = key
curDatabase.ReadRecord()

Dim curFields As BlAPI4A2.Fields =
curDatabase.Fields(BlAPI4A2.BlFieldKind.blfkDataField,
                                                    BlAPI4A2.BlFieldType.blftAll)

Dim i As Integer
For i = 1 To curFields.Count
  If Not (String.IsNullOrEmpty(curFields(i).DefinedQuestionText(1))) Then
    If (curFields(i).DefinedQuestionText(1).Contains("^")) Then
      lbResults.Items.Add("Record #" & curDatabase.ActiveRecord.FormID & ",
Field=" _
                          & curFields(i).IndexedName & ", Original Text=" _
                          & curFields(i).QuestionText(1, False) & ", Filled
Text=" _
                          & curFields(i).QuestionText(1, True))
    End If
  End If
Next i

curDatabase.Connected = False
curDatabase.DataFileName = Nothing
```

6. Conclusion

As shown in this paper, the information available via the BCP/API is rich in detail and can be used to accomplish many tasks. This Blaise “engine” makes it possible to create many utilities that are beyond the capabilities of Manipula, Maniplus, and Cameleon. Although the current Blaise documentation (4.7) is based upon VB6, there is a direct translation to the .Net environment. This is important since the development tools are always changing and VB6 is no longer supported by the Microsoft Corporation.

7. Appendix A

Below are tables comparing the use of VB commands in the documentation with the C# equivalent. An item in *italics* signifies it was not part of the documentation in VB6.

Database

Data?	External?	Database: VB	Database: C#
Yes		AccessMode	AccessMode
Yes		ActiveKey	ActiveKey
Yes		*ActiveRecord*	*ActiveRecord*
	Yes	AsciiFileSettings	AsciiFileSettings
		BlocksWithData	BlocksWithData
Yes		BOF	BOF
		CATI	*Cati*
		CAWI	*Cawi*
		Checks	Checks
Yes		Connected	Connected
Yes		CreateOnOpen	CreateOnOpen
Yes		DataChangedByRules	DataChangedByRules
Yes		DataChangedByUser	DataChangedByUser
Yes		DataEntryBehaviour	DataEntryBehaviour
Yes		DataFileName	DataFileName
		DataFilesInUse	DataFilesInUse
		DataFileVersionInfo	DataFileVersionInfo
		DictionaryAsField	DictionaryAsField
		DictionairyAsStatement	DictionairyAsStatement
		DictionaryCheckSum	DictionaryCheckSum
		DictionaryCreationDate	DictionaryCreationDate
		DictionaryCreationTime	DictionaryCreationTime
		DictionaryFileName	DictionaryFileName
		DictionaryFileVersionInfo	DictionaryFileVersionInfo
		DictionaryVersion	DictionaryVersion
Yes		EOF	EOF
	Yes	ExternalSearchPath	ExternalSearchPath
		FieldFilter	FieldFilter
Yes		FormID	FormID
Yes		FormStatus	FormStatus
		IMGLink	IMGLink
		InOutMode	InOutMode
		Internet	Internet
		InternetInformation	InternetInformation
		Keys	Keys
		KeyValue	KeyValue
		Languages	Languages
Yes		LastRecord	LastRecord
		Libraries	Libraries
		ModeLibrary	ModeLibrary
		ParallelDefs	ParallelDefs
Yes		PrimaryKeyFieldValues	PrimaryKeyFieldValues
Yes		RecordCount	RecordCount
Yes		RecordFilter	RecordFilter

Data?	External?	Database: VB	Database: C#
		RulesNavigator	RulesNavigator
		Screens	Screens
Yes		SmallErrorStack	SmallErrorStack
Yes		StorageFormat	StorageFormat
		SystemTypes	SystemTypes
		UsedDictionaries	UsedDictionaries
		UserTypes	UserTypes

Database

For Data?	Database methods: VB	Database methods:C#
Yes	CheckRecord()	CheckRecord()
Yes	ClearRecord()	ClearRecord()
Yes	CreateRecord()	CreateRecord()
Yes	DeleteRecord()	DeleteRecord()
Yes	ReadRecord()	ReadRecord()
Yes	ReadNextRecord()	ReadNextRecord()
Yes	ReadPreviousRecord()	ReadPreviousRecord()
Yes	*ReadRelativeRecord(OffSet)*	*ReadRelativeRecord(OffSet)*
Yes	*ReadRelativeRecords(OffSet, Count, BlaiseRecord)*	*ReadRelativeRecords(OffSet, Count, BlaiseRecord)*
Yes	Reset()	Reset()
Yes	UpdateRecord()	UpdateRecord()
Yes	WriteRecord()	WriteRecord()
	DefinedFields*(BlFieldKind, BlFieldType)*	*get_DefinedFields(BlFieldKind, BlFieldType)*
	DictionaryText*(Language)*	*get_DictionaryText(Language)*
Yes	Errors*(BlErrorKind)*	*get_Errors(BlErrorKind)*
	Fields*(BlFieldKind, BlFieldType)*	*get_Fields(BlFieldKind, BlFieldType)*
Yes	NextFieldOnRoute*(Field, AConditionSet)*	*get_NextFieldOnRoute(Field, AConditionSet)*
Yes	NextPageOnRoute(Question, Toggles)	NextPageOnRoute(Question, Toggles)
Yes	NextQuestionOnRoute(Question, Toggles)	NextQuestionOnRoute(Question, Toggles)
	Field*(Name)*	*get_Field(Name)*
	FieldDef*(Name)*	*get_FieldDef(Name)*
Yes	FirstQuestionOnRoute(Layout, Parallel, Toggles)	FirstQuestionOnRoute(Layout, Parallel, Toggles)
Yes	PreviousFieldOnRoute*(Field, AConditionSet)*	*get_PreviousFieldOnRoute(Field, AConditionSet)*
Yes	PreviousPageOnRoute(Question, Toggles)	PreviousPageOnRoute(Question, Toggles)
Yes	PreviousQuestionOnRoute(Question, Toggles)	PreviousQuestionOnRoute(Question, Toggles)
	QuestionAllowed(Question, Toggles)	QuestionAllowed(Question, Toggles)
	Statement*(Ident)*	*get_Statement(Ident)*

Field

Data?	Field methods: VB	Field methods:C#
	ArrayIndex	ArrayIndex
	Attributes	Attributes
	CategoryTexts(Language)	get_CategoryTexts(Language)
	ColumnTitles	ColumnTitles
	Database	Database
	DataType	DataType

Data?	Field methods: VB	Field methods:C#
	DefinedDescriptionText(Language)	get_DefinedDescriptionText(Language)
	DefinedFields(BlFieldKind, BlFieldType)	get_DefinedFields(BlFieldKind, BlFieldType)
	DefinedQuestionText(Language)	get_DefinedQuestionText(Language)
	DefinedRowTitles	DefinedRowTitles
	DefinedTag	DefinedTag
	DescriptionText(Language, Substitute)	get_DescriptionText(Language, Substitute)
	DisplayText	DisplayText
	DontKnowAllowed	DontKnowAllowed
	Editable	Editable
	EditInformation	EditInformation
	EditMode	EditMode
	EnumerationDisplayString	get_EnumerationDisplayString(ShowLabels)
	ErrorCounter(BlErrorKind)	get_ErrorCounter(BlErrorKind)
	Errors(ErrorKinds)	get_Errors(ErrorKinds)
	FieldDef	FieldDef
	FieldKind	FieldKind
Y	Fields(BlFieldKind, BlFieldType)	get_Fields(BlFieldKind, BlFieldType)
	IndexedName	IndexedName
	IsKeyField(BlKeyKind)	get_IsKeyField(BlKeyKind)
	IsParallel	IsParallel
	LocalName	LocalName
	Name	Name
	Parent	Parent
Y	QuestionText(Language, Substitute)	get_QuestionText(Language, Substitute)
	ReferredFieldName	ReferredFieldName
	RefusalAllowed	RefusalAllowed
	Remarked	Remarked
	RemarkText	RemarkText
	Required	Required
	RouteStatus	RouteStatus
	SetDisplayString	SetDisplayString
	Size	Size
	Status	Status
	Tag	Tag
Y	Text	Text
	TextAsSet	TextAsSet
Y	Value	Value
	AheadOnRoute(Field)	AheadOnRoute(Field)
	CopyDitto()	CopyDitto()
	Validate	Validate

FieldDef

FieldDef methods: VB	FieldDef methods:C#
BlockText(Language)	get_BlockText(Language)
Categories	Categories
Classification	Classification
DataType	DataType
Decimals	Decimals
DisplayDataType	DisplayDataType
DisplayElementDataType	DisplayElementDataType

ExternalInformation	ExternalInformation
IsArray	IsArray
IsEmbeddedBlock	IsEmbeddedBlock
IsSet	IsSet
IsTable	IsTable
LocalName	LocalName
MaxIndex	MaxIndex
MaxValue	MaxValue
MinIndex	MinIndex
MinValue	MinValue
Name	Name
RulesText (WithGeneratedParams, WithGeneratedRules)	get_RulesText(WithGeneratedParams, WithGeneratedRules)
Size	Size
Statements	Statements
TypeInformation	TypeInformation
CatNameToInt (Value)	CatNameToInt(Value)
GetDisplayText (FieldValue)	GetDisplayText(FieldValue)
IntToCatName(Value_)	IntToCatName(Value_)
ValidateClassificationCode(Value_)	ValidateClassificationCode(Value_)

Categories

Field methods: VB6	Field methods:C#/VB.NET
FieldDef.Categories.Count	FieldDef.Categories.Count
FieldDef.Categories.IndexOf(Name)	FieldDef.Categories.IndexOf(Name)
FieldDef.Categories[].Code	FieldDef.Categories[].Code
FieldDef.Categories[].Name	FieldDef.Categories[].Name
FieldDef.Categories[].Text(Language)	FieldDef.Categories[].get_Text(Language)
FieldDef.Categories[].TextDefined(Language)	FieldDef.Categories[].get_TextDefined(Language)

8. Appendix B

Differences between VB6 and .Net

For programmers familiar with VB6 and not so much .Net there are a few pitfalls that can be avoided with some care.

- Indexing
  VB6 has a base index of "1" and so does the BCP, such as Fields[1].IndexedName. .Net works with a base index of "0," so stepping through an indexed variable is slightly different.

VB6

```
Set oFields = _Database.DefinedFields(blfkAll, blftAll)

For I = 1 To oFields.count
   curName(I) = oFields(I).IndexedName
Next I
```

VB.Net

```
oFields = _Database.DefinedFields(BlFieldKind.blfkAll,
                                  BlFieldType.blftAll)

For I = 0 To oFields.Count - 1
   curName(I) = oFields(I - 1).IndexedName
Next I
```

C#.Net

```
oFields = _Database.get_DefinedFields((int)BlFieldKind.blfkAll,
                                      (int)BlFieldType.blftAll);

for (int I = 0; I < oFields.Count; I++)
   {
        curName[I] = oFields[I - 1].IndexedName;
   }
```

- Case sensitivity
  .Net is case-sensitive. This means that "count" is not the same variable as "Count." The compiler will let you know.

- Strong typing
  Assignments and parameters must have the same type. VB6 allowed free assignment between strings and integers and behind the scenes did the conversion. This sometimes lead to unexpected results.

- Defined variables
  All variables used in .Net must be declared, and initialized depending upon their use.

- Object orientation
  .Net is strongly object-oriented. It has a large impact on how memory and processing time works in the programs written. Follow good programming practices and declare and use objects locally, dispose or assign them to null/Nothing when finished, avoid heavy string use but use string builders, and use the IDisposable inheritance on your own classes.

9. Recommendations for working with .Net

Error trapping

It is highly recommended that you use Try/Catch/Finally for each procedure you write and do your best to handle errors. Use a simple log file and write errors to the log, especially for complex routines. This will greatly reduce the time and effort of making your program robust.

Debugging log

By the same regard, create a debugging log that is written when a debugging flag has been set.

.Net Development Environment

This is a very powerful tool for developing your applications, and again a great deal of time can be saved by learning as many features of the Visual Studio environment as possible.

Basic BCP/API recommendations

Retrieve collections all at the same time if you can afford the memory use, and then step through the collection using an integer index. This will allow you to quickly step through all the information. However, be warned: retrieving large quantities of data at one time makes a hit on your system resources.

Although the BCP/API does a good job of releasing objects, it behaves like strings in the .Net environment. That is, the objects are collected eventually when garbage collection happens. You can force garbage collection earlier by the GC() method. For this reason limit creation of new objects to those actually needed, keep the object definition as local as possible, avoid global objects that are held around a long time, and create classes that use the Dispose() inheritance.

10. Appendix C

10.1 Additional .Net tips

Shorthand for if/else

```
curDatabaseManager.DictionarySearchPath = pos > -1 ? txtBMIFile.Text.Substring(0, pos) : "\\";
```

Escaping string sequences

Use the backslash for each escape, such as '\\' to get the character '\', or the @ sign in front of a string to escape all the dereferences, such as @"c:\program files\thisandthat\sample.txt."

Adding a BCP/API reference to your .Net solution.

- Create a solution.
- Next, open the Solution Explorer and open the References tree. Right-click and Add a Reference. Choose the "COM" tab, and select the "Blaise 4.7 API Component" item. This will add both the BLAPI4A2.dll and the SiStrClA.dll to your project.
- If you don't like typing the BLAPI4A2 class name repeatedly you can make a reference at the top of your C# program with the using statement, or the imports statement in VB.