INTRODUCTION INTO SIMPL#

A QUICK GUIDE TO HELP YOU GET YOU STARTED

DO I HAVE TO LEARN A NEW LANGUAGE?

No! There is no need to learn a new language. SIMPL Windows and SIMPL+ languages are not going away. SIMPL# is merely an addition to these two great programming languages to make Crestron 3-Series Control Systems even more powerful.

WHAT IS SIMPL#

SIMPL# is the name given to the C# Libraries (that are within the Crestron sandbox), that can be called from SIMPL+. It will, for example, enable you to easily connect to HTTP(S) servers, parse XML, and even connect to SQL Servers.

This is not our own invented language. All C# language constructs are available. It's even developed and debugged in Visual Studio, not proprietary Crestron tools. We did provide a plugin for Visual Studio to create SIMPL# Libraries. This plugin makes sure that the compiled code works on a 3-Series, and fits within the Crestron Sandbox.

The Sandbox is in place to ensure that the C# code cannot negatively affect the 3-Series underlying architecture and subsystems.

WHAT DO I NEED

In order to start programming in SIMPL#, you'll need the following:

- 3-Series firmware 1.007.0019 or later
- SIMPL Windows 4.2.16 or later
- Device Database 50.00.004.00 or later
- Crestron SIMPL# Library plugin 1.0.0149 or later
- Visual Studio 2008 Professional, with Service Pack 1 installed
- Working knowledge of C#

WHEN TO USE SIMPL#?

Some of the areas where SIMPL# really shines are:

- Ability to parse XML natively; no need to write your own XML Parser
- Ability to connect to webservers natively.
- Ability to accept HTTP requests from other web services, or web clients.
- Ability to connect to SQL Server, for querying and updating databases.
- Ability to do floating point calculations.

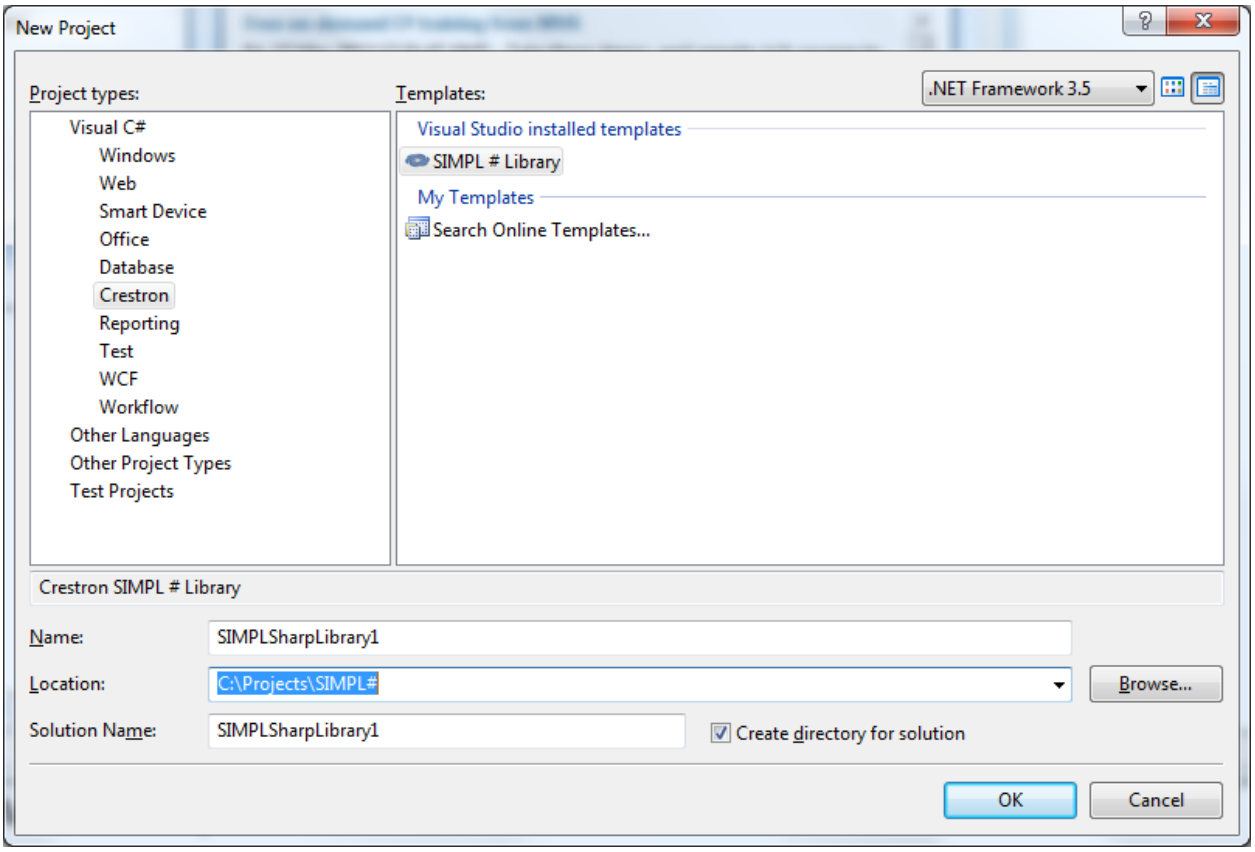
There are many more areas, but if you currently have a need for any of these, you'll probably benefit greatly from writing a SIMPL# library.

HOW TO GET STARTED

After you've determined you want to write a SIMPL# library, and you have installed all the tools required, start Visual Studio 2008.

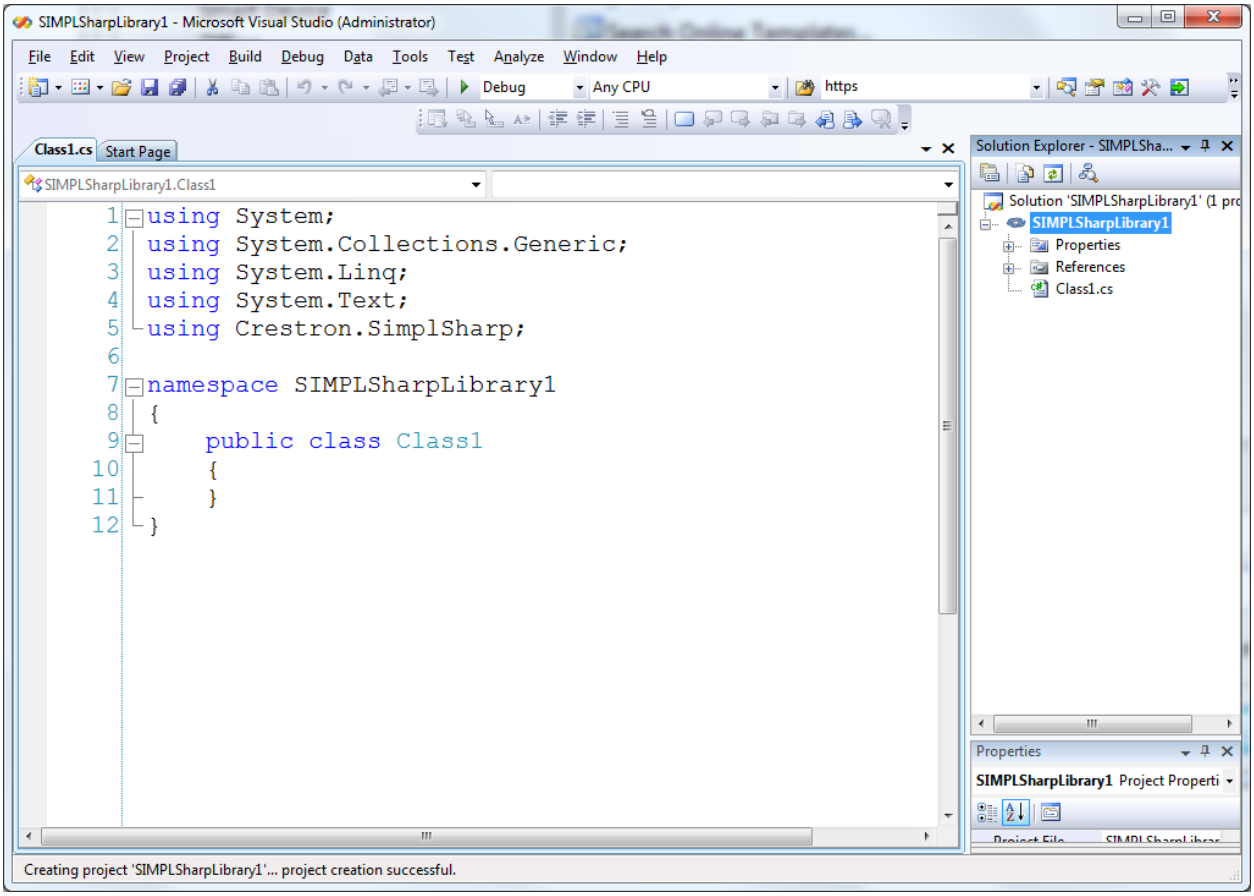
In Visual Studio, do **F**ile, **N**ew, **P**roject.

The new project dialog comes up. On the left, you should see Crestron in the list. When you select that, you can pick SIMPL# Library.



Fill in the names, and pick the right location. When done, click "**OK**"

An almost empty document comes up with an empty template:



The Crestron Plugin has already added all the references to the S# libraries that are needed. It also created the basis for a new C# Class. From here, you'll rename the namespace and the class, and start writing the code.

I HAVE A SIMPL# LIBRARY… NOW WHAT?

Once you have finished writing the SIMPL# Library, you should compile and build it in Visual Studio. Once the file builds without errors, a .CLZ file will be created in the BIN subfolder under the project folder. This CLZ file is what you'll be referring to from SIMPL+.

Once you have the library in place, you can start writing your SIMPL+ wrapper. Best practice is to keep SIMPL+ as light-weight as possible, and do all the "heavy lifting" in SIMPL#.

To link a library in SIMPL+, you need to add a reference to the SIMPL# Library. Use the **#USER_SIMPLSHARP_LIBRARY** keyword. There are two options:

- Copy the CLZ into the same folder as the .USP file; in this case, you can simply type:
  #USER_SIMPLSHARP_LIBRARY "MySimplSharpLibrary"
- If you don't want to copy the file over, but rather refer to the output of the VS Project, use an absolute path, such as: **#USER_SIMPLSHARP_LIBRARY** "C:\Projects\MySimplSharpLibrary\bin\MySimplSharpLibrary"

Once you have done this, you can right-click in the SIMPL+ editor, and select "Open API For…." To see all the exposed classes, methods, and properties.

Within SIMPL+, you can now start using the class(es) defined in the SIMPL# library. You do that simply by declaring a variable of the class type, such as:

```
MyClass myLocalClassVariable;
```

Please be aware that creating a variable of a C# class in SIMPL+, that the default constructor is automatically called. The line above is interpreted as:

```
MyClass myLocalClassVariable = new MyClass();
```

The local variable called `myLocalClassVariable` now has access to any methods, and properties defined in the class in C#, using the dot-notation. Please be aware that SIMPL+ is case sensitive when it comes to addressing C# methods, classes, and properties.

SIMPL# DATA TYPES COMPARED TO SIMPL+ DATA TYPES

Since you will be passing data back-and-forth between SIMPL+ and SIMPL#, it is good to know the comparison between the different data types

SIMPL# Datatype	Equivalent SIMPL+ Datatype	Can Declare in SIMPL# Library	Can Declare in SIMPL+ Module	Array Support
string	STRING	YES	YES	1D
SimplSharpString	STRING	YES	NO	1D
int	SIGNED_LONG_INTEGER	YES	YES	1D, 2D
uint	LONG_INTEGER	YES	YES	1D, 2D
short	SIGNED_INTEGER	YES	YES	1D, 2D
ushort	INTEGER	YES	YES	1D, 2D
struct	STRUCTURE	YES	YES	1D
class	CLASS	YES	NO	1D

HOW TO USE THE SAMPLE PROGRAM

The sample program included in this introduction is a SIMPL# library that connects to Flickr! to create a photo viewer on a SmartGraphics Xpanel or touch screen. It utilizes the HttpClient, and XML Parsing, as well as callbacks from SIMPL# into SIMPL+.

There are two folders:

- VSSolution: this holds the Visual Studio Solution for FlickrFeed. This is the SIMPL# Library. Once you've opened FlickFeed.sln, please compile it.
- SMWProject: this holds the SIMPL Windows program, SIMPL+ module, and Xpanel definition.

After you've compiled the C# code, please go into the SIMPL+ module, and update the absolute path where the SIMPL# Code is located.

Also, you should update the UserID (line 22) to your own Flickr ID.

First, compile the SIMPL+ module, and then compile the SIMPL Windows program. After startup, the SIMPL Windows program waits 30s before starting the SIMPL+ module.

Congratulations, you've just loaded your first SIMPL# Module!

SIMPL#/SIMPL+ IMPLEMENTATION DETAILS

NEW KEYWORDS

NEW RESERVED KEYWORDS

- namespace
- class
- delegate
- static
- EventHandler
- DelegateProperty

NEW SIMPL+ KEYWORDS

- RegisterEvent
- UnregisterEvent
- RegisterDelegate
- UnregisterDelegate
- EventArgs
- callback

SIMPL# LIBRARY IMPLEMENTATION

OVERVIEW

SIMPL# Libraries can expose namespaces, structure definitions, and classes. A class can expose variables, functions and properties. Only functions and variables that have matching SIMPL+ datatypes can be exposed to a SIMPL+ module. All other functions and variables are not available to be used outside of the SIMPL# Library.

NOTE: SIMPL# classes, structures and variables ARE case-sensitive

DATATYPES

The datatypes that can be exposed to SIMPL+ are contained in the following table:

SIMPL# Datatype	Equivalent SIMPL+ Datatype	Can Declare in SIMPL# Library	Can Declare in SIMPL+ Module	Array Support
String	STRING	YES	YES	1D
SimplSharpString	STRING	YES	NO	1D
Int	SIGNED_LONG_INTEGER	YES	YES	1D, 2D
Uint	LONG_INTEGER	YES	YES	1D, 2D
Short	SIGNED_INTEGER	YES	YES	1D, 2D
Ushort	INTEGER	YES	YES	1D, 2D
Struct	STRUCTURE	YES	YES	1D
Class	CLASS	YES	NO	1D

SIMPL# Function Return Type	Equivalent SIMPL+ Function Return Type	Can Declare in SIMPL# Library	Can Declare in SIMPL+ Module
String	STRING_FUNCTION	YES	YES
Int	SIGNED_LONG_INTEGER_FUNCTION	YES	YES
Uint	LONG_INTEGER_FUNCTION	YES	YES
Short	SIGNED_INTEGER_FUNCTION	YES	YES
Ushort	INTEGER_FUNCTION	YES	YES
Void	FUNCTION	YES	YES

Modules targeted for 3-Series now support nested structures and classes.

Classes and structures defined within SIMPL# Libraries can be derived from a base structure or class.

NOTE: SIMPL+ string arrays cannot be passed to SIMPL#.

EXAMPLE

The following code is an example of how structures and classes can be defined and used:

```
namespace SIMPLSharpLibrary1
{
    public struct tagBaseLibStruct
    {
        public string buf;
        public SimplSharpString simplSharpStr;
        public int i;
        public int[] arr;
    }

    public struct tagLibStruct
    {
        public tagBaseLibStruct baseLibStruct;
        public int j;
    }

    public class baseClass
    {
        public baseClass() { }
        public int baseClassInt;

        public void baseClassFn() {}
    }

    public class Class1 : baseClass
    {
        public tagLibStruct libStruct;

        public int x;

        public SimplSharpString str;

        public int classFn() { return 0; }
    }

}
```

CLASSES

OVERVIEW

Classes represent a collection of functions, structures and variables. SIMPL# classes are exposed to SIMPL+ and SIMPL+ can then create variables of this class type. Once defined, SIMPL+ can then call and use the class's functions and variables.

The following demonstrates how to do this:

SIMPL#:

```
namespace SIMPLSharpLibrary1
{
	public class Class1
	{
		int ID;
		public int GetID()
		{
			return ID;
		}
	}
}
```

SIMPL+:

```
	function InitializeSimplSharpObjects()
	{
		Class1 myClass;
		integer ID;


		ID = myClass.GetID();
	}
```

INITIALIZING STRUCTURES

OVERVIEW

All Objects declared in SIMPL# must be initialized. By default, they will be null. It is up to the programmer to ensure that all objects have been properly initialized.

A structure type only defines its containing members. When declaring a SIMPL# structure variable within SIMPL#, all appropriate objects must be initialized.

There is no way for SIMPL+ to initialize structure objects. This must be implemented within the SIMPL# class. To do this, a function can be written SIMPL# which can initialize and return the allocated object.

The following demonstrates how to do this:

SIMPL#:

```
namespace SIMPLSharpLibrary1
{
	public struct tagMyStruct
	{
		public string structstr;
	}

	public class Class1
	{
		public string myStr;

		public int InitStruct(ref tagMyStruct myStructArg)
```

```
            {
                  myStructArg.structstr = string.Empty;
                  myStructArg.structstr = "abc";

                  return 0;
            }
      }
}
```

SIMPL+:

```
      function InitializeSimplSharpObjects()
      {
            tagMyStruct myStruct;
            Class1 myClass;

            myClass.InitStruct( myStruct );
            myClass.myStr = "abc";

      }
```

REFERENCING STRUCTURE MEMBER VARIABLES FROM SIMPL+ IS CASE-SENSITIVE.

DELEGATES

OVERVIEW

Delegates are simply function pointers. In SIMPL#, you can define a signature for a function, create a variable of that delegate type, then assign a SIMPL+ function with a matching signature to that variable. With that, a function within your SIMPL+ module can be called from SIMPL#.

The easiest way to do this is as follows:

In SIMPL#, define a function's signiture

```
public delegate short DelegateComputeFn(void);
```

In SIMPL#, create a property of with this delegate type

```
public DelegateComputeFn ComputeFn { get; set; }
```

In SIMPL+, assign the SIMPL# property to a SIMPL+ function

```
RegisterDelegate (myClass, ComputeFn, ComputeCallbackFn);
```

In the above example, SIMPL#'s ComputeFn will call ComputeCallbackFn within SIMPL+.

`UnregisterDelegate` can be called to clear the delegate within SIMPL+. This will stop SIMPL# from calling the previously registered function in SIMPL+.

NOTE: Delegate functions are synchronous calls

EXAMPLE: CREATING AND CALLING A DELEGATE

The following code is an example of how to define a delegate within SIMPL# and create and call a callback function in SIMPL+.

SIMPL#:

```
namespace SIMPLSharpLibrary1
{
	// Define the delegate:
	public delegate short DelegateComputeFn(void);
	public delegate string DelegateFn(uint id);

	// Create a property in a class:
	public class DelegateTest
	{
		// non-static property
		public DelegateComputeFn ComputeFn { get; set; }

		// static property
		static public DelegateFn StaticFn { get; set; }

		private short Compute ( )
		{
			short ret = 0;

			// Call the mapped function in SIMPL+
			// * remember to check for null in case *
			// *   the function was never mapped!   *
			if (DelegateFn != null)
				ret = DelegateFn();

			return ret;
		}
	}
}
```

SIMPL+:

```
DelegateTest delTest;

function Init()
{
```

```
        // register non-static property
        RegisterDelegate (delTest, ComputeFn, ComputeCallbackFn);

        // register static property
        RegisterDelegate (DelegateTest, StaticFn, StaticCallbackFn);
}

function Close()
{
        // register non-static property
        UnregisterDelegate (delTest, ComputeFn );

        // register static property
        UnregisterDelegate (DelegateTest, StaticFn );
}

callback signed_integer_function ComputeCallbackFn()
{
        return (0);
}

callback string_function StaticCallbackFn (long_integer id)
{
        return ("");
}
```

DELEGATE FUNCTION ARGUMENTS

The SIMPL+ callback function must match the SIMPL# delegate's signature. The function's return type and all of its arguments must be the same data type as what is defined within SIMPL#. If any argument type or return type differ, the callback will not be matched and will not be called during runtime.

The following table shows how to map SIMPL# data types to SIMPL+ data types:

SIMPL# Datatype	SIMPL+ Datatype (Argument)	SIMPL+ Datatype (Function)
string	STRING	STRING_FUNCTION
SimplSharpString	STRING	STRING_FUNCTION
int	SIGNED_LONG_INTEGER	SIGNED_LONG_INTEGER_FUNCTION
uint	LONG_INTEGER	LONG_INTEGER_FUNCTION
short	SIGNED_INTEGER	SIGNED_INTEGER_FUNCTION
ushort	INTEGER	INTEGER_FUNCTION
<user_class>	<user_class>	

EVENT HANDLERS

OVERVIEW

Events provide a way for SIMPL# to provide notifications to SIMPL+. A SIMPL+ Module can subscribe to a SIMPL# event, and when SIMPL# raises that event, the subscribed SIMPL+ event handler function will be called.

NOTE: Event Handler functions are asynchronous calls

EXAMPLE 1 - BASIC

In the simplest case, defining an event using C#'s EventHandler class is used. When the event is triggered, the sender (this), along with C#'s EventArgs class is passed.

SIMPL#:

```
namespace SIMPLSharpLibrary1
{
      public event EventHandler MyEvent;
```

```
	public class ControlSystem
	{
		public void TriggerEvent1Handler()
		{
			MyEvent(this, new EventArgs());
		}
	}
}
```

SIMPL+:

```
	ControlSystem myClass;

	function Init()
	{
		RegisterEvent (myClass,MyEvent,MyEvnt_Hndlr);
	}

	eventhandler MyEvnt_Hndlr(ControlSystem sender, EventArgs args)
	{
	}
```

EXAMPLE 2 – USING DELEGATES (WITHIN NON-STATIC CLASSES)

In the next case, a custom event handler is created using a delegate. Once the delegate is defined, an event is created with this delegate type.

SIMPL#:

```
namespace SIMPLSharpLibrary1
{
	public delegate void CustomHandler1(object sender, EventArgs e);
	public event CustomHandler1 MyEvent;

	public class ControlSystem
	{
		public void TriggerEvent1Handler()
		{
			MyEvent(this, new EventArgs());
		}
	}
}
```

SIMPL+:

```
	ControlSystem myClass;

	function Init()
	{
		RegisterEvent (myClass,MyEvent,MyEvnt_Hndlr);
	}
```

```
	eventhandler MyEvnt_Hndlr(ControlSystem sender, EventArgs args)
	{
	}
```

EXAMPLE 2 – USING DELEGATES (WITHIN STATIC CLASSES)

In the next case, a custom event handler is created using a delegate. Once the delegate is defined, an event is created with this delegate type.

SIMPL#:

```
namespace SIMPLSharpLibrary1
{
	public delegate void CustomHandler1(EventArgs e);
	public event CustomHandler1 MyEvent;

	public class ControlSystem
	{
		public void TriggerEvent1Handler()
		{
			MyEvent(new EventArgs());
		}
	}
}
```

SIMPL+:

```
	ControlSystem myClass;

	function Init()
	{
		RegisterEvent (myClass,MyEvent,MyEvnt_Hndlr);
	}

	eventhandler MyEvnt_Hndlr(EventArgs args)
	{
	}
```

EXAMPLE 2 – DERIVING FROM EVENTARGS

In this case, a class is declared deriving from C#'s EventArgs class. That class can then be populated and passed to the SIMPL+ event handler function.

SIMPL#:

```
namespace SIMPLSharpLibrary1
{
	public class MyEventArgs : EventArgs
	{
		public int ID { get; set; }
```

```
        }

        public delegate void CustomHandler(object sender, MyEventArgs e);
        public event CustomHandler MyEvent;

        public class ControlSystem
        {
              public void TriggerEvent1Handler()
              {
                    MyEvent(this, new MyEventArgs());
              }
        }
}
```

SIMPL+:

```
        ControlSystem myClass;

        function Init()
        {
              RegisterEvent (myClass,MyEvent,MyEvnt_Handler);
        }

        eventhandler MyEvnt_Hndlr(ControlSystem sender, MyEventArgs args)
        {
        }
```