The Xbase^{++} Guide through the Clipper RDD Jungle

The Xbase^{++} Guide through the Clipper RDD Jungle 1

The Xbase++ Guide through the Clipper RDD Jungle

Abstract: This article outlines important issues resulting from concurrent database access using different database drivers (RDD) and Database Engines (DBE). The configuration of Xbase++ DBEs is discussed for concurrent usage with Clipper's RDDs. How to access databases concurrently with Xbase++, Comix, Six, FoxPro and Visual Foxpro is explained. A generic DBESYS procedure is finally presented, that provides the correct DBE configuration for various scenarios.

DatabaseEngine versus Replaceable Database Driver

Xbase++ is designed to be compatible with the Clipper programming language. Clipper's default database and index file formats are DBF and NTX, which are the defaults for Xbase++ as well. Clipper is able to access different database and index file formats by means of Replacable Database Drivers (RDD). Xbase++ is able to do the same by means of its Database Engines (DBE).

What is the difference between RDD and DBE? The major difference is that Clipper's RDDs must be **linked** to an EXEcutable file, while Xbase++'s DBEs are **loaded** by an EXE at runtime. A Clipper programmer has to decide which RDD is available to an application when the application is built. An Xbase++ programmer can answer the same question while the Xbase++ application is running.

The Xbase++ approach for accessing different database and index file formats via DBEs extends Clipper's RDD concept. DBEs are proven to be extremely flexible. This is possible since DBEs are encapsulated within Dynamic Loadable Libraries (DLL). The entire functionality for accessing different file formats with the same set of CREATE, USE, APPEND, DELETE etc. commands is possible since Xbase++ applications can load the required Database Engine at runtime.

What DBESYS.PRG is good for

Each Xbase++ application calls three INIT PROCEDUREs on start-up: ErrorSys(), DbeSys() and AppSys(). That means: these three procedures are executed before the Main() procedure is called. Each of these procedures is provided in PRG source code (see: \ALASKA\XPPW32\SOURCE\SYS) so that they can be modified to suit your needs.

The ErrorSys() procedure installs the default error handling routine, just as in Clipper.

AppSys() initializes the default output device depending on how an application is linked (Console or GUI application = VIO or PM mode).

DbeSys() finally loads the DBEs that must be present before PROCEDURE Main() is executed.

Note : as you see DBEs are loaded after AppSys() and/or ErrorSys(). So no DBEs are available within AppSys() or ErrorSys() procedure. This has an implication when users are overloading AppSys() i.e. to start up with a fancy screen and the application tries to read some information from configaration tables. This is not recommended as you can not easily open a table in AppSys(). If you need to overload i.e. AppSys() and you need DBEs here leave AppSys() blank and move your code to Main() procedure.

The default DbeSys() procedure calls three functions: *DbeLoad()* for loading a DBE, *DbeBuild()* for combining data (i.e. DBF) and index (i.e. NTX or CDX) components and *DbeSetDefault()* for selecting the default database engine. It is used for opening a database file when a DBE is not specified for creating/opening a file.

If you are not familiar with these three functions, take a look at the default DBESYS.PRG file and read the description for these three functions in the Xbase^{++} online help.

About file formats and Clipper RDDs

A detailed description on file formats for DBF and index files is available in the internet at www.wotsit.org, for example. We are going to discuss the major differences that must be taken into consideration when an Xbase^{++} application accesses data concurrently with different RDDs.

DBF/DBT

xBase database files (DBF) support a special data type: Memo. This data type is used to store textual data of variable length to disk. Memo data is not stored in a DBF file but in a separate file (DBT). Clipper and Xbase^{++}'s DBFDBE use the DBT file specification for handling Memo data. DBT file format does not allow to store binary data.

DBF/FPT

A different Memo file format was introduced by FoxPro. It has the extension FPT and requires less storage space for Memo data than the DBT file format. Xbase^{++}'s FOXDBE uses the FPT file format for Memo data. This FTP file format also allows to store any binary data (JPGs, BMPs, Xbase^{++} Persistant Format (XPF), ...) which is not possible with DBT file format.

NTX

An NTX file contains index data created by the INDEX ON command. The NTX file format was introduced by Clipper. Xbase++'s NTXDBE uses this file format for index files. An NTX file stores only one index for one DBF file.

CDX

A CDX file contains index data created by the INDEX ON command. The CDX file format was introduced by FoxPro and is supported by Clipper RDDs as well as by Xbase++'s DBEs. The advantage of the CDX file format is that data for multiple indexes can be stored within one file.

FoxPro

FoxPro has introduced major enhancements to the xBase database and index file formats. There is the FPT and CDX file format for Memo and Index files. In addition, FoxPro made the transition to National Language Support on the file level. I.e. Codepage information is stored in the CDX file. The evolution from DOS (console based) to Windows (GUI based) is reflected in FoxPro by changing the default character set from OEM to ANSI.

FoxPro's switch from OEM to ANSI is a major source for confusion.

SIX/Comix

The SIX and Comix RDDs have been very popular for Clipper programmers. Both RDDs use the FoxPro 2.x CDX file format for fast indexing and rely on the OEM character set. The CDXDBE of Xbase++, however, uses Visual FoxPro's default settings, which is the ANSI character set. The CDXDBE must be configured for accessing SIX/Comix data.

Successful migration and concurrency

Each developer who intends to run a 32-bit Xbase++ application concurrently with a 16-bit Clipper or FoxPro application must be aware of differences in file formats. What do I use? DBF/DBT/NTX (Clipper) or DBF/FPT/CDX (FoxPro,Six,Comix) or ANSI or OEM...? The list could be extended with codepage issues that were introduced with FoxPro/Visual FoxPro.

The popular SIX and Comix RDDs, for example, can be used concurrently with Xbase++ applications when the Xbase++ DBE is configured properly. The default configuration of Xbase++ DBEs, however, disallows concurrent access with SIX or Comix RDDs.

You must configure the Xbase++ DBEs for concurrent access with 16-bit applications. This is subject of our generic DbeSys() routine.

A generic DbeSys() for 16-bit RDDs

Xbase++'s default DbeSys() procedure is automatically linked to an application unless you create your own DbeSys(). In this case, your DbeSys() is linked and the default behavior is defined by your code.

As an Xbase++ application just loads DBFDBE and NTXDBE for compatibility reasons we need to extent the list of DBEs which should be loaded. Here comes FOXDBE and CDXDBE into mind.

Note : a DBE (technically a DLL) must - and can - only be loaded once. You then can use it for building different DBEs depending on your specific needs. So i.e. you have to load CDXDBE once while you can build/combine different DBEs with it.

Let us concentrate on CDXDBE for any further explanations. CDXDBE will be used for Visual Foxpro 5 and higher compatibility, Comix and SIXCDX compatibility. When loading CDXDBE it will be setup to Visual Foxpro 5 and higher compatibility mode by default. You need to configure it for using it in Comix or SIXCDX mode.

This will be accomplished with dbeInfo() function. Comix i.e. requires

```
DbeInfo( COMPONENT_ORDER, CDXDBE_MODE, CDXDBE_FOXPRO2X )
```

Attention : The Xbase++ DBE concept supports two usage scenarios for DBEs. First of all you can use a data/storage DBE (DBFDBE, FOXDBE) as they are by simply dbeLoad()ing the DBE. But this does not allow i.e. to index a table and this is not what we really need. The second option is to create a virtual compound DBE which now combines different DBEs and different types of DBEs. Most common is to build a compound DBE as of a data DBE (DBFDBE, FOXDBE) and an order DBE (NTXDBE, CDXDBE). By the way, this unique Xbase++ feature allows to index an ASCII SDF file (see \ALASKA\XPPW32\SOURCE\SAMPLES\BASICS\DBE for a sample).

The dbeInfo() settings will not be stored within a compound DBE. The settings provided by dbeInfo() will only influence/setup the core DBE (i.e. CDXDBE.DLL) which is part of a compound DBE. The information which DBE should be setup will be passed through the first parameter to dbeInfo(): the data DBE (COMPONENT_DATA) or the order DBE (COMPONENT_ORDER). To be straight: to dbeBuild() a Visual Foxpro compatible DBE and a Comix compatible DBE needs to use CDXDBE within both compound DBE. So setting CDXDBE per dbeInfo() does change settings within all compound DBEs which use CDXDBE.

The goal of the article is to show how Xbase++ developers can use any combination of DBEs within their applications. This makes it necessarry i.e. to load CDXDBE once and to use it within several compound DBEs. But each resulting compound DBE needs its specific setup. So Visual Foxpro needs different settings as to use Comix compatible indexes.

To overcome this issue we explain how to overload a few functions / commands which depend on the DBE settings. These are dbCreate(), dbUseArea() and dbeSetDefault() functions and of course the USE command which will be pre-processed to dbUseArea(). Those are the only functions which are critical in the area as these are responsible for the mode in which the DBE works. Luckily the resulting generated/created/used DBO (Data Base Object) inherits the settings of the DBEs and stores them internally. This means from the moment when a table has been created and/or is in use it inherits all its DBE settings and uses them. The DBO (the open table) does not rely on any later change of any DBE. So you are free from then on.

The trick is to overload dbCreate(), dbUseArea() and dbeSetDefault() and prepare the DBEs with the setting they need. This then happens right before the function will be executed. And this guarantees that the appropriate setting are valid when using the DBE.

The pre-processor directive

```
#xtranslate dbUseArea([<param,...>]) => ;
           _dbUseArea([<param>])
```

will be pre-processed to the function here:

```
FUNCTION _DbUseArea( lNewArea, cDBE, cFileName, cAlias, ;
                     lShared, lReadonly )
   IF cDBE == NIL
      cDBE := dbeSetDefault()
   ENDIF
   SetDbeSwitches( cDBE )
RETURN( dbUseArea( lNewArea, cDBE, cFileName, cAlias,   ;
                   lShared, lReadonly ) )
```

The same works with dbCreate() and dbeSetDefault(). See attached DBESYS.PRG.

In the end the entire modifications for using different DBEs (RDDs in Clipper terms) are to include a small DBESYS.CH file. This covers the translation to its pendants which take care about the DBE settings. Plus you need to add the modifed DBESYS.PRG to your project and thus overload the standard DbeSys() procedure with it. That's all which is required.

Happy Xbase^{++}ing,

Your Alaska Software Team