FBlockManager

(Development Guide)

2010-10-01

Axel Kesseler

Purpose ____ *2*
Components ____ *2*
IFBlockAccessor ____ **3**
IFBlockAccessorEvents ____ **3**
IReportSink ____ **4**
FBlockAccessorBase ____ **4**
DefaultFBlockAccessorEvents ____ **4**
Sequences ____ *4*
Task Creation ____ **4**
Task Destruction ____ **5**
FBlock Connection ____ **5**
FBlock Disconnection ____ **6**
Writing an FBlock Application ____ *6*
General Steps ____ **7**
Creating an FBlock Application using a DLL ____ **9**
Creating an FBlock Application using an EXE ____ **11**
Connecting an FBlock ____ **12**
Disconnecting from an FBlock ____ **13**
Handling the Instance ID ____ **13**
Using the Report Sink ____ **13**
Abbreviations ____ *14*

Purpose

This guide shows how to develop a so-called function block application. Such an application is used to simulate the functionality of an FBlock and will be managed through the FBlock-Manager application, which is written in C#. Furthermore, every of the function block simulations is based on the usage of K2L's ATS testing environment.

Components

The overall FBlockManager consists of an executable which represents the user interface (UI) application to manage multiple function block simulations, a dynamic link-library (DLL) which provides a C# class collection needed to handle the communication between the UI and its assigned FBlock simulations and of course the FBlock simulations itself. See Figure 1 to become acquainted with all involved components. Because of the nature of this document an explanation of the user interface application will not be in the focus right here.

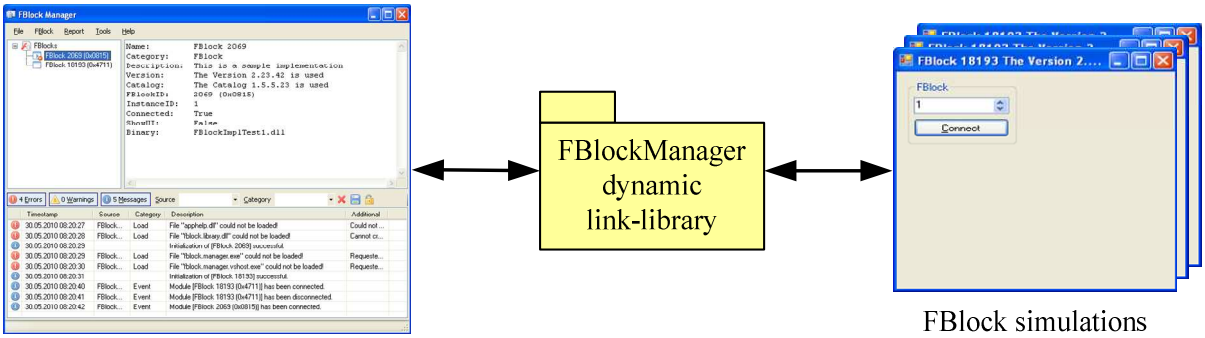

Figure 1: Component communication overview

The dynamic link-library contains the declarations of the interfaces, the events and some base class implementations. To get an impression how the single classes collaborate see Figure 2. The interfaces exposed by the DLL defining the "contract" how a communication between the FBlockManager application and its assigned FBlocks has to be done. In general this means, it is possible to implement the interfaces by yourself but you shouldn't do this because some of the required management stuff is already covered by the provided base classes. Therefore, the easiest way to create a function block application is to derive an own class from the abstract class *FBlockAccessorBase* and to implement the needed methods like shown in Figure 2.

IFBlockAccessor

The interface *IFBlockAccessor* represents the core interface and defines properties and methods used to collect needed information and to control the behavior of the function block application from within the FBlockManager application like Connect and Disconnect. This interface is already implemented by the abstract class *FBlockAccessorBase*.

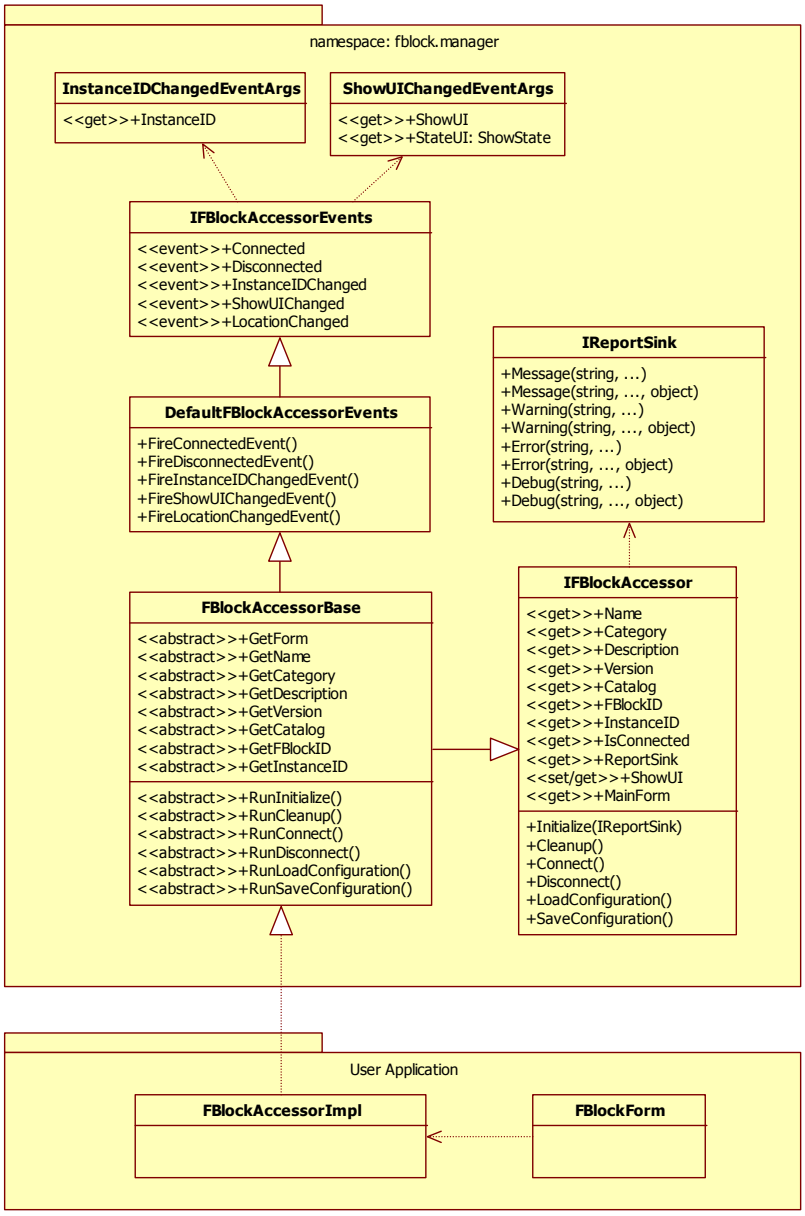

Figure 2: FBlockManager library class overview

IFBlockAccessorEvents

The interface *IFBlockAccessorEvents* contains the event definitions which are used to inform the FBlockManager application about various state changes like Connected and Disconnected. Note that this interface is already implemented by class *DefaultFBlockAccessorEvents*.

IReportSink

The interface *IReportSink* represents the definition of a logging sink which should be used by a function block implementation to report its current state, warnings or errors. An instance to the report sink is handover to the function block application during its initialization process. This means the report sink is not available until the method Initialize has been called!

FBlockAccessorBase

The class *FBlockAccessorBase* implements the interface *IFBlockAccessor* and should always be used to derive an own class from. Otherwise the additional management functionality must to be implemented by oneself!

DefaultFBlockAccessorEvents

The class *DefaultFBlockAccessorEvents* implements the interface *IFBlockAccessorEvents* and is used to report some state changes to the FBlockManager application. Note there is no need to inherit from this class directly because the base class *DefaultFBlockAccessorEvents* already supports all methods to fire state change events!

Sequences

Task Creation

If the FBlockManager application loads an FBlock's binary file the creation procedure will be executed as shown in Figure 3. Note that the class *FBlockAccessorBase* does additional management work like form creation, additional initialization and configuration loading.

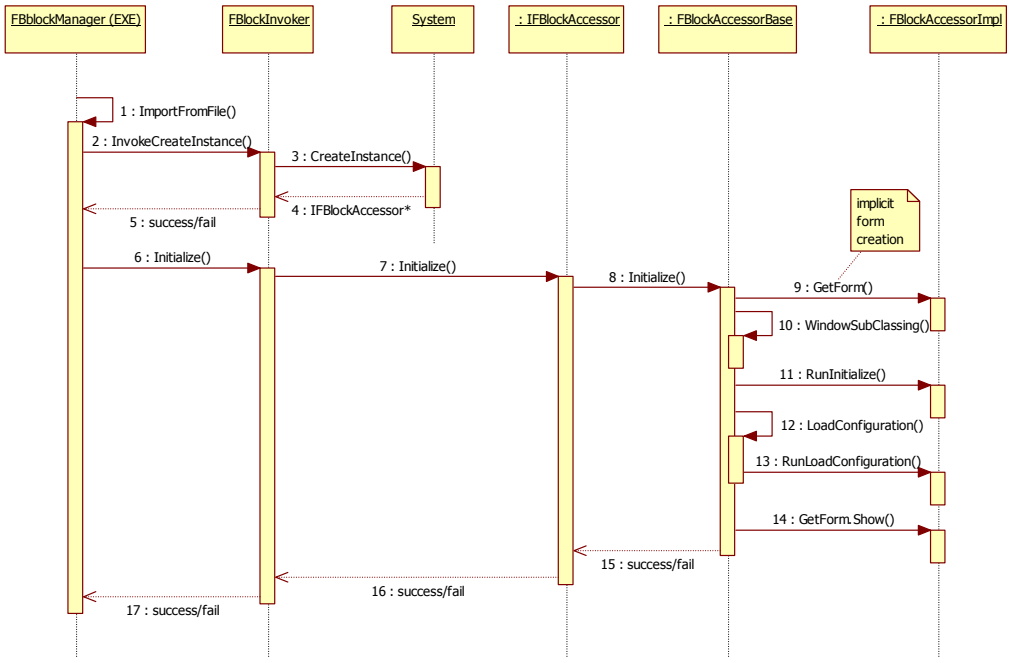

Figure 3: Task creation sequence

Maybe you recognized the self-call to method WindowsSubClassing shown in Figure 3. This is a well-know programming trick to catch and handle “important” Windows events. In this special case Windows Sub Classing is used to suppress the Windows message WM_CLOSE and therefore to avoid the FBlock binary’s main window from being closed by clicking the Close button in the form’s upper right corner. As result the FBlock’s main window will be put into hidden state and the application remains available until the FBlockManager application will be closed!

Task Destruction

If the FBlockManager application closes or unloads an FBlock’s binary file the destruction sequence will be executed as shown in Figure 4. Note that the class *FBlockAccessorBase* does additional management work like saving the configuration, additional de-initialization and form closing. Indeed, the destruction procedure looks very similar to the creation procedure except the additional Disconnect action. An explicit undo of the Windows Sub Classing is not necessary because the form will be really closed and destroyed afterwards.

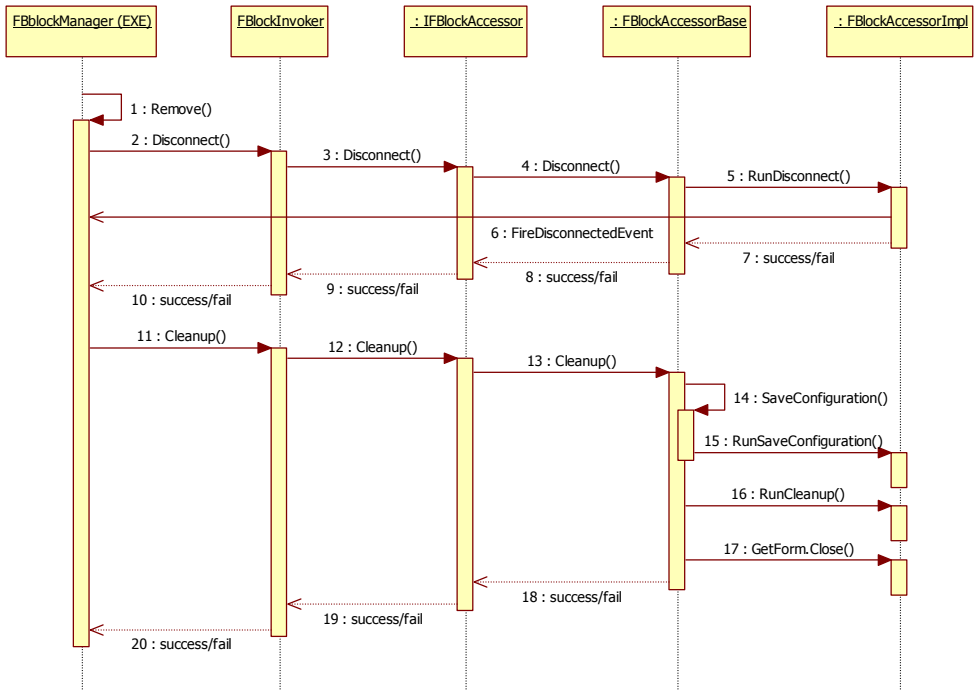

Figure 4: Task destruction sequence

FBlock Connection

If the FBlockManager application tries to connect an FBlock application to its counterpart the connection procedure will be executed as shown in Figure 5. This is very easy to understand because an FBlock simulator needs its communication partner. The only important thing right here is that the implementation of an *FBlockAccessorBase* derived class has to fire the Connected event if the connection could be made! Please do not forget to fire this event because otherwise the FBlockManager application is unable to reflect this state change to the user. See section *Connecting an FBlock* for an example.

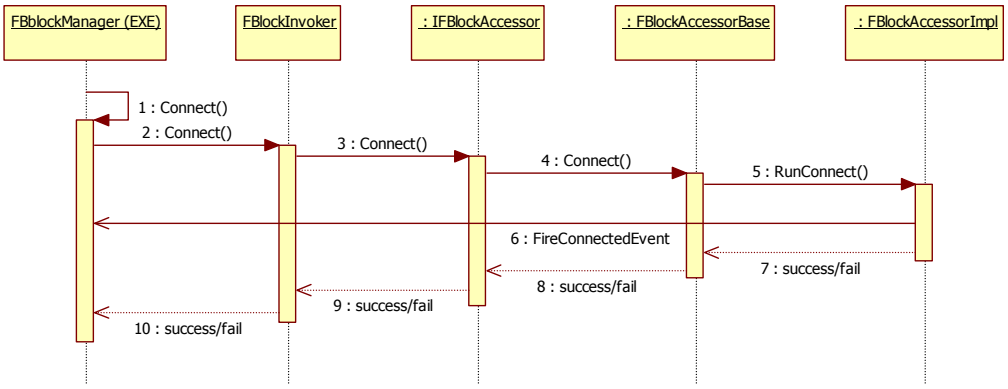

Figure 5: FBlock connection sequence

FBlock Disconnection

If the FBlockManager application tries to disconnect an FBlock application from its communication partner the disconnection sequence will be executed as shown in Figure 6. This is very alike to the above explained connection behavior. The only important thing right here is that the implementation of an *FBlockAccessorBase* derived class has to fire the Disconnected event if the connection could be closed! Please do not forget to fire this event because otherwise the FBlockManager application is unable to reflect this state change to the user. See section *Disconnecting from an FBlock* for an example.

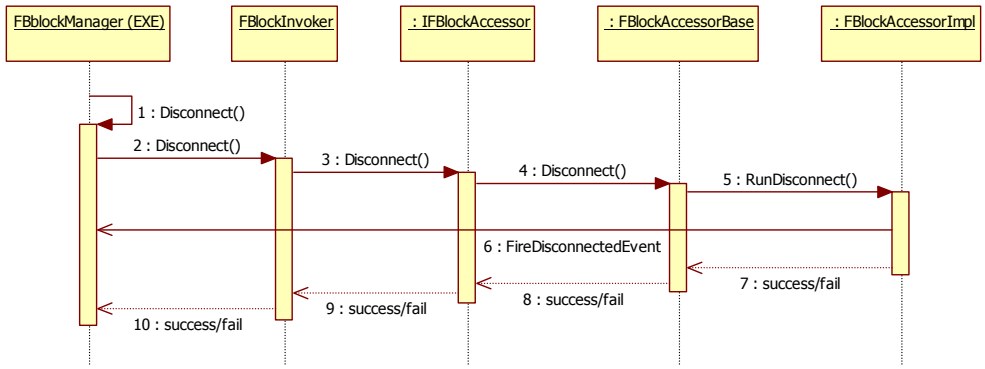

Figure 6: FBlock disconnection sequence

Writing an FBlock Application

This section shows how to implement an FBlock application which then can be used with the FBlockManager. Basically it is possible to create such an application as an executable as well as a dynamic link-library. But to prevent users from starting the program without the FBlockManager it's a good idea to build an FBlock application's release as a dynamic link-library! Therefore an executable should be used only for development and testing purpose. The distinction between both types (EXE and DLL) will be discussed later in this document.

General Steps

Independent of an FBlock application's type (EXE or DLL) some general steps are always necessary. See following list for each of those steps.

1. Create a new C# project (usually a WinForm application).
2. Add the FBlockManager's DLL (fblock.library.dll) to the list of references.
3. Create a new public class and derive it from class *FBlockAccessorBase*.
4. Implement all inherited abstract methods of class *FBlockAccessorBase*.
5. Provide appropriated information in each derived method.
6. Build the project and load it into the FBlockManager (fblock.manager.exe).

Hint: Implementation of inherited abstract methods best works using the Developer Studio's context menu. To do it, right click the name of the abstract class (in this case *FBlock-AccessorBase*) and choose menu item Implement Abstract Class from the context menu (see Figure 7). Afterwards a lot of new code can be found in the source file.

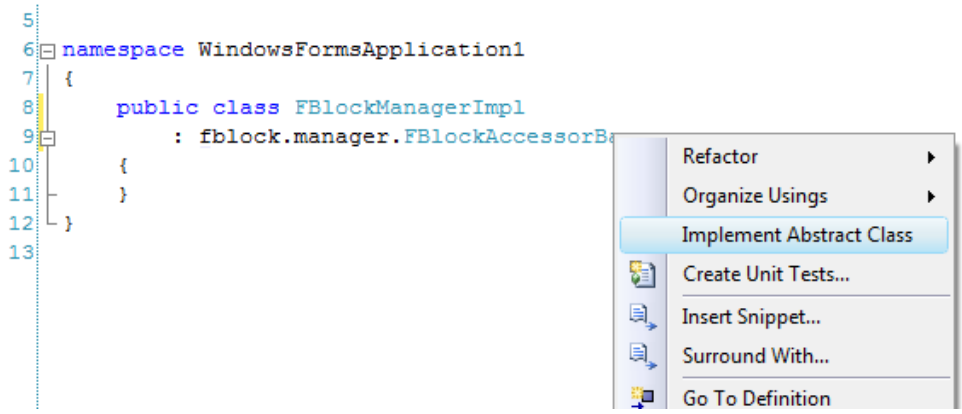

Figure 7: How to implement an abstract class

Now it seems to be a good idea to have a brief look at the generated methods to understand what is needed to be done by each method.

GetFBlockID()

This is one of the most important methods because the function block identifier is essential to track the underlying FBlock through the whole system. Always return the real function block identifier and if possible avoid hardcoded return values.

GetInstanceID()

This is another very important method because the FBlock's instance identifier is necessary to track an FBlock's instance through the whole system. Always return the up-to-date instance identifier and always fire the Instance ID Change to inform the FBlockManager! See section *Handling the Instance ID* later in this document.

GetForm()

This method is relevant for internal window handling and management and therefore a very important method (especially for the FBlockManager). This method should always return a valid instance of the FBlock application's main window. Otherwise the FBlockManager is unable to run the FBlock application!

GetName()

This method returns the name of the implemented function block. If possible this method should return the real FBlock name. Please avoid hardcoded return values. Keep in mind the method's return value will only be displayed and is not checked by the FBlockManager.

GetDescription()

This method should return a short description so that the user gets a brief instruction about the FBlock's purpose. Keep in mind the method's return value will only be displayed and is not checked by the FBlockManager. Therefore, only provide appropriated information right here.

GetVersion()

This method returns a string containing the version number of the FBlock application's binary file. This information could be important for bug tracking and problem reporting. But keep in mind the method's return value will only be displayed and is not checked by the FBlock-Manager. Therefore, only provide appropriated information right here.

GetCatalog()

This method returns a string containing the version of the underlying function block catalog (FCat). This information could be important for bug tracking and problem reporting. But keep in mind the method's return value will only be displayed and is not checked by the FBlock-Manager. Therefore, only provide appropriated information right here.

GetCategory()

This method returns a string containing a type descriptor. Usually this descriptor will be set to FBlock but sometimes it might be set to Shadow. Keep in mind the method's return value will only be displayed and is not checked by the FBlockManager. Therefore, only provide appropriated information right here.

RunInitialize()

This method is used to handle additional initializations. If no additional initialization stuff is required this method should not fail! Be aware if this method returns unsuccessful then the complete initialization procedure will be aborted and the FBlock application's binary file will not be added to the FBlockManager application! See Figure 3 for more details.

RunLoadConfiguration()

This method enables an FBlock application to load its own configuration (usually from a file). If configuration loading is not required this method should not fail! Be aware if this method returns unsuccessful then the complete initialization procedure will be aborted and the FBlock application's binary file will not be added to the FBlockManager application! See Figure 3 for more details.

RunConnect()

This method enables an FBlock application to execute its connection procedure. Such a connection is usually made by creating, initializing and connecting a K2L Automotive generated FBlock. This method should return successful when a connection could be established or if the connection was already made. After connecting an FBlock it is very important to fire the Connected event because otherwise the FBlockManager application will not recognize this occurrence! See Figure 5 to get an overview about the connection behavior and see also section *Connecting an FBlock* for a more detailed description.

RunDisconnect()

This method enables an FBlock application to execute its disconnecting procedure. Such a disconnection is usually made by disconnecting from a K2L Automotive generated FBlock. This method should return successful if the disconnection could be made or if the FBlock has already been disconnected. After disconnecting from an FBlock it is very important to fire the Disconnected event because otherwise the FBlockManager application will not recognize this occurrence! See Figure 6 to get an overview about the disconnection behavior and see also section *Disconnecting from an FBlock* for a more detailed description.

RunSaveConfiguration()

This method enables an FBlock application to save its configuration (usually into a file). If it isn't required to save the configuration this method should return with success. Otherwise the FBlockManager application will report this event as an error! But the cleanup procedure is not aborted in this case. See Figure 4 for more details.

RunCleanup()

This method is used to execute additional cleanups. If no additional cleanup stuff is required this method should return with success. Otherwise the FBlockManager application will report this event as an error! But the cleanup procedure is not aborted in this case. See Figure 4 for more details.

Creating an FBlock Application using a DLL

Basically it will be a good idea to implement an FBlock application as a dynamic link-library because users are not encouraged to start such programs directly. But sometimes it might be a better idea to have an executable instead. See section *Creating an FBlock Application using an EXE* under what circumstances it is better to have an executable file.

Before continuing with the “real” implementation it is necessary to make some assumptions to ensure, that only the important aspects will be discussed. See following enumeration for the list of tasks needed to prepare.

1. A new C# application of type “Class Library” has been created.
2. A new Form named *MainForm* has been added to the project.
3. A class with name *FBlockManagerImpl* which is derived from base class *FBlockAccessorBase* has been added to the project. The class already overrides its inherited abstract methods. See also section *General Steps*.
4. An additional constructor taking an instance of class *FBlockManagerImpl* as parameter has been added to class *MainForm*.
5. The property InstanceID, the method Connect and the method Disconnect have already been added to class *MainForm*.
6. Additionally, a class *VersionData* has been added to the project which provides global information about the underlying FBlock The class may also be used with the applications About box.
7. Furthermore, a DLL containing the *MediaPlayer* sample FBlock has been generated and added to the project using K2L’s MAG.

After above preparations, replacing all existing “throw new” statements with an appropriated content should be the next step. See following code snipped for the implementation results.

```
public class FBlockManagerImpl : fblock.manager.FBlockAccessorBase
{
  private MainForm mainForm = null;

  public FBlockManagerImpl()
  {
      // Instantiate the main form.
      this.mainForm = new MainForm(this);
  }

  public override int GetFBlockID()
  {
      // Obtain the real FBlock's ID.
      return K2L.Automotive.Test.MediaPlayer.FBlockId;
  }

  public override int GetInstanceID()
  {
      // Return with currently used Instance ID.
      return this.mainForm.InstanceID;
  }

  public override System.Windows.Forms.Form GetForm()
  {
      return this.mainForm;
  }

  public override string GetName()
  {
      // Obtain the real FBlock's name.
      return K2L.Automotive.Test.MediaPlayer.FBlockName;
  }
```

```
    public override string GetDescription()
    {
        return VersionData.Description;
    }

    public override string GetVersion()
    {
        return VersionData.Version;
    }

    public override string GetCatalog()
    {
        return VersionData.Catalog;
    }

    public override string GetCategory()
    {
        return VersionData.Category;
    }

    public override bool RunInitialize()
    {
        return true; // Do not fail!
    }

    public override bool RunLoadConfiguration()
    {
        return true; // Do not fail!
    }

    public override bool RunConnect()
    {
        return this.mainForm.Connect();
    }

    public override bool RunDisconnect()
    {
        return this.mainForm.Disconnect();
    }

    public override bool RunSaveConfiguration()
    {
        return true; // Do not fail!
    }

    public override bool RunCleanup()
    {
        return true; // Do not fail!
    }
}
```

The above sample implementation offers a maximum of flexibility especially because of the usage of the real function block name and its identifier which are directly taken from the generated FBlock code.

Creating an FBlock Application using an EXE

Due to the fact that a dynamic link-library is good idea to prevent users from executing such a file directly, is a DLL as binary output very disturbing during development time. Under that condition it could be better to have an executable instead. To achieve this is in C# quite easy because the differences between EXE files and DLLs are not that much. The only things

which are really needed are to choose “Windows Application” as the binary’s output type and to provide an appropriated main function (usually located within file Program.cs).

To change the program’s output type from DLL into EXE open the project’s properties pane and move to page Application. Here you choose “Windows Application” from the combo box below Output Type. That’s it.

The easiest way to obtain an appropriated Main function is to create an independent project of type Windows Forms Application and thereafter to copy the file Program.cs into the own project. Do not forget to adapt the namespace after adding this file to your project! See following code snipped for additional modifications.

```
static class Program
{
  /// <summary>
  /// The main entry point for the application.
  /// </summary>
  [STAThread]
  static void Main()
  {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);

    // Adaptions according to FBlockManager's behavior.
    FBlockManagerImpl impl = new FBlockManagerImpl();
    impl.RunInitialize();
    impl.RunLoadConfiguration();
    Application.Run(impl.GetForm());
    impl.RunSaveConfiguration();
    impl.RunCleanup();
  }
}
```

With this it becomes possible to switch between an executable and a dynamic link-library as output format whenever it is needed simply by changing the Output Type on the project’s properties pane.

Connecting an FBlock

As already mentioned an FBlock application’s main form should provide a public method to connect to a MOST function block which is called by class *FBlockManagerImpl*. Furthermore, such method has the duty to inform the FBlockManager application about a successful connection by firing the Connection event. See following code snipped how to fire this event.

```
public bool Connect()
{
  // Do anything to connect to MOST function block.

  // Inform FBlockManager about granted connection.
  if (this.connected && this.manager != null)
    this.manager.FireConnectedEvent();

  return this.connected;
}
```

Disconnecting from an FBlock

The same as with connecting an FBlock application applies to disconnecting such an application too. This means an FBlock application should also provide a public method to handle the disconnection stuff and the method is called by class *FBlockManagerImpl*. Furthermore, this method has the duty to inform the FBlockManager application about successful disconnecting from its MOST function block by firing the Disconnection event. See following code snipped how to fire this event.

```
public bool Disconnect()
{
  // Do anything to disconnect from MOST function block.

  // Inform FBlockManager about successful disconnection.
  if (!this.connected && this.manager != null)
    this.manager.FireDisconnectedEvent();

  return this.connected;
}
```

Handling the Instance ID

Because of the nature that an instance ID can vary between two connections the FBlock application needs to reflect such changes to the FBlockManager. The best way to achieve this is to fire the Instance ID Changed event every time the instance ID has been modified. See following code snipped how to fire this event.

```
private void numInstanceID_ValueChanged(object sender, EventArgs args)
{
  if (this.manager != null)
    this.manager.FireInstanceIDChangedEvent(
      Convert.ToInt32(this.numInstanceID.Value)
    );
}
```

The above code snipped assumes a number edit box within the FBlock application's UI which is linked to a member function that handles all value changes.

Using the Report Sink

To use the report sink from within an FBlock application it is only needed to share the reference to the *FBlockAccessorBase* derived class. But keep in mind, the report sink will not be available until the FBlockManager calls method Initialize of interface *IFBlockAccessor*! And in case of an executable file which is not started by the FBlockManager a report sink will never be available except it will be implemented by oneself. This exactly means always check the reference returned by property ReportSink of interface *IFBlockAccessor* before using it! See following code snipped to get an impression how to use FBlockManager's report sink.

```
if (this.manager != null && this.manager.ReportSink != null)
  this.manager.ReportSink.Message("Hello world", "FBlock", "Test");
```

Using the ReportSink to capture internal application states and events is always a good idea. But keep in mind, more than one application requesting the FBlockManager to print important information. Therefore, distinguish between messages, warnings and errors carefully. Furthermore, always provide a source identifier and an appropriated category in each message. This makes it easier to filter the FBlockManager's output panel for special messages.

It is also possible to handover an object reference in each report sink function. This is very helpful if you need to print for example an exception. So, just put a caught exception into the object parameter of an error message and the FBlockManager will print the object using the ToString method.

As shown in Figure 2 the interface *IReportSink* also provides methods to print out debugging messages. But be aware debugging massages will only be printed if the FBlockManager's executable file is compiled as Debug version. Otherwise such messages will be suppressed!

Abbreviations

ATS	Automotive Test System, a C# based framework to build MOST applications
DLL	Dynamic Link Library, a binary collection of executable functions
EXE	Extension of Windows executable files
FBlock	The contract how a MOST function block communicates with its Shadows
FCat	Function block Catalog, a description of a MOST function block
ID	IDentifier, usually a number used as handle to an object
K2L	Name of the company which publishes various tools regarding to MOST
MAG	MOST Application Generator, a tool to generate FBlock binary files
MOST	Multimedia Oriented Systems Transport, the communication protocol
UI	User Interface, the visible part of an application