Python database access

Ken Zook

August 27, 2009

Contents

1 Introduction

IronPython is a free Python program based on .NET Runtime 2.0, so it allows calls to the FDO layer of FieldWorks. This allows Python programs to access and modify the database similar to C# and other programs that interface with the .NET Runtime. Python can also be used in the bulk edit tools in Flex. See Bulk edit issues.doc, section 5.5.1 for information on this.

Note: In order to support the possibility of using the Firebird database engine in addition to Micrsoft SQL Server, and due to limited length of names in Firebird, some class, property, and procedure names were shortened in FieldWorks 5.4 compared to earlier versions. The spreadsheet, Model name changes.xls, lists the changes that were made. If you had programs for older versions you may need to make a few of these name changes for it to continue to work in FieldWorks 5.4 and later.

Caution: As with any method for modifying the database outside of a FieldWorks program, if you do not know what you are doing, you can inadvertently damage the data. FieldWorks applications may no longer run or it could do damage in a way that will not show up until later. You can freely use IronPython to explore existing data, but be *extremely* cautious about making *any* changes to the database. Any time you plan to do this, make sure you *first* back up the data and check carefully what you did before going on. It is safer to use FDO via IronPython than using SQL directly in the database. However, it *is still* possible to cause irreparable damage with either approach.

Other versions of Python that can handle COM interfaces adequately should be able to work with ISilDataAccess, IVwOleDbDA, and IVwCacheDa, IOleDbEncap, and TsString-related COM interfaces to access and modify the database without using FDO. See http://fieldworks.sil.org/objectweb for details on these interfaces.

2 Installation

You can download a free copy of IronPython at www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython. Here is how to install IronPython:

1. Download IronPython-1.0.1-Bin.zip from the web site.
2. Unzip the file into a directory (e.g., in a Python directory under the FieldWorks directory). You will probably want to move the files from the IronPython-1.0.1 directory up to the Python directory to make access easier.
3. If desired, add the directory containing ipy.exe to your path to make access easier.
4. To allow IronPython to work with SIL Encoding Converters, create this registry key:
   HKEY_LOCAL_MACHINE\SOFTWARE\Python\PythonCore\2.5
   and add an InstallPath string variable that gives the path to your ipy.exe.

3 FDO information

IronPython accesses the FieldWorks database through the FieldWorks Data Objects (FDO) layer. This is an object-oriented business layer that makes access to the database more intuitive. FDO is written in C# based on .NET 2.0, so its methods can be accessed from IronPython.

FDO provides

- generated classes for every class in the FieldWorks conceptual model
- generated methods on those classes for every property defined in the FieldWorks conceptual model, and
- many hand-written methods on common classes providing additional functionality.

See fdoHelp.doc for an introduction to FDO. The current FDO programmer documentation is in FDO.chm, which was generated from the C# source code. Additional COM interfaces needed to deal with Strings and other information are in COMInterfaces.chm. They were also generated from C# source code.

In FDO.chm, use the index to find any FieldWorks class and get a list of all members on that class. This includes the generated property methods as well as hand-written methods. For each one, follow the link to discover the arguments to the method and the return information. Use a similar process in COMInterfaces.chm to get information on COM interfaces.

4 Getting started in IronPython

To start IronPython, open a DOS box on the c:\Program Files\SIL\FieldWorks directory so it can access FDO.dll and related FieldWorks DLLs. To run the program from the DOS box, type 'ipy' (including a path if ipy.exe cannot be located along the path). This gives you an IronPython prompt (e.g., >>>) in which you type Python commands. To exit from IronPython, type Ctrl+Z and press Enter.

To access a FieldWorks database in IronPython:

1. Import the .NET Common Language Runtime system, then import FDO.dll.
2. Import System methods if you plan to use time or GUID properties.
3. In order to create FieldWorks objects and access all their methods, import the different database modules, and COMInterfaces.
4. Get a cache object on the database you want to access, then get the LangProject from the cache.
5. The following IronPython commands can be entered at the Python prompt to do these things.:

```
import clr
clr.AddReference("FDO")
from SIL.FieldWorks.FDO import *
from System import *
from SIL.FieldWorks.FDO.Cellar import *
from SIL.FieldWorks.FDO.Ling import *
from SIL.FieldWorks.FDO.Scripture import *
from SIL.FieldWorks.FDO.Notebk import *
from SIL.FieldWorks.FDO.LangProj import *
clr.AddReference("COMInterfaces")
from SIL.FieldWorks.Common.COMInterfaces import *
cache = FdoCache.Create("TestLangProj")
lp = cache.LangProject
```

Note: For FieldWorks 4.0.1 or earlier, you'll also need to insert these additional lines before the COMInterfaces lines above:

```
clr.AddReference("FwKernelLib")
from FwKernelLib import *
.from SIL.FieldWorks.FDO.Cellar.Generated import *
from SIL.FieldWorks.FDO.Ling.Generated import *
from SIL.FieldWorks.FDO.Scripture.Generated import *
```

```
from SIL.FieldWorks.FDO.Notebk.Generated import *
from SIL.FieldWorks.FDO.LangProj.Generated import *
```

In the Create method above, use the name of your database. From the LangProject object, you can navigate the ownership hierarchy to get to any object owned by the language project. The FDO cache holds information read from the database. When you make changes to the cache, these changes are immediately written to the database.

Note: Case is critical when dealing with class and method names.

IronPython provides some support for investigating objects. After getting the 'lp' object above representing the language project, enter this command:

```
dir(lp)
```

This gives an alphabetical list of the methods available. One of these is LexDbOA. You can get some information on this method with this command:

```
print lp.LexDbOA.__doc__
```

You can also find out what class a variable represents:

```
lp.__class__
```

You get more helpful information by referring to FDO.chm.

There are three ways you can run an IronPython program:

A. Start IronPython in a DOS box with 'ipy', then type your Python commands directly at the IronPython prompt.
B. Enter the Python commands into a text file and start IronPython in a DOS box with 'ipy'. Then copy your commands from the text file and paste them into the DOS box at the IronPython prompt and press Enter to execute the last line (when needed). To paste them into a DOS box, right-click and choose Paste.
C. Enter the Python commands into a text file (e.g., test.py) and execute the text file directly using this DOS command:

```
ipy test.py
```

4.1 Working with Python.NET

(This section is thanks to Craig Farrow.) Python.NET can also be used to access the FDO.dll. Python.NET supports standard Python C libraries such as CElementTree (used by the NLTK library) and others, which aren't supported in IronPython. However, there are some differences in the way Python.NET and IronPython behave:

- The DLL has to be on the path for Python.NET to import the namespace. IronPython doesn't seem to care.
- IronPython automatically references ScrFDO.dll, but Python.NET doesn't, so it has to be explicitly referenced.

The following code snippet works in both versions of Python to get access to FdoCache:

```
# This line seems to be the best way to get IronPython
# and Python.NET to both work:
# * IronPython doesn't support .pth files;
# * Python.NET doesn't import the name-space without the DLL being
#   on the path.
import sys
sys.path.append("c:\\program files\\sil\\fieldworks\\")

import clr
```

```
# For some unknown reason FdoCache create fails with a TypeLoadException
# under Python.NET if ScrFDO is not added here.
# IronPython works fine without this.
clr.AddReference("FDO")
clr.AddReference("ScrFDO")

from SIL.FieldWorks.FDO import FdoCache

db = FdoCache.Create() # No name opens the first db on the default server
print "\t\t", db.ServerName, db.DatabaseName
```

5 Working with owning properties

FDO suffixes owning properties to indicate the type of property:

- -OA ⇨ owning atomic
- -OC ⇨ owning collection
- -OS ⇨ owning sequence

This program prints the headword for all entries in the lexicon:

```
import clr
clr.AddReference("FDO")
from SIL.FieldWorks.FDO import *
cache = FdoCache.Create("TestLangProj")
lp = cache.LangProject
lexicon = lp.LexDbOA
for entry in lexicon.EntriesOC :
  print entry.ReferenceName
```

Since LangProject_LexDb is an atomic owning property, FDO provides the LexDbOA method to return the LexDb. LexDb_Entries is an owning collection, so FDO provides the EntriesOC method to access this property. The 'for' loop in this example gets each entry and prints the ReferenceName, which actually returns a string with the HeadWord. FDO provides a HeadWord method that returns a TsString, but IronPython cannot instantiate TsStrings at this point.

If you want to instantiate a single entry and the first sense in that entry, instead of processing the loop, use these methods, assuming 'lexicon' is set to the LexDb:

```
hvoEntry = lexicon.EntriesOC.HvoArray[0]
entry = CmObject.CreateFromDBObject(cache, hvoEntry)
hvoSense = entry.SensesOS.HvoArray[0]
sense = CmObject.CreateFromDBObject(cache, hvoSense)
```

The EntriesOC method on LexDb returns an FdoOwningCollection class. One of these methods is HvoArray, which returns an array of integers corresponding to the database Ids of the owned objects. Hvo in FieldWorks stands for 'Handle to a Viewable Object', but anymore this is usually represents the object ids from the database. Once you have the Database Id, use the CreateFromDBObject method on CmObject to instantiate that object in the cache and load in basic properties from the database. In the first case above, Flex provides an entry object representing the first entry in the Entries array.

The SensesOS method on LexSense returns an FdoOwningSequence class. One method is HvoArray, which returns an array of database Ids. The example above picks the first one and instantiates a LexSense from this. With these entry and sense objects, you can access the properties on each one.

Once you have an object, you may want to access various information about that object:

```
print sense.Hvo
print sense.OwnerHVO
print sense.OwningFlid
print sense.OwnOrd
print sense.ClassID
```

- Hvo returns the Hvo or database Id of the sense.
- OwnerHVO returns the Hvo of the owning object.
- OwningFlid returns the owning Field Id (flid) on the owning object that owns the sense.
- OwnOrd returns the ord value from the database that determines the sequence in a sequence property.
  Note: Numbers do not have to be sequential.
- ClassID returns the 'clsid' for the sense.

Any time you need to know what methods are available, or the arguments and return values for methods, refer to FDO.chm.

6 Working with reference properties

FDO suffixes reference properties to indicate the type of property:

- -RA ⇨ reference atomic
- -RC ⇨ reference collection
- -RS ⇨ reference sequence

In many places you need an LgWritingSystem Id to pick from multilingual properties. You can get this information from the LangProject 'lp' in several ways:

```
wsv = lp.CurVernWssRS.HvoArray[0]
wsa = lp.CurAnalysisWssRS.HvoArray[0]
```

The CurVernWssRS method on LangProject is a reference sequence property that returns an FdoReferenceSequence class. Use the HvoArray method on this class to get the first writing system Hvo. You can also use the CmObject.CreateFromDBObject method as illustrated above to get the actual LgWritingSystem object. This is *not* needed to access multilingual properties—only the Hvo.

Simpler methods can give the top vernacular and top analysis writing system values, as long as you only need one writing system from each property:

```
wsv = lp.DefaultVernacularWritingSystem
wsa = lp.DefaultAnalysisWritingSystem
```

The SemanticDomainsRC method on LexSense returns an FdoReferenceCollection containing the semantic domains referenced by the sense. Once you have a sense, use this code to print the names of the semantic domains for the sense:

```
for sem in sense.SemanticDomainsRC :
  print sem.Name.GetAlternative(wsa)
```

The SenseTypeRA method on LexSense is an atomic reference property that returns a CmPossibility:

```
sensType = sense.SenseTypeRA
```

For both owning and reference properties, atomic properties return an actual instance of an object, while the collection and sequence properties return an array of Hvos that *must* be instantiated if you want the actual objects.

Suppose you need to set the sense type for a sense to the ‘primary’ CmPossibility in the Sense Types list. This takes more work than doing the same thing in SQL. You need to first go to the list and then iterate through the items to find the desired one, instead of simply using a query to return the item with the desired name. One way to do this, assuming you already have the lexical database (lexicon), sense (sense), and default analysis ws (wsa):

```
stList = lexicon.SenseTypesOA
for st in stList.PossibilitiesOS :
  if st.Name.GetAlternative(wsa) == "primary" :
    break
sense.SenseTypeRA = st
```

This gets the Sense Type list from the SenseTypesOA property of LexDb. It then iterates through the PossibilitiesOS property looking for one with a ‘primary’ name. It finds it and then breaks from the loop and sets the sense type for the sense to that item. Other complications have not been addressed. For example, what should happen if the list does not contain the desired item? If the list is hierarchical, you also need to recursively search through the SubPossibilitiesOS property for each item as well.

To add or remove items in reference sequences or collections, look at FDO methods on FdoReferenceSequence, FdoReferenceCollection, and FdoVector.

7 Working with basic properties

7.1 Strings

FieldWorks uses various methods for storing strings on objects, depending on the amount of information it needs to store. See ‘Conceptual model overiew.doc’ section 1.5.1 for background on this.

The Python console is very limited in showing Unicode characters. Python defaults to ASCII when dealing with str string functions (e.g., str.upper(s)). When dealing with Unicode strings you should use unicode functions (e.g., unicode.upper(s)). When specifying a Unicode string in source code, you should add ‘u’ in front of the quoted string. (e.g., u'αββα'). However, if you paste this in the console mode the Beta (U+3b2) will be incorrectly converted to ANSI \xdf. In a Unicode string, you can mix actual Unicode code points with quoted Unicode values (e.g., u'α\u03b2\u03b2α' is the same as the previous string but will work properly in console mode). \u is used with 4 hexadecimal digits to represent values up to U+FFFF. For higher values, use \U plus 8 hexadecimal digits (e.g., u'α\U000F2090βα'). By writing values to a UTF-8 encoded file, you can obtain the real Unicode values without the console mode incorrectly converting them. See Section 10 for an example of this.

7.1.1 FieldWorks strings

Whenever you access a String, BigString, MultiString, or MultiBigString property, you will need to work with some underlying COM interfaces that allow FieldWorks to embed formatting, writing systems, and other objects inside the string. These interfaces are discussed in sections 8.2 and 8.3.

Note that FieldWorks stores strings in Unicode NFD (normalization decomposed) format. When you get a string from the database it will be in NFD, so your Python program will

need to handle this properly. When storing strings, FieldWorks will automatically convert any string to NFD.

7.1.2 MultiUnicode/MultiBigUnicode

LexSense_Gloss is an example of a MultiUnicode string. Assuming 'sense' is a sense object and 'wsa' is the analysis writing system, this command retrieves the string from the gloss:

```
gloss = sense.Gloss.GetAlternative(wsa)
```

This sets the gloss for the wsa writing system:

```
sense.Gloss.SetAlternative("apple", wsa)
```

7.1.3 MultiString/MultiBigString

LexSense_Definition is an example of a MultiString FieldWorks String. Assuming 'sense' is a sense object and 'wsa' is the analysis writing system, this command retrieves the raw Unicode string from the definition:

```
definition = sense.Definition.GetAlternative(wsa).Text
```

If you want details on the formatting of a string, you'll need to get a TsString and use properties and methods described in section 8.2 to get the formatting. The following method returns a TsString for the defintion.

```
tss = sense.Definition.GetAlternativeTss(wsa)
```

To set a MultiString or MultiBigString value, you first need to construct a TsString with appropriate formatting. Section 8.3 describes methods for creating TsStrings. You should make sure every TsString defines a writing system for all code points in the string. Here's one way to set the definition for a sense.

```
tisb = TsIncStrBldrClass.Create()
tisb.SetIntPropValues(FwTextPropType.ktptWs.value__, FwTextPropVar.ktpvDefault.value__, wsa)
tisb.Append("A small round object.")
sense.Definition.SetAlternative(ITsIncStrBldr.GetString.Call(tisb), wsa)
```

Note, it is possible to set a definition using the following code, but this should never be used because the string will have no formatting, which makes it invalid.

```
sense.Definition.SetAlternative("A small round object.", wsv)
```

7.1.4 String/BigString

LexSense_ScientificName is an example of a FieldWorks String. Assuming 'sense' is a sense object, this command retrieves the raw Unicode string from the scientific name:

```
name = sense.ScientificName.Text
```

If you want details on the formatting of the string, you'll need to get a TsString and use properties and methods described in section 8.2 to get the formatting. The following method returns a TsString for the scientific name.

```
tss = sense.ScientificName.UnderlyingTsString
```

To set a String or BigString value, you first need to construct a TsString with appropriate formatting. Section 8.3 describes methods for creating TsStrings. You should make sure every TsString defines a writing system for all code points in the string. Here's one way to set the scientific name for a sense.

```
tisb = TsIncStrBldrClass.Create()
tisb.SetIntPropValues(FwTextPropType.ktptWs.value__, FwTextPropVar.ktpvDefault.value__, wsa)
```

```
tisb.Append("tulip.")
sense.ScientificName.UnderlyingTsString = ITsIncStrBldr.GetString.Call(tisb)
```

Note, it is possible to set a scientific name using the following code, but this should never be used because the string will have no formatting, which makes it invalid.

sense.ScientificName.Text = "tulip"

7.1.5 Unicode/BigUnicode

CmPossibility_HelpId is an example of a Unicode string. Assuming ‘poss’ is a CmPossibility object, this command retrieves the string from the HelpId:

```
help = poss.HelpId
```

This sets the HelpId:

```
poss.HelpId = “ARC203”
```

7.2 Other Properties

7.2.1 Booleans

LexEntry_ExcludeAsHeadword is an example of a Boolean property. Assuming ‘entry’ is an entry object, this command retrieves the Boolean result:

```
bool = entry.ExcludeAsHeadword
```

This sets the Boolean:

```
entry.ExcludeAsHeadword = “true”
```

7.2.2 Integers

LexEntry_HomographNumber is an example of an integer property. Assuming ‘entry’ is an entry object, this command retrieves the homograph number:

```
hom = entry.HomographNumber
```

This sets the Boolean:

```
entry.HomographNumber = 1
```

7.2.3 Time

LexEntry_DateCreated is an example of a time property. Assuming ‘entry’ is an entry object, this command retrieves the create time:

```
time = entry.DateCreated
```

The time returned here is a .NET DateTime object. Its methods are available when you execute the Python ‘from System import *’ command.

This sets the create date—the first to the current time and the second to any specified time:

```
entry.DateCreated = DateTime.Now
entry.DateCreated = DateTime.Parse("11/7/2006 12:45:58 PM")
```

7.2.4 GUIDs

LexEntry_Guid is an example of a GUID property. Assuming ‘entry’ is an entry object, this command retrieves the GUID:

```
guid = entry.Guid
```

The GUID returned here is a .NET GUID object. Its methods are available when you execute the Python 'from System import *' command.

This sets the GUIDs—the first to a new GUID and the second to any specified instance:

```
entry.Guid = Guid.NewGuid()
entry.Guid = Guid("edef982a-f69a-4793-95fb-f4398e4a2ddf")
```

7.2.5 GenDates

RnEvent_DateOfEvent is an example of a GenDate. At this point, FDO has not implemented access to GenDates, so you cannot use it to read or set these values.

7.2.6 Binary

Binary fields are implemented in FDO in different ways. Fields that hold style information return an ITsTextProps interface. However, IronPython fails to instantiate this COM interface at this point. UserView_Details is a property that returns a .NET Byte Structure. Assuming 'uv' is a UserView object, this returns the Byte Structure:

```
bytes = uv.Details
```

Using the print command, it prints 'System.Byte[]5, 0, 0, 0)'.

This sets the property with the Byte Structure:

```
uv.Details = bytes
```

8 Working with COM interfaces

FieldWorks uses COM interfaces for a number of the underlying functions such as working with FieldWorks strings and accessing the MetaDataCache that provides information about classes and properties. Some COM interfaces implement IDispatch, and are easier to access via Python. Most of these underlying interfaces do not currently implement IDispatch. For these interfaces, you need to use GetValue and Call methods.

To access properties, you would use Interface.PropertyName.GetValue(comObject).

To execute methods, you would use Interface.PropertyName.Call(comObject, parameters…).

For definition of string-related COM interfaces, see ComInterfaces.chm. For definitions of text properties, see TextServ.idh.

8.1 Accessing class/property metadata information

In section 5 we described how you can access the ClassId and OwningFlid for any object in the databse. If you want to find out more information about the class and field definitions, you need to use the MetaDataCache. From an FDO cache, you can access a MetaDataCache as follows:

```
mdc = cache.MetaDataCacheAccessor
```

From the MetaDataCache you can access various information about classes and senses. For information on MetaDataCache methods, refer to IFwMetaDataCache in ComInterfaces.chm. For examples, the following methods retrieve the name of class 13 (CmPerson), the name of field 13001 (Alias), number of fields defined in the model (around 821), and the number of classes in the model (around 173).

```
mdc.GetClassName(13)
```

```
mdc.GetFieldName(13001)
mdc.FieldCount
mdc.ClassCount
```

COM interfaces that return arrays need to use special marshalling to allow Python to access the unmanaged memory used by the underlying COM code. The following code produces a list of classes, giving the class number and name for each class.

```
clidCount = mdc.ClassCount
clidList={}
clidListSize = clidCount - 1
clidArray = MarshalEx.ArrayToNative(clidCount, int)
mdc.GetClassIds(clidCount,clidArray)
clidList=MarshalEx.NativeToArray(clidArray, clidCount, int)
for id in clidList:
  print id, mdc.GetClassName(id)
```

The following code produces a list of fields, giving the field number and name for each class.

```
flidCount = mdc.FieldCount
flidList={}
flidListSize = flidCount - 1
flidArray = MarshalEx.ArrayToNative(flidCount, int)
mdc.GetFieldIds(flidCount,flidArray)
flidList=MarshalEx.NativeToArray(flidArray, flidCount, int)
for id in flidList:
  print id, mdc.GetFieldName(id)
```

The FDO cache provides some methods that access MetaDataCache information directly without making the calls directly from the MetaDataCache (e.g., GetClassName, GetFieldsOfClass, and GetFieldType).

8.2 Accessing FieldWorks strings

When you use a String, BigString, MultiString, or MultiBigString property, you will need to work with underlying COM interfaces that allow FieldWorks to embed formatting, writing systems, and other objects inside the string. These interfaces are described in ComInterfaces.chm. To access information from a string, you use the ITsString interface.

Assuming tss is a TsString that you have created or accessed from a string property, you can get the raw Unicode characters using the Text property (entire string) or GetChars method (range from begin offset to end offset) of the ITsString interface.

```
ITsString.Text.GetValue(tss)
ITsString.GetChars.Call(tss,0,3)
```

The number of characters in the string can be obtained using the Length property.

```
print ITsString.Length.GetValue(tss)
```

A TsString holds a sequence of one or more runs. All of the code points in a given run share the same properties, such as writing system, style, and formatting. ITsString has a number of methods for accessing runs within the string. You can iterate through the runs using the RunCount property (returns the number of runs in the string) and the get_RunText method (returns the raw Unicode characters in the run).

```
for irun in range(ITsString.RunCount.GetValue(tss)):
  print ITsString.get_RunText.Call(tss, irun)
```

To obtain the properties of a run or a single code point within a run, you use the ITsTextProps interface which provides methods for accessing a bundle of text properties. There are two basic types of text properties: integer properties and string properties.

Integer properties return an integer that represents a writing system, a point size, an enum indicating bold or italic, and similar types of integer property. A string property returns a Unicode string that typically represents a style name, font name, or other string property. The get_Properties method returns the properties for a given run, while the get_PropertiesAt method returns the properties of a code point at a certain offset in a string.

```
runProps = ITsString.get_Properties.Call(tss, 1)
runProps = ITsString.get_PropertiesAt.Call(tss, 35)
```

Once you have the bundle of properties, the ITsTextProps interface provides methods to access the properties. The IntPropCount and StrPropCount properties return the number if integer and string properties contained in the bundle.

```
ITsTextProps.IntPropCount.GetValue(runProps)
ITsTextProps.StrPropCount.GetValue(runProps)
```

The GetIntProp method returns an integer property at the specified offset within the bundle of integer properties. This method returns an array of three integers. You can access the three parts using array indexes as shown below. The first item in the array is the value of the property, the second is the type of property, and the third is the variant value of the property. This first example illustrates a writing system integer property with a type of 1 (ktptWs), a variant of 0 (ktpvDefault) and a value of 40717 (the id of the writing system for this run).

```
ITsTextProps.GetIntProp.Call(runProps, 0)
#(40717, 1, 0)
ITsTextProps.GetIntProp.Call(runProps, 0)[0]
#40734 = value = writing system id
ITsTextProps.GetIntProp.Call(runProps, 0)[1]
#1 = type = FwTextPropType.ktptWs.value__
ITsTextProps.GetIntProp.Call(runProps, 0)[2]
#0 =variant = FwTextPropVar.ktpvDefault.value__
```

This example illustrates an italic integer property with a type of 2 (ktptItalic), a variant of 3 (ktpvEnum) and a value of 2 (kttvInvert) which flips the italic flag from what it was before.

```
ITsTextProps.GetIntProp.Call(runProps, 0)
#(2, 2, 3)
ITsTextProps.GetIntProp.Call(runProps, 0)[0]
#2 = value = FwTextToggleVal.kttvInvert.value__
ITsTextProps.GetIntProp.Call(runProps, 0)[1]
#2 = type = FwTextPropType.ktptItalic.value__
ITsTextProps.GetIntProp.Call(runProps, 0)[2]
#3 =variant = FwTextPropVar.ktpvEnum.value__
```

The GetStrProp method returns a string property at the specified offset within the bundle of string properties. This method returns an array of two items. You can access the two parts using array indexes as shown below. The first item in the array is the string value of the property, the second is the type of string property. This example illustrates a named style property that causes the text to default to the formatting contained in the specified style from a stylesheet. The property type is 133 (ktptNamedStyle) and the value is ‘Emphasized Text’ which is defined in the style sheet for the lexicon.

```
ITsTextProps.GetStrProp.Call(runProps, 0)
#('Emphasized Text', 133)
ITsTextProps.GetIntProp.Call(runProps, 0)[0]
# Emphasized Text = value = name of a style
```

```
ITsTextProps.GetIntProp.Call(runProps, 0)[1]
#133 = type = FwTextPropType. ktptNamedStyle.value__
```

8.3 Creating FieldWorks strings

There are several ways to create TsStrings. If you only need a single run with no properties other than a writing system, you can use the TsStrFactory MakeString method. The following example uses this method to create a string using the wsa writing system.

```
tsf = TsStrFactoryClass.Create()
tss2 = ITsStrFactory.MakeString(tsf,"A simple string.", wsa)
ITsString.Text.GetValue(tss2)
#'A simple string.'
```

Note: For FieldWorks 4.0.1 or earlier, instead of using TsStrFactoryClass.Create(), you'll need to use TsStrFactoryClass().

If you need more than a simple string, you would normally use the incremental string builder (ITsIncStrBldr interface). With this string builder, you set the properties you want, append the text for that property, then set/clear properties for the next run, append text for that run. After building the string, use the GetString method to return the final TsString. The following example creates a string 'This is a discussion about las casas.' where the writing system for 'las casas' is the vernacular writing system and the rest of the string is the analysis writing system. The word, 'discussion', uses the Emphasized Text style, which would be italic in the lexicon, if you you haven't redefined this style. In addition to 'las casas' being in a different writing system, the italic flag for this text is inverted from the normal text. In this case, since the surrounding text is not italic, 'las casas' will be italic.

```
tisb = TsIncStrBldrClass.Create()
tisb.SetIntPropValues(FwTextPropType.ktptWs.value__, FwTextPropVar.ktpvDefault.value__, wsa)
#SetIntPropValues(int tpt, int nVar, int nVal) where tpt = 1, nVar = 0, nVal = 40734
tisb.Append("This is a ")
tisb.SetStrPropValue(FwTextPropType.ktptNamedStyle.value__, "Emphasized Text")
tisb.Append("discussion")
tisb.SetStrPropValue(FwTextPropType.ktptNamedStyle.value__, "")
tisb.Append(" about ")
tisb.SetIntPropValues(FwTextPropType.ktptWs.value__, FwTextPropVar.ktpvDefault.value__, wsv)
tisb.SetIntPropValues(FwTextPropType.ktptItalic.value__, FwTextPropVar.ktpvEnum.value__,
FwTextToggleVal.kttvInvert.value__)
tisb.Append("las casas")
tisb.SetIntPropValues(FwTextPropType.ktptWs.value__, FwTextPropVar.ktpvDefault.value__, wsa)
tisb.Append(".")
tss = tisb.GetString()
```

When getting strings from builders, sometimes the simple tisb.GetString() method will work, but at other times you need to specify the more precise code ITsIncStrBldr.GetString.Call(tisb).

Note: For FieldWorks 4.0.1 or earlier, instead of using TsIncStrBldrClass.Create(), you'll need to use TsIncStrBldrClass().

If you want to modify an existing string, you would use an TsStrBldr interface that allows you to modify an existing string by inserting or deleting text, or changing properties for some range within the string. The following example continues from the previous example and changes 'is' to 'was' in the tss string.

```
tsb = ITsString.GetBldr.Call(tss)
ITsStrBldr.RunCount.GetValue(tsb)
#5
ITsStrBldr.Text.GetValue(tsb)
```

```
#'This is a discussion about las casas.'
runProps = ITsStrBldr.get_PropertiesAt.Call(tsb,5)
ITsStrBldr.Replace.Call(tsb,5,7,"was",runProps)
ITsStrBldr.Text.GetValue(tsb)
#'This was a discussion about las casas.'
ITsStrBldr.RunCount.GetValue(tsb)
#5
tss = ITsStrBldr.GetString(tsb)
ITsString.Text.GetValue(tss)
#'This was a discussion about las casas.'
```

A TsStrBldr has many of the same functions as a TsString as far as examining runs, text, and properties. But in addition, it allows you to make changes to the copy of the string in the TsStrBldr. Notice the final argument for the Replace method is an ITsTextProps interface. In this case we get the properties from the first character we are replacing and use those properties for replaced text. When done, you can get the completed string out of the TsStrBldr using the GetString property.

Other interfaces for working with text properties are ITsPropsBldr and ITsPropsFactory.

9 Adding and deleting objects

9.1 Adding objects

Assuming ‘entry’ is an entry object, you can add a new LexEtymology object to the entry using

```
ety = LexEtymology()
entry.EtymologyOA = ety
```

For this to work, you have to make sure the name space that defines LexEtymology is loaded. For FieldWorks 4.0.1 and earlier, you would use

```
from SIL.FieldWorks.FDO.Ling import *
from SIL.FieldWorks.FDO.Ling.Generated import *
```

For versions after FieldWorks 4.0.1 you don’t need the Generated line.

Here’s a more complete example that adds a LexEtymology object to the entry, along with filling in information on the LexEtymology.

```
ety = LexEtymology()
entry.EtymologyOA = ety
ety.Form.SetAlternative("formal", wsa)
ety.Source = "French"
tisb = TsIncStrBldrClass.Create()
tisb.SetIntPropValues(FwTextPropType.ktptWs.value__, FwTextPropVar.ktpvDefault.value__, wsa)
tisb.Append("This was borrowed from French, which borrowed the word from Germanic")
ety.Comment.SetAlternative(tisb.GetString(), wsa)
```

9.2 Deleting objects

To delete an object, use the DeleteObject method on the cache which takes an ‘hvo’. This deletes the object, along with *everything* it owns and removes references to it. In most cases this is sufficient. For some things when you delete one object, some other object should also be deleted since it is no longer needed.

Assuming ‘sense’ is a sense you want to delete, you can execute this command:

```
cache.DeleteObject(sense.Hvo)
```

For an atomic owning property, you can also use this form of the command:

```
cache.DeleteObject(entry.EtymologyOAHvo)
```

10 Example dumping information from the lexicon

Here is a simple IronPython program that lists all lexical entries with their top-level senses listing the entry headword and sense gloss and definition for each one, separating the fields with a semicolon and space. Note this method of listing the definition ignores any formatting on the definition, but just returns the raw Unicode characters.

```
import clr
clr.AddReference("FDO")
from SIL.FieldWorks.FDO import *
cache = FdoCache.Create("TestLangProj")
lp = cache.LangProject
lexicon = lp.LexDbOA
wsv = lp.DefaultVernacularWritingSystem
wsa = lp.DefaultAnalysisWritingSystem
for entry in lexicon.EntriesOC :
  headword = entry.ReferenceName
  for sense in entry.SensesOS :
    glos = sense.Gloss.GetAlternative(wsa)
    definition = sense.Definition.GetAlternative(wsa).Text
    print headword + "; " + glos + "; " + definition
```

This works fine for data that displays on the console. However, if your lexicon has Unicode characters that do not display, the following alternative will write the data to a UTF-8 file so that you can actually see your data. This example also uses the AutoLoadPolicy option that puts the cache in a mode that loads all data for a given table in one query rather than individual queries for each entry. This results in a much faster dump of data, especially for large databases.

```
import clr
import sys
clr.AddReference("COMInterfaces")
from SIL.FieldWorks.Common.COMInterfaces import *
setenc=sys.setdefaultencoding
setenc("utf-8")
out=open("dict.txt",'w')
clr.AddReference("FDO")
from SIL.FieldWorks.FDO import *
cache = FdoCache.Create("TestLangProj")
cache.VwOleDbDaAccessor.AutoloadPolicy = AutoloadPolicies.kalpLoadForAllOfObjectClass
lp = cache.LangProject
lexicon = lp.LexDbOA
wsv = lp.DefaultVernacularWritingSystem
wsa = lp.DefaultAnalysisWritingSystem
for entry in lexicon.EntriesOC :
  headword = entry.ReferenceName
  for sense in entry.SensesOS :
    glos = sense.Gloss.GetAlternative(wsa)
    definition = sense.Definition.GetAlternative(wsa).Text
    out.write(headword + "; " + glos + "; " + definition + "\n")

out.close()
```

11 Example changing strings in Interlinear Texts

In FieldWorks 4.0.1 or older, it is not possible to do a search and replace in the baseline of interlinear texts. The next version implemented a search and replace dialog, but it only

works on a single text. If you have many texts, it may be useful to use a Python program to make the changes.

The following program illustrates one way of doing this. It is written for FieldWorks 4.0.1 or older. In this case, the 'search' must be set to the string you want to change, and 'replace' must be set to the replacement text. Note that this program will change the 'search' text regardless of whether it is in the middle of a word or not. The FdoCache.Create command needs to be set to the database name you want to change. To use this program, save the contents to a UTF-8 file with a .py extension, then in a command window, execute the python file. When done, it will print a message telling how many strings were changed.

```
def Fix(s, search, replace):
  changed = False
  if s is None:
    return
  if s.find(search) >= 0:
    s = s.replace(search, replace)
    changed = True
  if s.find(search.capitalize()) >= 0:
    s = s.replace(search.capitalize(), replace.capitalize())
    changed = True
  if changed == True:
    return s

search = u"éκ̄"
replace = u"εκν"
import clr
import System
import System.Text
search = System.String.Normalize(search, System.Text.NormalizationForm.FormD)
replace = System.String.Normalize(replace, System.Text.NormalizationForm.FormD)
clr.AddReference("FDO")
from SIL.FieldWorks.FDO import *
from System import *
clr.AddReference("FwKernelLib")
from FwKernelLib import *
clr.AddReference("COMInterfaces")
from SIL.FieldWorks.Common.COMInterfaces import *
cache = FdoCache.Create("TestLangProj")
lp = cache.LangProject
runsChanged = 0
for text in lp.TextsOC:
  for par in text.ContentsOA.ParagraphsOS:
    tss = par.Contents.UnderlyingTsString
    tsb =  ITsString.GetBldr.Call(tss)
    if ITsString.Text.GetValue(tss) is not None:
      for irun in range(ITsStrBldr.RunCount.GetValue(tsb)):
        run = ITsStrBldr.get_RunText.Call(tsb, irun)
        strNew = Fix(run, search, replace)
        if strNew is not None:
          runProps = ITsStrBldr.get_PropertiesAt.Call(tsb,irun)
          bounds = ITsStrBldr.GetBoundsOfRun.Call(tsb, 0)
          ITsStrBldr.Replace.Call(tsb,bounds[0],bounds[1],strNew,runProps)
          runsChanged = runsChanged + 1
    par.Contents.UnderlyingTsString = ITsStrBldr.GetString.Call(tsb)
print "Changed " + runsChanged.__str__() + " runs."
```

There are several issues involved in this example.

- In order to include UTF-8 characters in the file, you'll need to use an editor such as Word or ZEdit that support UTF-8. In order for Python to interpret the file as UTF-8, it must include a BOM at the beginning of the file.
- FieldWorks stores strings in NFD (Normalization Form Decomposed). Your program must be prepared to handle this appropriately when reading data from the database. In this case, we use the .NET normalization methods to ensure that 'search' and 'replace' are set to NFD.
- When specifying strings, you need to use the 'u' prefix to identify them as Unicode.

12 Miscellaneous examples

The following code will print the id of the vernacular writing system with the name 'Kalaba'.

```
import clr
clr.AddReference("FDO")
from SIL.FieldWorks.FDO import *
from System import *
cache = FdoCache.Create("TestLangProj")
lp = cache.LangProject
wsa = lp.DefaultAnalysisWritingSystem
for ws in lp.VernWssRC:
  if ws.Name.GetAlternative(wsa) == "Kalaba":
   break

print ws.Hvo
```

The following code lists the id, ICULocale, and name for all writing systems in the database:

```
import clr
clr.AddReference("FDO")
from SIL.FieldWorks.FDO import *
from System import *
cache = FdoCache.Create("TestLangProj")
lp = cache.LangProject
wsa = lp.DefaultAnalysisWritingSystem
for ws in cache.LanguageEncodings:
  print ws.Hvo.ToString() + "; " + ws.ICULocale + "; " + ws.Name.GetAlternative(wsa)
```

Output for this in TestLangProj is as follows:

```
40716; en; English
40719; fr; French
40721; de; German
40724; es; Spanish
40729; fr__IPA; French (IPA)
40731; xsta; Stacked Diacritics
40733; xkal; Kalaba
40735; ar_IQ; Arabic (Iraq)
40737; dv; Divehi
40739; zh; Chinese, Mandarin
41131; xfwt; FwTest
```

13 Accessing FieldWorks source code

If you want to access FieldWorks source code repository to help understand anything about FieldWorks, you can use this process:

1. Go to http://www.perforce.com/perforce/downloads/ntx86_64.html and download Core Perforce Windows Installer, and P4Win Installer.
2. Install the downloaded files.
3. On first screen accept default 'User Installation'
4. Accept defaults until Perforce Client Configuration page. On this page, choose
   Server Port: src.sil.org:1934
   Username: anonymous
5. After installation, run program from Start...All Programs...Perforce...P4Win
6. In Client Workspace Wizard, accept default of 'Select an existing client workspace'
7. In the Client/owner list choose anonymous anonymous
8. Choose Setting...Use Current as Default
9. For everything, you can expand depot, then right-click fw and choose Sync...Force sync to Head Revision

14 Using FlexApps

Craig Farrow has written a collection of utilities called FlexApps, which can be found at http://craigstips.wikispaces.com/FlexApps. These include:

- *FlexCards* is a language learning application for use with FieldWorks databases.
- *FlexicanJumpingBean* jumps to places in FLEx from a taskbar icon.
- *CustomUtilities* which shows examples of what you can do with the core module, such as listing parts of speech, listing writing systems, and reading the lexicon.
- A function to extract texts from a FLEx database.

More details can be found at Craig's website, as well as instructions how to install it.

15 Using the Natural Language Toolkit (NLTK)

from Steve Miller
The NLTK is an open source project containing natural language processing capabilities, including a Chart Parser, Shift-Reduce Parser, POS Concordance, Collocations, and much more. It is well documented. You can find it at www.nltk.org.

The NLTK book is very readable, and I recommend it. You can see the online version at http://www.nltk.org/book, or you can pick up a copy from O'Reilly Publishers.

The NLTK can work on plain text files, HTML, and other files. One of the easiest things to do is to copy a text to a plain text file (.txt). Then use the NLTK functions on it.

15.1 Collocation Example

For example, to use the NLTK collocate function:

1. Install Python. (An installer is available in Windows.)
2. Install NLTK. (An installer is available in Windows.)
3. Put your text into a text file. (That is, a plain text file, with a .txt extension.)
4. Do the following commands, using Python's interface called IDLE (which is installed with Python):

```
from __future__ import division
import nltk, re, pprint
f = open('C:\My_File.txt')
raw = f.read()
tokens = nltk.word_tokenize(raw)
text = nltk.Text(tokens)
text.collocations()
```

This last will give you a deprecation warning if you are using Python 2.6. You can ignore that. But then it will give you your collocations.

15.2 Collocation Example Using FlexApps

FLEx exports its interlinear documents in a variety of formats, but we do not presently export its baseline to a vanilla text file. However, Craig Farrow has written a utility and to extract texts from FieldWorks databases, and included it in his FlexApps. (See the preceding section.) This is the code he sent:

```
# Load the Flex DB Access module from FlexLibs

from FLExDBAccess import FLExDBAccess

# If your data doesn't match your system encoding (in the console) then
# redirect the output to a file: this will make it utf-8.
## BUT This doesn't work in IronPython!!
import codecs
import sys
if sys.stdout.encoding == None:
    sys.stdout = codecs.getwriter("utf-8")(sys.stdout)

FlexDB = FLExDBAccess()

# No name opens the first db on the default server
if not FlexDB.OpenDatabase(dbName = "", verbose = True):
    print "FDO Cache Create failed!"
    sys.exit(1)

# Do some analysis of the Texts
import nltk, re, pprint
```

```
texts = FlexDB.lp.TextsOC

for t in texts:
    print "====", t.Name, "===="
    content = []
    for p in t.ContentsOA.ParagraphsOS:
        print " > ", p.Contents.Text
        if p.Contents.Text:
            content.append(p.Contents.Text)
    ## Need to tokenise according to language
    ## nltk.word_tokenize doesn't work with many modified letters.
##    tokens = nltk.word_tokenize(" ".join(content))
    tokens = " ".join(content).split()
    text = nltk.Text(tokens, name=t.Name)
    print text
    print text.collocations()
```

Craig sent the following notes:

I guess people might want to feed all the texts into one collocation analysis, and that is easily achieved, too. There are issues relating to parsing a text -- eg what are word-building characters, etc. I've just used the basic 'split' function (breaks at whitespace) in that example. word_tokenize() made a mess of my texts since it didn't recognise my IPA as word-building. So thought and customisation may be required in each situation.