The Eiffel FoxPro Library

Giving Eiffel access to xBase Database Files (DBFs)

The dBase-III/IV/V or "xBase" languages of the 1980s and 1990s left a very big mark on the computing industry. One of the xBase languages to evolve was Foxpro from Fox Software, which was eventually purchased by Microsoft and became Microsoft Visual FoxPro, reaching version 9.x.

At the core of this technology was (and is) the DBF (i.e. database file). The DBF not only served as a vehicle for persisting data to long term storage, but became (in the case of Foxpro) the storage mechanism for its own code. Thus, the use of the DBF was core and prolific. However, this choice would also doom the language system to being excluded from the .NET family when Microsoft began its project of shepherding various languages into that programming paradigm.

Nevertheless, the DBF has had a profound impact on the computing industry and in its various forms has built another form of common data interchange between systems. Moreover, the DBF protocol and standard has become the basis for a number of modern database respositories, including Microsoft SQL Server (at whose heart beats the core technologies gleaned from Visual FoxPro itself).

This library represents a step towards providing Eiffel programmers and systems access to DBFs and their data content. While it is not a complete or exhaustive treatment, it does offer the basics and a pattern for accessing DBFs more completely in the future.

Why incomplete? The motivation for creating this library was born (as many things are) as a product of the mother of invention: necessity. Our project consists of a number of "legacy" systems. Some of these systems are written in FoxPro 2.x, whereas others are written in Visual FoxPro 9. Those systems communicate with each other by way of shared DBFs. Our Eiffel project code needed to participate in this intra-system communication. This presented us with a choice and in the end, we chose to write a library in Eiffel that would know how to read and write from basic DBFs at enough of a level as to facilitate communication to and from legacy FoxPro systems. Thus, this library contains just enough code to handle this need and no more.

What there is

The Eiffel library as we have coded it provides the capacity to read and write basic DBFs, with a subset of its full-set of data type complement. FoxPro 2.x has a smaller complement of data types than does Visual FoxPro, so the FoxPro 2.x standard is the least common denominator between the two and is partially what we coded for. Therefore, we specifically targeted the following data types for the Eiffel Foxpro library:

1. *Character (a STRING in Eiffel)*
2. *Numeric (a REAL_32 variant)*

3. *Float (another REAL_32 variant)*
4. *Date (a DATE in Eiffel)*
5. *Logical (a BOOLEAN in Eiffel)*

While the legacy systems we were coding for also included the Memo and General data types, those systems did not cross-communicate with DBFs where those data types were used. So, we simply ignored them.

Still—even though we only needed to code for five data-types from the legacy FoxPro 2.x, we were inclined to include a few more. The ones we included beyond the five above were included because they were so close to the five that it was hardly any labor at all to code them up. At least one more data-type we coded into the Eiffel Foxpro library for no other reason than it was a challenge and because we could!

Here is the full list of what the Eiffel Foxpro library is coded to both read and write from DBFs:

1. *Numbers*
    a. *Numeric (from Eiffel REAL_32)*
    b. *Float (from Eiffel DECIMAL)*
    c. *Float (from Eiffel REAL_32)*
    d. *Integer (from Eiffel INTEGER_32)*
    e. *Currency (from Eiffel DECIMAL)*
2. *Dates*
    a. *Date (from Eiffel DATE)*
    b. *Datetime (from Eiffel DATE_TIME)*
3. *Booleans*
    a. *Logical (from Eiffel BOOLEAN)*
4. *Strings*
    a. *Character (from Eiffel STRING)*

When compared to the Eiffel ECMA-367 Standard Basic Types (§ 8.30), the coverage from Eiffel to the DBF is much clearer:

1. *BOOLEAN*
    a. *DBF: Logical*
2. *CHARACTER*
    a. *DBF: None*[1]
3. *INTEGER*
    a. *DBF: Integer*
4. *NATURAL*

[1] *Loosely realized in FoxPro DBF as a Character "string" of 1 in size precision.*

    a. *DBF: None*[2]
5. *REAL*
    a. *DBF: Float*
    b. *DBF: Double*
    c. *DBF: Numeric*
6. *DECIMAL*[3]
    a. *DBF: Currency*
7. *POINTER*
    a. *DBF: None*[4]
8. *STRING*
    a. *DBF: Character*

From the list of Basic Types, you may notice that these do not include data types of DATE and DATE_TIME. Nevertheless, these are "basic types" for FoxPro DBFs and are included in the library's capacity for reading from and writing to a DBF.

What there is not

The early versions of dBase and Foxbase included only a few data types in their universe. Over the years, those basic data types were expanded. Today, databases (especially RDBMS) include a nice assortment of data types including geospatial and others. The intermediate years of Visual FoxPro did see growth in the number of data types capable of being persisted in the database. These include:

Data type	Description	Size (bytes)	Range
Character	Any text	1 to 254	Any characters
Varchar	Any text	1 to 254	Any characters
Varbinary	Binary data	1 to 254	Binary data

[2] *Loosely specified in a DBF as a positive integer.*
[3] *Class DECIMAL is not presented in the ECMA-367 standard as a Basic Type, and while it is not an expanded type (yet), it does represent a distinct departure from class REAL by virtue of controlled precision. It is closely related to the FoxPro Currency data-type, so it is included in this list as its own Basic Type.*
[4] *While there is no direct support of POINTER in a DBF, one might consider Primary Keys and Foreign Keys as pointers of sorts and loose analogs of an Eiffel POINTER, but not completely analogous.*

Date	Chronological data consisting of month, year, and day	8	When using strict date formats, {^0001-01-01}, January 1st, 1 A.D to {^9999-12-31}, December 31st, 9999 A.D.
Date Time	Chronological data consisting of month, year, day, and time	8	{^0001-01-01}, January 1st, 1 A.D to {^9999-12-31}, December 31st, 9999 A.D., plus 00:00:00 a.m. to 11:59:59 p.m.
Numeric	Integers or fractions	1 to 20	- .9999999999E+19 to .9999999999E+20
Float	Integers or fractions	1 to 20	- .9999999999E+19 to .9999999999E+20
Integer	Integers	4	-2,147,483,647 to 2,147,483,647
Double		1 to 20	- .9999999999E+19 to .9999999999E+20
Currency	Monetary amounts	8 bytes	- 922337203685477.5807 to 922337203685477.5807
Logical	Boolean value of true or false	1 byte	True (.T.) or False (.F.)
Memo	Any text	In the DBF 4 bytes/memo In the FPT allocated in	Any characters

		chuncks based on SET BLOCKSIZE	
General	data and host program	In the DBF 4 bytes/memo In the FPT allocated in chuncks based on SET BLOCKSIZE	OLE Documents
Blob	Binary data	In the DBF 4 bytes/memo In the FPT allocated in chuncks based on SET BLOCKSIZE	Binary data

Some of these are not included in the present iteration of this library (perhaps in the future—expand them as you care or like).

- *Varchar*
- *Varbinary*
- *Memo*
- *General*
- *Blob*

Varchar is really printable STRING_32 in a field limited to boundaries of 1 to 254 CHARACTER_32. It may be simple to code and might be low-hanging fruit to immediate extension of the library.

Varbinary is printable and non-printable STRING_32, having the same limits and perhaps qualification as "low-hanging-fruit" for library extension in the near future!

The data types for Memo, General, and Blob are more involved and will require more work for writing and reading due to the precise nature of their specifications which are not already represented in available Eiffel library code, such as STRING_32. Therefore, these data types are not as easily attained as the varchar and varbinary types.

Indexing not present

As our first goal was to capture the needs our own legacy FoxPro systems, we chose not to code for the use of FoxPro or dBase indexes and indexing files. There are several types. NONE of those types was coded for in this library as those needs were not within the boundaries of our initial necessity for our legacy systems. Therefore, they are ripe for extension, but much more involved than simple data storage and persistence.

We recommend great care be taken when addressing the needs of indexing DBF data as this is an entire field unto itself and demands and requires great care before embarking upon it (to our current level of understanding of the matter—that is).

Round-trip Mapping

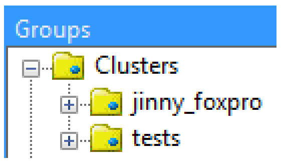
This library is designed (and tested as) a round-trip mechanism—that is—data may begin in Eiffel code, be exported (written) to FoxPro DBFs and then read back into Eiffel from the DBF file, where the end result is precisely the same as how it started. Moreover, data generated in and by FoxPro systems ought to be well formed and on-par with its Eiffel counterpart once arriving in Eiffel from the FoxPro legacy DBF file.

You will find tests and testing that demonstrate this process both internally within the Eiffel FoxPro library and externally, by writing to an actual DBF (created on-the-fly) and then read back in, parsed, and translated from the raw DBF data into Eiffel counterpart types.

Clusters and Classes: The lay-of-the-land

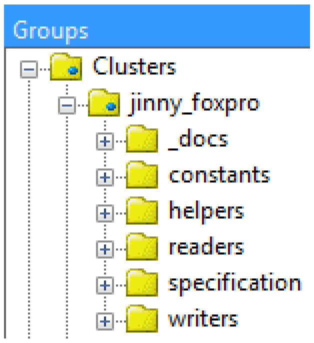
The Eiffel FoxPro library consists of a few primary classes, some supporting classes, and testing classes. You will also find documentation classes, which are temporary "hacks" designed to hold library and cluster documentation until Eiffel Studio has been thought through more thoroughly and thoughtfully as how to have a mechanism for presenting documentation. Nevertheless, the primary and secondary classes are of immediate interest for you (presuming reading this documentation represents a need on your part).

Targets: Library and Testing



There are two primary library targets: Library (foxpro) and Testing (test). As you can surmise, the primary target your project needs to use is "foxpro". The only reason for using the "test" *target would be to correct bugs or extend the library coverage. So, this section of the documentation will focus on the "foxpro" target.*

Target: FoxPro Clusters



The FoxPro target contains a number of clusters: _docs, constants, helpers, readers, specification, and writers. The "_docs" cluster has documentation for the library at the library level (not covered here). The constants cluster is (hopefully) self-explanatory, but see the classes for specific details as to what the constants mean. There are some constants that are quite meaningful (and have orienting stories), especially those involving dates and date-time data types.

The primary class of the helper cluster is the FP_BYTE_HELPER, which is a class that assists other library classes in handling FoxPro-specific byte operations. For example: In many cases, 2, 4, and 8 byte values (i.e. INTEGER_16 or INTEGER_32) are stored in the DBF as bytes in reverse order (for whatever reason FoxPro requires). There are features of the FP_BYTE_HELPER that assist with this need, when reading and writing these "reversed-byte" collections to and from the DBF file.

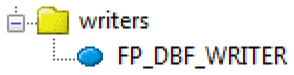
The readers and writers cluster have classes which perform the work of preparing and then reading or writing Eiffel data objects to their DBF complements.

Finally, the specification cluster contains common classes which define the structure of DBF header sections, subrecord sections, and data storage in the DBF file.

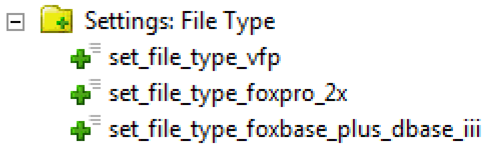
Primary and Secondary Classes

Out of the clusters described above, there are a number of classes to pay special attention to when using the Eiffel Foxpro library. These classes are:

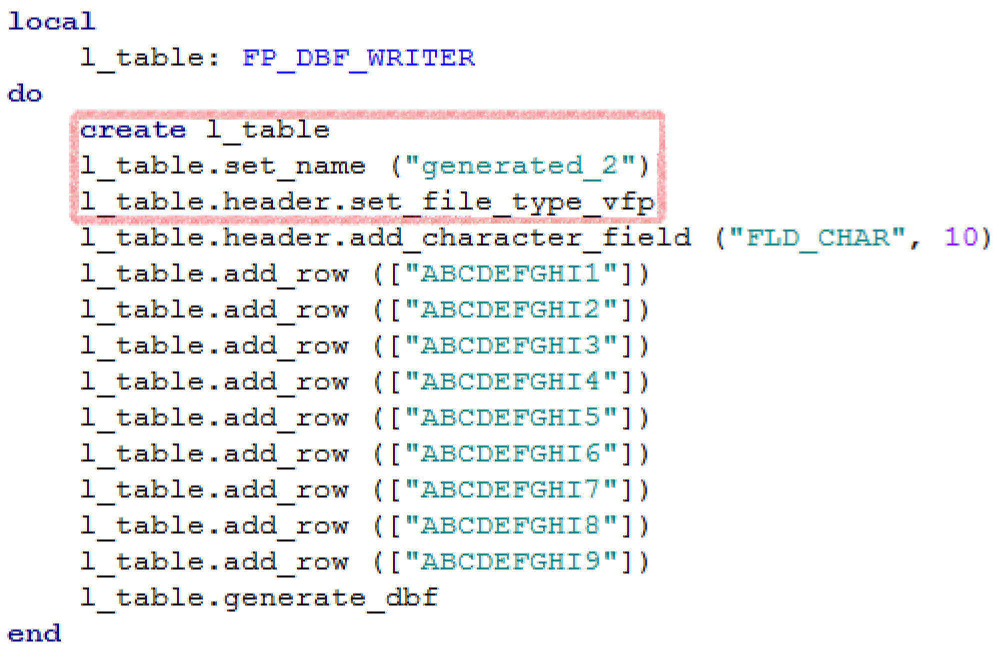
FP_DBF_WRITER



When you want to write Eiffel data objects to a DBF file, use the FP_DBF_WRITER. The basic process for creating a writer is this:

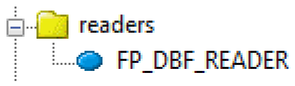
1. *Create an instance of an FP_DBF_WRITER.*
2. *Provide name and file type specifications (e.g.*
   Settings
   set_name
   and
   FP_HEADER
   
   Settings: File Type
   set_file_type_vfp
   set_file_type_foxpro_2x
   set_file_type_foxbase_plus_dbase_iii
   features).
3. *Provide field specification(s) (e.g. calls to `add_*', where "*" = field data type).*
4. *Provide data in the form of calls to `add_row', passing a TUPLE, with elements who's types match the field specifications in the same order as the fields were specified in step #3 above.*

Once one has provided the above information (in order), then a call to `generate_dbf" will create a DBF in whatever location you specify.



```
local
    l_table: FP_DBF_WRITER
do
    create l_table
    l_table.set_name ("generated_2")
    l_table.header.set_file_type_vfp
    l_table.header.add_character_field ("FLD_CHAR", 10)
    l_table.add_row (["ABCDEFGHI1"])
    l_table.add_row (["ABCDEFGHI2"])
    l_table.add_row (["ABCDEFGHI3"])
    l_table.add_row (["ABCDEFGHI4"])
    l_table.add_row (["ABCDEFGHI5"])
    l_table.add_row (["ABCDEFGHI6"])
    l_table.add_row (["ABCDEFGHI7"])
    l_table.add_row (["ABCDEFGHI8"])
    l_table.add_row (["ABCDEFGHI9"])
    l_table.generate_dbf
end
```

FP_DBF_READER



readers
FP_DBF_READER

When you want to read DBFs, their header, fields, and data into Eiffel objects. Reading is somewhat the opposite of the writing process, but not entirely. While writing requires one to specify many things (header and field specifications as well as data), reading has no such requirements. Thus, reading a DBF from disk into memory is quite simple. Fetching the data out of the results is where it gets a little tricky and complicated. Thus, getting the DBF read is simple:

1. *Create an instance of an FP_DBF_READER.*
2. *Call {FP_DBF_READER}.read_dbf (a_file_name: STRING)*

```
local
    l_reader: FP_DBF_READER
    l_dbf_name: STRING
do
    l_dbf_name := "empty_table.dbf"
    dbf_printer (l_dbf_name)
    create l_reader
    l_reader.read_dbf (l_dbf_name)
```
[5]

If the file is successfully read, the data will be gathered for you into the feature: `last_data_content', which is an ARRAYED_LIST [TUPLE [...]].

```
last_data_content: ARRAYED_LIST [attached like record_anchor]
        -- `last_data_content' read from file.
```

```
record_anchor: detachable TUPLE [detachable ANY, detachable ANY, detachable ANY,
        -- Record TUPLE type anchor.
```

The complex part of reading a DBF is iterating over the ARRAYED_LIST and gathering data from each TUPLE.

NOTE: An examination of the `record_anchor' TUPLE will reveal that it is a TUPLE with 255 detachable ANY items. Thus, the FP_DBF_READER is capable of reading any DBF with 1 to 255 fields of any of the supported data types (see previous sections above).

Extendability Thoughts

One can envision a single class with a "reader" and a "writer" that wraps the above two notions and then descends into classes that are about specific DBFs with specific static structures, which is precisely how we will use this library in our legacy code needs! In such a way the complexities of setup (WRITER) and scanning of read data (READER `record_anchor') can be hidden in features which already know or redefine to specific knowledge about a particular DBF and its structure (e.g. perhaps you have a DBF containing CONTACT information where the structure and types of data are stable).

---

[5] *Note that the call to "dbf_printer (l_dbf_name) is a test-only support feature. See the FP_DBF_READER_TEST_SET for more information on this feature.*