

# Getting Started with Instrument Control Using Microsoft Visual Studio's C# Language

## Intro

Are you tasked with automating the setup and control of instruments for a test sequence? Do you prefer the C# programming language over others but not sure where to start? Whether you are a new programmer looking for some help and stepping stones, or a seasoned Test Engineer looking for just a hint of guidance, this piece may be just the thing for you. In the following pages you will find:

- A general overview of the downloading and installation of the Microsoft Visual Studio 2019 Integrated Development Environment
- Steps for starting up your first C# .NET Framework project
- Steps for adding the VISA COM reference tools
- An overview on the general building blocks you will need for connecting to, sending commands to, and receiving data from your connected instrumentation.
- Building and running your simple instrument-controlling C# application

For brevity, we have specifically skipped discussing things like good coding practices, syntax and how to do what with C#, and any debugging information. There are plenty of online resources for you to refer to without us recreating those discussion points here.

## Prerequisites

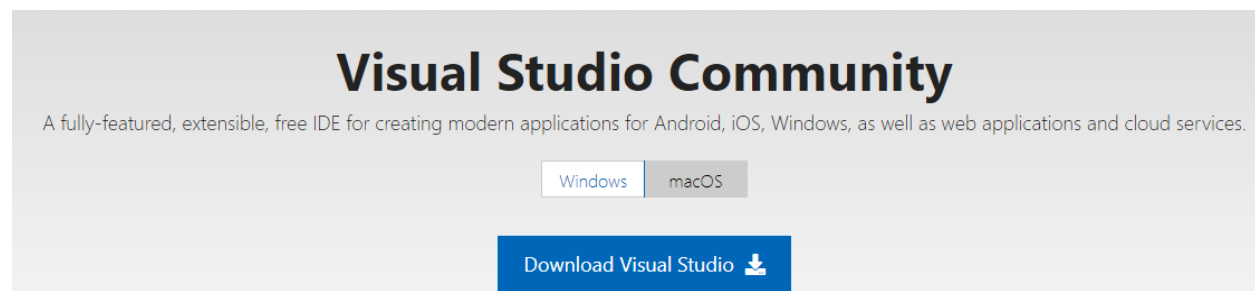
Prior to progressing through this document, you will need to ensure you have some programming knowledge and additional necessary tools installed on your computer. These include:

- A general or basic background in computer programming
- National Instruments VISA 17.5 (or higher) Runtime or Full Version

## Download Visual Studio Community

Download the free version of Visual Studio from the Microsoft website:

<https://visualstudio.microsoft.com/vs/community/>

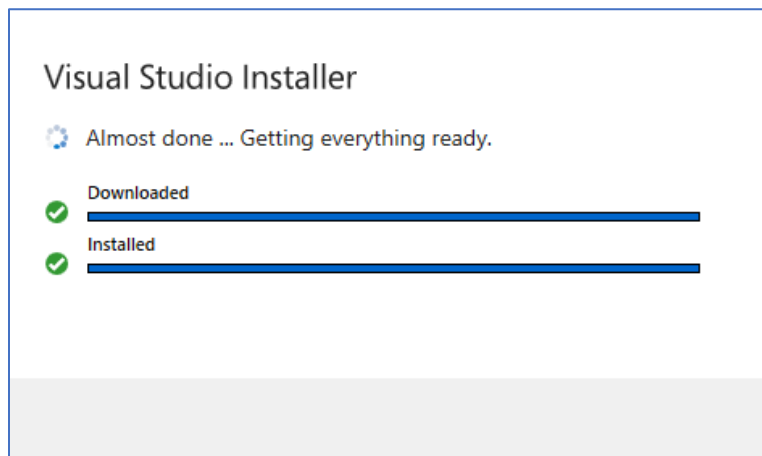


What you will need to use the VS Community version:

- A pulse
- A Hotmail or Outlook account
- A passion for writing code
- A network connection

## Install and Select the Basic Programming Tools

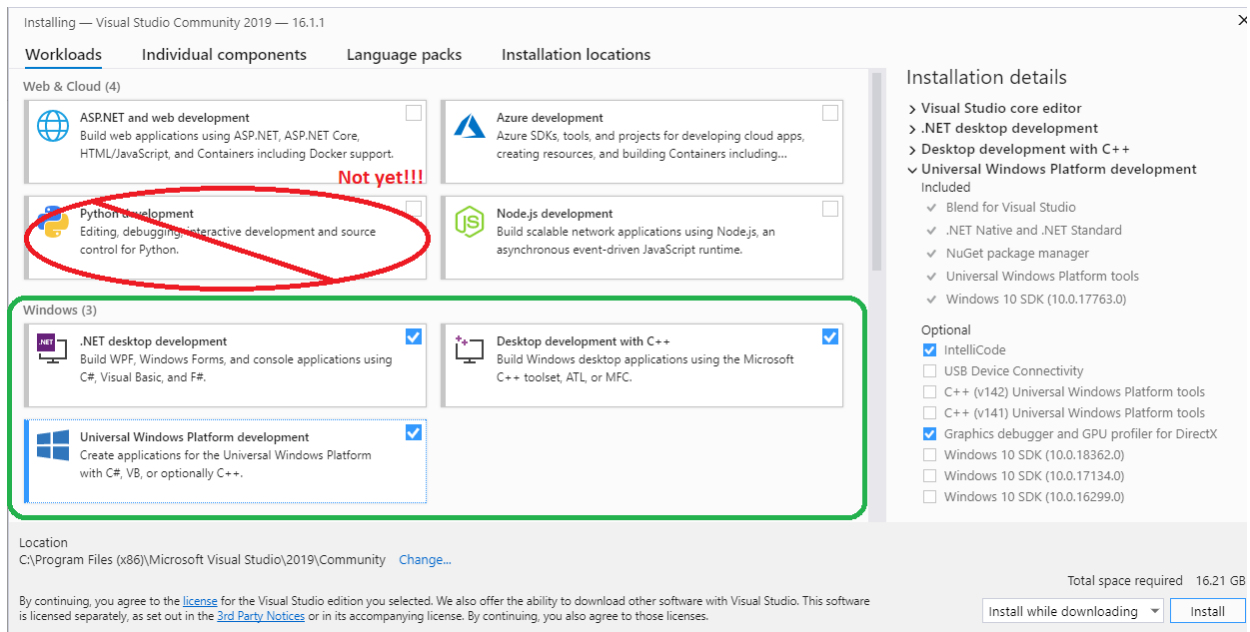
Run the installer and it will perform a quick check of your computer so that it can download the necessary installation items for the setup.



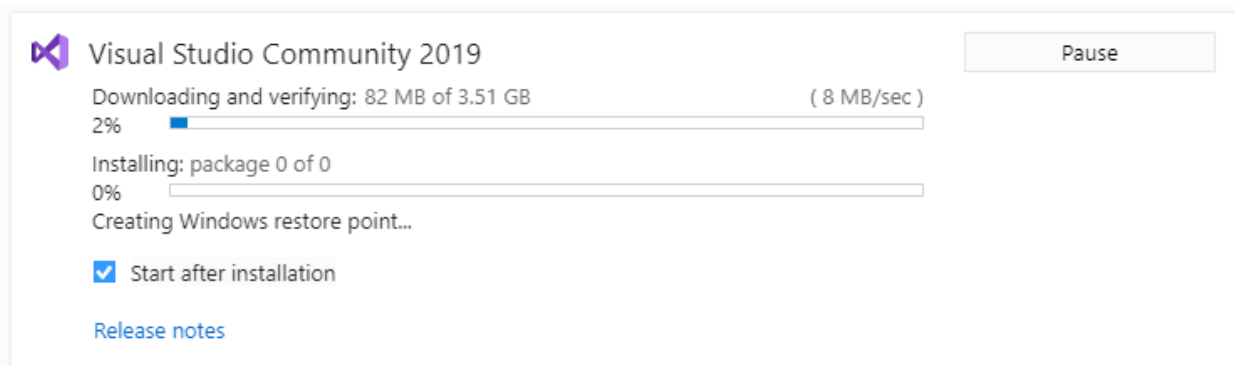
You will then be presented with the following screen which presents you with options for customizing your VS installation. Because we are only concerning ourselves with programming in C#, only choose the following three items for now (outlined in green in the image):

- .NET desktop development
- Desktop development with C++
- Universal Windows Platform development

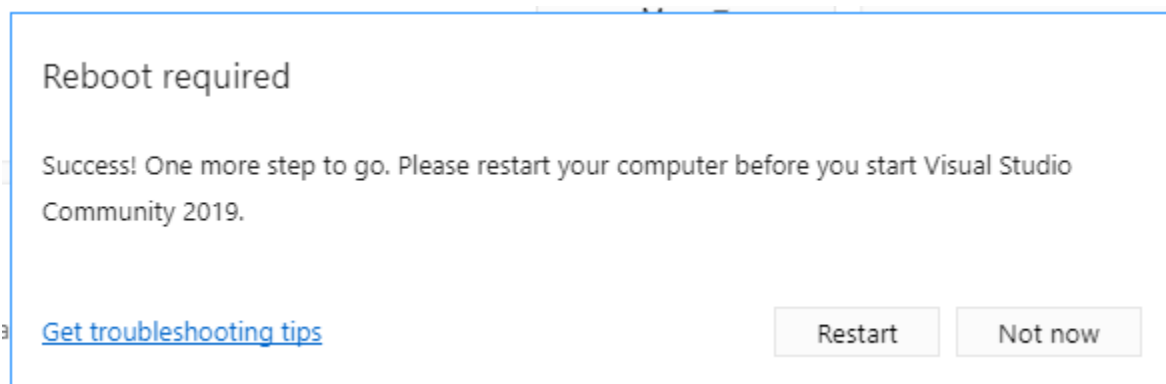
Note that we have specifically deterred you from including the Python development as part of the installation. While Python is a powerful language itself, we avoid adding per this installer so that any existing Python version does not face a secondary installation to contend with. We encourage you to explore this in the future but hold off for the purposes of this exercise.



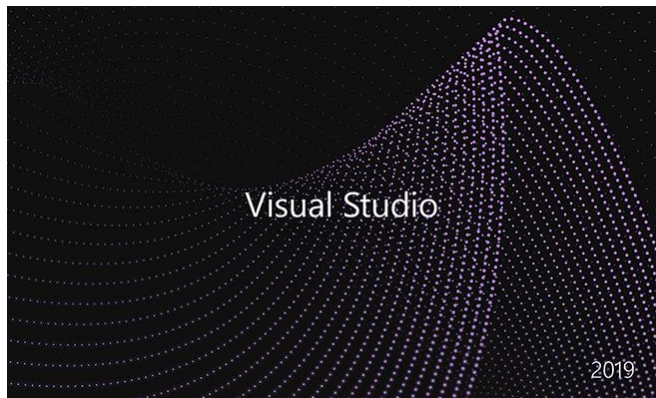
After clicking the Install button, you will see the VS version begin its download, verify, and install of the software tools.



As with most successful Windows application installations, a reboot is required at some point.



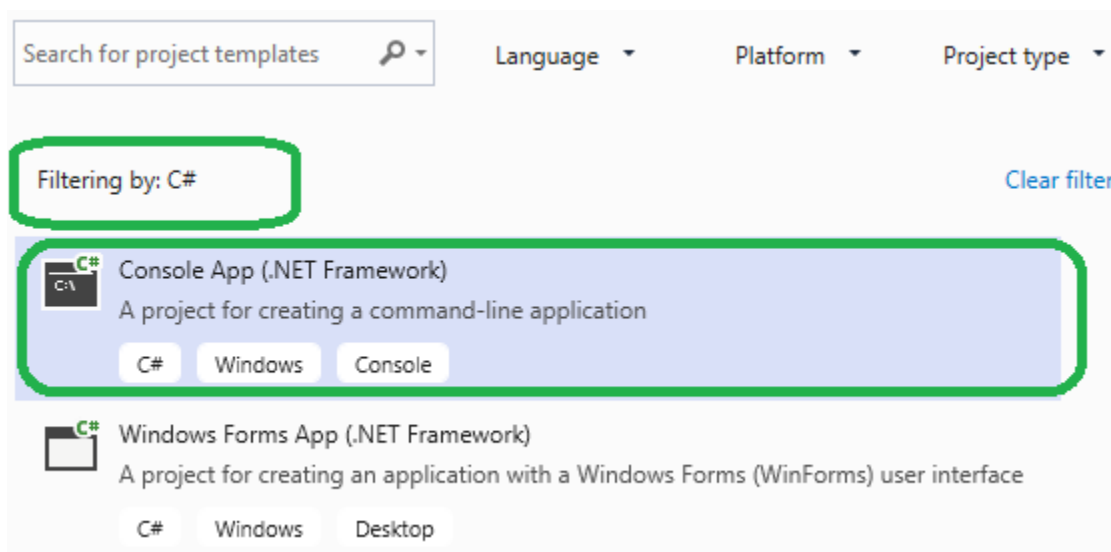
To verify a successful installation, search for Visual Studio 2019 in your list of installed applications and launch.



## Create Your First C# Program

### Create the starter project

After Visual Studio launches, you will locate and click the **Create a new project** button. From there you will need to locate the Console App (.NET Framework) option



Next, name your project using the input field provided. To keep things consistent, enter "My\_First\_C\_Sharp\_Project" as the name then click the **Create** button.

# Configure your new project


Console App (.NET Framework) C# Windows Console

Project name

My\_First\_C\_Sharp\_Project

Location

C:\Users\██████\source\repos

Solution name 

My\_First\_C\_Sharp\_Project

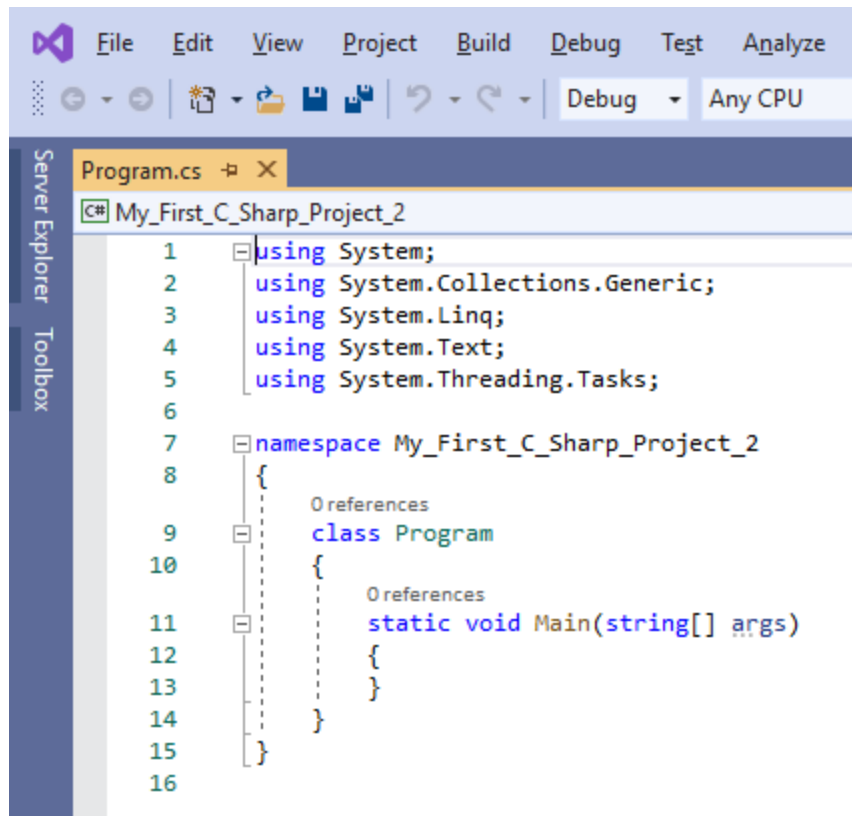
☐

Place solution and project in the same directory

Framework

.NET Framework 4.7.2

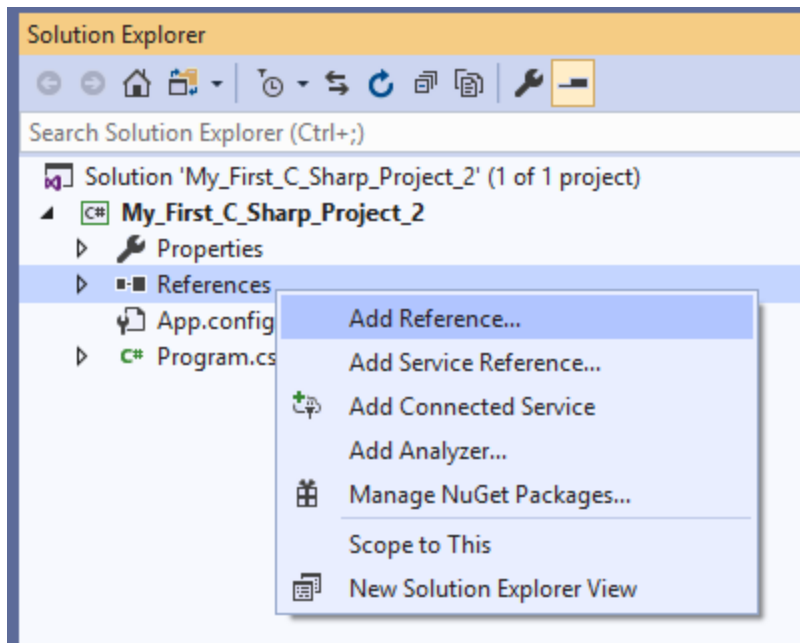
At this point, your project will be created, and you will be presented with the basic setup for your Program.cs file – the main source file for your C# program.



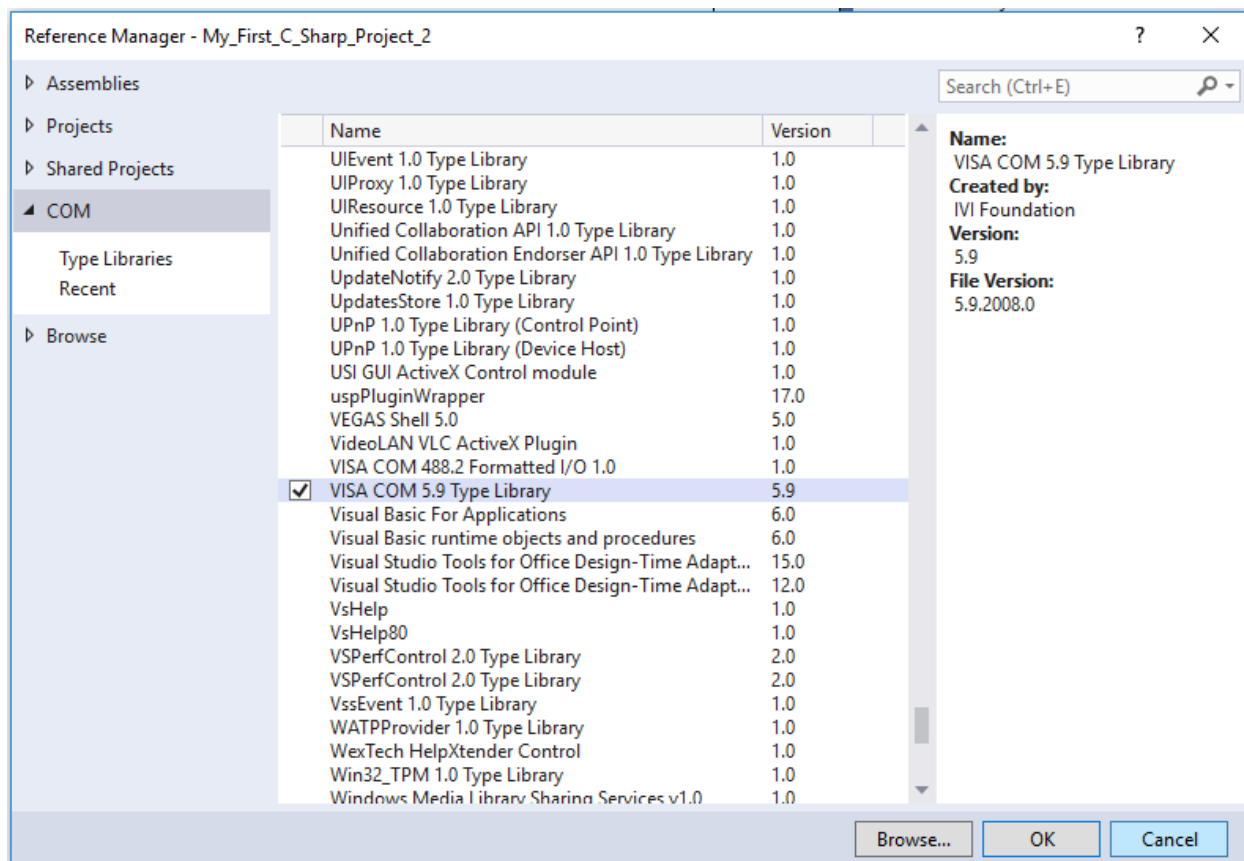
Note that while we see the traditional “Hello World!” tag line as text that will print to the console window, we will be going much further than this.

### Adding the VISA Reference

On the right-hand side of the Visual Studio interface you should see a pane entitled Solution Explorer. From the items displayed, right-click on the **References** text and navigate to **Add Reference**.

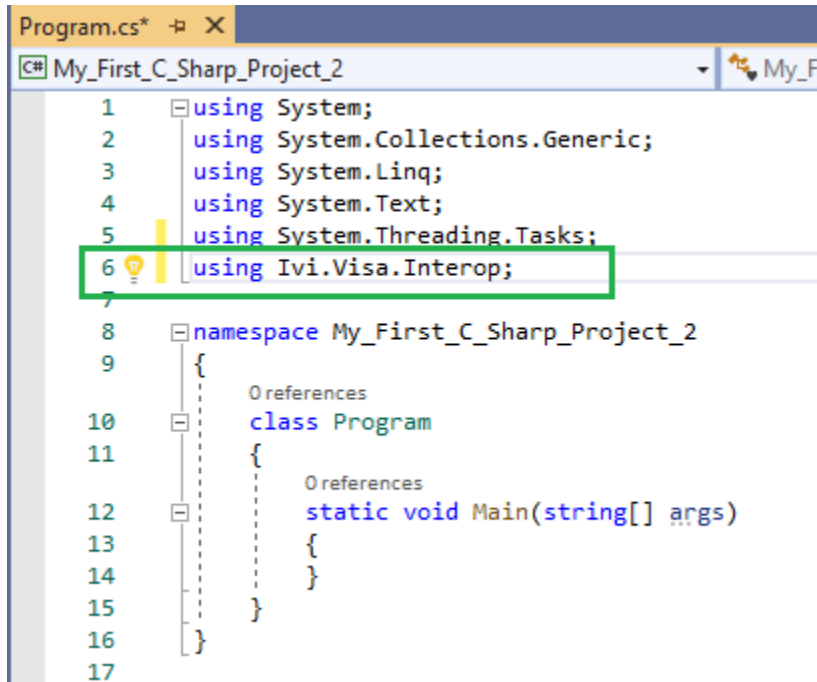


Select the COM category and scroll down the list of names to locate the **VISA COM 5.9 Type Library**; add a check to the box adjacent to it; and click the **OK** button.



Side note, but perhaps your VISA COM Type Library is not 5.9 as seen in this document. This is likely because you have a different VISA version installed than the one we listed above in the Prerequisites section. No matter – if you have the VISA COM Type Library of any version go ahead and add it as it is a safe assumption it will work. While there are likely some changes in the underlying code in the different versions you will most likely find that there is little or no difference in the tools implemented here.

Finally, to use the tools within the VISA COM library, you will need to tell your source file to use it as well. You will do this by entering the text “using Ivi.Visa.Interop;” at the end of the list of other pre-populated reference items in your editor:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using Ivi.Visa.Interop;
7
8 namespace My_First_C_Sharp_Project_2
9 {
10     class Program
11     {
12         static void Main(string[] args)
13         {
14         }
15     }
16 }
17
```

### A Quick Note on the Use of Wrapper Functions

Appendix A is populated with a completed and more elaborate version of the example you are about to read through. One of the things you should take note of are the series of wrapper functions – functions which bundle other functions and variables – used in this example. These have been provided to help bring some additional clarification to the VISA tools they leverage. While the Visual Studio Intellisense feature shares brief pop-up information and suggestions as you type in your commands, the wrapper functions share more verbose information per the comments found within their declarations. Along with sharing details of the function purpose, parameters, return value, and revision details, using the wrappers enables us to customize the use of reference library methods and attributes while also providing additional information to users of all skill levels.

For instance, a common operation used with instrument control is a **query**. Think of this as asking a question: you make a statement in the form of a question and expect an answer. While some VISA API implementations have a query function built in, the VISA COM library used in this Visual Studio example



does not. However, we can (and have) created a query function of our own using the VISA `WriteString()` and `ReadString()` functions:

```
1reference
static string Instrument_Query(FormattedIO488 instrument_control_object, string command)
{
```

We will leave the task of research and investigation of the VISA COM library options and the C# programming language in your hands.

### **Build the Basic Control Tools with Our Template Functions**

Basic control of your instrument can be broken down into three main elements:

- Connecting to your instrument
- Sending and receiving information (settings, readings, etc.)
- Disconnecting from your instrument

The following steps will guide you through each of these.

Start by adding a resource manager object just after the `Main()` declaration:

```
0references
static void Main(string[] args)
{
    /*
     * First step: Open the resource manager and assigns it to an object variable
     */
    ResourceManager resource_manager = new ResourceManager();
}
```

Yes, this does seem like an extra step not listed in the three bullets above. However, the Resource Manager is the layer of VISA responsible for managing connections to instruments and identifying attributes about them in a relatively non-invasive manner. It can look at all resources connected to your computer without disrupting other programs which might be using them. You do, however, need it active prior to taking control of your target instrument.

With the Resource Manager activated – or instantiated – you can now create an object that will be used to connect to and manage the transactions with your instrument. Below you will notice that `my_instrument` is the object which is first created to become your instrument connection. The instrument identification string tells the Resource Manager what communications protocol is being used (GPIB, USB, Ethernet, or RS232) and what address at which to look for and identify the instrument you want to control. We use the Resource Manager and `FormattedIO488 (instrument)` objects along with the instrument identification string to make the connection to the instrument.

```

/*
 * Second step: Create a FormattedIO488 object to represent the instrument
 * you intend to communicate with, and connect to it.
 */
FormattedIO488 my_instrument = new Ivi.Visa.Interop.FormattedIO488();

string instrument_id_string = "GPIB0::18::INSTR";
Int16 timeout = 20000; // define the timeout in terms of milliseconds
// Instrument ID String examples...
// LAN -> TCPIP0::134.63.71.209::inst0::INSTR
// USB -> USB0::0x05E6::0x2450::01419962::INSTR
// GPIB -> GPIB0::16::INSTR
// Serial -> ASRL4::INSTR
Connect_To_Instrument(ref resource_manager, ref my_instrument, instrument_id_string, timeout);

```

After you connect to the instrument you can build your test sequence with your choice of commands written to the equipment and in some cases reading information back from it. As noted earlier, a query is the combination of a write followed by a read. Below we loop a query for the instrument identification string to be reported back to the program so that it can be printed to the program console.

```

/*
 * Third step: Issue commands to the instrument and receive responses to
 * be printed to the program console window.
 */
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(Instrument_Query(my_instrument, "*IDN?"));
}

```

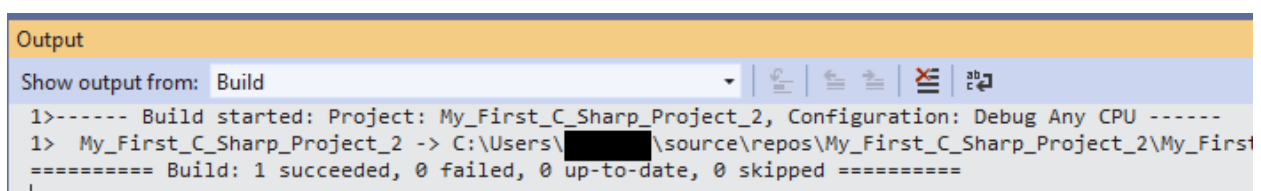
After your test sequence is complete, it is good practice – and sometimes a necessity – to disconnect from the instrument.

```

/*
 * Fourth step: Close the instrument object and release it for use
 * by other programs.
 */
Disconnect_From_Instrument(ref my_instrument);

```

You can then build your program by navigating to **Build->Build Solution** or using the Ctrl+Shift+B hotkey combination.



The screenshot shows the Visual Studio Output window. The 'Show output from:' dropdown is set to 'Build'. The output text reads: '1>----- Build started: Project: My\_First\_C\_Sharp\_Project\_2, Configuration: Debug Any CPU ----- 1> My\_First\_C\_Sharp\_Project\_2 -> C:\Users\████████\source\repos\My\_First\_C\_Sharp\_Project\_2\My\_First\_C\_Sharp\_Project\_2\ 1> ===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ====='.

You can then run your program by navigating to **Debug->Start Debugging** or using the F5 hotkey. When you do, you will see that the instrument you have connected to reports its identification string.

There is plenty more to learn and understand with respect to C# programming and using the Visual Studio IDE (things like setting breakpoints, stepping through and into your code line-by-line, etc.), but we leave that for you to learn on your own. Your challenge – using the tools and information presented here – is to collect the series of commands you will need to control your equipment and successfully implement them. Good luck!

## Conclusion

As noted earlier, we have only covered the very basic necessities of remote instrument programming here. If you have not done so already, we strongly encourage you to locate a C# programmer's reference to have on hand as you continue your journey. Below (in the Appendix) you will find a complete and more elaborate version of the example we just reviewed from which you can copy and paste the core functions and their usage.

## Appendix A – Example Code

The following code is the basic construction that this document walks the reader through. This code (along with numerous other examples) can be found at <https://github.com/tektronix/keithley>.

```
/*=====
Copyright 2019 Tektronix, Inc.
See www.tek.com/sample-license for licensing terms.
=====*/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Ivi.Visa.Interop;
using System.Diagnostics; // needed for stopwatch usage

namespace My_First_C_Sharp_Project_2
{
    class Program
    {
        static Boolean echo_command = true;

        static void Main(string[] args)
        {
            // Create a Stopwatch object and capture the program start time from the system.
            Stopwatch myStpWtch = new Stopwatch();
            myStpWtch.Start();

            /*
             * First step: Open the resource manager and assigns it to an object variable
             */
            ResourceManager resource_manager = new ResourceManager();

            /*
```

```

    * Second step: Create a FormattedIO488 object to represent the instrument
    * you intend to communicate with, and connect to it.
    */
    FormattedIO488 my_instrument = new Ivi.Visa.Interop.FormattedIO488();

    string instrument_id_string = "GPIB0::18::INSTR";
    Int16 timeout = 20000; // define the timeout in terms of milliseconds
    // Instrument ID String examples...
    // LAN -> TCP/IP0::134.63.71.209::inst0::INSTR
    // USB -> USB0::0x05E6::0x2450::01419962::INSTR
    // GPIB -> GPIB0::16::INSTR
    // Serial -> ASRL4::INSTR
    Connect_To_Instrument(ref resource_manager, ref my_instrument, instrument_id_string, timeout);

    /*
    * Third step: Issue commands to the instrument and receive responses to
    * be printed to the program console window.
    */
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(Instrument_Query(my_instrument, "*IDN?"));
    }

    /*
    * Fourth step: Close the instrument object and release it for use
    * by other programs.
    */
    Disconnect_From_Instrument(ref my_instrument);

    // Capture the program stop time from the system.
    myStpWtch.Stop();

    // Get the elapsed time as a TimeSpan value.
    TimeSpan ts = myStpWtch.Elapsed;

    // Format and display the TimeSpan value.
    string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}:{3:00}. {4:000}",
        ts.Days, ts.Hours, ts.Minutes, ts.Seconds,
        ts.Milliseconds / 10);
    Console.WriteLine("RunTime " + elapsedTime);

    // Implement a keypress capture so that the user can see the output of their program.
    Console.WriteLine("Press any key to continue...");
    char k = Console.ReadKey().KeyChar;
}

static void Connect_To_Instrument(ref ResourceManager resource_manager, ref FormattedIO488
instrument_control_object, string instrument_id_string, Int16 timeout)
{
    /*
    * Purpose: Open an instance of an instrument object for remote communication and establish the
    communication attributes.
    *
    * Parameters:
    * resource_manager - The reference to the resource manager object created external to this function. It
    is passed in
    * by reference so that any internal attributes that are updated when using to
    connect to the
    * instrument are updated to the caller.
    *
    * instrument_control_object - The reference to the instrument object created external to this function.
    It is passed
    * in by reference so that it retains all values upon exiting this function,
    making it
    * consumable to all other calling functions.
    *
    * instrument_id_string - The instrument VISA resource string used to identify the equipment at the
    underlying driver
    * level. This string can be obtained per making a call to Find_Resources() VISA
    function and
    * extracted from the reported list.
    */
}

```

```

        *      timeout - This is used to define the duration of wait time that will transpire with respect to VISA
read/query calls
        *
        *      prior to an error being reported.
        *
        *      Returns:
        *          None
        *
        *      Revisions:
        *          2019-06-04      JJB      Initial revision.
        */
instrument_control_object.IO = (IMessage)resource_manager.Open(instrument_id_string, AccessMode.NO_LOCK,
20000);
// Instrument ID String examples...
//      LAN -> TCP/IP0::134.63.71.209::inst0::INSTR
//      USB -> USB0::0x05E6::0x2450::01419962::INSTR
//      GPIB -> GPIB0::16::INSTR
//      Serial -> ASRL4::INSTR
instrument_control_object.IO.Clear();
int myTO = instrument_control_object.IO.Timeout;
instrument_control_object.IO.Timeout = timeout;
myTO = instrument_control_object.IO.Timeout;
instrument_control_object.IO.TerminationCharacterEnabled = true;
instrument_control_object.IO.TerminationCharacter = 0x0A;
return;
}

static void Disconnect_From_Instrument(ref FormattedIO488 instrument_control_object)
{
    /*
    *      Purpose: Closes an instance of and instrument object previously opened for remote communication.
    *
    *      Parameters:
    *          instrument_control_object - The reference to the instrument object created external to this function.
It is passed
    *
    *          in by reference so that it retains all values upon exiting this function,
making it
    *
    *          consumable to all other calling functions.
    *
    *      Returns:
    *          None
    *
    *      Revisions:
    *          2019-06-04      JJB      Initial revision.
    */
instrument_control_object.IO.Close();
return;
}

static void Instrument_Write(FormattedIO488 instrument_control_object, string command)
{
    /*
    *      Purpose: Used to send commands to the instrument.
    *
    *      Parameters:
    *          instrument_control_object - The reference to the instrument object created external to this function.
It is passed
    *
    *          in by reference so that it retains all values upon exiting this function,
making it
    *
    *          consumable to all other calling functions.
    *
    *          command - The command string issued to the instrument in order to perform an action.
    *
    *      Returns:
    *          None
    *
    *      Revisions:
    *          2019-06-04      JJB      Initial revision.
    */
if (echo_command == true)
{
    Console.WriteLine("{0}", command);
}
instrument_control_object.WriteString(command + "\n");

```

```

        return;
    }

    static string Instrument_Read(FormattedIO488 instrument_control_object)
    {
        /*
         * Purpose: Used to read commands from the instrument.
         *
         * Parameters:
         *     instrument_control_object - The reference to the instrument object created external to this function.
It is passed
making it
         *                                     in by reference so that it retains all values upon exiting this function,
         *                                     consumable to all other calling functions.
         *
         * Returns:
         *     The string obtained from the instrument.
         *
         * Revisions:
         *     2019-06-04      JJB      Initial revision.
         */
        return instrument_control_object.ReadString();
    }

    static string Instrument_Query(FormattedIO488 instrument_control_object, string command)
    {
        /*
         * Purpose: Used to send commands to the instrument and obtain an information string from the instrument.
         *     Note that the information received will depend on the command sent and will be in string
         *     format.
         *
         * Parameters:
         *     instrument_control_object - The reference to the instrument object created external to this function.
It is passed
making it
         *                                     in by reference so that it retains all values upon exiting this function,
         *                                     consumable to all other calling functions.
         *
         *     command - The command string issued to the instrument in order to perform an action.
         *
         * Returns:
         *     The string obtained from the instrument.
         *
         * Revisions:
         *     2019-06-04      JJB      Initial revision.
         */
        Instrument_Write(instrument_control_object, command);
        return Instrument_Read(instrument_control_object);
    }
}

```

## References

TBD

## Revisions

Revision	Date	Authored by	Notes
1.0	2019-06-03	Josh Brown	Hey, you gotta start somewhere
