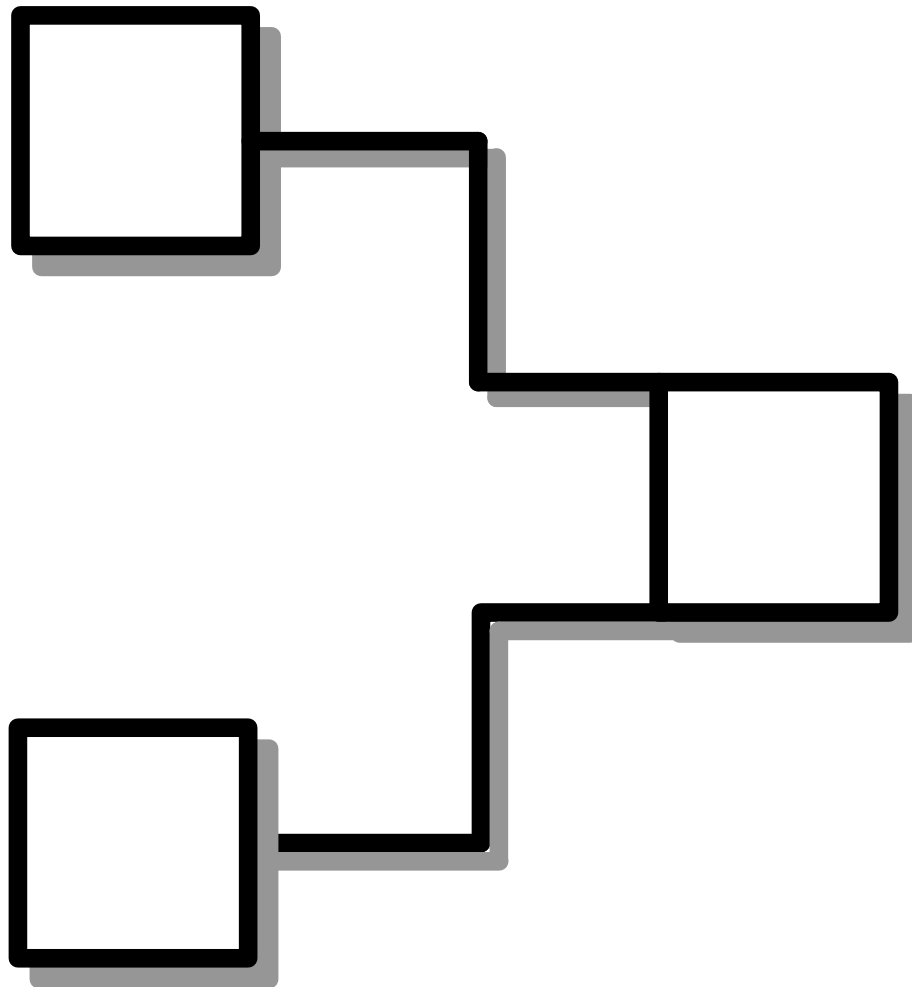


2104002–001 rev. AA

Totalflow[®]

EZ Block[™] **Builder Software**

Maintenance Guide



TOTALFLOW
MEASUREMENT & CONTROL SYSTEMS

ABB

Intellectual Property & Copyright Notice

©2009 by ABB Inc., Totalflow ("Owner"), Bartlesville, Oklahoma 74006, U.S.A. All rights reserved.

Any and all derivatives of, including translations thereof, shall remain the sole property of the Owner, regardless of any circumstances.

The original US English version of this manual shall be deemed the only valid version. Translated versions, in any other language, shall be maintained as accurately as possible. Should any discrepancies exist, the US English version will be considered final.

Notice: This publication is for information only. The contents are subject to change without notice and should not be construed as a commitment, representation, warranty, or guarantee of any method, product, or device by Owner.

Inquiries regarding this manual should be addressed to ABB Inc., Totalflow Products, Technical Communications, 7051 Industrial Blvd., Bartlesville, Oklahoma 74006, U.S.A.

EZ Block Builder Software Maintenance Guide

Software Version 1.0

1	Introduction.....	3
2	Build Environment.....	3
2.1	Release mode build issues with Visual Studio 2008 and .NET 3.5 SP1	3
2.2	Build procedure.....	5
2.3	Building the help file	5
3	Design Notes.....	7
3.1	Application Settings.....	7
4	Classes and methods	9
4.1	Function blocks (EZ blocks).....	9
4.2	Communications	10
4.3	Extensions	11
4.4	PropertiesControl	11
4.5	Attribute classes.....	12
5	Serialization	14
6	Adding new EZ Blocks to the project.....	15
7	Line Routing	29
8	Class Listing.....	31

1 Introduction

This document is intended to serve as an aid for developers tasked with maintaining and/or extending the EZ Block Builder application, which was developed by ProtoLink, Inc. (www.ProtoLink.com) for ABB. It provides a road map to the documented source code. The document assumes that the developer is familiar with the features and use of the application and has read and thoroughly understands the contents and terminology of the EZ Block Builder Requirements Specification.

2 Build Environment

The source code for “EZBlockBuilder.exe” was written in C# and developed using Microsoft Visual Studio 2008, Professional Edition, with Service Pack 1 (Version 9.0.30729.1 SP) and Microsoft .NET Framework, Version 3.5, Service Pack 1.

This software was designed to execute on a computer running Windows XP Professional SP2.

2.1 Release mode build issues with Visual Studio 2008 and .NET 3.5 SP1

At the time of the writing of this document, there is a problem with Microsoft's .NET 3.5 SP1 bootstrapper that may cause attempts to create a release mode build of EZ Block Builder to fail. The following workaround may be required in order to get a release-mode build of this project to complete. The explanation that follows is taken from the Microsoft Visual Studio 2008 Service Pack 1 Readme file.

If the .NET Framework 3.5 SP1 bootstrapper package is selected in the Prerequisite dialog box for a Setup project or in ClickOnce publishing, and also the "Download prerequisites from the same location as my application" option is selected, the following build error is shown:

The install location for prerequisites has not been set to 'component vendor's web site' and the file 'dotNetFx35setup.exe' in item 'Microsoft.Net.Framework.3.5.SP1' cannot be located on disk.

To resolve this issue:

Update the Package Data

1. Open the [Program Files]\Microsoft SDKs\Windows\v6.0A\Bootstrapper\Packages\DotNetFx35SP1 folder or %ProgramFiles(x86)%\Microsoft SDKs\Windows\v6.0A\Bootstrapper\Packages\DotNetFx35SP1 on x64 operating systems

2. Edit the Product.xml file in Notepad.
3. Paste the following into the <PackageFiles> element:
<PackageFile Name="TOOLS\clwireg.exe" />
<PackageFile Name="TOOLS\clwireg_x64.exe" />
<PackageFile Name="TOOLS\clwireg_ia64.exe" />
4. Find the element for <PackageFile Name="dotNetFX30\XPSEPPSC-x86-en-US.exe" and change the PublicKey value to:
3082010A0282010100A2DB0A8DCFC2C1499BCDAA3A34AD23596BDB6CB
E2122B794C8EAAEBFC6D526C232118BBCDA5D2CFB36561E152BAE8F0D
DD14A36E284C7F163F41AC8D40B146880DD98194AD9706D05744765CEA
F1FC0EE27F74A333CB74E5EFE361A17E03B745FFD53E12D5B0CA5E0DD0
7BF2B7130DFC606A2885758CB7ADBC85E817B490BEF516B6625DED11DF
3AEE215B8BAF8073C345E3958977609BE7AD77C1378D33142F13DB62C9A
E1AA94F9867ADD420393071E08D6746E2C61CF40D5074412FE805246A216
B49B092C4B239C742A56D5C184AAB8FD78E833E780A47D8A4B28423C3E
2F27B66B14A74BD26414B9C6114604E30C882F3D00B707CEE554D77D2085
576810203010001
5. Find the element for <PackageFile Name="dotNetFX30\XPSEPPSC-amd64-en-US.exe" and change the PublicKey value to the same as in step 4 above
6. Save the product.xml file

Download and Extract the Core Installation Files

1. Navigate to the following URL: <http://go.microsoft.com/fwlink?LinkID=118080>
2. Download the dotNetFx35.exe file to your local disk.
3. Open a Command Prompt window and change to the directory to which you downloaded dotNetFx35.exe.
4. At the command prompt, type:
dotNetFx35.exe /x:
This will extract the Framework files to a folder named "WCU" in the current directory.
5. Copy the contents of the WCU\dotNetFramework folder and paste them in the %Program Files%\Microsoft SDKs\Windows\v6.0A\Bootstrapper\Packages\DotNetFx35SP1 folder (%ProgramFiles(x86)%\Microsoft SDKs\Windows\v6.0A\Bootstrapper\Packages\DotNetFx35SP1 on x64 operating systems). Note: Do not copy the WCU\dotNetFramework folder itself. There should be 5 folders under the WCU folder, and each of these should now appear in the DotNetFx35SP1 folder. The folder structure should resemble the following:
 - o DotNetFx35SP1 (folder)
 - o dotNetFX20 (folder)
 - o dotNetFX30 (folder)
 - o dotNetFX35 (folder)
 - o dotNetMSP (folder)
 - o TOOLS folder)
 - o en (or some other localized folder)

- dotNetFx35setup.exe (file)

You may now delete the files and folders you downloaded and extracted in steps 2 and 4.

2.2 Build procedure

To build the EZ Block Builder application, load the FunctionBlockDesigner solution file (FunctionBlockDesigner.sln) into Visual Studio.

The solution is comprised of two projects. The FunctionBlockDesigner project contains the files needed to build EzBlockBuilder.exe. The Setup project takes assemblies EZBlockBuilder.exe with various support files to create an installer program.

After loading the solution, set the solution configuration to “Release” mode and hit “F6” to begin the build. If a problem was encountered during the build operation, refer to the section titled Release [mode build issues with Visual Studio 2008 and .NET 3.5 SP1](#) to ensure that any required patches have been applied.

The following files and directories will be created and placed in the solution’s “Setup\Release” directory:

- Setup.exe
- Setup.msi
- DotNetFX35SP1
- WindowsInstaller3_1

The entire the entire contents of the “Setup\Release” will be required for an install.

For a full installation, run Setup.exe (not Setup.msi).

2.3 Building the help file

The project’s user guide was converted into a “chm” file using EC Software’s Help & Manual, Professional Edition, Version 4.5.1 Build 1371.

If modifications to the User Guide or help file are required, first make the modifications to the Microsoft Word document titled UserGuide.doc.

Save a copy of the user guide as an RTF file (UserGuide.RTF) and import this file into Help & Manual.

After importing the RTF file, each section of the help file will need to be examined. Some adjustments to the layout and formatting of the help file will be necessary since the RTF import procedure does not perfectly maintain the look of the MS Word User Guide.

Any hyperlinks created in the user guide will be lost during this import procedure so each hyperlink in the file will need to be manually adjusted. Note: For small changes to the

EZ Block Builder Software Maintenance Guide

document it may be faster to simply modify help file manually and bypass the RTF import step and avoid having to re-format the entire document.

Additional modifications will need to be made to the positioning of graphics in the help file to make them appear to be in the same (or similar) positions to the graphics in the user guide.

After modifications to the help file are complete, build the help file by pressing CTRL-F9 inside the Help & Manual Editor.

The output of this procedure should be a file named "UserGuide.chm". Copy this file over the old UserGuide.chm in the EZ Block Builder's main solution directory and the next time a release build of EZ Block Builder is performed, the new UserGuide.chm will be included in the build.

3 Design Notes

3.1 Application Settings

The application stores the defaults for project setting values in the file `EZBlockBuilder.exe.config`, located in the executable's directory. The user may modify many of these settings via the user interface. These modifications are stored in a `user.config` file located in the `C:\Documents and Settings\USER\Local Settings\Application Data\FunctionBlockDesigner\` subdirectories where "USER" is the name of the current user's login. The customized settings for the user are loaded each time the application runs.

The following settings are used by the application:

- `TooltipsEnabled`: When this Boolean value is true, tooltips are displayed when the cursor hovers over a function block in the work area. When false, they are not displayed.
- `AliasListFile`: This string value contains the full name and path to the currently selected alias list file.
- `AliasListEnabled`: This boolean value is true when an alias list has been selected for use; false otherwise.
- `RecentFiles`: This collection contains the names of recently opened EZ Block Builder program files.
- `LastFile`: This string value contains name of the most recently opened EZ Block Builder program file.
- `MaxRangesPerRequest`: This integer contains the maximum number of ranges allowed in a device poll request.
- `RemoteTimeout`: This integer value contains the timeout value (in seconds) for a remote device connection.
- `PCCUSecurityCode`: This string value contains the security code used for connecting to the device.
- `LocalCommPort`: This integer value contains the COM port selected for device communications.
- `UseLocalMode`: When true, the application uses local communication methods to talk to the device when in serial mode. When false it uses remote communication methods when in serial mode.
- `ProfileFile`: This string contains the name of the currently active profile.
- `ProfileEnabled`: When this boolean value is true, the application uses profiles; when false, it does not.
- `SnapToGrid`: This boolean value governs if function blocks placed in the work area are arranged along a defined grid or if they may be placed at any location in the work area.
- `ShowPins`: This Boolean value determines if the input and output pins are displayed on function blocks visible in the work area.

EZ Block Builder Software Maintenance Guide

- `UseSerial`: When true, the application uses serial communication methods to talk to the device (RS-232 or USB). When false it uses IP communication methods (network).
- `TargetIPAddress`: This string contains the IP address the application will connect with during IP mode communications
- `TargetIPPort`: This unsigned integer contains the port value the application will use to connect with the device during IP mode communications.

4 Classes and methods

A design goal for the EZ Block Builder was to design the software so new blocks could be added at a later time relatively quickly and without excessive modification to the rest of the code base.

To that end, much of the EZ Block Builder code was designed with the idea that the blocks themselves would be fairly sparse, ideally containing just enough information to describe its data model and any other code required for specialized operations.

The following sections describe classes and methods that are likely to be important to examine when performing code maintenance or upgrades.

4.1 *Function blocks (EZ blocks)*

The `FunctionBlock` base class implements much of the functionality required of EZ blocks. (During most of the development process of Version 1.0, EZ blocks were referred to as Function blocks and the code retains references to Function blocks. The terms, for purposes of this document, are interchangeable.) All EZ blocks have an associated class that extends the `FunctionBlock` base class. The methods described in the remainder of this section are relevant to extending the functionality of EZ blocks while detailed instructions on the actual procedure for creating new EZ blocks is provided in the section titled [Adding new EZ blocks to the project](#).

protected void Initialize()

`Initialize` should be called at the end of every `FunctionBlock` derived class's constructor. This method initializes the input and output pin lists for the `FunctionBlock` and synchronizes them with their associated array assignments by calling `SynchAppIDandArrayValues()`.

public virtual bool IsVisibleTo(FunctionBlock fb)

`IsVisibleTo` returns true if the supplied `FunctionBlock` can see the pins of the current function block. Certain function blocks may exhibit special behavior such that pins that would normally be visible to one another should not be visible. For example, `ComplexInputs` should be invisible to `ComplexOutputs` and vice versa. These "helper" function blocks are there to service the device function blocks. By default, every function block is visible to every other function block as long as they share the same parent.

public virtual bool IsFullyConfigured()

This method returns true if all the input and output pins of this function block are fully configured; false otherwise.

public virtual void OnDrop(PageCanvas canvas)

This method is called when a function block is dropped onto the PageCanvas. The Default action is to do nothing. The PageCanvas will automatically add the function block. This method should be overridden if custom actions need to be performed when particular type of function block is dropped.

public virtual string HelpString()

This method returns a brief description of the function block. It is displayed in the property window when the function block is selected.

public void SynchroniseAppIDandArrayValues()

This method gathers all properties of the function block pins that have the attribute [ArrayAssignment]. It then sets the AppID and Array values for these pins to the appropriate values.

public virtual bool IsUploadedToDevice()

This method is called to determine if this function block is a block that needs to be uploaded to the physical device or if it is a construct such as a Complex block or an External Location block which does not. By default, this method returns true and should only be overridden to return false when the function block should not be uploaded to a device.

public virtual Pin GetPinByName(string strPinName)

Given the label for an InputPin, this method returns its instance.

4.2 Communications

The Communications class handles the interaction between EZ Block Builder and the target device.

static public bool ConnectToDevice()

Before any file upload or download operations may begin, EZ Block Builder must establish a connection with the target device. This method attempts to establish that connection.

**static public void SaveOperationsAppToDevice(
string strTempFileLocationOnDisk, int iAppID, int iInstance)**

This method is called to initiate a file upload operation after the user chooses to save a file to the device. The file is first saved to a temporary location on disk and then the contents of that file are uploaded to the device. The operations application to be saved is the app displayed in the current tab on the work area.

**static public string ReadOperationsAppFromDevice(int iAppID, int iInstance,
string strFilename)**

This method is called to initiate a file download operation after the user chooses to read a file from the device. The file is saved to filename specified or to a temporary file if no filename is given. The method returns the path and filename where the downloaded file was stored.

4.3 Extensions

The following extension methods are used in the project to provide a simple way to perform common operations on several primitive classes.

public static string Description(this Enum value)

This is an extension method for the Enum class. This extension returns the text defined in the [Description("some text")] attribute of an enumerated type.

Usage:

```
MyEnum obj = MyEnum.SomeEnumeratedValue;
string strDescription = obj.Description();
```

public static string ToStringNullSafe(this object value)

This extension method for the object class will return the normal ToString value of an object unless the object is null in which case it returns an empty string. Unlike ToString(), ToStringNullSafe() will not cause an exception when obtaining the string value of a null object.

public static string ToStringFormattedForDisplay(this TimeSpan ts)

This extension method for the TimeSpan class will return the given time formatted as HH:MM:SS

4.4 PropertiesControl

The PropertiesControl class contains the logic for automatically producing the UI needed for configuring EZ Blocks. This control is displayed in the Properties pane on the right side of the EZ Block Builder's main window.

public FunctionBlock Target

To specify the function block to be displayed in the property control, set the Target property.

public bool ValidateSettings()

If any UI element in the property control is unable to commit its property values then this is an invalid configuration and the method will return false. This could occur, for example, if a user entered a text string in a control that was expecting a numeric entry. If all the settings are valid, then the method will return true.

public void ConfigureFBProperties(FunctionBlock fbSelected)

Given a function block, this method will check to see if it has any properties or methods that are decorated with the [FBConfigurableProperty] tag. If it does, this method will populate the properties window grid with a row that describes the property or method and that allows the user to change the property value or call the method.

4.5 *Attribute classes*

Three custom attribute classes are used through the project. This section describes the function of these three classes.

FBConfigurablePropertyAttribute

The [FBConfigurableProperty] attribute and is used inside the application's FunctionBlock classes to decorate properties that are configurable via the property window. For example, a FunctionBlock-derived class may define the following property:

```
[FBConfigurableProperty]
public string StringOp
{
    get { return strValue; }
    set { strValue = value; }
}
```

In the example shown above, the properties window will contain an entry for "StringOp" and will allow the user to enter a new string value for the property. Note that both `get` and `set` methods must be provided in order for this feature to work properly. The user may also specify the `Description` parameter like this:

```
[FBConfigurableProperty(Description="Enter a string.")]
public string StringOp
{
    get { return strValue; }
    set { strValue = value; }
}
```

In this case the text "Enter a string." will appear as the description in the property grid instead of "StringOp". In order to work properly with the properties window, the return type of the property must be one of the following:

- Pin
- string
- int
- double
- any user defined enumeration

The `SortOrder` option determines the order the properties appear in the property window with lower numbered sort orders appearing above higher numbered Sort Orders.

Additional options include `MinValue` and `MaxValue` for setting boundaries on a property's numeric values and `NumLines` for specifying how many lines the property setting should take inside the property pane.

ArrayAssignmentAttribute

The [ArrayAssignment] attribute and is used inside the application's FunctionBlock classes to decorate properties that are present in the device. For example, to define a property with an array assignment location of 24, a class could define the following property:

```
[ArrayAssignment(Location=24)]
public string Description
{
    get { return strValue; }
    set { strValue = value; }
}
```

An Array assignment of 9 is given to the string “Description” shown in the above example.

It is important to set the ArrayAssignment attribute since this is the attribute that will be checked when programming the device in order to associate uploaded elements with their proper array assignment.

ExternalAssignmentAttribute

This [ExternalAssignment] attribute is used inside the application's ExternalLocationFunctionBlock class to decorate properties that represent pins providing access to registers external to the current application. For example, to define a property as an External Assignment, a class would define the following property:

```
[ExternalAssignment]
public OutputPin Output
{
    get { return outputPin; }
    set { outputPin = value; }
}
```

5 Serialization

Serialization is the process of persisting the state of an object to and from storage. The application can serialize a `Program` in XML form to or from disk and too or from an X-Series device.

The application makes use of the `IXmlSerialize` interface and the `XmlArchive` class. The `IXmlSerialize` interface has a method named `SerializeAll`.

This method takes a single argument, a reference to an `XmlArchive` object. An `XmlArchive` object is always associated with a .NET `Stream` or `Stream` derived object. The only public constructor for the `XmlArchive` class takes a reference to a `Stream` object, a boolean indicating whether the `Archive` object will be used for reading or writing (true for writing), and a string to define the default namespace of the object being stored. Once an `XmlArchive` object is created, it can be used for reading or writing – depending on the boolean used in construction. An `XmlArchive` object can never be used for both reading and writing.

Internally, the `XmlArchive` class uses the .NET `XmlDocument` class to read `Streams` and the `StreamWriter` class to write them. The `XmlArchive` class provides generic `Write` and `WriteCollection` methods for writing primitive types such as ints, strings, floats, etc or collections of these primitives respectively. In addition the `XmlArchive` class provides a `WriteClass` method that can write any object that implements the `IXmlSerialize` interface. For reading, the `XmlArchive` class provides methods analogous to the previously describe write methods with the notation “Read” instead of “Write”.

In addition to the `Program` class implementing the `IXmlSerialize` interface, all types of objects contained in the `Program` class that need to be serialized (such as EZ Blocks) also implement the `IXmlSerialize` interface. In general, each type of object is responsible for being able to serialize itself to and from an `XmlArchive` object. Special attention is required when dealing with derived classes that implement `IXmlSerialize` when the class they inherit from also supports `IXmlSerialize`. In this case the base class should declare the `SerializeClass` method as virtual. The overridden method in the derived class should always call the base classes `SerializeClass` method as the first thing it does. The key concept is that each class is responsible for only data items that it contains directly and not for any items in the base class or further derived classes. This makes all serialization code local to a single class. If a new data member is added to a class the only serialization code that needs to be modified is the `SerializeClass` method in that class.

One advantage of this approach is that it allows very fine grained control of the serialization of individual objects in a complex hierarchy of objects. The `Program` class contains several collections of other objects. Each of these objects is potentially complex and can contain many individual elements as well as collections of other objects.

6 Adding new EZ Blocks to the project

EZ Block Builder was designed with the idea that new EZ blocks would be added to the project without the need for excessive modifications or extensive user interface or graphics changes.

This section describes the steps required for adding an EZ block to the project by way of an example. In this example the programmer will add a new EZ block named “HiLoFunctionBlock” to the project.

1) Define a specification for the EZ Block

Before an EZ block can be added to the tool, the following attributes must be defined:

- The number of inputs it takes
- Zero or more configurable properties
- An output
- A property summary format
- The register number for the “Capacity” setting inside the device.

The inputs, properties, and output of an EZ block are stored within specific arrays in the Operations application on the X-Series device. The following tables describe the requirements for the example HiLo EZ block and provide the array assignments for its constituent components. A table describing the property summary format is also provided. The tool will display the property summary on the graphical depiction of the block. With this information, a programmer can now begin the process of adding the new EZ block to the application.

Note that the transfer function for the HiLo block is not required because it is not relevant to the operation of the EZ Block designer. The array assignments given in the example specification below are fictitious, as is the HiLo block itself. These values are used only for purposes of illustrating the process of creating a new EZ Block.

HiLo Block Inputs:

Name	Pin Label	Array Assignment
R1	R1	9
R2	R2	10

HiLo Block Properties:

Name	Type	Array Assignment	Enumeration List Items
Description	String	24	N/A

Operation	Enumeration	8	<table border="1"> <thead> <tr> <th>String</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>None</td> <td>0</td> </tr> <tr> <td>High Value</td> <td>1</td> </tr> <tr> <td>Low Value</td> <td>2</td> </tr> </tbody> </table>	String	Value	None	0	High Value	1	Low Value	2
String	Value										
None	0										
High Value	1										
Low Value	2										

HiLo Block Output:

Name	Array Assignment
N/A	7


HiLo Block Property Summary Format:

Line	Format
1	FB Type : Description
2	Operation

Register assignments:

Register	Value
Capacity	50

2) Create a 16x16 pixel GIF image for the function block.

 Freeware such as Paint.Net is adequate for creating simple icons with a transparent background. This is the graphic that will be displayed in the toolbox and in the property pane of the application.

3) Add GIF image to the project

Copy the GIF to the project's "images" folder and add the icon to the project by right-clicking on "images" in the Solution Explorer and selecting **Add | Existing Item...**

4) Create the HiLoFunctionBlock class

Add a new class to the project and name it "HiLoFunctionBlock".

5) Edit the class

The commented HiLo block code is given below. Refer to the **bolded** comments in the code below for detailed explanations of important features of the code.

```
// FunctionBlockDesigner.Extensions provides access to helpful
// extension methods such as ToStringNullSafe that prevents
```

```

// the developer from having to manually
// perform a null check on an object
// before performing a ToString() operation

using FunctionBlockDesigner.Extensions;
using System.ComponentModel;
using System.Windows.Media;
using System.Windows;

namespace FunctionBlockDesigner
{
    // All EZ blocks (also referred to as function blocks in this code)
    // must inherit from the base class FunctionBlock

    public class HiLoFunctionBlock : FunctionBlock
    {
        // According to the example specification, the HiLo block contains
        // an Operation property that has three enumeration list items
        // None:0 , High Value:1 , Low Value:2
        // The HiLoOperation enumeration below is used to produce this mapping
        // The text that appears inside the [Description] attributes for each
        // entry is the text that will appear in the "Hi Lo Operation" combo
        // box in the properties pane.

        public enum HiLoOperation
        {
            [Description("None")]      None = 0,
            [Description("High Value")] Hi = 1,
            [Description("Low Value")]  Lo = 2
        };

        // Define all the input and output pins as class variables
        // Each will be initialized in the constructor

        private Pin m_R1;
        private Pin m_R2;
        private Pin m_Output;

        // The HiLo Operation variable is defined here and is
        // given a default value of "None".
        private HiLoOperation m_eOperation = HiLoOperation.None;

        // The following three property getters and setters define the public
        // properties of the HiLo blocks input pins, output pin, and
        // Enumeration operation.
        // Every pin definition of an EZ block class will
        // follow the general pattern outlined below.

        // The [FBConfigurableProperty] attribute and is
        // used inside the application's FunctionBlock classes to decorate
        // properties that are configurable via the property window.
        // For example, if a FunctionBlock-derived class defines
        // the following property:
        //

```

```

// [FBConfigurableProperty]
// public string StringOp
// {
//   get { return strValue; }
//   set { strValue = value; }
// }
//
// then the properties window will contain an entry for "StringOp"
// and will allow the user to enter a new string value
// for the property. Note that both get and set methods
// must be provided in order for this feature to work properly.
// The user may also specify the Description parameter like this:
//
// [FBConfigurableProperty(Description="Enter a string.")]
// public string StringOp
// {
//   get { return strValue; }
//   set { strValue = value; }
// }
//
// In this case the text "Enter a string." will appear as the
// description in the property grid instead of "StringOp".
//
// In order to work properly with the properties window,
// the return type of the property must be one of the following:
// Pin
// string
// int
// double
// any user defined enumeration
//
// The SortOrder option determines the order the properties
// appear in the property window with lower numbered
// sort orders appearing above higher numbered Sort Orders
//
// Additional options include MinValue and MaxValue
// for setting boundaries on a property's numeric values and
// NumLines for specifying how many lines the property
// setting should take inside the property pane. Refer to other
// FunctionBlocks in the application for examples on how to use
// these additional options.

// The [ArrayAssignment] attribute and is used inside the
// application's FunctionBlock classes to decorate
// properties that are present in the device.
// In the Pin R1 shown below, for example,
// the Array Assignment 9 is given to this property.
// This was the array assignment given to R1 in the example
// specification. It is important to set the ArrayAssignment
// attribute since this is the attribute that will be checked
// when programming the device in order to associate the Pin
// with the array assignment.

```

EZ Block Builder Software Maintenance Guide

```
[FBConfigurableProperty(SortOrder = 1)]
[ArrayAssignment(9)]
public Pin R1
{
    get { return m_R1; }
    set { m_R1 = value; }
}

[FBConfigurableProperty(SortOrder = 2)]
[ArrayAssignment(10)]
public Pin R2
{
    get { return m_R2; }
    set { m_R2 = value; }
}

[FBConfigurableProperty]
[ArrayAssignment(24)]
public Pin Output
{
    get { return m_Output; }
    set { m_Output = value; }
}

[FBConfigurableProperty(Description = "HiLo operation")]
[ArrayAssignment(8)]
public HiLoOperation Operation
{
    get { return m_eOperation; }
    set { m_eOperation = value; }
}

// The Name property is a special property that is common to each
// EZ block.No SortOrder option is required in the
// [FBConfigurableProperty]attribute since the name always appears
// at the very top of the property pane. It is still important
// to specify the ArrayAssignment value. The "Description" field
// in X-Series device is the same thing as the
// "Name" of the EZ Block. It is given the array assignment 24 as
// required by the example specification.

[FBConfigurableProperty]
[ArrayAssignment(24)]
public override string Name
{
    get { return m_strName; }
    set { m_strName = value.Trim(); }
}

// Every EZ block must override the ImageResourcePath property
// and return the relative path to the associated GIF image file
// that will be used as the icon for the block. Project images
// reside in the "images" subfolder as shown below.
```

```

public override string ImageResourcePath
{
    get { return @"images\HiLo.gif"; }
}

public HiLoFunctionBlock()
    : base()
{
    // Assign the m_strType and m_strName strings
    // to be the name of the EZ Block. In most cases
    // the string used for both will be identical however
    // they may be different if it is desired that the
    // name shown to the user m_strName is different from the
    // name used internally to reference the class.

    m_strType = "HiLo";
    m_strName = m_strType;

    // The input and output pins are initialized next.
    // Note that each pin is assigned a PinType.
    // The PinType determines if a pin is an input or output
    // and what kind of other pins it can be connected to.
    // There are four types of Pins supported:
    // ValueConsumer - A pin that can read a single value
    // ValueSupplier - A pin that can supply a value
    // ReferenceConsumer - A pin that can read a device address
    // ReferenceSupplier - A pin that can supply a device address
    // For most EZ Blocks, the input pins will be ValueConsumers
    // and the output pin will be a ValueSupplier.

    R1 = new Pin(this, PinType.ValueConsumer, "R1");
    R2 = new Pin(this, PinType.ValueConsumer, "R2");
    Output = new Pin(this, PinType.ValueSupplier);

    // Call the base class initializer as the last
    // act of the constructor
    base.Initialize();
}
}
}

```

6) Enable drag/drop capabilities

Now that the HiLo EZ Block has been added to the project, it must be added to the list of acceptable drag/drop types known to the work area's canvas.

In PageCanvas.cs, add `typeof(HiLoFunctionBlock)` to the static array of `Types` contained in `s_dragDropAcceptableTypes`.

7) Place HiLo in the toolbox

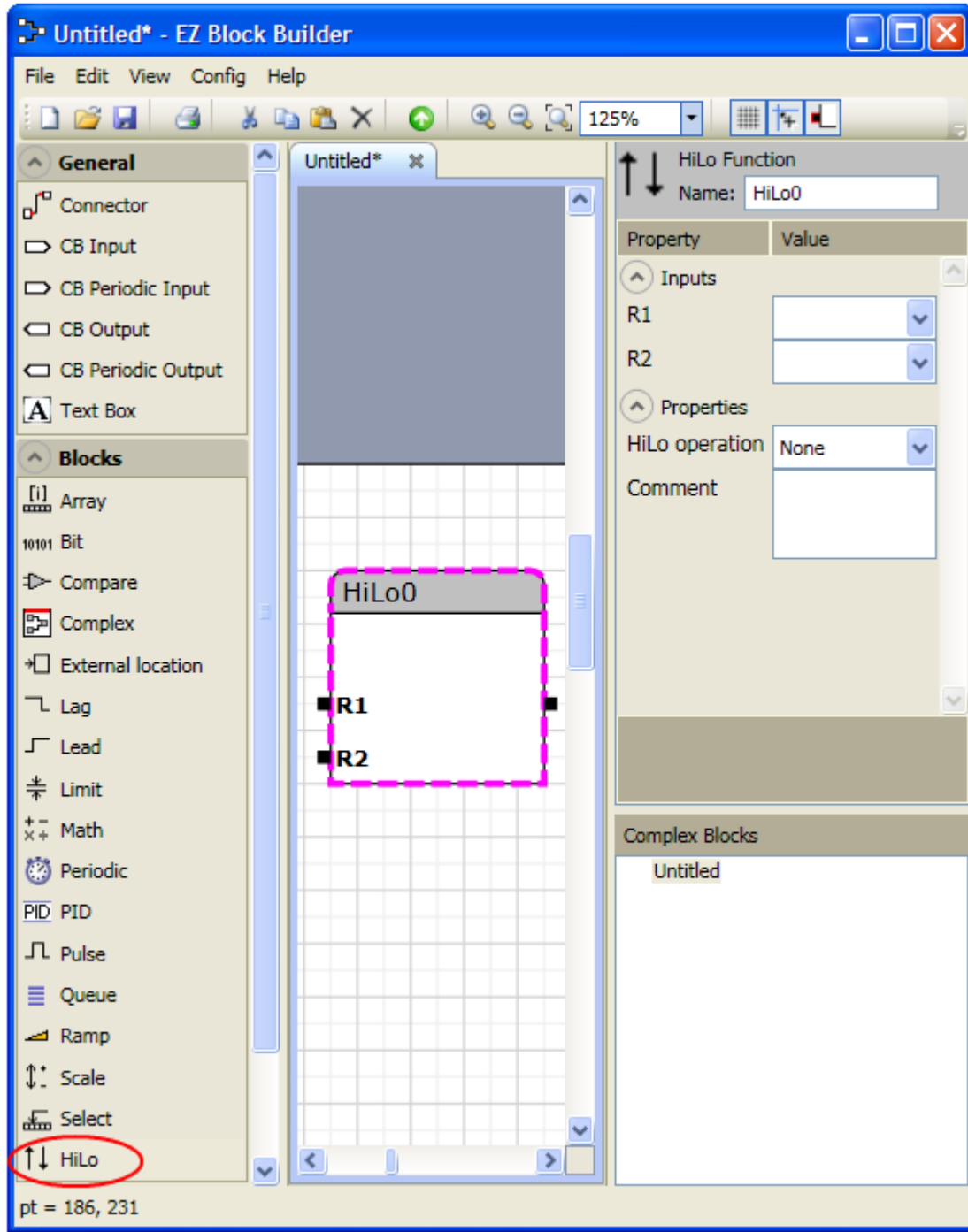
In order for HiLo to appear in EZ Block Builder's toolbox, a single line of code is needed.

In the `InitializeToolbox()` method of `MainWindow.cs`, add the following line.

```
listBox1.Items.Add(new ToolBoxItem("HiLo", typeof(HiLoFunctionBlock),  
@"images\HiLo.gif", "HiLo Block"));
```

8) (Optional) Run the application

This is all the code needed to enable the HiLo function block for inside the UI. More work will be required to get serialization and device communication working but running the application now will produce the following:



Note that the HiLo object now appears in the toolbox along side its icon. A user can drag the HiLo block onto the work area and see that it is named "HiLo0", it has input pins

labeled R1 and R2, and one output pin. In the property pane, the R1 and R2 inputs have associated combo boxes. The HiLo block's pins are fully functional and if other EZ blocks are added to the work area, the HiLo block can be connected to them. A "HiLo operation" property combo box is present pre-populated with the options "None", "High Value", and "Low Value" as called for in the specification. Entries for name and comment fields also appear. A tooltip is also available for the block and is displayed when a user hovers over the block when the global tooltips option is enabled.

This functionality was achieved by defining a few properties and writing a class constructor. No graphics code was required, no UI code was required for populating the property pane, and no conditional logic was required for getting the EZ block to operate in conjunction with other types of EZ Blocks.

As stated above, some additional code will be required in order to customize the HiLo block, add serialization, and to make it function during device uploads but the above code is all that's needed to produce a starting point for these customizations.

The following steps will add these needed additional features.

9) Add remaining features to the EZ block

The code listed below should be inserted into the HiLoFunctionBlock class to add serialization, custom drawing, comparison checking and other features to the block. Each code segment is commented in detail in bold text. This code completes the HiLoFunctionBlock class.

```
// If special drawing code is needed to render text or  
// other visual elements inside the EZ block, override  
// RenderFunctionBlockVisualContent.  
// In the example specification for the HiLo block, we  
// are to display the HiLo operation selected by the user  
// inside the block. The following code shows how to  
// accomplish this.  
  
internal override void RenderFunctionBlockVisualContent  
(DrawingContext dc, int iHighlight, double dOffsetX, double dOffsetY)  
{  
    // Each enumerated value in the HiLoOperation has  
// a [Description] attribute that contains a text string.  
// The following line of code, get's that string and  
// places it in a FormattedText object.  
  
    FormattedText text = FormatText(m_eOperation.Description());  
  
    // The following code draws the formatted text centered  
// horizontally in the block and 70 pixels below the top  
// of the block.  
  
    double dwidth = text.Width;
```

EZ Block Builder Software Maintenance Guide

```
        dc.DrawText(text, new Point(dOffsetX + (m_dWidth - dWidth) / 2,
dOffsetY + 70));
    }

    // EZ Block Builder has the ability to check each function block
    // in a program to determine if it is fully configured.
    // Unconfigured blocks can be reported to the user so he can
    // make changes as necessary.
    // The FunctionBlock's base class implementation of IsFullyConfigured
    // will interrogate each input and output to see if it is connected
    // and return false if it detects a problem.
    // Some EZ Blocks may want to override this method however in order
    // to perform checking that is specialized to the block.
    // For example, HiLo's implementation of IsFullyConfigured will
    // return true even when its inputs are configured when the
    // selected operation is "None".

    public override bool IsFullyConfigured()
    {
        // Check to see if the input pins are required for this
        // configuration

        if (Operation != HiLoOperation.None)
        {
            // The operation is not "None", so only return "true" if
            // both inputs are configured.

            return (R1.IsFullyConfigured() &&
                R2.IsFullyConfigured());
        }

        // In this example, HiLo is considered to be fully
        // configured any time the operation is set to "None".

        return true;
    }

    // The Help string is the text that appears in the lower portion of
    // the property window when the block is selected.
    public override string HelpString()
    {
        return "HiLo block";
    }

    // GetHashCode must be overridden using the normal rules
    // for .NET and C# to produce a suitable hash code.
    // As a general rule, GetHashCode() should be overridden
    // any time Equals() is overridden.

    public override int GetHashCode()
    {
        string strHash = string.Format("{0}{1}{2}{3}", base.GetHashCode(),
R1.ToStringNullSafe(), R2.ToStringNullSafe(), Operation.ToStringNullSafe());
        return strHash.GetHashCode();
    }
}
```

EZ Block Builder Software Maintenance Guide

```
// Override the Equals method so that a comparison of two
// HiLo objects that are configured identically produce
// a result of "true"

public override bool Equals(object obj)
{
    if (obj is HiLoFunctionBlock)
    {
        // Call the base class Equals
        // operation to compare the class name
        // and type fields.

        if (base.Equals(obj))
        {
            HiLoFunctionBlock fb = (obj as HiLoFunctionBlock);

            // This method will return true only when the object
            // being compared is an identically configured
            // HiLo block.

            return ((R1.Equals(fb.R1) &&
                R2.Equals(fb.R2) &&
                (Operation == fb.Operation)));
        }
    }

    return false;
}

// Serialization is the process of loading an EZ block
// from a file, saving it to a file, loading it into
// memory for a copy operation, or reading it from
// memory for a paste operation.

// In most cases, the SerializeAll method for an EZ block
// will be exactly as shown below. SerializeAll is called
// during file save/load operations to begin the
// process of serializing the class.

public override void SerializeAll(XmlArchive ar, string strInstance)
{
    ar.SerializeClass(strInstance, this);
}

// SerializeBody is where the work of saving and loading
// the EZ block to/from an XML file occurs.
// This usually involves writing code to save and
// load the various block properties as described
// in the code below.

public override void SerializeBody(XmlArchive ar)
{
    // The first thing that must be done during a read
    // or write operation is to call the base
```

```

// class implementation of SerializeBody.
// This will cause the user-configurable
// properties that are common to all EZ blocks
// (such as the Name of the block,
// the position of the block in the work area,
// etc.) to be serialized.

base.SerializeBody(ar);

// The XMLArchive that is passed to SerializeBody
// contains a boolean named IsStoring.
// This value will be true when it is desired to
// write the object to an XML file and
// false when it is desired to read the object from
// an XML file

if (ar.IsStoring)
{
    // This is a write operation.
    // Any customized properties must be written.
    // In the case of the example HiLo block there are
    // three user-configurable properties that need
    // to be saved: R1, R2, and the Operation property
    // Note that the Output pin is not being serialized
    // because it is a supplier of data, not a consumer.

    ar.Write("R1", m_R1);
    ar.Write("R2", m_R2);

    // The following notation saves the Operation
    // enumeration as an integer value.

    ar.Write<int>("Operation", (int)m_eOperation);
}
else
{
    // This is a read operation.
    // Any customized properties must be read.
    // In the case of the example HiLo block there are
    // three user-configurable properties that need
    // to be read: R1, R2, and the Operation property

    // Reading a pin is a two step process.
    // First, read the pin by passing a reference to the
    // pin to the XMLArchive's Read method.

    ar.Read("R1", ref m_R1);

    // Next, "hook" the pin to the Function block by
    // setting the pin's FunctionBlock property to this.
    // This pattern will be repeated for every EZ block
    // pin that is read. This is shown below when R2
    // is also read from the XML file.

    m_R1.FunctionBlock = this;
}

```

EZ Block Builder Software Maintenance Guide

```
// All pins follow the same read procedure as outlined
// above.

ar.Read("R2", ref m_R2);
m_R2.FunctionBlock = this;

// The HiLo Operation enumeration is stored in XML as
// an integer value so the following code, reads that
// value and casts it to a HiLoOperation.
// If for any reason, the read operation is unsuccessful,
// m_eOperation is left unchanged.

int iReadEnumVal = 0;
m_eOperation = ar.Read("Operation", ref iReadEnumVal) ?
(HiLoOperation)iReadEnumVal : m_eOperation;
    }
}

// This additional Serialize method is used for binary
// serializations that take place during copy/paste operations.
// It follow much of the same pattern used for the XML
// serialization used above.

public override void Serialize(ProtoLink.Archive ar)
{
    // The base class serialization method must
    // be called first

    base.Serialize(ar);

    if (ar.IsStoring)
    {
        // This is a write operation.
        // Any customized properties must be written.
        // In the case of the example HiLo block there are
        // three user-configurable properties that need
        // to be saved: R1, R2, and the Operation property
        // Note that the Output pin is not being serialized
        // because it is a supplier of data, not a consumer.

        ar.Write(m_R1);
        ar.Write(m_R2);

        // Enumeration values such as m_eOperation may
        // be stored as integer values

        ar.Write((int)m_eOperation);
    }
    else
    {
        // This is a read operation.
        // Any customized properties must be read.
        // In the case of the example HiLo block there are
        // three user-configurable properties that need
```

```

// to be read: R1, R2, and the Operation property

// Reading a pin is a two step process.
// First, read the pin by passing a reference to the
// pin to the XMLArchive's Read method.

ar.Read<Pin>(ref m_R1);
ar.Read<Pin>(ref m_R2);

// Next, "hook" the pin to the Function block by
// setting the pin's FunctionBlock property to this.
// This pattern will be repeated for every EZ block
// pin that is read.

m_R1.FunctionBlock = this;
m_R2.FunctionBlock = this;

// The HiLo Operation enumeration is stored as
// an integer value so the following code, reads that
// value and casts it to a HiLoOperation.

int iOperation = ar.ReadInt32();
m_eOperation = (HiLoOperation)iOperation;
    }
}

```

10) Configure EZ block capacity location

Each EZ block that has a corresponding function inside the device has a capacity register. This capacity register, which resides inside the device, contains a value which is the number of instances of a particular block type that is present in the program. In the example specification, the register location is "1".

Add the following value to the FBCapacityRegister enumeration of the Communications class:

```
[Description("HiLo")] HiLo = 50
```

Examine the ComputeCapacities() method of the Communications class. The method has an "if" block that sets a capacity table to a value depending on the EZ Block type. Insert the following code in the "if" block:

```

...
else if (fb is HiLoFunctionBlock)
{
    dictCapacityTable[FBCapacityRegister.HiLo]++;
}
...

```

11) Add ability to read EZ block from device

When a HiLo block is read directly from the device registers instead of from an XML file, EZ Block Builder must instantiate the block for use by the end-user.

Examine the `LoadDeviceConfigurationFromRegisters()` method of the `LoadFileFromDeviceWindow` class. The method contains a `switch` statement that checks the `FBCapacityRegister` value of the current register dictionary.

Add the following code to the `switch` block:

```
case Communications.FBCapacityRegister.HiLo:
    HiLoFunctionBlock hiLoFB = new HiLoFunctionBlock();
    ReadNonPeriodicRegistersFromDevice(iAppID, iInstance,
        dictCurrentRegister[kvpEntry.Key]++, hiLoFB);
    AddFunctionBlockFromDevice(pNewProgram, hiLoFB, m_dX, m_dY);
    break;
```

12) Test the application

The example HiLo block is now fully implemented. Since the HiLo block does not actually exist on the device, attempting to upload or download the block will, of course, fail, however the steps followed in the creation of this example block can be reused when creating new EZ blocks for use with the EZ Block Builder application.

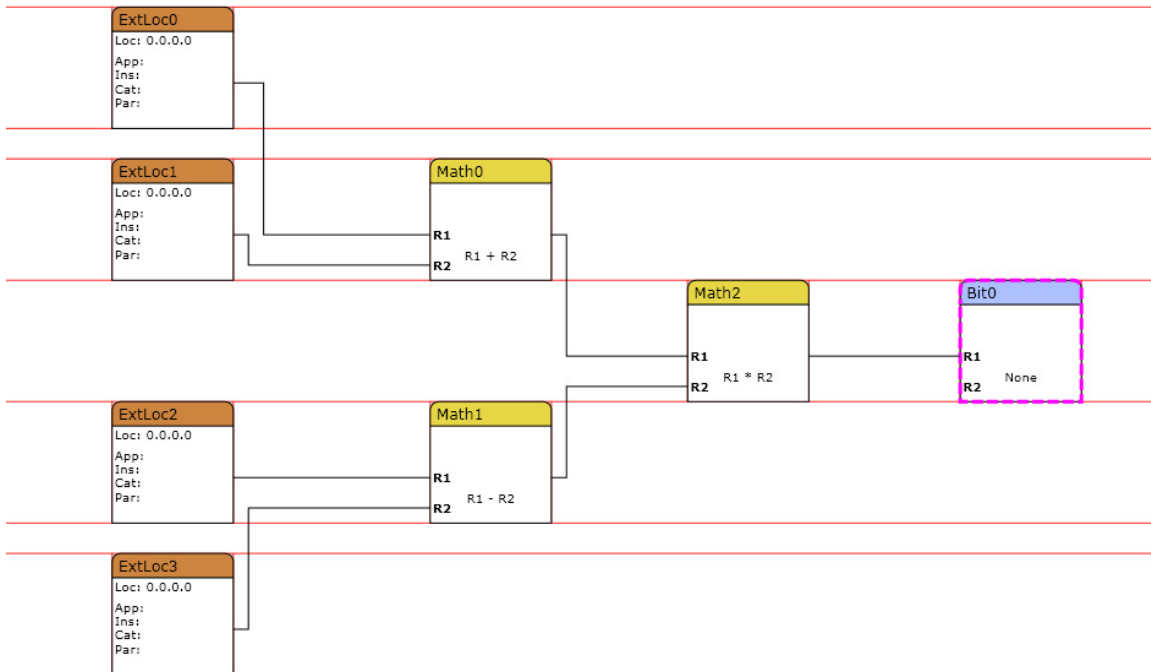
7 Line Routing

EZ Block Builder automatically routes lines (Connectors) that connect function blocks.

EZ Block Builder uses an implementation of the “Corner Stitching” data structures and algorithms described in a paper by John K. Ousterhout entitled “Corner Stitching: A Data Structuring Technique for VLSI Layout Tools”. The basic corner stitching data structure is the Tile. All the workspace is divided into non-overlapping tiles. The tiles can either represent occupied space (used by a `FunctionBlock`) or empty space (available to route lines through). Each tile maintains a reference to its immediate neighbors at its corners. These references are the “stitches”. There are eight stitches for each tile: `TopLeft`, `TopRight`, `BottomLeft`, `BottomRight`, `LeftTop`, `LeftBottom`, `RightTop` and `RightBottom`. There are various algorithms for basic operations such as inserting tiles, finding all of a tile’s neighbors along a specific edge, etc.

Another paper used for reference was “A Novel Algorithm for Line Routing in Hierarchical Diagrams” by Tobias Reinhard, et al. It describes methods for finding optimum routes for lines connecting non-space blocks.

The diagram below shows a sample of how the tiling is used. The space tiles are outlined in red. They are not shown during the normal operation of the program. All lines that connect function blocks must be routed through space tiles.



If an EZ Block Builder program is laid out in such a way that there is not a possible path (a block is completely surrounded and boxed in by other function blocks for instance), then the line routing algorithm defaults to a simple route that doesn't avoid other blocks.

The line routing algorithm contains a post processing step to look for line intersections after all lines have been routed. The algorithm then adds "line hops" over all intersecting lines to make the program more readable.

8 Class Listing

Class	Description
AboutDialog	This window displays “About EZ Block Builder” dialog
AliasListCategory	This class is used to serialize a Category element of an Alias list to XML.
AliasCreationWindow	This window displays the “Create Alias” dialog, launched when the user selects the “Create Alias” button from an External Location function block’s property page.
AliasListArchive	This class is used to read and write alias list files.
AliasListEntry	This class is used to serialize an entire alias entry list to XML.
AliasListParameter	This class is used to serialize a Parameter element of an Alias list to XML.
AppTypeManager	This class is used to process the AppTypes.xml file
Archive.cs	This class contains methods used for function block copy/paste operations.
ArrayAssignmentAttribute	This class defines the [ArrayAssignment] attribute and is used inside the application's FunctionBlock-derived classes to decorate properties that are present in the device.
ArrayFunctionBlock	This class defines the capabilities of an Array Function Block
AssignmentControl	This class contains the control used in the Options dialog for configuring the various assignments.
AssignmentWindow	This is the shell class for the options dialog
BitFunctionBlock	This class defines the capabilities of a Bit Function Block
BoundaryItem	This class is used in lists to encapsulate exposed pin classes contained in function blocks.
CommConfigDialog	This window allows the configuration of the communication setting options.
Communications	This class contains the logic required for connecting and communicating with X-series devices.
CompareFunctionBlock	This class defines the capabilities of a Complex Function Block
ComplexFunctionBlock	This class defines the capabilities of a Complex Function Block

ComplexInputBase	This abstract class derives from FunctionBlock and provides the base functionality for the ComplexValueInput and ComplexReferenceInput classes.
ComplexOutputBase	This abstract class derives from FunctionBlock and provides the base functionality for the ComplexValueOutput and ComplexReferenceOutput classes.
ComplexReferenceInput	This class derives from ComplexInputBase and provides the functionality required to provide reference inputs to a complex function block.
ComplexReferenceOutput	This class derives from ComplexInputBase and provides the functionality required to provide reference outputs to a complex function block.
ComplexValueInput	This class derives from ComplexInputBase and provides the functionality required to provide value inputs to a complex function block.
ComplexValueOutput	This class derives from ComplexInputBase and provides the functionality required to provide value outputs to a complex function block.
ConnectionStatusControl	This class defines a user control that displays the current connection status with an X-Series device.
Connector	This class defines a program elements used to express a relationship between two function blocks.
ConnectorEndBlock	This helper class is used by the Connector class to define the function of the connector's endpoints.
ConnectorEndVisual	This helper class is used by the Connector class to define the visual properties of the connector's endpoints.
Constants	This class contains constant values used throughout the application.
DevAppListViewDataItem	This wrapper class is used by various list views when a text representation of an App ID is desired.
Device	This class contain a variety of methods used to interact with the device durinf file upload and download operations.

DeviceApplication	This data class encapsulates information about applications that exist within the X-Series device.
DrawingPage	This class is used to represent a printable page of the work area
DrawingPagePaginator	This class is used to determine how to divide the work area into separate printable pages.
ExtensionMethods	This class defines the various C# extension methods used throughout the application.
ExternalAssignmentAttribute	This class defines the [ExternalAssignment] attribute and is used inside the application's External Location Function Block class to decorate properties that represent pins providing access to registers external to the current application
ExternalLocationFunctionBlock	This class defines the capabilities of an External Location Function Block
FBConfigurablePropertyAttribute	This class defines the [FBConfigurableProperty] attribute and is used inside the application's Function Block classes to decorate properties that are configurable via the property window.
FBPropertySummaryWindow	This class defines the window displayed as function block tooltips.
FunctionBlock	This class serves as the base class for the other function block classes and provides a set of common methods and properties.
FunctionBlockVisual	A function block drawing visual class.
FunctionBlockVisualPair	A wrapper class that associates a function block visual with a function block.
HSegment	This class is used to represent a horizontal line segment. This class is used for line routing purposes.
HSegmentCollection	This class contains a collection of VSegments.
InputPinVisual	An input pin drawing visual class.
InputPropertyControl	A wrapper class for a combo box used to represent input properties in the function block's properties pane.
IXmlSerialize	This interface provides the contract needed for objects to read themselves from disk and to write themselves to disk.
LagFunctionBlock	This class defines the capabilities of a Lag Function Block

EZ Block Builder Software Maintenance Guide

LeadFunctionBlock	This class defines the capabilities of a Lead Function Block
LimitFunctionBlock	This class defines the capabilities of a Limit Function Block
ListViewItemStyleSelector	A utility class that performs alternate-row shading on list views
LoadFileFromDeviceWindow	This class displays a window that can be used to select a program file from a connected device for downloading into the application.
LoadSelectionWindow	This informational dialog is presented when the user must decide if the application should load a program directly from the device's registers or from a previously saved program file.
LogEvent	A helper class used for logging events.
MainWindow	The application's main window containing the work area, property pages, toolbox, menus and other program elements.
MathFunctionBlock	This class defines the capabilities of a Math Function Block
OptionsManager	The options window contains various global settings such as alias list and profile selections.
OutputPinVisual	The output pin's drawing visual
OutputPropertyControl	A wrapper class for a combo box used to represent an output property in the function block's properties pane.
PageCanvas	This class is the canvas used in the work area of the main window of the application.
PageSetupWindow	This window allows the user to configure various options regarding the work area and printed page layouts.
PaperSize	This class is used to encapsulate the page size settings used during printing operations.
PeriodicFunctionBlock	This class defines the capabilities of a Periodic Function Block
PIDFunctionBlock	This class defines the capabilities of a PID Function Block
Pin	This class is used by function blocks to define how they may be connected with other function blocks. The pin class may be configured to behave as an input or output.

EZ Block Builder Software Maintenance Guide

Plane	This class is used for line routing purposes. The Plane object is composed of multiple non-overlapping Tile objects that spatially represent the entire work area.
PrintPreviewWindow	This class is used to show a WYSIWYG representation of the printed output.
PriorityQueue	This class is used in the routing logic of the Connector class to help determine the appropriate tile to use during line drawing operations
ProfileArchive	This class encapsulates the functionality required to read and write profile XML files.
ProfileInstanceNameWindow	This window allows the user to name or rename instances associated with a profile entry.
ProfileListViewDataItem	This wrapper class is used by the profile listview to hold the data associated with a profile entry.
ProfilePropertiesWindow	This window displays detailed information about the currently selected profile.
Program	This class represents a complete application program. Serializing this class results in an application program file.
ProgramControl	This class associates a program with a page canvas so that it may be displayed visually.
ProgramTabItem	This class represents an individual tab of in the work area's tab control.
ProgramTreeView	This class displays a program's complex function block hierarchy in visual form.
ProgramValidationWindow	This window contains the functionality required to validate if a program is complete and ready for upload to a device.
PropertiesControl	This window is displayed as part of the main window and contains detailed information about the currently selected function block. A user may modify the configuration of the selected function block using the user interface elements provided by this class.
PulseFunctionBlock	This class defines the capabilities of a Pulse Function Block
QueueFunctionBlock	This class defines the capabilities of a Queue Function Block
RampFunctionBlock	This class defines the capabilities of a Ramp Function Block

EZ Block Builder Software Maintenance Guide

Range	This class is used during X-series device communications to specify a set of registers on which to operate.
RegisterPromptWindow	This window is displayed when the application needs to determine if an application register should be re-used or reserved.
SaveFileToDeviceWindow	This window allows a user to selected a location within an X-Series device to save a device and to begin the save operation.
ScalarBox	This wrapper class contains the UI element used to enter numeric data in the property page.
ScaleFunctionBlock	This class defines the capabilities of a Scale Function Block
Segment	This class represents a line segment used for line routing purposes. It is the base class for HSegment and VSegment.
SelectFunctionBlock	This class defines the capabilities of a Select Function Block
TextBoxBlock	This class represents the capabilities of a Text Box Block.
Tile	This class is used for line routing purposes. The PageCanvas class has a single instance of a Plane class. The Plane is divided into multiple non overlapping Tile objects. Tiles can represent areas that contain FunctionBlocks as well as areas that contain empty space.
TimeSpanBox	This wrapper class contains the UI element used to enter time information in the property page.
ToolBoxItem	This class is used to visually represent items that may be dragged onto the work area from the toolbox.
Variable	This class is used during X-series device communications to specify a variable entity inside the device.
VSegment	This class is used to represent a vertical line segment. This class is used for line routing purposes.
VSegmentCollection	This class contains a collection of VSegments.
XmlArchive	This class contains methods used to serialize program information to disk in XML form.

EZ Block Builder Software Maintenance Guide

XSeriesAddress	This class encapsulates the address values required for writing and reading data from an X-series device.
XSeriesComm	This class contains methods that manage the lifecycle of X-series communication sessions.



ABB Inc.

Totalflow Products

7051 Industrial Blvd.

Bartlesville, Oklahoma 74006

Tel: USA (800) 442-3097

International 001-918-338-4880

2104002-001 (AA)

