XBase++ Programming For VFP Developers

Geek Gatherings 2015 Roger Donnay

Contents

Introduction

Xbase++ has extended the capabilities of the language beyond what is available in Visual FoxPro. There are new variable types and language concepts that can enhance the programmer's ability to create more powerful and more supportable applications.

The flexibility of the Xbase++ language is what makes it possible to create libraries of functions that can be used dynamically across multiple applications. The preprocessor, code blocks, ragged arrays and objects combine to give the programmer the ability to create their own language of commands and functions and all the advantages of a 4th generation language that will run on 32-bit or 64-bit Windows operating systems.

This workshop will also show how these language concepts can be employed to create 3rd party add-on products to Xbase++ that will integrate seamlessly into Xbase++ applications.

The Xbase++ language in incredibly robust and it could take years to understand most of its capabilities, however when migrating VFP applications, it is not necessary to know all of this. I have aided many Clipper and FoxPro developers with the migration process over the years and I have found that only a basic introduction to the following concepts are necessary to get off to a great start:

- The Xbase++ Project. Creation of EXEs and DLLs.
- The compiler, linker and project builder .
- Console mode for quicker migration of Fox 2.6 apps.
- INIT and EXIT procedures, DBESYS, APPSYS and MAIN.
- The DBE (Database engine)
- LOCALS, STATICS, PRIVATE and PUBLIC variables.
- STATIC functions.
- New data types such as Code Blocks and Objects, also detached locals.
- The Reference operator, the Alias operator, and the Index operator.
- Multi-Dimensional arrays, including ragged arrays.
- The Pre-Processor and how to create your own commands.
- The Error Handler

When it is time to improve on the application and evolve it into a modern application, then an introduction to the following concepts are necessary:

- Xbase Parts for GUI elements.
- Multi-threading and workspaces.
- Class creation and usage, including methods and instance variables.
- ActiveX.
- The graphics engine and owner-drawing.
- Understanding the garbage collector.
- Introduction to SGL (Structured GUI Language) programming.

Language compatibility is the reason that Xbase++ is the most natural migration path for VFP developers. The main objective of this workshop is to show VFP developers how to get started with Xbase++ without the use of the WorkBench. It is understood that VFP developers are used to a Microsoft-style development environment in which everything is integrated into a Visual Studio, including the editor, form designer, compiler, etc. The purpose of this workshop is not to teach a new environment, but instead to explain the nuts and bolts of an Xbase++ application and to give attendees options for creating the kind of environment that will suit their needs. We will show examples of Xbase++ applications and how these applications are developed and maintained using a variety of environmental techniques and programming styles.

There is often resistance from VFP developers who have only used an environment which supports drag and drop techniques for form design. The first thing they want to see is some kind of project management environment or a simple methodology to convert their existing VFP applications to Xbase++. This workshop is not going to make such promises but instead is going to show development methods that are at least as productive and usually much more so than Visual tools while keeping the developer well-grounded in understanding the Xbase++ language and what is going on under the hood.

Finally, we will discuss some ideas where Donnay Software and VFP developers can put their heads together to create a “hybrid” development environment that allows VFP developers to continue designing screens using their familiar drag and drop Microsoft tools but then translate the resultant metadata into source code that can be compiled by Xbase++.

So let’s get started by first getting acquainted with the elements of an Xbase++ application.

The Xbase++ SOURCE CODE files

Source code is contained in .PRG, .CH, .ARC and .XPJ files. PRG files contain the source code for functions, procedures and classes. CH files contain the source code for definitions and pre-processor directives. ARC files contain the source code for resource files that contain .JPG,

.BMP, cursors, etc. XPJ files contain the source for the project builder that compiles and links the entire project.

The Xbase++ COMPILER

The COMPILER, XPP.EXE, converts Xbase++ source code into object (.OBJ) files which are later linked together into .EXE or .DLL files. A compiled language has many advantages over interpretive languages such as FoxPro. Interpreters cannot validate code until runtime and therefore cannot insure that variables are typed correctly, that the code syntax is correct or even that called functions exist. The compiler performs the function of pre-processing commands into executable expressions, insures that variables and database fields are correctly typed and are not ambiguous, and provides other valuable features that help the programmer to prevent coding mistakes and help the debugging process. A compiled application runs much faster than an interpreted application (`Figure 1`).

```
XPP test.prg /c /l /p  -> creates test.obj and test.ppo
```

Figure 1

The Xbase++ LINKER

The LINKER, ALINK.EXE, links all the .OBJ files of a project into an .EXE or .DLL file that can run on 32-bit or 64-bit Windows operating systems. When a .DLL file is created, a .LIB file of the same name as the .DLL file is also created (`Figure 2`).

```
ALINK test.obj, mylib.lib
```

Figure 2

The .LIB file consists only of a list of public functions and procedures that are contained in the .DLL file. This file is then used to link the .DLL to other .EXEs or .DLLs so that all .DLLs are automatically loaded when the .EXE is run. The Windows .EXE loader dynamically links all function and procedure calls between .EXEs and .DLLs at the time that they are loaded into memory. .EXEs and .DLLs are commonly referred to as "runtime binaries" and are necessary to be distributed with the application files. Source files, .OBJ files, .RES files, .LIB files are not required to be distributed with the application. If, however, the Xbase++ application is used by other programmers as a set of .DLLs which link to their own application, then .LIB files are necessary.

The PROJECT file and PROJECT builder

The Xbase++ project builder (PBUILD.EXE) makes it simple to list all the files that make up an application and create the end product which may consist of multiple .EXEs and .DLLs. The source file is an .XPJ file that looks something like `Figure 3`:

```
[PROJECT]
    COMPILE       = xpp
    COMPILE_FLAGS = /q /w
    DEBUG         = yes
    GUI           = yes
    LINKER        = alink
    BUTTONTEST.XPJ

[BUTTONTEST.XPJ]
    BUTTONTEST.EXE
    BUTTONTEST.DLL

[BUTTONTEST.EXE]
    Buttontest.arc
    BUTTONTEST.PRG
    BUTTONS.LIB
```

Figure 3 – ButtonTest.Xpj project file

```
C:\PBUILD buttontest    >> creates runtime binaries: BUTTONTEST.EXE, BUTTONS.DLL
                        >> creates intermediates: BUTTONS.LIB, BUTTONS.RES,
                           BUTTONTEST.OBJ,BUTTONS.OBJ.
```

Figure 3a – Command Line to compile ButtonTest.xpj

The beauty of the Xbase++ project file is that intermediate dependencies are not required to be listed for PBUILD to determine how to build the project. Only source files and .LIB files need to be listed.

Procedures: INIT, EXIT, APPSYS, DBESYS and MAIN

All Xbase++ apps must have a MAIN procedure, and APPSYS procedure and a DBESYS procedure in the source code.

APPSYS - This procedure is the first one called and is used to set up application parameters. If the programmer does not include an APPSYS procedure in his/her source, then Xbase++ automatically uses a default APPSYS procedure which creates and displays an XbpCrt() class object for use with text-based code such as Clipper and Foxpro 2.6 code.

DBESYS - This procedure is the next one called and is used to load the database engine(s) used in the application. This is where a connection to a database server may be initiated or where other DBEs may be loaded. If the programmer does not include a DBESYS procedure in his/her source, then Xbase++ automatically uses a default DBESYS which loads the DBFNTX dbe for Clipper compatibility.

INIT -This procedure is the next one called and is optional. This is where initialization code is called before calling the MAIN procedure.

MAIN - This procedure is the next one called and is the only one that is required. This is the entry point for running the main application.

EXIT - This procedure is the last one called and is executed immediately after the QUIT command or after returning from the MAIN procedure. This is the last exit point for the application and is used for cleanup such as deleting files, disconnecting from a database server, etc.

PUBLIC FUNCTIONS and STATIC FUNCTIONS

PUBLIC FUNCTIONS are callable from anywhere in the application and by any DLLs loaded by the application. They are also callable from a macro because the function name exists in the symbol table. There can be no duplicates of public functions in the same EXE or DLL, however public function names can be duplicated in different DLLs. This is undesirable however, because it can be ambiguous as to which function actually is called in the application.

STATIC FUNCTIONS are callable only from functions that are in the same source file as the STATIC function. They are also not callable from a macro because a STATIC function does not exist in the symbol table. There can be no duplicates of static functions in the same source file, however there can be an unlimited number of duplicates within an application.

The DATABASE ENGINE (DBEs)

All database operations, i.e. dbSkip(), dbGoTop(), dbSetFilter(), RecNo(), etc. are routed through the DBE layer of Xbase++. Xbase++ supports DBEs for FoxPro, dBase, Clipper, Advantage Server, and ODBC databases. The database engine gives the programmer the ability to write code that is independent of the database being used. The default dbe is DBFNTX, which supports .DBF databases and .NTX indexes (Clipper compatible).

DBESYS

DBESYS.PRG contains the source code for the default DbeSys() Procedure that is automatically called before the Main() procedure (Figure 4).

```
*************************************************************
* DbeSys() is always executed at program startup
*************************************************************
PROCEDURE dbeSys()
/*
 *   The lHidden parameter is set to .T. for all database engines
 *   which will be combined to a new abstract database engine.
 */
LOCAL aDbes := { { "DBFDBE", .T.},;
                 { "NTXDBE", .T.},;
                 { "DELDBE", .F.},;
                 { "SDFDBE", .F.} }
LOCAL aBuild :={ { "DBFNTX", 1, 2 } }
LOCAL i

  /*
   *   Set the sorting order and the date format
   */
  SET COLLATION TO AMERICAN
  SET DATE TO AMERICAN

  /*
   *   load all database engines
   */
  FOR i:= 1 TO len(aDbes)
      IF ! DbeLoad( aDbes[i][1], aDbes[i][2])
         Alert( aDbes[i][1] + MSG_DBE_NOT_LOADED , {"OK"} )
      ENDIF
  NEXT i

  /*
   *   create database engines
   */
  FOR i:= 1 TO len(aBuild)
      IF !DbeBuild( aBuild[i][1], aDbes[aBuild[i][2]][1],aDbes[aBuild[i][3]][1])
         Alert( aBuild[i][1] + MSG_DBE_NOT_CREATED , {"OK"} )
      ENDIF
  NEXT i

RETURN
```

Figure 4 – Generic DbeSys.prg good for most applications

Selecting a Database engine is a simple as using the dbeSetDefault() function.

CONSOLE MODE

Xbase++ supports a non-gui console mode for migration of Clipper or FoxPro 2.6 applications. By default, console mode is always enabled via the AppSys() function which is always called before Main(). APPSYS.PRG contains the default source code (Figure 5).

```
PROCEDURE AppSys()

#define DEF_ROWS       25
#define DEF_COLS       80
#define DEF_FONTHEIGHT 16
#define DEF_FONTWIDTH   8

LOCAL oCrt, nAppType := AppType(), aSizeDesktop, aPos

// Compute window position (center window on the Desktop)
aSizeDesktop    := AppDesktop():currentSize()
aPos            := { (aSizeDesktop[1]-(DEF_COLS * DEF_FONTWIDTH))  /2, ;
                     (aSizeDesktop[2]-(DEF_ROWS * DEF_FONTHEIGHT)) /2  }

// Create XbpCRT object
oCrt := XbpCrt():New ( NIL, NIL, aPos, DEF_ROWS, DEF_COLS )
oCrt:FontWidth  := DEF_FONTWIDTH
oCrt:FontHeight := DEF_FONTHEIGHT
oCrt:title      := AppName()
oCrt:FontName   := "Alaska Crt"
oCrt:Create()

// Init Presentation Space
oCrt:PresSpace()

// XbpCrt gets active window and output device
SetAppWindow ( oCrt )

RETURN
```

Figure 5

Console mode sends all legacy output such as ?, MENU TO, @..SAY, @..PROMPT, SetColor(), etc. to the application window which is created from the XbpCrt() class. This insures that most Clipper and FoxPro 2.6 legacy code will execute correctly and display output to the console window.

The Xbase++ PRE-PROCESSOR

In my opinion, the pre-processor gets the "most valuable feature" award when it comes to programmer productivity. This is because it provides the abstraction concepts necessary to reduce the number of lines of code needed to create enterprise-level applications. In fact, my product eXpress++ is only possible due to the pre-processor and code blocks.

Look at this code example in Figure 6:

```
SELECT INTO BOISE NAME,PHONE WHERE ZIP="83706" ORDER BY PHONE
```

Figure 6

This is a classic SQL SELECT statement which simplifies a Query-based method of data access and is why SQL is so popular a language.

Due to the fact that compiled languages really do not understand commands such as this, the elements must be parsed into something that can be understood, and so there must be a parser that would convert the command to a set of ordered parameters that can be passed to a function or an object.

The SQL function input parameters may look something like Figure 7:

```
SQLSelect( aColumns, cWhere, cOrderBy, cInto )
```

Figure 7

If you were given the task of writing an SQL Query Parser you could use the Xbase++ pre-processor to accomplish this task (Figure 8).

```
#command SELECT [INTO <into>] <columns,...> [WHERE <where>] [ORDER
BY <order>];
  => SQLSelect( {<(columns)>}, <(where)>, <(order)>, <(into)> )
```

Figure 8

The above SELECT statement would be converted to the following by the pre-processor during the compile process (Figure 9):

```
SQLSelect( {"NAME", "PHONE"}, 'ZIP="83706"', "PHONE", "BOISE" )
```

Figure 9

Most Xbase++ commands are defined in STD.CH. Look at this file to get a better understanding of how the pre-processor is used to create the Xbase++ command set.

Declarations

Variables are declared LOCAL or PRIVATE at the beginning of any function or procedure. LOCAL variables are visible only within the function where declared. PRIVATE variables are visible within the function where declared and all functions called by the function where declared. LOCALS are not added to the symbol table, therefore they cannot be used in macros or saved to an XPF memory file. LOCALS should always be used instead of PRIVATES other than under special conditions. If a called function needs access to a variable in the parent function, then the variable should be passed.

Private

All received parameters are always declared as LOCAL unless using the PARAMETERS statement which assigns them as PRIVATE.

Public

A variable can be declared PUBLIC anywhere in the application. PUBLIC variables are also visible anywhere in the application.

Static

A variable can be declared STATIC at the top of a source file or at the top of a function. If it is declared at the top of the source file, then it is visible by all functions declared in that source file. If it is declared at the top of the function, then it is visible only within that function.

STATICs should be used with caution, because even though they have limited visibility, they keep their value throughout the application and could become memory hogs. STATICS should never be used as a method of passing values between functions.

Local

LOCAL variables should be used whenever possible, especially when writing functions or classes that are reusable. This not only protects the value from being overwritten but also they improve performance due to they are not stored in the symbol table.

LOCALS may even be used in functions that return a code block to be evaluated later. Such LOCALS are referred to as "detached locals" because they exist after a function ends and thus are detached from the declaring function. The LOCAL variable references are removed from memory only after there are no more references to the code block and they have all been destroyed (`Figure 10`).

```
FUNCTION Main()

LOCAL bElapTime := ElapTime(), i

FOR i := 1 TO 10
  ? Eval(bElapTime)
  Sleep(50)
NEXT

WAIT

RETURN nil

* ---------

STATIC FUNCTION ElapTime()

LOCAL nSeconds := Seconds()

RETURN {||Seconds()-nSeconds}
```

Figure 10 – Example of a detached local

All variable types may contain values of any data type

- C – Character
- D – Date
- N – Numeric
- L – Logical
- A – Array
- B – Codeblock
- - Object

The MACRO Operator

The MACRO (&) operator is basically a runtime compiler. It can compile and execute any valid expression that contains PUBLIC functions, PUBLIC variables, PRIVATE variables, DATABASE FIELDS, and PUBLIC or PRIVATE object iVars and Methods. It will not work with strings that contain references to LOCAL or STATIC variables or STATIC functions. A MACRO function can even be called with local variables (`Figure 11` and `Figure 12`).

Examples of VALID macros:

```
cString := 'MsgBox("The time is: " + Time())'
&(cString)

PRIVATE cTime := Time()
cString := 'MsgBox("The time is: " + cTime)'
&(cString)

PRIVATE oXbp := XbpStatic():new():create()
cString := 'oXbp:setCaption("Testing")'
&(cString)

LOCAL dDate := Date(), cTime := Time()
cString := "PostDateAndTime"
&(cString)(dDate,cTime)
```

Figure 11 – Examples of valid macros

```
LOCAL cTime := Time()
cString := 'MsgBox("The time is: " + cTime)'
&(cString)

STATIC soXbp
soXbp := XbpStatic():new():create()
cString := 'soXbp:setCaption("Testing")'
&(cString)
```

Figure 12 – Examples of invalid macros

ARRAYS

Arrays in Xbase++ are virtually unlimited in size and capability. Arrays can contain any data type and may have a multiple dimensions. Arrays are defined in a variety of ways and can be manipulated with a variety of functions.

Declaring an array can be done in several ways (Figure 13):

```
LOCAL aArray := {}                  // creates an empty array

LOCAL aArray[10]                    // creates an array of 10 elements

LOCAL aArray[5,6]                   // creates a 2 dimensional array of 30 elements

LOCAL aArray[10,10,10]              // creates a 3 dimensional array of 1000 elements

LOCAL aArray := Array( 10,10,10 ) // same as above
```

Figure 13

Arrays are manipulated with functions like AAdd(), ADel(), AIns(), ARemove().
Any element of an array can be any data type including another array.

Arrays are simply pointers to areas of memory therefore many pointers can point to the same array and manipulate or access the same data. When an array is passed to another function, only the pointer is passed, not the array contents. This allows the function to change the contents of the original array in the calling function.

Arrays can be "ragged" with any type of non-symmetrical structure (Figure 14).

```
// ragged array
aArray := { 1, Date(), Time, { 3,4,5 }, .f., {||Directory()} }

// load a 2-dimensional array with the contents of a disk folder.
aDir := Directory()
```

Figure 14

The INDEX Operator

The INDEX Operators [] are used to access an array element or a string character (Figure 15).

```
aArray[2,3] // returns a pointer to the 3rd element of the 2nd array element of aArray.

cString[10] // returns the value of the 10th character of cString.
```

Figure 15 – Examples of index operators

The REFERENCE operator

The REFERENCE operator (@) is used to pass a value by reference rather than by value. Basically, this means that a pointer to a variable in memory is being passed rather than the contents of the variable. All objects and array are automatically passed by reference, therefore the @ symbol is not required (Figure 16).

```
LOCAL dDate := Date()

GetDate( @dDate )
```

Figure 16 – Example of the @ (reference) operator

The function GetDate() can change the value of dDate which will then change the value in the calling function.*

The reference operator can even be used with database fields and object iVars.

The ALIAS operator

The ALIAS (->) operator is used with database operations to connect a field name or expression to a specified database. This reduces code and insures that a database work area is selected for a specified operation. If the ALIAS operator is NOT used, then the programmer risks that an operation may be performed on a wrong database work area (Figure 17 Figure 17).

```
SELECT customer
DO WHILE !Eof()
  ? cust_nmbr
  dbSkip()
ENDDO
CLOSE

SELECT invoice
DO WHILE !Eof()
  ? inv_nmbr
  dbSkip()
ENDDO
CLOSE
```

Figure 17 – Accessing a table or database without the Alias operator

```
DO WHILE !customer->(Eof())
  customer->(dbSkip())
  ? customer->cust_nmbr
ENDDO
customer->(dbCloseArea())

invoice->(dbSkip())
DO WHILE !invoice->(Eof())
  ? invoice->inv_nmbr
  invoice->(dbSkip())
ENDDO
invoice->(dbCloseArea())
```

Figure 17a – Accessing a table or database using the Alias operator

CODE BLOCKS

The code block is one of the most powerful data types of Xbase++ because it is used to store a snippet of code that can be executed at any later time using the Eval() function. This is much more powerful than using a Macro because it can store a comma-delimeted list of expressions to evaluate and can be passed an unlimited number of parameters. The return value of the last expression is the value returned from Eval(). Unlike macros, code blocks can contain references to LOCAL and STATIC variables and STATIC functions. Code blocks () can be stored in arrays or memory variables and can even be saved to disk using Var2bin(`Figure 18`).

```
bEval := {|a,b,c|c := a+b, c += 20, c}

n := Eval(bEval, 3,6)
? n  //  displays 29

Example 2: aEval := { {||MyFunc1()}, {||MyFunc2()}, {||MyFunc3()} }
bEval := aEval[2]
Eval(bEval) // Runs MyFunc2()
```

Figure 18

ERROR HANDLING

ERRORSYS.PRG contains the source for the default error handler in Xbase++. The function ErrorSys() is automatically called on startup before the Main() function (`Figure 19`).

```
************************************
* Install default error code block
************************************
PROCEDURE ErrorSys()
   ErrorBlock( {|o| StandardEH(o)} )
RETURN
```

Figure 19

The function ErrorBlock() is used to post a code block that is evaluated every time there is a runtime error. By default this will call StandardEH() and pass it the Error() object. Many times it is necessary to trap runtime errors so they will not be displayed by the standard error handler but will give the user a more friendly error message or even an opportunity to retry. The below example uses BEGIN SEQUENCE .. RECOVER .. END SEQUENCE .. LOOP to recover from an error. For FoxPro developers, this is similar to TRY .. CATCH .. RECOVER.

In the example, the UDF DbfNtxUse() opens a DBF file and its index files. The information for creating the index files is passed in an array. When an index file does not exist, a runtime error occurs and the function Break() branches out of the error block to the RECOVER statement and passes an error object to it. After RECOVER, the missing index file is created (`Figure 20`).

```
#include "Error.ch"

PROCEDURE Main

LOCAL aIndexFiles

aIndexFiles := ;
   { ;
     { "CUSTOMRA", "CUSTNR" }, ;
     { "CUSTOMRB", "Upper(LAST_NAME + FIRST_NAME)" }, ;
     { "CUSTOMRC", "ZIP" } ;
   }

DbfNtxUse( "CUSTOMER", aIndexFiles )

RETURN

PROCEDURE DbfNtxUse( cDbfFile, aIndexFiles )
LOCAL n, nMax, bSaveErrorBlock, oError, cIndexFile, cIndexKey

USE (cDbfFile) NEW                // Open DBF file

n := 1
nMax := Len( aIndexFiles )

// Define error block
bSaveErrorBlock := ErrorBlock( {|e| Break(e)} )

 DO WHILE n <= nMax

   BEGIN SEQUENCE                 // Get index data
   cIndexFile := aIndexFiles[n,1]
   cIndexKey  := aIndexFiles[n,2]
   OrdListAdd( cIndexFile )    // Open index file
   n++

   RECOVER USING oError

   IF oError:osCode == 2       // Error: file does not exist

     USE (cDbfFile) EXCLUSIVE // open DBF file exclusively

     INDEX ON &cIndexKey ;    // create index file
           TO &cIndexFile

     USE (cDbfFile)           // Open file SHARED without
                                    // index files
     n := 1
     LOOP                     // Back to BEGIN SEQUENCE

    ENDIF

   END SEQUENCE
 ENDDO

ErrorBlock(bSaveErrorBlock)       // Restore old error code block

 RETURN
```

Figure 20

Xbase++ is capable of creating and using source code in an object-oriented style. This includes the declaration of classes and the instantiation of classes. Below (Figure 21) is an example of a very simple class that contains a few iVars and methods. This class creates and writes to a text file.

```
#include "fileio.ch"
CLASS textfile

EXPORTED:
VAR handle
VAR pointer
METHOD init, close, write

ENDCLASS

METHOD textfile:init( cFileName )
IF !Empty(cFileName)
  IF FExists(cFileName)
    ::handle := FOpen( cFileName, FO_WRITE )
  ELSE
    ::handle := FCreate( cFileName )
  ENDIF
  ::pointer := FSeek(::handle,0,FS_END)
ELSE
  ::pointer := 0
  ::handle := 0
ENDIF
RETURN self

METHOD textfile:close()
FClose(::handle)
RETURN self

METHOD textfile:write( cText )
IF !Empty(cText)
  FWrite( ::handle, cText + Chr(13) + Chr(10) )
  ::pointer := FSeek(::handle,0,FS_END)
ENDIF
RETURN self

* --------
* Instantiation and usage of the class

FUNCTION Main()

LOCAL oText,i

oText := TextFile():new( 'MyText.Txt' )
IF oText:handle > 0
  FOR i := 1 TO 20
    oText:write('This is line ' + Alltrim(Str(i)))
    ? oText:pointer
  NEXT
  oText:close()
ENDIF
wait

RETURN nil
```

Figure 21 – Declaration of a class

XBASE PARTS

Xbase Parts is the terminology that Alaska Software uses to describe a set of GUI controls that are used to create robust GUI applications. Xbase Parts are all classes (Figure 22) that start with Xbp*.

```
XbpDialog()     // Create the main Dialog window for an application or
window.
XbpPushButton() // Create a pushbutton
XbpBrowse()     // Create a columnar browse listing of an array or
database.
XbpMenu()       // Create a submenu or menu item
XbpTabPage()    // Create a Tab Page

.. many more

Sample code:

#include "appevent.ch"

FUNCTION Main()

LOCAL oDlg, oButton, nEvent := 0, mp1, mp2, oXbp

oDlg := XbpDialog():new(AppDeskTop(),,{ 100,100 }, { 400,400 } )
oDlg:title := 'My Window'
oDlg:create()

oButton := XbpPushButton():new( oDlg:drawingArea,, {50,50}, {300,280} )
oButton:caption := 'Push Me!'
oButton:activate := {||MsgBox('I have been pushed!')}
oButton:create()
oButton:setFontCompoundName('36.Arial Bold')

// Need event loop to process all events
DO WHILE nEvent # xbeP_Close
  nEvent := AppEvent( @mp1, @mp2, @oXbp, .1 )
  IF nEvent > 0
    oXbp:handleEvent( nEvent, mp1, mp2 )
  ENDIF
ENDDO

RETURN nil

* -------

PROC appsys ; return
```

Figure 22

Xbase++ can execute code in multiple threads, simultaneously. Look at the below code. You can run this code by building THREAD.XPJ and then running THREAD.EXE. Note how the messages from the threads are interlaced indicating that they are running simultaneously (Figure 23).

```
FUNCTION Main()

LOCAL oThread, lThread1Active := .f., lThread2Active := .f.

oThread := Thread():new()
Sleep(5)
oThread:start({||ThreadLoop(@lThread1Active)})

oThread := Thread():new()
Sleep(5)
oThread:start({||ThreadLoop(@lThread2Active)})

DO WHILE lThread1Active .OR. lThread2Active
  Sleep(1)
ENDDO

RETURN

* ---------

STATIC FUNCTION ThreadLoop( lActive )

LOCAL i

lActive := .t.

FOR i := 1 TO 5
  ? 'I am running in thread ' + Alltrim(Str(ThreadId()))
  Sleep(50)
NEXT

USE xtest
XTEST->(dbGoTo(ThreadID()*2))

i := 0
DO WHILE i++ < 5 .AND. !XTEST->(Eof())
  ? 'I am record number ' + Alltrim(Str(XTEST->(RecNo()))) + ' in thread ' + ;
    Alltrim(Str(ThreadId()))
  XTEST->(dbSkip())
  Sleep(50)
ENDDO

XTest->(dbCloseArea())
lActive := .f.

RETURN nil
```

Figure 23

Each thread also contains a workspace in which up to 65,000 database aliases may be opened. The workspace in a thread is not affected by database operations in other threads, making the workspace "thread-safe".

Look at the above code. Note how the same database can be opened in 2 different threads and be accessing different records.

ACTIVEX (COM and OCX controls)

Xbase++ supports all third-party ActiveX controls. ActiveX controls that have a visual component (OCX) are usually handled by the XbpActiveXControl() class. Here is an example of code that creates a Web Browser using the ActiveX control that is part of Internet Explorer. The CLSID for this control is "Shell.Explorer" (Figure 24).

```
#Pragma Library("ASCOM10.LIB")
#include "appevent.ch"

FUNCTION Main()

LOCAL oDlg, oWebBrowser, nEvent := 0, mp1, mp2, oXbp

oDlg := XbpDialog():new(AppDeskTop(),,{200,200},{700,500})
oDlg:title := 'Alaska Software'
oDlg:resize := {||oWebBrowser:setSize(oDlg:drawingArea:currentSize())}

oWebBrowser := XbpActiveXControl():new(oDlg:drawingArea,,
                  {0,0},oDlg:drawingArea:currentSize())

oWebBrowser:clsId := 'Shell.Explorer'
oWebBrowser:create()
oWebBrower:navigate('http://alaska-software.com')

DO WHILE nEvent # xbeP_Close
  nEvent := AppEvent(@mp1,@mp2,@oXbp,1)
  IF nEvent > 0
    oXbp:handleEvent(nEvent,mp1,mp2)
  ENDIF
ENDDO

RETURN nil

PROC appsys ; RETURN
```

Figure 24

In FoxPro there is a very easy way to export records into MS-Excel as in Figure 25:

```
COPY FIELDS cust_name, zip, address to c:\testing TYPE XLS
```

Figure 25

The below code shows how to create this functionality using the preprocessor to create the command and CreateObject() to connect to Excel (Figure 26).

```
#Pragma Library("ASCOM10.LIB")

#command COPY TO <(file)> TYPE XLS ;
          [FIELDS <flds,...>] ;
          [   FOR <for>] ;
          [ WHILE <whl>] ;
          [  NEXT <nxt>] ;
          [RECORD <rcd>] ;
          [ <rst: REST>] ;
          [   VIA <dbe>] ;
          [ ALL ] ;
=> _dbExport( 'C:\Temp\ExcelData', {<(flds)>}, __EBCB(<for>),
__EBCB(<whl>),<nxt>, <rcd>, <.rst.>, <dbe> ) ;
   ;M->_oExcel := CreateObject("Excel.Application");
   ;M->_oExcel:DisplayAlerts:=.f.;
   ;M->_oBook:= M->_oExcel:workbooks:Open('C:\temp\ExcelData.dbf');
   ;M->_oBook:SaveAs(<(file)>+".xls", -4143 );
   ;M->_oExcel:Quit();
   ;M->_oExcel:Destroy();
   ;Ferase('C:\temp\'ExcelData.Dbf')

* Usage of the COPY TO .. TYPE XLS command
FUNCTION Main()

USE customer VIA "FOXCDX"

COPY TO Customer TYPE XLS   // This will end up in
C:\Users\<login>\Documents
COPY TO (CurDrive() + ":\" +CurDir() + "\Customer") TYPE XLS
  // This will end up in current directory

RETURN nil
```

Figure 26

The GRAPHICS engine

Xbase++ supports a robust set of graphic functions. These functions start with Gra*() and can be used to paint graphs, images, gradients, etc. in a window or on any Xbase++ control that supports "owner-drawing".

Here is a sample program that uses Gra functions to draw a gradient background on a custom static object (Figure 27 Figure 27):

```
FUNCTION Main()

LOCAL oDlg, oStaticG, oStatic, nEvent, mp1, mp2, oXbp

oDlg := XbpDialog():new(AppDeskTop(),,{200,200},{200,200})
oDlg:title := 'Gradient Static'
oDlg:create()

oStaticG :=
GradientStatic():new(oDlg:drawingArea,,{0,0},oDlg:drawingArea:currentSize()
)
oStaticG:color := {100,0,0} // red
oStaticG:gradientStep := 2
oStaticG:create()

oStatic := XbpStatic():new(oStaticG,,{30,80},{180,20})
oStatic:caption := 'Gradient'
oStatic:setFontCompoundName('18.Lucida Console Bold')
oStatic:setColorBG(XBPSYSCLR_TRANSPARENT)
oStatic:create()

nEvent := 0
DO WHILE nEvent # xbeP_Close
  nEvent := AppEvent(@mp1,@mp2,@oXbp,1)
  IF nEvent > 0
    oXbp:handleEvent(nEvent,mp1,mp2)
  ENDIF
ENDDO

RETURN nil
```

Figure 27

```
* --------------
CLASS GradientStatic FROM XbpStatic

EXPORTED:

VAR gradientStep
VAR color

* ----------
INLINE METHOD Init(a,b,c,d,e,f,g,h)

::gradientStep := 1
::color := { 0,100,0 } // green
::XbpStatic:init(a,b,c,d,e,f,g,h)
::XbpStatic:drawMode := XBP_DRAW_OWNER

RETURN self

* ----------
INLINE METHOD Create(a,b,c,d,e,f,g,h)

::XbpStatic:create(a,b,c,d,e,f,g,h)

RETURN self

* -----------
INLINE METHOD Draw( oPS, aInfo )

LOCAL aColors, i, nHeight := ::currentSize()[2]
LOCAL aStartPos := {aInfo[4][1],aInfo[4][2]}
LOCAL aEndPos   := {aInfo[4][3],aInfo[4][4]}

aColors := AClone(::Color)

FOR i := 1 TO nHeight/2
  aColors[1] += ::gradientStep
  aColors[2] += ::gradientStep
  aColors[3] += ::gradientStep
  GraSetColor( oPS, GraMakeRGBColor(aColors))
  GraBox( oPS, aStartPos, aEndPos, GRA_OUTLINE, 0, 0 )
  aStartPos[2]++
  aEndPos[2]--
NEXT

RETURN self

ENDCLASS
```

Figure 27a

The Garbage Collector

Xbase++ has a running thread that is used for memory cleanup. When a memory variable no longer has any reference, it is "released" from memory. For example, LOCAL variables are released at the end of the function in which they are called, unless they are "detached", then they are only released when the codeblock referencing them is released. PRIVATE variables are released at the end of the function in which they are called.

The garbage collector is responsible for de-allocating system memory whenever variables are released. It is a good practice to use LOCAL and PRIVATE variables to insure that memory is properly de-allocated. PUBLIC and STATIC variables are never released, therefore it is not a good practice to store arrays or objects to PUBLIC and STATIC variables. If a STATIC variable must be used to temporarily point to an array or object, then the only way to release the memory used by the array or object is to set the STATIC variable to NIL.

Introduction to Structured GUI Langage (SGL) programming

SQL, otherwise known as Structured Query Language, has been instrumental in simplifying data access in large applications and therefore has become predominant in most languages. This is because **SQL is a descriptive language**. SQL describes the desired dataset rather than coding the methods to achieve that result. It eliminates the need for all that complicated code because that code resides on the SQL server.

So I started putting together in my mind a new concept, which I will refer to now as **SGL (Structured GUI Language).** I wanted to make it possible to describe all the elements of a complicated GUI screen in a structured language that was easy to write and also to understand. It needed to have the ability to code the screens in a set of commands in which the command name described the type of GUI control and each command parameter described the properties of the control, including its relationship to all other controls. In Clipper and FoxPro 2.6, there was a similar concept that created @ .. SAY .. GETs on the screen and then managed the navigation through the screen via a “reader” command named READ. It occurred to me at this approach could be applied to every possible kind of GUI control. The advantages would be numerous.

1. It would make it easy to convert existing Clipper text-based code, FoxPro 2.6 text-based code, and VFP form meta-data to Xbase++ GUI-based code.

2. It would eliminate the need for Clipper and VFP programmers to learn object-oriented programming and Xbase parts, in the same way that SQL programming eliminates the need to learn all the db*() database functions, except when it comes to GUI there are many more controls and properties to learn.

3. A structured GUI language would dramatically simplify the design of complicated GUI screens and make it unnecessary to use a form designer.

4. The commands are Xbase++ code therefore they are part of the .PRG source and can be combined with the business logic. This makes the code much easier to read and maintain than disjointed concepts like form designers.

5. Because the commands are Xbase++ code, they can include debugging information entered by the programmer.

6. The programmer can easily visualize the screen design and the relationships between GUI controls by just looking at the source code.

7. The programmer can write code in a syntax that is very forgiving and does not require knowing the order of parameters or the names of methods and iVars.

I knew that everything had to be written entirely in Xbase++ for this concept to become real and I also knew that Xbase++ supported the very features necessary to make this happen.

1. A powerful pre-processor capable of creating commands with numerous optional clauses.

2. The ability to bind LOCAL variables and Instance variables to the naming of GUI objects for the creation of parent/child relationships and linking the data to the GUI object.

3. A multi-dimensional, ragged array system that served as a container for every GUI control, every property, every parent/child relationship and every callback.

4. The ability to create new classes that inherited from Xbase Parts and other classes which will be used by the reader to create the screens from the definitions in the array.

5. The ability to create a reader function which will build and render the screens from the information in the array.

6. The ability to control events in the reader for screen management and navigation.

7. The ability to give the programmer the option of using pixel-based coordinates or text-based coordinates.

8. The ability to give the programmer the option of using a top-down coordinate system or a bottom-up coordinate system.

Linear vs Lateral Programming

I have used both linear and lateral programming techniques over the years and I have come to find that linear coding not only creates faster development but also easier maintainability because it is always easier to read and understand the source code.

In a linear approach, the application screen design and the business logic are usually in the same source file. Maintaining the code seldom requires opening more than one file and the

programming can all be done within the same editing program. Applications are usually broken down into modules in which the source for an entire module may exist in only a few functions or class methods. A module is usually a program that encapsulates all the functionality for a given task or database manager. For example, a module may be a "customer" screen which provides for browsing, searching and data-entry of customers. Because all tasks supporting customer management are related, it makes sense that all the code should be in a single source file

whenever practical. In a linear programming style, the code that designs the screen elements should be placed in a top-down structure that easily defines the relationships between all the GUI controls. The more function calls or class methods that are required to do this, the more disjointed is the code and the harder it is to follow the logic and the structure.

Language in which the screen design is not done in code at all requires many more editing tools, and such tools do not lend themselves to any concept of linear programming. On the contrary, this is a more lateral style and when taken to an extreme can even be referred to as "atomic." Visual Objects was an example of a programming language that had everything in a repository with each VARIABLE, FUNCTION, COMMAND, CLASS, METHOD, iVAR store as an "atom".

Visual Objects failed for many reasons but mostly it was very difficult for a programmer to use his/her own editor for writing a program. Data driven programs have the same set of problems in which much of the code is not code at all but instead a set of records in a database that defines screen output. Alaska Software has been struggling for years trying to adapt Xbase++ to the needs of the FoxPro community because Visual FoxPro developers could only create screens from a design tool and not from code. In the early days of Xbase++, Alaska Software decided that the language was more important than the "development environment". The language was so powerful that each programmer could create their own development environment. I used the Xbase++ language to create my own debugging tools, a dot-prompt interpreter, and a command-based GUI language. Clipper programmers did not complain because they were used to writing code to accomplish a given task. Visual FoxPro programmers come from the Microsoft world in which they were given everything needed to create a program without having to embrace the GUI elements of the language, therefore Alaska Software is finding themselves in the unenviable position of having to create an entirely new programming environment even though it will have absolutely no effect on producing a better end application.

Bobby Drakos have been talking about this for years and have been wondering why Visual Foxpro developers have been so slow at jumping aboard whereas Clipper developers were instant believers and their applications are the proof. I have been to every SWFox/SWXbase conference in the past 5 years and have seen no further movement towards Xbase++ even though Visual Foxpro applications have no other language in which to migrate. So we thought we would make an effort to expose Visual FoxPro developers to an idea that they can use right now and not wait for Polar Fox or some other environment.

An SGL Architecture

I have seen some resistance to the SGL idea from seasoned programmers because they are transfixed on the idea that this isn't really programming and that a command-based language for GUI is like going backward rather than going forward. This is an illogical argument in the same way that programmers who refuse to use SQL say that only ISAM can handle the complexities of a robust database browsing system. I too had that opinion for many years but I evolved to understand why SQL became more popular over the years and ISAM programming became less popular. ISAM was not replaced by SQL. On the contrary, SQL engines are all driven by ISAM navigation and are mostly written in function based or object-oriented languages. The idea of

SQL came about because it was taking too long for application programmers to get the datasets they needed. They had to have a language that could improve their productivity and give them reliable results. SGL solves the same problem for GUI screen design that SQL solves for database management.

Before starting this project, I had to write down some objectives to insure that this would not soon become another extinct idea.

1. Easy adaption to existing application code for migration purposes.

2. Good performance.

3. Must be written entirely in Xbase++.

4. Must be able to evolve in features and robustness.

5. Must be able to create applications as complex and robust as those written entirely in object-oriented or function based code.

6. Must increase programmer productivity 10 to 20 times over native language programming even when writing entirely new applications.

7. Must be a "language extension" rather than a "framework". Applications written in the SGL syntax must not have a specific look or functionality. Flexibility is an absolute must.

8. Must have no restrictions in the use of LOCAL, PRIVATE, STATIC and PUBLIC variables. Should allow application code that uses only LOCAL variables. Must NOT REQUIRE any PUBLIC variables or PUBLIC functions to bind GUI elements or other GUI screens to each other.

9. Must have no restrictions in the development of multi-threaded applications.

10. Must allow the ability to create any kind of GUI screen or relationships to other GUI screens following all the rules and capabilities of Xbase Parts including Parent, Owner, Sibling relationships and screen modality.

11. Must be compatible with screens and business logic written using native language programming, ie SGL objects can be intermingled with all other Xbase++ GUI objects.

12. GUI classes used by the SGL language must be Xbase Parts or derived from Xbase Parts and respond to the same methods and iVars.

13. Must offer features that are difficult to develop using other methodologies.

14. Installation must have a directory structure that makes it simple to deploy a patch for fixing bugs or adding a feature without needing to wait for a new release.

15. An application that uses the SGL language must be allowed to be any common programming structure, be it function based or object oriented, ie. SGL commands should be usable within OOPS methods or functions.

16. The SGL language must have a common structure for all GUI controls. For example subclauses like WHEN, HIDE, FONT, COLOR, RESIZE, SIZE, LOCATION, PARENT, VALID, GOTFOCUS, LOSTFOCUS, DRAG, DROP, TOOLTIP, MESSAGE, SUBCLASS, etc. must have the same meaning for each GUI command.

17. The programmer must be able to SUBCLASS a GUI command to build custom features into the underlying class architecture for each command.

18. The SGL language commands must not conflict with or replace any native Xbase++ commands.

19. The SGL language must allow flexibility in handling of data types that are bound to GUI controls. For example, a color may be passed as a numeric value, a RGB array or a code block.

20. The underlying architecture must also handle all the screen management and event management such as navigation rules, resizing rules, scaling, drag and drop, etc.

21. Must be well documented and include a plethora of examples.

The next step - eXpress++

What I had in mind became my product eXpress++ and it has been evolving now for 16 years along with hundreds of applications based on this concept. We will spend time this afternoon showing how to use eXpress++ and how to get support from the eXpress++ community.

Attendees of this session will receive eXpress and a 1 year subscription.

There are no strings attached. This is the full package including source code.

Conclusion

Getting started with Xbase++ is not a difficult process if you have any programming experience at all, but it is especially easy if you have experience in the Xbase dialect languages. You are about to enter a new era of powerful programming capability. Congratulations!!

Thank you for your attendance at this workshop.