Black Forest Software Corporation's

Crossfire Library

Author:	**Daniel Martini**
Copyright:	**© Black Forest Software Corporation, 1993, All rights reserved.**
Product:	**Paradox for Windows MDI/intra-application communications system**

Summary: Crossfire is a Paradox for Windows compiled library (XFire.LDL) containing methods that offer enhanced application window management functions. The functions are geared primarily to the management of multiple forms in an application and provide easy-to-use MDI (Multiple Document Interface) programming techniques, as well as simplifying the passing of data between forms, and handling auto-incrementing fields.

Introduction

Crossfire is a Paradox for Windows ObjectPAL library, XFire.LDL, that makes ObjectPAL MDI programming easier. MDI or Multiple Document Interface, interpreted loosely refers to an application that allows several child windows open on the desktop at once in an organized fashion. Used properly, MDI gives an application unparalleled flexibility and a highly polished look and feel. However, MDI programming in Paradox for Windows is difficult even for experienced professional ObjectPAL programmers.

Crossfire was designed to make MDI programming easy by placing the burden of window management on a pre-programmed ObjectPAL library, and to remove some of this burden from the developer. The Crossfire library is very easy to use and understand and can add remarkable power to Paradox for Windows applications. Now, no matter how your user navigates though your program, you will have instant reference to, and control of, any and all forms that they have open at a given time. Simply register the library in each form in your application using the provided API (application programming interface) and you can begin using the suite of MDI methods now at your control.

Crossfire is written entirely in ObjectPAL and as such uses many methods which provide information about forms to do its work. In fact two methods, MDITile and MDICascade, place direct calls to the appropriate menuAction statements to tile and cascade windows, for your convenience. All other Crossfire methods are written to expand on several built-in ObjectPAL methods and replace others.

Many of the native Paradox for Windows methods that provide information about forms on the desktop can be used to simulate Crossfire's behavior, but require significant coding to do so. One of the benefits of using Crossfire is that rather than attaching to another window's title to get its form handle you simply refer to that form by name. Many programmers use the window title to provide information to the user and doing so change the title many times during program execution. This makes getting a form handle based on the title difficult and tedious since you must maintain record of the titles' various states at runtime and provide this information to all forms that may need it. Additionally, if you hard code form titles in your code to permit attaching to them, and you decide later to alter these titles at design-time, you must change all explicit references to the titles to reflect the new values. This too can become very tedious. Dynamic arrays are used extensively in Crossfire to provide a logical association between the name of a form and the various handles available for it. As long as you know the name of a form, you have access to its handle.

The Crossfire library is targeted at Paradox for Windows users and developers who have at least a basic understanding of ObjectPAL programming. It is not necessary to understand how to build a library to use Crossfire, but the user is expected to understand rudimentary ObjectPAL programming concepts such as declaring variables and calling methods for Crossfire to be useful.

For those who do know how to construct libraries, Crossfire will prove eminently useful since it can offer a great deal to the programmer interested in reducing coding time and who is interested in reusable code. Crossfire routines are used in a custom library in exactly the same way as they are used in forms, giving you the ability to centralize your window handling program code. The example application included with Crossfire, SoftWatch, shows how to integrate Crossfire with forms as well as with another custom library of your creation. SoftWatch is a full-featured, "real world" application containing a wealth of useful ObjectPAL programming techniques and is provided with full source code, except for that of the Crossfire library itself which will be provided upon product registration.

Intra-application Communication

Intra-application communication (passing data and messages between forms) has always been regarded as tricky at best in ObjectPAL programming. Crossfire makes this daunting and tedious task easier. By registering the Crossfire Library in each form in your application, the channels are opened to **directly** assign values between forms. To send or receive detailed information and data among open forms in an application, simply call one of the many methods available in the library for this purpose.

In Paradox for Windows and ObjectPAL, there is no such animal as a truly "global" variable. You cannot declare a variable that exists at the **application level** which can be used as a means of shuttling values from form to form. This does have advantages, though they may not at first be apparent. Global variables as found in other languages such as BASIC or XBase can provide an easy way to move global data between entities, but this comes at a price. Global variables used wantonly can make a program difficult to follow and often with disastrous results.

When you consider the complexity of the Paradox for Windows containership model, the way objects may be owned or "contained" by other objects, and the inherent risk associated with global variables that can be changed anywhere and anytime, you might imagine how they could get you into trouble. If you were to place references to a global variable at several locations in a complex form, and then place references to the same global variable in several locations in several other complex forms, and you discovered that the variable was taking on a rogue value *somewhere* at runtime*,* you have what professional programmers call a **serious problem**.

Nonetheless, the need to pass data among form windows in an application is a very real one. The way this is done is via a library. Libraries in ObjectPAL should be thought of differently than libraries in other languages. In other languages, libraries are simply files that contain pre-programmed routines that are used during program execution by referencing them in your program at the appropriate places in your code. ObjectPAL libraries serve the same purpose, but in addition to containing executable code, they may also contain variables that can hold data at runtime. Several forms can share a library at the same time and by extension share the same data area within a library as well. As long as a library remains open and referenced by at least one form, the data area is preserved. Other forms may tap into this data area by opening the library with the GlobalToDeskTop parameter, so long as methods exist to provide access to the data values.

You see, library variables cannot be considered truly global either. They are directly accessible to methods and procedures in the library, but not directly accessible outside of the library. However, library methods have the ability to pass values into and out of the library's variables easily enough.

Typically, ObjectPAL programmers pass data between forms by creating a small library of methods and variables which is then used as a vehicle for form to form communication. The library should contain a full complement of methods that allow you to send data to its variables and to retrieve those values again at a later time, usually from another form. At the simplest level, methods can be created to pass each of the standard data types, but for more complex applications such limitations would prove challenging at best.

Beyond the issue of passing data from form to form is the issue of controlling one form from another at runtime. The most common way to manipulate a remote form is to open it and keep the handle from that opening operation or to attach a form variable to another form's title. Additionally, the form.wait() method is used frequently as a way of handling a second form during program execution. This has the unfortunate side-effect of making the application somewhat "modal". Although there are times when making a form appear modal to the user is desirable, a Windows application that is too rigid in nature tends to defeat the purpose of using a windowing environment. Not to mention that it can make a perfectly good program appear amateurish to the users of the system.

Data may be passed between forms in two distinct ways using the Crossfire library. The first technique is based on the concept of sending and receiving data to and from library variables. The Crossfire methods that take this approach work with dynamic arrays. A **globalDataBag** is an internal Crossfire dynamic array used for storing global data. By defining a custom type similar to this in your forms, you can pass a variety of data from a form to one or more other forms in one fell swoop. The concept is simple; in your **type** declaration window create a type called AnyBag as a dynamic array of AnyType.

```
type
; Creating an AnyBag
AnyBag = dynarray[] AnyType
endType
```

Now you can use this type to declare variables of your own that can be passed into Crossfire for distribution to other forms easily using the **xf_GDBSet** and **xf_GDBGet** methods.

Briefly, here is a simple example of how you would send and receive data between forms:

In the form sending the data, load your values into a local variable of type AnyBag and send them to the globalDataBag by calling a method, xf_GDBSet.

```
; formOne::SendData::pushButton
method pushButton(var eventInfo Event)
var
        localDataBag AnyBag
endVar

localDataBag["CompanyName"] = "Black Forest Software Corp"
localDataBag["NumberOfEmployees"] = 30
```

```
; Send data to the GDB (global data bag)
Crossfire.xf_GDBSet(localDataBag)

endmethod
```

In another form, you can retrieve these values like so:

```
; formTwo::ReceiveData::pushButton
method pushButton(var eventInfo Event)
var
        myDataBag AnyBag
        CustomerName String
        CustomerSize SmallInt
endVar

; Get the data from the GDB (global data bag) into myDataBag
myDataBag = Crossfire.xf_GDBGet()

; Grab the values from the localDataBag
CustomerName = myDataBag["CompanyName"]
CustomerSize = myDataBag["NumberOfEmployees"]

endmethod
```

That's just about as hard as it gets. Of course, there other methods to complement these, but most of the work of passing data can be done with xf_GDBSet and xf_GDBGet. There are three such custom types used by the Crossfire library for various purposes.

The second method of passing information between forms using Crossfire is a little more sophisticated, but a lot more powerful. Where Crossfire shines is in its ability to provide pointers to any open form during application execution. Without regard to the way your users advance through your application, opening and closing forms in sometimes random fashion, you can get information about which forms are open and even get pointers to the forms themselves by calling Crossfire methods. The only thing you need to get a handle to a form is that form's name. The form name should be understood to mean the name of the form and not the title of that form. The form name may be accessed and changed by right-clicking on the title bar in design mode. The default name for a new form is **#Form1** and should be changed to something more meaningful early in development.

Since Crossfire can provide handles to any and all open forms from wherever you happen to be calling from, values such as object properties can be easily retrieved from other forms by using these handles. Likewise, you can execute methods for any object on any other form as well. You get and set values and execute methods by using the form's handle and the full object name.

```
method pushButton(var eventInfo Event)
var
        formName String
```

```
          formHandle Form
          seedNumber SmallInt
          ECode SmallInt
endVar

seedNumber = 100
formName = "MyProductForm"

ECode = Crossfire.xf_MDIGetFormHandle(formName,formHandle)

if ECode = 0 then
          ; Directly assign the seed value to a field on that form
          formHandle.MyTFrame.MyField = seedNumber
else
          ; An error occurred; report it
          msgStop("Crossfire Error",xf_UTErrorMessage(ECode))
endif
endmethod
```

This example is a simple one, and much more complex manipulations are possible. You may wish to perform some action on ALL open forms without knowing the names of those forms. Crossfire provides this ability with equal ease. Consider the following method that closes all forms but the one we are calling from.

```
method pushButton(var eventInfo Event)
var
          formBag dynArray[] String
          formName String
          formHandle Form
          selfName String
endVar

; Get the current form's name
formHandle.attach()
selfName = formHandle.name

; Get ALL open form names
Crossfire.xf_MDIGetAllFormNames(formBag)

forEach formName in formBag
          ; Get a handle for each of the open forms
          Crossfire.xf_MDIGetFormHandle(formBag[formName],formHandle)

          ; If the currently held handle is not for the form we're in,
          ; close that form
          if formHandle.name <> selfName then
                    formHandle.close()
          endif
endForEach

endmethod
```

You provide the name of the form and Crossfire will get the handle, if it is open. Please note again: The form name is not to be confused with the form title. A form's name may be accessed and changed by right-clicking on the forms title bar in design mode. The default name for new forms is **#Form1**, and should be changed from this default to distinguish it from other forms in you application.

The previous code example was redundant since Crossfire provides you with similar high-level methods already built-in, but it does begin to explain some of the power of the library.

The reason both techniques of communication discussed above are provided in Crossfire is that variables in a form are not part of the form's true hierarchy and thus cannot be referenced the way objects on a form can. The GDB methods and the MDI methods work together to provide you with a full suite of tools to communicate data and execute methods between forms at will.

A Word About Libraries

Libraries are files that contain user-created methods, procedures and data. You can create libraries of custom code to be reused by a single form, several forms, or several applications. Methods and procedures are created for a library exactly the same way that you would create them for a form. In fact, many libraries come about because the developer determined that a custom method on a form could be used by other forms in an application and so then added them to a library where they could be shared.

Libraries, indeed, have much in common with forms. They have **uses**, **type**, **const**, **var** and **proc** windows and built-in **open**, **close** and **error** methods. Of course, a library's main purpose is to house user created methods and procedures.

After a library has been created, its methods can be made available to forms by performing a few setup chores in the forms with which they are intended to be used. To use a library's methods in a form, you must provide the form with certain information about the library. The form's **uses** window must contain **prototypes** for the methods that will be called from the library. A prototype is nothing more than a line of code that names the method and declares its parameters and return values, if any exist. If a uses window refers to ObjectPAL library methods, the **uses** clause must contain the suffix "ObjectPAL". For example:

```
uses ObjectPAL
xf_MDIMinimizeAll() SmallInt
endUses
```

This tells the form that the library method xf_MDIMinimizeAll is an ObjectPAL method, it takes no parameters, and it returns a SmallInt (small integer) type. Now declared, this method can be called from within the form.

There are a few other details that must be provided to the form before the library may be used. The library must first be opened by the form, and a library variable is used to do this. A library variable should be declared in one of the forms **var** windows, usually at the form level so that the library is visible from anywhere on the form.

```
var
Crossfire Library
endVar
```

Any custom types used as parameters in the method prototypes must also be entered in the form that uses the custom methods. Crossfire uses three such custom types and are entered into your form's type window as follows:

```
Type
        ; Crossfire Types
        AnyBag          = dynArray[] AnyType
        FormBag         = dynArray[] Form
        StringBag       = dynArray[] String
```

```
endType
```

Finally, now that we declared our library variable, entered our "**uses ObjectPAL**" prototypes and our custom types, the library itself must be opened before the methods within it can be used. Again, for many cases it is a good idea to open the library at the form level for broadest visibility.

```
; Form level open method
method open(var eventInfo Event)
if eventInfo.isPreFilter() then
        ; This code executes for each object on the form.
else
        ; This code executes only for the form.
        doDefault

        ; Open the Crossfire library
        if NOT Crossfire.open(":CROSSFIRE:XFIRE.LDL",GlobalToDesktop) then
                msgStop("Error:", "Crossfire library could not be opened")
                self.close()
        endif
endif
endmethod
```

The library methods whose prototypes are listed in the forms **uses** window then become accessible to the form. It is not necessary to list prototypes for all of a library's methods in a form. It is only necessary to list those methods that the form will be calling. The three custom types mentioned earlier must also be pasted into your forms to support the Crossfire method prototypes.

An advantage to listing all of a library's method prototypes in a form is that it relieves you of the burden of trying to keep the uses window in synch with the methods you are actually using. As you gain proficiency with Crossfire library, you will probably find yourself using more and more of its methods to exploit the power available to you. Rather than repeatedly going back to the **uses** windows of all of your forms to update them with the method prototypes you wish to use, you should consider listing all of the library's method prototypes in the **uses** window early in development.

Using Crossfire Library

Using Crossfire library in your applications is straightforward, provided that you follow a few simple guidelines. First, it is a good idea to create an alias called **CROSSFIRE** interactively in Paradox for Windows. The sample application provided with the Crossfire library, SoftWatch, **requires** an alias named CROSSFIRE to run.

An alias easily is created in Paradox for Windows by selecting **Aliases...** from the **File** menu. In the resulting dialog box, **Alias Manager**, press the New button. Then, enter "CROSSFIRE" as the **Database Alias**, choose the "STANDARD" **Driver Type** and enter the full DOS path name where Crossfire library is stored in the **Path** edit region. Then select **Save As...** and confirm that you wish to save the alias in the ODAPI configuration file. Once this has been done, the example applications will know where to find Crossfire library, and should run correctly.

In your applications that use Crossfire, it is recommended that **all** forms are registered with Crossfire upon opening. You need not register your dialog box forms with Crossfire, but sometimes it is desirable to do so for special effects. The way to register a form with Crossfire is to first open the library and then place a call to the method, **xf_MDIConstructor**. When closing a form, call the method, **xf_MDIDestructor**. These two methods are all that are required to allow the Crossfire MDI methods to function properly. The constructor creates an entry for a form in Crossfire's dictionary. This dictionary cannot be manipulated directly, but is a network of interconnected data structures used internally to manage forms on the desktop. After the library is opened and the constructor is called, the MDI methods in Crossfire are immediately available. GDB methods do not require the constructor and destructor calls to use them. Only the MDI methods require these calls.

The destructor, as you might imagine, removes references to the calling form from Crossfire's internal dictionary structures. The constructor and destructor calls are imperative to keeping the dictionary structures up to date. While these methods may actually be called at any time, the results of improperly updating Crossfire's dictionary will be unpredictable, so registration at the form level open and close methods are recommended.

Please note that most of the methods in the Crossfire library require that the custom types and the method prototypes, listed later in this document, be pasted or typed into your form's **Type** window and the form's **Uses** window respectively. This is a very easy operation to perform and you can even cut the information directly from this document and paste it into your forms without modification as will be explained later in the "Crossfire Custom Types" and "Uses ObjectPAL Window Prototypes" sections.

If you adhere to the few simple rules noted above, the Crossfire library methods are readily available to the forms in your applications. You will undoubtedly develop your own approach to using the library, but in its simplest form, you can think of it as a global form manager. While you actually control how the application flows, you will typically make requests to Crossfire for certain resources that it maintains. If you want handles to a form or several forms, you do not need to attach a variable to another form's title, which may be constantly changing, nor do you have to explicitly open one form from another just to get the handle. You simply declare a form variable, and request the handle.   Then you can work with its objects directly and discard the variables and the handles until you need them again. The services that Crossfire provides remove much of the tedium of tracking resources yourself and lets you focus on the problems your application is designed to solve.

The best way to learn to use the Crossfire library is to explore the source code in SoftWatch, the sample application that ships with Crossfire. SoftWatch is an excellent example of how to use Crossfire in a "real world" application and contains a large amount of source code that illustrates many advanced Paradox for Windows programming constructs.

Reference

Data Passing Services

xf_GDBEmpty()
xf_GDBGet() AnyBag
xf_GDBGetValue(GDBTag String) AnyType
xf_GDBSet(GDB AnyBag)
xf_GDBSetValue(GDBTag String, NewGDBValue AnyType)
xf_GDBSwapValue(GDBTag String, NewGDBValue AnyType) AnyType

MDI Services

xf_MDICascade()
xf_MDICloseAll(editTerm String) SmallInt
xf_MDIConstructor() SmallInt
xf_MDIDestructor()
xf_MDIGetFormHandle(formName String, var formHandle Form) SmallInt
xf_MDIGetAllFormHandles(var formHandleBag FormBag)
xf_MDIGetAllFormNames(var formNameBag StringBag)
xf_MDIGetAllFormTitles(var formTitleBag StringBag)
xf_MDIIsFormOpen(formName String) Logical
xf_MDIMinimizeAll()
xf_MDIMinimizeAllButSelf()
xf_MDITile()
xf_MDIUnMinimizeAll()

Utility Services

xf_SEQIncrement(tableName String, var fieldName String) Number
xf_UTErrorMessage(xf_ErrorCode SmallInt) String

xf_GDBEmpty

METHOD Empties the contents of the globalDataBag. The globalDataBag is the central data container used by Crossfire for passing data between forms. xf_GDBEmpty removes all existing values in preparation for sending in new data.

PROTOTYPE
xf_GDBEmpty()

EXAMPLE

```
; ClearGlobalDataBag::pushButton
method pushButton(var eventInfo Event)

; Clear the GDB
Crossfire.xf_GDBEmpty()

endmethod
```

xf_GDBGet

METHOD Retrieve the contents of the globalDataBag into a dynamic array. The globalDataBag is the central data container used by Crossfire for passing data between forms. xf_GDBGet permits copying the contents of the globalDataBag into a local dynamic array.

RETURNS The contents of the globalDataBag.

PROTOTYPE
xf_GDBGet() AnyBag

EXAMPLE

```
; RetrieveGlobalData::pushButton
method pushButton(var eventInfo Event)
var
        myLocalDataBag dynArray[] AnyType
endVar

; Copy the contents of the GDB
myLocalDataBag = Crossfire.xf_GDBGet()

myLocalDataBag.view()          ; displays the values retrieved from the GDB

endmethod
```

xf_GDBGetValue

METHOD Get the value of a globalDataBag element. The globalDataBag is the data container inside Crossfire library used for passing information between forms. xf_GDBGetValue provides a means of quick retrieval of a single value from the globalDataBag. GDBTag is the index of the globalDataBag element you are interrogating.

PARAMETERS

GDBTag	Tag of the element to be retrieved

RETURNS The value of the dynamic array element in GDBTag, or "" if none

PROTOTYPE

xf_GDBGetValue(**GDBTag** String) AnyType

EXAMPLE

```
; RetrieveCustomerName::pushButton
method pushButton(var eventInfo Event)
var
        custName String
endVar

; Get the CustomerName element from the GDB
custName = Crossfire.xf_GDBGetValue("CustomerName")

if custName <> "" then
        msgInfo("Customer Name:",custName)
else
        msgStop("PROBLEM","Customer name was empty!")
endif
endmethod
```

xf_GDBSet

METHOD Pass the contents of a dynArray, the parameter *dBag*, to the globalDataBag. The globalDataBag is the central data container used by Crossfire for passing data between forms. xf_GDBSet permits copying the contents of a dynamic array to the globalDataBag. Any elements of the globalDataBag of the same name as those being sent via xf_GDBSet will automatically be overwritten.

PARAMETER

dBag A dynamic array of values to copy to the globalDataBag. An AnyBag (a dynarray of AnyType) should be used to declare dBag.

PROTOTYPE

xf_GDBSet(**dBag** AnyBag)

EXAMPLE

```
; PassCustomerInfo::pushButton
method pushButton(var eventInfo Event)
var
        CustomerInfo dynArray[] AnyType
endVar

CustomerInfo["custName"]        = "BFS Corp."
CustomerInfo["acctNum"]         = "BFS-1234"
CustomerInfo["zipCode"]         = "22172"
CustomerInfo["acctBal"]         = 250.00

; Send the CustomerInfo dynamic array to the GDB for later retrieval
Crossfire.xf_GDBSet(CustomerInfo)

endmethod
```

xf_GDBSetValue

METHOD Sets the value in a globalDataBag element. The globalDataBag is the data container inside Crossfire library used for passing information between forms. xf_GDBSetValue provides a means of quickly passing a single value to the globalDataBag. GDBTag is the index of the globalDataBag element you are assigning.

PARAMETERS

GDBTag	Tag of the dynamic array element to be set
NewGDBValue	New value to be assigned

PROTOTYPE

xf_GDBSetValue(**GDBTag** String, **NewGDBValue** AnyType)

EXAMPLE

```
; PassCustomerName::pushButton
method pushButton(var eventInfo Event)
var
        custName String
endVar

custName = "Black Forest Software Corporation"

; Send the CustomerName element to the GDB
Crossfire.xf_GDBSetValue("CustomerName",custName)

endmethod
```

xf_GDBSwapValue

METHOD Swap values in a globalDataBag element. The globalDataBag is the data container inside Crossfire library used for passing information between forms. xf_GDBSwapValue provides a means of quickly swapping the contents of an element in the globalDataBag. GDBTag is the index of the globalDataBag element you are exchanging.

PARAMETERS

GDBTag	Tag of the dynamic array element to be swapped
NewGDBValue	New value to be assigned

RETURNS The old value of the GDBTag, or "" if none

PROTOTYPE
xf_GDBSwapValue(**GDBTag** String, **NewGDBValue** AnyType) AnyType

EXAMPLE

```
; SwapContactName::pushButton
method pushButton(var eventInfo Event)
var
        oldContactName,
        firstContactName,
        newContactName String
endVar

firstContactName = "Harry Benson"

Crossfire.xf_GDBSetValue("ContactName", firstContactName)

; Next line displays "Harry Benson"
msgInfo("Current Contact Name:",Crossfire.xf_GDBGetValue("ContactName"))

newContactName = "Ian Malcom"

; Swap the ContactName element to the GDB
oldContactName = Crossfire.xf_GDBSwapValue("ContactName",newContactName)

; Next line displays "Ian Malcom"
msgInfo("New Contact Name:",Crossfire.xf_GDBGetValue("ContactName"))

; Next line displays "Harry Benson"
msgInfo("Old Contact Name:",oldContactName)

endmethod
```

xf_MDICascade

METHOD Cascade all forms windows on the desktop by calling a Paradox for Windows menuAction expression. This method is redundant and only provided for convenience.

PROTOTYPE
xf_MDICascade()

EXAMPLE

```
; CascadeWindows::pushButton
method pushButton(var eventInfo Event)

Crossfire.xf_MDICascade()

endmethod
```

xf_MDICloseAll

METHOD

Attempts to close all open forms on the desktop. xf_MDICloseAll only closes forms that are registered with Crossfire library. If a form is found to be in "edit" mode, xf_MDICloseAll handles it according to the value passed in the parameter *editTerm*. If *editTerm* is "CANCELRECORD", Crossfire cancels the record and then attempts to end edit mode. If *editTerm* is "POSTRECORD", Crossfire attempts to post the record and then attempts to end edit mode. If the operation is unsucessful, Crossfire returns a numeric error code indicating the result of the operation.

PARAMETER

editTerm

A string indicating how to handle a form found in edit mode. Either "CANCELRECORD" or "POSTRECORD" may be used. A null string, "", will cause xf_MDICloseAll to assume the default behavior which is the same as "POSTRECORD".

RETURNS

An error code, zero if successful, non-zero otherwise.

PROTOTYPE

xf_MDICloseAll(editTerm String) SmallInt

EXAMPLE

```
; CloseAndPostForms::pushButton
method pushButton(var eventInfo Event)
var
	ECode SmallInt
endVar

ECode = Crossfire.xf_MDICloseAll("POSTRECORD")

; If a problem occurred, such as a keyViolation, Crossfire will not be able to close the
; form.  Default behavior can be changed to cancel the record by using the constant
; "CANCELRECORD" instead
if ECode <> 0 then
	MsgStop("PROBLEM",Crossfire.xf_UTErrorMessage(ECode))
endif
endmethod
```

xf_MDIConstructor

METHOD Registers the current form with the Crossfire library. The constructor creates an MDI entry in the library that the library uses to represent that form. All forms in an application should call xf_MDIConstructor in the form's open method and call xf_MDIDestructor in the form's close method. Forms that do not call the constructor and the destructor will not be affected by calls to other Crossfire methods and can disrupt the services provided by the library.

RETURNS An error code, zero if successful, non-zero otherwise.

PROTOTYPE
xf_MDIConstructor() SmallInt

EXAMPLE

```
; CustomerForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter() then
        ; This code executes for each object on the form.
else
        ; This code executes only for the form.
        ; Open the Crossfire library
        if Crossfire.open(":CROSSFIRE:XFIRE",GlobalToDesktop) then

                ; Initialize the MDI entry
                ECode = Crossfire.xf_MDIConstructor()

                ; Test to see if there was a problem
                if ECode <> 0 then
                        msgStop("ERROR:",Crossfire.xf_UTErrorMessage(ECode))
                        ; Do something appropriate
                        ; The form will close automatically
                else
                        maximize()
                endif
        else
                msgStop("PROBLEM", "Crossfire library could not be opened")
                self.close()
        endif
endif
endmethod
```

xf_MDIDestructor

METHOD De-registers the form with Crossfire library. The destructor removes the MDI entry used to represent the current form from Crossfire library. All forms in an application should call xf_MDIConstructor in the form's open method and call xf_MDIDestructor in the form's close method. Forms that do not call the constructor and the destructor will not be affected by calls to other Crossfire methods and can disrupt the services provided by the library.

PROTOTYPE
xf_MDIDestructor()

EXAMPLE

```
; CustomerForm::close
method close(var eventInfo Event)
if eventInfo.isPreFilter() then
        ; This code executes for each object on the form.
else
        ; This code executes only for the form.
        doDefault

        ; Destroy the MDI entry
        Crossfire.xf_MDIDestructor()
endif
endmethod
```

xf_MDIGetFormHandle

METHOD Acquires a Form handle for the *formName* requested. This method assigns the handle to the variable parameter, *formHandle*, given by the programmer. The form named in *formName* must be open and registered to Crossfire library. If a handle cannot be retrieved, a numbered errorcode is returned.

PARAMETERS

formName	name of form for requested handle
formHandle	form variable to receive the requested handle, passed by reference

RETURNS An error code, zero if successful, non-zero otherwise.

PROTOTYPE
xf_MDIGetFormHandle(**formName** String, var **formHandle** Form) SmallInt

EXAMPLE

```
; PresentProductForm::pushButton
method pushButton(var eventInfo Event)
var
        formName String
        productForm Form
endVar

formName = "Product"

if Crossfire.xf_MDIGetFormHandle(formName,productForm) = 0 then
        productForm.bringToTop()
else
        productForm.open("PROD.FSL")
endmethod
```

xf_MDIGetAllFormHandles

METHOD Returns a dynamic array of all open form handles in the *formHandleBag* parameter.

PARAMETER
formHandleBag the dynamic array of form handles, or FormBag, to contain the handles, passed by reference. The dynamic array, *formHandleBag*, is emptied by Crossfire before new values are assigned. If the dynamic array contains values before the call to this method, all values will be lost. The index or subscript of the returned array is name of the form and the element value of the form is the form handle.

PROTOTYPE
xf_MDIGetAllFormHandles(var **formHandleBag** FormBag)

EXAMPLE

```
; AreFormsEditing::pushButton
method pushButton(var eventInfo Event)
var
        formHandleBag dynArray[] Form
        formID String
endVar

Crossfire.xf_MDIGetAllFormHandles(formHandleBag)

forEach formID in formHandleBag
   if formHandleBag[formID].editing then
        msgInfo("Guess what?","Form: "+formHandleBag[formID].name+" is editing!")
   endif
endForEach

endMethod
```

xf_MDIGetAllFormNames

METHOD Returns a dynamic array with all open form names into the *formNameBag* parameter. The dynamic array, *formNameBag*, is emptied by Crossfire before new values are assigned. If the dynamic array contains values before the call to this method, all values will be lost. The index or subscript of the returned array is the name of the form and the element value of the form is also the form name.

PARAMETER

formNameBag the dynamic array of strings, or StringBag, to contain the form names, passed by reference.

PROTOTYPE

xf_MDIGetAllFormNames(var **formNameBag** StringBag)

EXAMPLE

```
; CheckCustomerOrSalespersonForms::pushButton
method pushButton(var eventInfo Event)
var
        formNameBag dynArray[] String
        tmpForm Form
endVar

Crossfire.xf_MDIGetAllFormNames(formNameBag)

; Open customer form if need be
if not formNameBag.contains("Customer") then
   tmpForm.open("CUSTFRM.FSL")
   tmpForm.hide()
endif

; Open sales form if need be
; NOTE:The tmpForm variable can be immediately reused since we can get the form
; handles later whenever we like
if not formNameBag.contains("Sales") then
   tmpForm.open("SALESFRM.FSL")
   tmpForm.hide()
endif

endMethod
```

xf_MDIGetAllFormTitles

METHOD Returns a dynamic array with all open form titles into the *formTitleBag* parameter.

PARAMETER

formTitleBag the dynamic array of strings to contain the returned form titles, passed by reference. The index or subscript of the returned array is the name of the form and the element value of the form is the form title.

PROTOTYPE

xf_MDIGetAllFormTitles(var **formTitleBag** StringBag)

EXAMPLE

```
; ViewOpenFormTitles::pushButton
method pushButton(var eventInfo Event)
var
	formTitleBag dynArray[] String
endVar

Crossfire.xf_MDIGetAllFormTItles(formTItleBag)

formTitleBag.view()
; Shows the current window titles for all open forms registered with
; Crossfire library.  See the SoftWatch Window Manager for a practical
; use of these titles

endMethod
```

xf_MDIIsFormOpen

METHOD Determines whether formName is open and registered to Crossfire. If the form is open the method returns a logical True, otherwise it returns False.

PARAMETERS

formName	name of form to test

RETURNS True if the form is open and registered with Crossfire, False otherwise.

PROTOTYPE

xf_MDIIsFormOpen(**formName** String) Logical

EXAMPLE

```
; TestProductForm::pushButton
method pushButton(var eventInfo Event)
var
	formName String
endVar

formName = "Product"

if Crossfire.xf_MDIIsFormOpen(formName) then
	msgInfo("Form is open",formName)
else
	msgInfo("Form is NOT open",formName)
endmethod
```

xf_MDIMinimizeAll

METHOD Minimize all open forms on the desktop that are registered with Crossfire library, including the current one.

PROTOTYPE
xf_MDIMinimizeAll()

EXAMPLE

```
; MinimizeAllWindows::pushButton
method pushButton(var eventInfo Event)

; Minimizes all open forms registered with Crossfire library
Crossfire.xf_MDIMinimizeAll()

endmethod
```

xf_MDIMinimizeAllButSelf

METHOD Minimize all form windows on the desktop that are registered with Crossfire library except for the form that called this method.

PROTOTYPE
xf_MDIMinimizeAllButSelf()

EXAMPLE

```
; UnClutterDesktop::pushButton
method pushButton(var eventInfo Event)

; Minimizes all open forms registered with Crossfire library
; Except for the current form window
Crossfire.xf_MDIMinimizeAllButSelf()

endmethod
```

xf_MDITile

METHOD Tile all open form windows in the desktop by calling a Paradox for Windows menuAction expression. This method is redundant and only provided for convenience.

PROTOTYPE
xf_MDITile()

EXAMPLE

```
; TileWindows::pushButton
method pushButton(var eventInfo Event)

Crossfire.xf_MDITile()

endmethod
```

xf_MDIUnMinimizeAll

METHOD Restores all minimized open forms on the desktop that are registered with Crossfire library, and brings the currently active form to the top of the stack.

PROTOTYPE
xf_MDIUnMinimizeAll()

EXAMPLE

```
; ReClutterDesktop::pushButton
method pushButton(var eventInfo Event)

; Minimizes all open forms registered with Crossfire library
; Except for the current form window
Crossfire.xf_MDIMinimizeAllButSelf()

;  ...Do something provocative here...

; Restores the minimized forms to their previous positions
Crossfire.xf_MDIUnMinimizeAll()

endmethod
```

xf_SEQIncrement

METHOD	Generates an auto-incremented number. xf_SEQIncrement will look for a table in the working directory called **XFSEQ.DB**. If the table is not found, xf_SEQIncrement will automatically create it in the working directory. Auto-incremented numbers are tracked by *tableName* and *fieldName* allowing you to maintain several separate parallel tracks for a single table, by field name. You must pass a *tableName* and a *fieldName* to xf_SEQIncrement to retrieve an auto-increment number.

PARAMETERS

tableName	The name of the the table to receive the generated number.
fieldName	The name of the the field to receive the generated number.

RETURNS	An auto-incremented number if successful, -1 otherwise.

PROTOTYPE

xf_SEQIncrement(**tableName** String, var **fieldName** String) Number

EXAMPLE

```
; GetAutoIncrementValue::pushButton
method pushButton(var eventInfo Event)
var
        autoGenNumber Number
endVar

autoGenNumber = Crossfire.xf_SEQIncrement("CUSTOMER","CUSTOMERCODE")

if autoGenNumber <> -1 then
        CustomerCode = SalesCode.value+String(INT(autoGenNumber))
else
        msgStop("PROBLEM", "Could not generate a number for customer")
endif
endmethod
```

xf_UTErrorMessage

METHOD Returns an error message translation for a Crossfire error constant

PARAMETERS

xf_ErrorCode	A SmallInt Crossfire error code

RETURNS A string translation for a Crossfire error constant

PROTOTYPE

xf_UTErrorMessage(**xf_ErrorCode** SmallInt) String

EXAMPLE

```
; FormHandleErrorTest::pushButton
method pushButton(var eventInfo Event)
var
        formName String
        formHandle Form
        seedNumber SmallInt
        ECode SmallInt
endVar

seedNumber = 100
formName = "MyProductForm"

ECode = Crossfire.xf_MDIGetFormHandle(formName,formHandle)

if ECode = 0 then
        ; Directly assign the seed value to a field on that form
        formHandle.MyTFrame.MyField = seedNumber
else
        ; An error occurred; report it
        msgStop("Crossfire Error",xf_UTErrorMessage(ECode))
endif
endmethod
```

Crossfire Custom Types

The following is a listing of custom types that should be typed into the **Type** window of the forms that use Crossfire library. Most Crossfire methods require these custom types to operate and the Crossfire prototypes will expect them to be in your form's type window. You can copy them directly from the page below and paste them into your forms without modification.

```
Type

; Crossfire Types
AnyBag          = dynArray[] AnyType
FormBag         = dynArray[] Form
StringBag       = dynArray[] String

endType
```

"Uses ObjectPAL" Window Prototypes

The following is a listing of method prototypes that should be typed into the **uses** window of the forms that use Crossfire library. Prototypes for all methods used by a form should be entered into that form's uses window. In fact, you can copy them directly from the page below and paste them into your forms without modification.

```
Uses ObjectPAL

;Crossfire Data Passing Services
xf_GDBEmpty()
xf_GDBGet() AnyBag
xf_GDBGetValue(GDBTag String) AnyType
xf_GDBSet(GDB AnyBag)
xf_GDBSetValue(GDBTag String, NewGDBValue AnyType)
xf_GDBSwapValue(GDBTag String, NewGDBValue AnyType) AnyType

;Crossfire MDI Services
xf_MDICascade()
xf_MDICloseAll(editTerm String) SmallInt
xf_MDIConstructor() SmallInt
xf_MDIDestructor()
xf_MDIGetFormHandle(formName String, var formHandle Form) SmallInt
xf_MDIGetAllFormHandles(var formHandleBag FormBag)
xf_MDIGetAllFormNames(var formNameBag StringBag)
xf_MDIGetAllFormTitles(var formTitleBag StringBag)
xf_MDIIsFormOpen(formName String) Logical
xf_MDIMinimizeAll()
xf_MDIMinimizeAllButSelf()
xf_MDITile()
xf_MDIUnMinimizeAll()

;Crossfire Utility Services
xf_SEQIncrement(tableName String, var fieldName String) Number
xf_UTErrorMessage(xf_ErrorCode SmallInt) String

endUses
```

SoftWatch Example Application

Crossfire is provided with an example program, **SoftWatch**, that demonstrates some of the services provided by Crossfire. SoftWatch is an application that tracks the ordering and registration of software products sold by a small software vendor. The application is provided as an example and although it is not proposed that SoftWatch contains all of the business logic required for such an endeavor in "real life", the application contains many useful OPAL programming techniques as well as insights to the power and usefulness of Crossfire library.

Source code for SoftWatch is provided with Crossfire library with the exception of the source code for the Crossfire library itself which shall be distributed only in its compiled form. Source code for the Crossfire library will be provided upon registration of the product. All of the forms in SoftWatch and the application-specific library, SWATCH.LSL, which controls many of SoftWatch's features are open for examination. There are many interesting programming techniques in SoftWatch such as centralized menu handlers and examples of using multiple libraries in a program among many others.

No tutorial is provided for using SoftWatch, so to uncover many of its techniques and suprises, you will have to roll up your sleeves and wade in. As a quick start, however, we offer the following information.

Briefly, you start SoftWatch by changing your working directory to the directory where SoftWatch resides. Next, run SWATCH.FSL or SWATCH.SSL and proceed through the menu structures as they are presented. Both pulldown menus and popup menus are presented during program execution. The popup menus are always available by depressing the right mouse button (provided that the mouse is configured for right-handed operation, of course). Most Paradox for Windows users will already know that the right mouse click will have no such effect if the mouse pointer is hovering over a field object; move the pointer over a non-field object for the popup menus to appear.

The records on the forms may be manipulated (added, edited, deleted, etc) when the form is in edit mode. They are not in edit mode by default. Either press F9, or select "Edit" from the menus to go into edit mode, and F9 again or select "End Edit" to end edit mode (the menus are context sensitive to a degree). If you have several forms on the desktop at once, the popup menus will be context sensitive relative to the form that you "right-click" over.

The menu "Windows" is available from any form and provide some of the simpler functions provided by Crossfire. They are, or rather should be, standard "issue" in most MDI applications. Tile, tiles. Cascade, cascades. Tidy Desktop, minimizes all forms but the current one and Restore Desktop un-minimizes all forms that Tidy Desktop affected. A unique selection is Window Manager which provides a dialog similar to Windows Task Manager. Open several of the SoftWatch windows at once to see how changes in context are handled by the system.

Some fields on the Order form (customer name, sales code) have examples of custom validation/lookup dialogs which are a cut above the standard table lookup dialogs otherwise provided. These dialogs also provide examples of the xf_GDB... data-passing methods and other techniques. Pressing F1 in the Model field of the Order form's table frame displays another lookup. SoftWatch will, however, allow you to enter an un-validated free form item in the Model field to account for unexpected sales transactions.

Please note that if you are running SoftWatch on a Novell network, you should increase the number of allowable open file handles with the line FILE HANDLES = 100 in your NET.CFG or SHELL.CFG files. This will allow you to open all of SoftWatch's windows on the desktop at once without incident.

Although simpler applications would have sufficed to demonstrate some of the Crossfire methods, SoftWatch was designed to contribute a few new ideas and illuminate some ill-understood concepts. In SoftWatch you will find techniques for working with TCursors, MROs, action event handling, enhanced lookups, coordinating multiple libraries, centralized menu handling, quickly printing the current record, auto-incremented numbers, intra-application passing of data and messages, and many others. Some of these use the Crossfire library but many are provided with full source code. It is hoped that the extra effort will contribute in some way to the user's knowledge of Paradox for Windows in addition to attempting a modest display of the library methods.

Limitations in Crossfire version 1.0

All forms opened simultaneously in an application using the Crossfire library must have unique names. Only one copy of each uniquely named form may be opened at runtime. This limitation refers to the form name, not the form title. The form name is accessed and changed by right-clicking on the form's title bar in design mode. The default form name for new forms is **#Form1** and should be changed to something more meaningful during development.

The sample applications **require** an alias named **CROSSFIRE** available in Paradox for Windows to run. An alias easily is created in Paradox for Windows by selecting **Aliases...** from the **File** menu. In the resulting dialog box, **Alias Manager**, select the New button, enter "CROSSFIRE" as the Database Alias, choose the "STANDARD" Driver Type and enter the full DOS path name where Crossfire library is stored in the Path edit region.

Please note that if you are running SoftWatch on a Novell network, you should increase the number of allowable open file handles with the line FILE HANDLES = 100 in your NET.CFG or SHELL.CFG files. This will allow you to open all of SoftWatch's windows on the desktop at once without incident.

If you discover any anomalies with this version of the Crossfire Library or would like to suggest enhancements for inclusion in a future version of Crossfire Library, please forward the information via Compuserve to Daniel Martini, CIS ID 71033,1722.

Registration

Please feel free to use the Crossfire Library and the source code for SoftWatch in your custom Paradox for Windows application as you see fit. You may distribute the unregistered version Crossfire Library whole or in part with your applications as long as you do not remove or obstruct the "unregistered version" notice dialog box from any application using the library. Crossfire Library may be distributed as part of a custom Paradox for Windows application to be sold to a single client under this license. Crossfire library may not be sold whole or in part as part of any product sold as a Paradox for Windows programming utility or programming aid. Crossfire Library may not be distributed whole or in part as part of a shareware or retail product without the express written permission of Black Forest Software Corporation. By using Crossfire Library you are acknowledging that you have read and agree with the terms of this notice.

If you find Crossfire Library or the SoftWatch application useful, a registration fee for the library of $25.00 (US currency) + shipping and handling ($4.00 USA and Canada, $15.00 outside USA and Canada) would be greatly appreciated.

When you purchase the licensed version of Crossfire Library you will receive a disk with the Crossfire library (XFIRE.LSL) in source form or a file sent via Compuserve Mail containing the Crossfire library in source form, whichever you choose.

Make checks or money orders payable to: Black Forest Software Corporation

Mail to:
Black Forest Software Corporation
3618 Wharf Lane
Triangle, VA   22172
USA

Please allow 2-3 weeks for delivery via mail.