# 03 Working with SIMPL# Series

This document assumes the user is familiar with SIMPL Windows and SIMPL+ programming.

VisualStudio 2008 SP1 is required with Crestron SIMPLSharp plugin.

## Basic TCP/IP Connection

Continuing from the code written in documents 01 and 02 SIMPL Sharp Libraries we will be covering the topic of creating a TCP/IP client connection (direct socket) from a SIMPL# Library to a TCP/IP Server.

## Why use this method?

The TCP/IP symbol in SIMPL will complete the same function as what we are about to demonstrate with a few key restrictions: the port number and the IP Address (target server) cannot be modified on the fly, the returned string is restricted to 255 characters and the TCP/IP symbol will reserve an IPID within the IP Table while direct sockets do not.

Wait a minute; SIMPL+ has the ability to perform direct socket connections that will not take up an IPID, doesn’t have the character restriction and you can modify the port number and IP Address on the fly…so why do this in SIMPL#?

SIMPL# – use a chainsaw to cut down a tree.

SIMPL+ – use a hacksaw to cut down a tree.

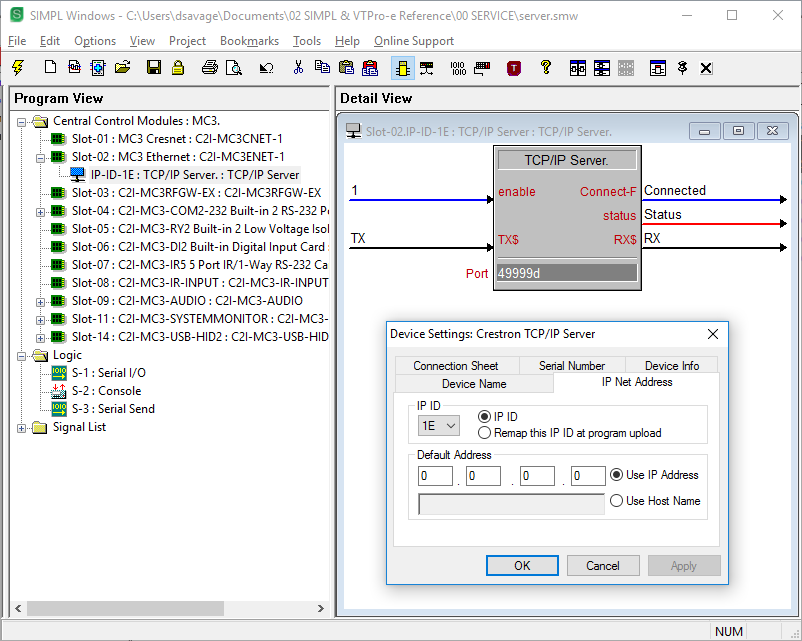
SIMPL – use your teeth.

Although SIMPL+ would work for most things, there is much more power, additional tools and functions within SIMPL# that we can use down the road (encryption for secure connections, JSON parsing, etc). We need to get a basic connection working first before we can start using the more advanced features.

## Preparing our Server program to test the Client connection

I am working on a 3-series processor with a 2nd 3-series processor running a program with a TCP/IP Server symbol to accept my incoming connection request (I tried this across multiple program slots on the same processor with no luck…that’s what EICs are for).

The Server symbol requires an IP address to be configured (right-click on the symbol). I am using 0.0.0.0 as this means that connections from ANY IP Address will be accepted. The port has been set as 49999 to reside in the dynamic range (anyone can use these numbers and there is no overlap with an existing service).

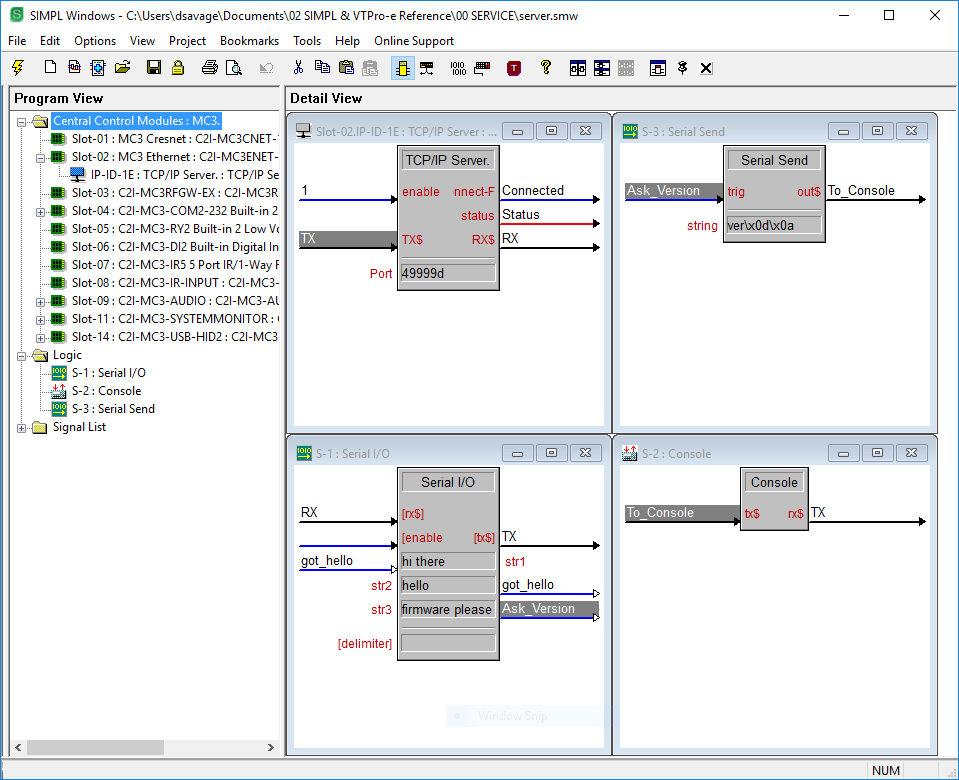


There is also some code to deal with data into the server and back from the server.

If the server receives a “hello” from the client it will respond with a “hi there”.

If the server receives a “firmware please” from the client it will send the “ver\x0d\x0a” command to the CONSOLE of the processor and send the response out through the server back to the client.

The incoming message of “firmware please” was not sent directly to the CONSOLE as I wanted to validate the information the client was asking for first. The SIO waits for an exact match before trigging the SEND to transmit the command.



Save, Compile and Load this program to ‘server’ processor.

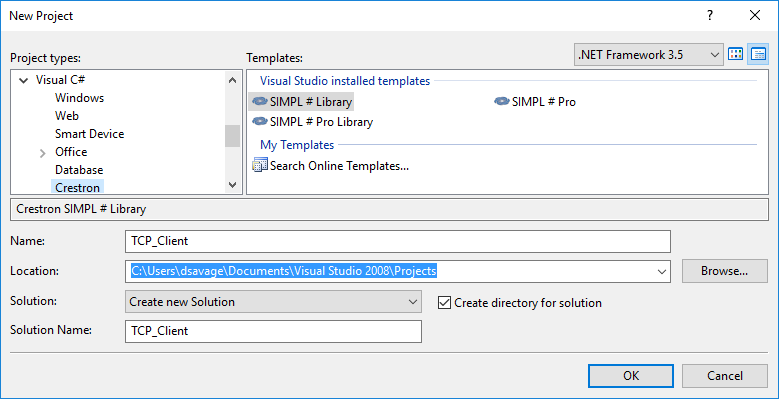
## Preparing the Client programs

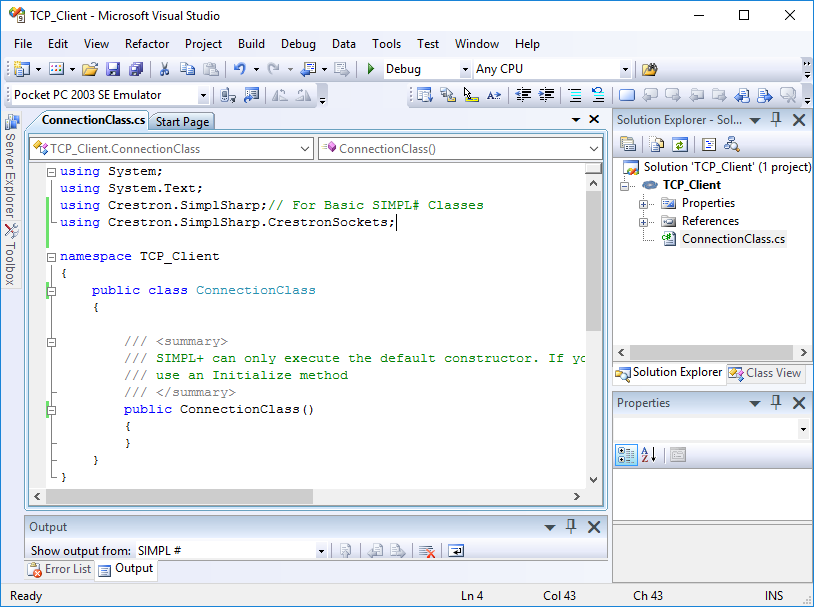
Since we are working with a SIMPL# Library, there will be three programs we will need;

1. SIMPL for the user input and to print the result to a touchpanel (or debugger)
2. SIMPL+ module to bridge between SIMPL and SIMPL#
3. SIMPL# to complete the TCP/IP connection and return data

### SIMPL#

Create a new SIMPL# Library file;

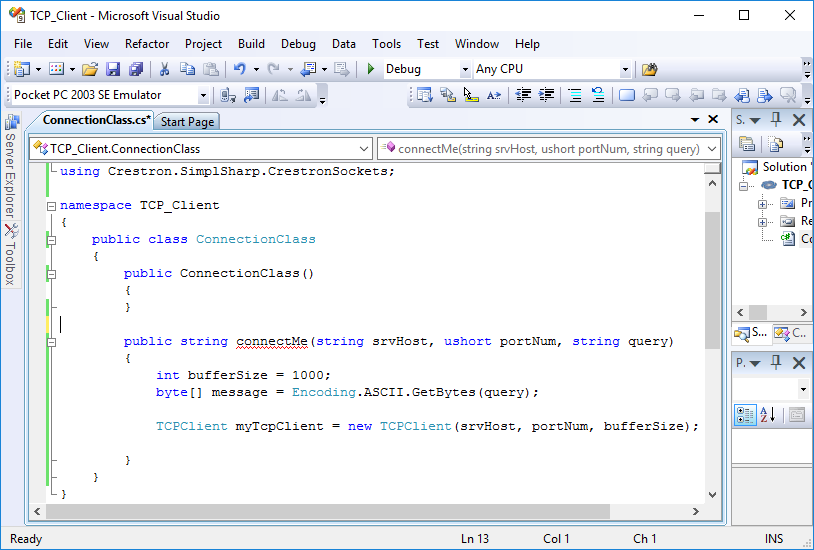




Changes made from the file created; adding of the **‘using Crestron.SimplSharp.CrestronSockets;’**, renaming of the public class to **‘public class ConnectionClass’**, constructor to **‘public ConnectionClass()’** and the .cs file to **‘ConnectionClass.cs’**.

**Crestron.SimplSharp.CrestronSockets** – This is the namespace that has the functions we are allowed to use for direct sockets. It is different from the System.Net.Sockets namespace that also has similar functions, but is not allowed in the Crestron C# Sandbox. You only find this out when you compile your Visual Studio project…guess how I found out…

Create the public SIMPL# method that will be called from SIMPL+.



**‘public string connectMe(string srvHost, ushort portNum, string query)’** – defining parameters in your public method allows you to pass in custom values from SIMPL+ when you call the method. In this case, the server IP Address (**srvHost**), the port number (**portNum**), and the string you want to send to the server (**query**).

**‘int bufferSize = 1000;’** – this is the size (in bytes) of the bucket you are using to collect the data coming back from the server. An IP connection is bi-directional so you have to define how much data you are able to take back in at once.

**‘byte[] message’** – like the datatype ‘**int**’ the ‘**byte[]**’ declaration creates a bucket that stores the datatype of ‘**byte**’. Using the ‘**[]**’ after the datatype makes the bucket (variable) called ‘**message**’ able to store an array of values of the type ‘**byte**’.

An array is an ordered list of items. If an array is declared like this in SIMPL#; ‘**int[] myArray = new int[5];**’ then the variable ‘**myArray**’ can now hold 5 different values of integers. If I want to know the value stored in one of the 5 locations in the array I can ask for it like this in SIMPL#;

**Int x = myArray[3];**

*\*\*\*Here is the tricky part…arrays don’t count starting at 1. They start counting at 0. So in my 5-value array,* ***myArray[3]*** *is actually the 4th value in the list.*

In SIMPL+ you absolutely need to say how big your arrays are when you declare them. Seeding values into your array is at least a 2 step process as you can’t declare and assign at the same time (this has to do with SIMPL+ needing to know how much memory it needs at compile time to reserve for the program to run);

**Integer myArray[10];**

**Integer x;**

**For(x = 0 to 9) // 0 to 9 is 10 steps!!**

**{**

**myArray[x] = *<some value you set>;***

**}**

SIMPL# does not need to know how much memory to reserve for most variables and arrays and has the ability to dynamically allocate memory (while the program is running on the processor!).

**‘byte[] message = Encoding.ASCII.GetBytes(query);’** – taking the whole line into context now…each time the method ‘**connectMe**’ is called from SIMPL+, a new SIMPL# array (‘**message**’) will be created with a size equal to the number of bytes in the string ‘**query**’. If ‘**query**’ is set to “**hello**”, the array ‘**message**’ be 5 values long.

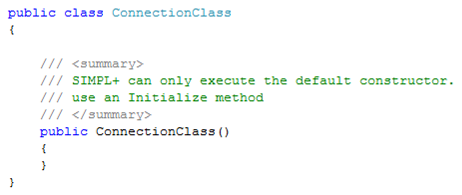
Why are we turning a string into an array of bytes? This will be covered shortly.

**‘TCPClient myTcpClient = new TCPClient(srvHost, portNum, bufferSize);’** – here we are creating/instantiating an object of the class TCPClient. TCPClient is a class inside the CrestronSockets namespace that was added earlier. Once we have this object, we can access methods from within the TCPClient class.

When we create an object of a class we run what is known as a Constructor. A Constructor allows you add in options or add things in from the beginning when you create your object of a class.

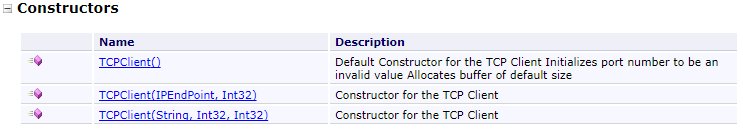
The class we are creating ‘**ConnectionClass**’ has what is known as a Default Constructor. This is a public function that has nothing in it to begin with, but must always be present.

You may have noticed ‘**public ConnectionClass()**’ when you created your SIMPL# Library file;

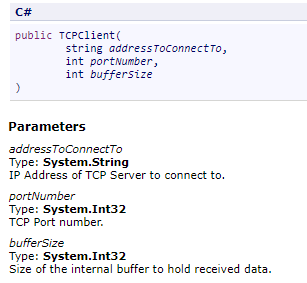


When you create your class object in SIMPL+, this method is run. As mentioned in the text above, SIMPL+ will only run the default constructor. SIMPL# however has the ability to run a class’ other Constructors (a class can have more than one type of Constructor defined in it…this is called Overloading the Constructor).

In the case of the SIMPL# ‘**TCPClient**’ class it has 3 Constructors to choose from;



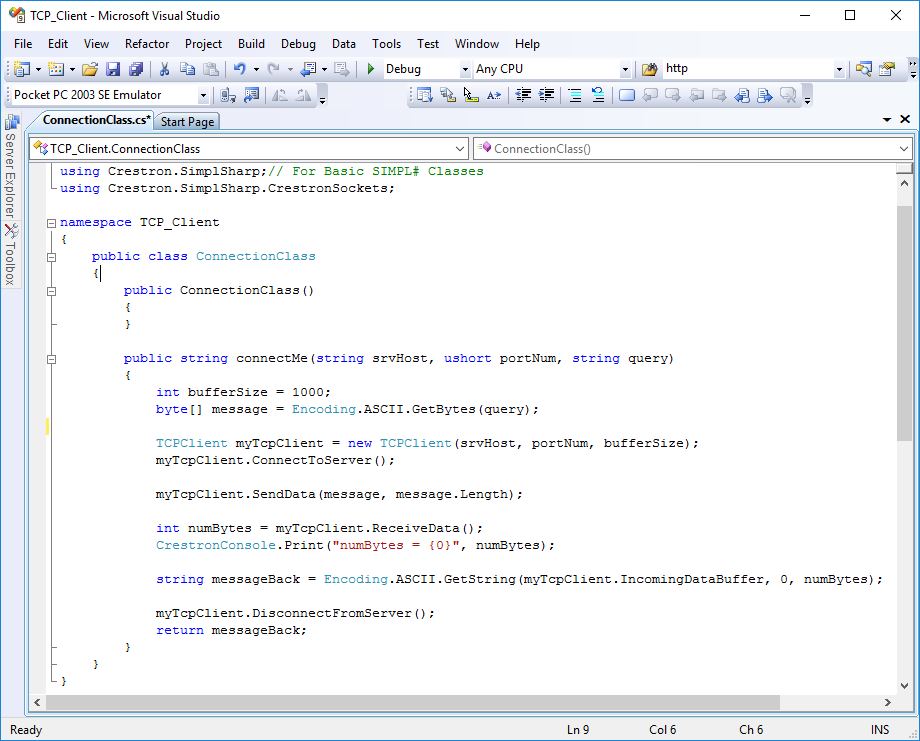
Choosing the 3rd item for more details;



*(screenshots from reference.crestron.com)*

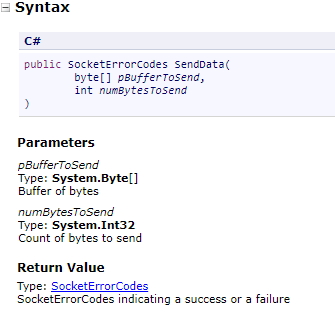
We choose the 3rd one so we can feed in these parameters as soon as the object is created.

Now we will connect to the server, send the data, handle the response, disconnect and return data to the SIMPL+ function that called this SIMPL# method.



**‘myTcpClient.ConnectToServer();’** – our object of the TCPClient class now has access to some methods! ‘**ConnectToServer()**’ is the method that will take our server IP Address and port number to initiate the 3-way handshake that happens during a TCP/IP connection.

**‘myTcpClient.SendData(message, message.Length)’** – the SendData method in the TCPClient class requires 2 parameters (arguments) to be passed in;



*(screenshots from reference.crestron.com)*

The first argument need to be of the ‘**byte[]**’ array datatype. We created this when we declared the variable ‘**message**’.

‘**message.Length**‘ – the second argument is how many bytes of the array we mentioned in the first argument that we actually want to send to the server.

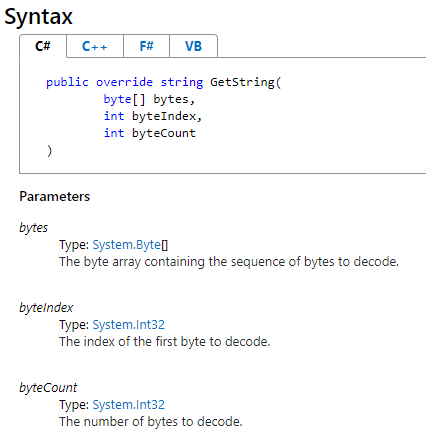
‘**.Length**’ – if you go back to **01 SIMPL Sharp Libraries – Introduction** step 5, I explained how the datatype ‘**int**’ is a class that contains its own methods. Classes can also contain properties that describe information about themselves. If you go back to **02 SIMPL Sharp Libraries – Delegates and Events** on page 3 I described a property of a delegate. This property contained two operators within it called ‘**get**’ and ‘**set**’. If you need to change the information about a property you use the ‘**set**’ operator, and if you want to read the property information you use the ‘**get**’ operator. Using the statement ‘**.Length**’ is a way of reading (**get**ting) the length property of the byte array ‘**message**’.

This is the equivalent in SIMPL+ to the following; ‘**integer stringLength; stringLength = len(“my string message”);**’.

**‘int numBytes = myTcpClient.ReceiveData();’** – the ‘**ReceiveData()**’ method returns the number of bytes that are currently sitting in the incoming buffer of the TCP/IP connection. This should be the entire message received back from the server. We need this value as an integer to know how far into the incoming buffer to read (how full is the bucket).

**‘CrestronConsole.Print(“numBytes = {0}”, numBytes);’** – in order to debug your SIMPL# code you will use a similar method as SIMPL+ using Trace or Print statements. This statement will print out a custom made string in the console of the processor (also in SIMPL debugger). Instead of using %s or %i to denote the datatype of what you are inserting into your string, SIMPL# uses {0}, {1}, {2}…etc to denote which of the comma-separated arguments listed will get printed in the string. This phrase prints out the number of bytes received from the server.

**‘string messageBack = Encoding.ASCII.GetString(myTcpClient.IncomingDataBuffer, 0, numBytes);’** – the data we got back from the server is stored in a property of our ‘**myTcpClient**’ object. Here we are **get**ting the bytes that are sitting in the ‘**IncomingDataBuffer**’ property (response from the server is stored in a byte array!), starting from the first byte in the array (0 is the first value in an array) and ending after the correct number of bytes (‘**numBytes**’). We then convert the bytes (‘**Encode.ASCII.GetString**’) into the string format.



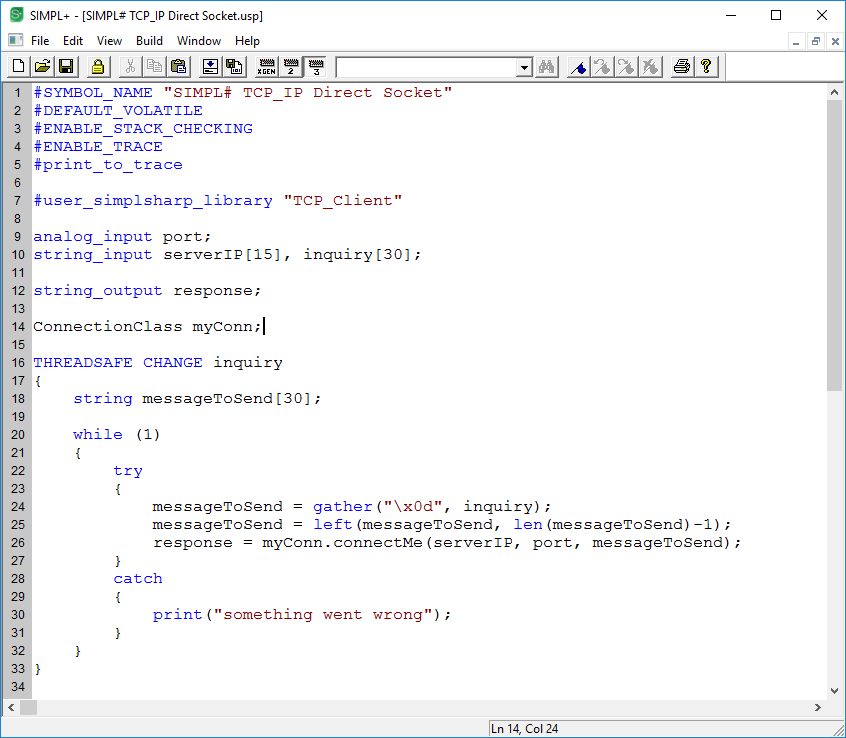
*(screen shot from Microsoft - https://msdn.microsoft.com/en-us/library/38b953c8(v=vs.110).aspx)*

**‘myTcpClient.DisconnectFromServer();’** – close the connection when you are done with it.

**‘return messageBack’** – as our method ‘**connectMe**’ is a string method, we need to return string data to whatever function in SIMPL+ called this SIMPL# method.

### SIMPL+

Create the logic to use the SIMPL# library;

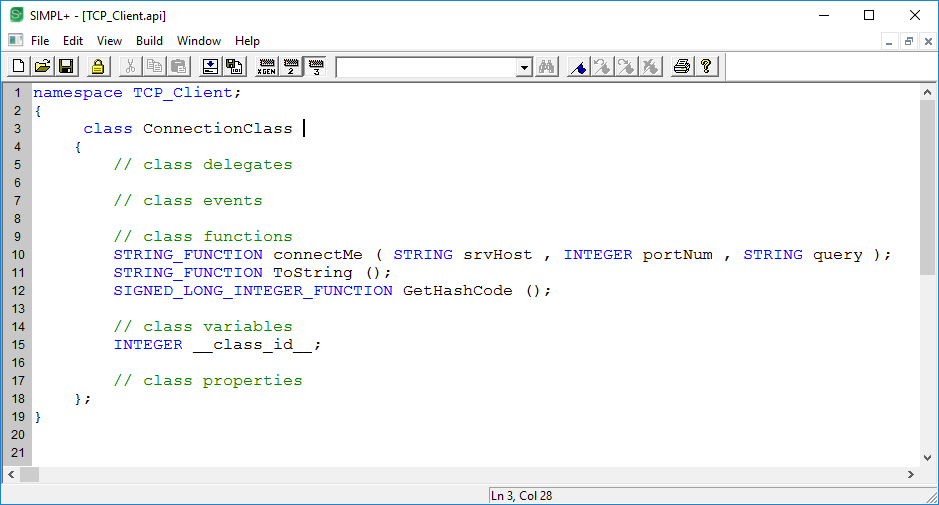


**‘#user\_simplsharp\_library “TCP\_Client”’** – this points the SIMPL+ program to the SIMPL# library (all files in the project are currently saved in the same folder).

**‘ConnectionClass myConn;’** – creating and instance (or object) of the SIMPL# class so we can access its methods from SIMPL+.

**‘THREADSAFE CHANGE inquiry’** – each time a new string message is sent to this module, I want to open the SIMPL# socket, send the inquiry, close the socket and send the response to an output. This happens very quickly and we will be able to see the status of the socket connection changing in the server program.

How to we start this socket connection from SIMPL+? We need to call the SIMPL+ method that handles this for us. If you open the API for TCP\_Client you will find the following;



**‘STRING\_FUNCTION connectMe( STRING srvHost , Integer portNum , STRING query );’** – this is the method we need to execute. There are 3 arguments/parameters that need to be fed in from SIMPL+ in order for it to work; an IP address, a port number and the message we would like to send.

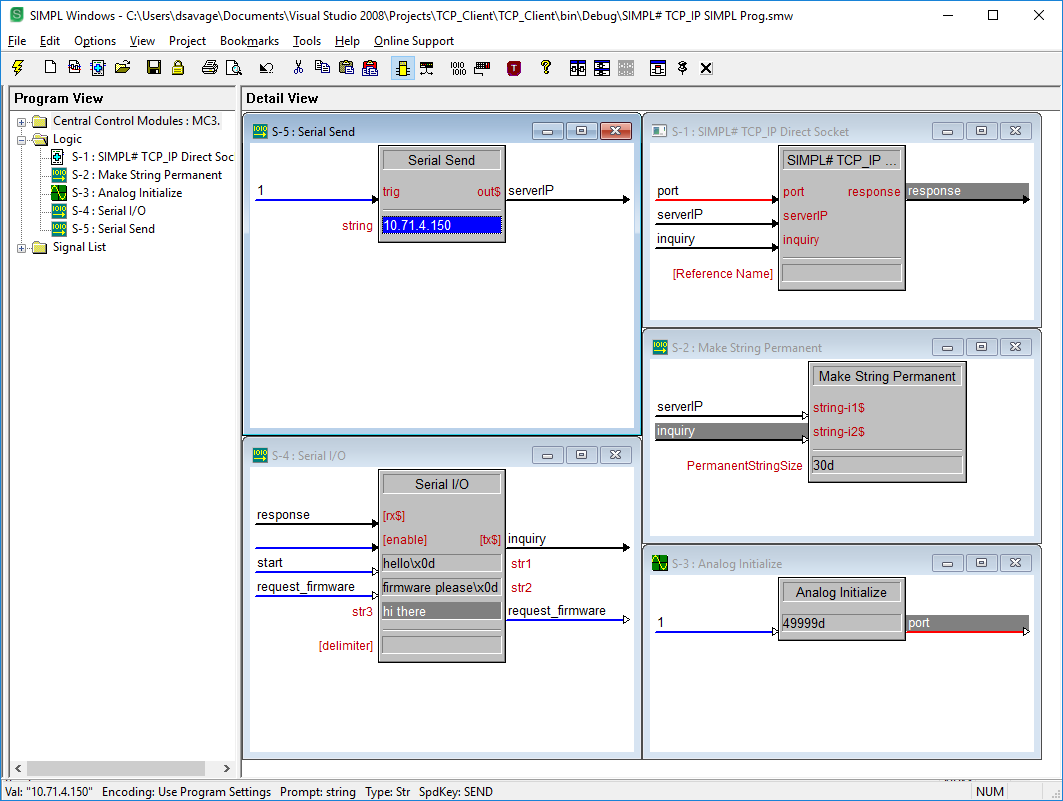
Back in SIMPL+;

**‘response = myConn.connectMe(serverIP, port, messageToSend);’** – we first do some parsing work on the incoming inquiry to remove the “**\x0d**”, then we use this line to submit the result as part of the SIMPL# method call. ‘**serverIP**’ and ‘**port**’ are supplied from the SIMPL program to the SIMPL+ module. Assigning this method call to a SIMPL+ module output ‘**response**’ allows us to send the message and pass the result of a SIMPL# connection back up to SIMPL in one line of code.

**‘try {} catch {}’** – so far none of the code we have looked as has included any way to handle a failure in the code. Without exception handling, if a piece of code gets a result it doesn’t know what to do with it can either perform unexpectedly or stop functioning. Best case it will post an error in the logs and the program will keep running. Worst case, the processor will ‘lock up’. In SIMPL+, the try/catch block allows us to attempt to run a piece of code and if it doesn’t work…do something else! In this case it will merely print a message to the console that “**something went wrong**” and keep running. Once we have a few connection methods squared away we will go further into what to do if a connection doesn’t work.

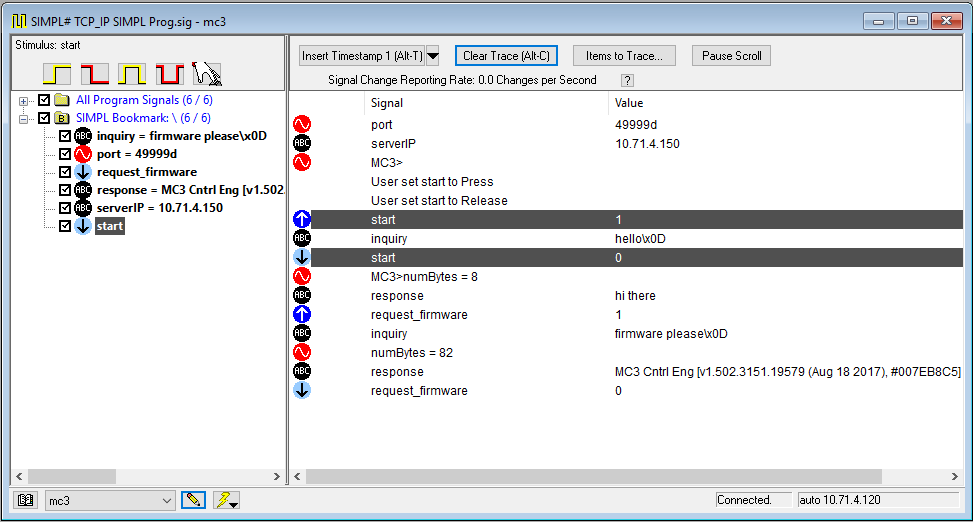
### SIMPL

Create the logic to use the SIMPL+ Module;

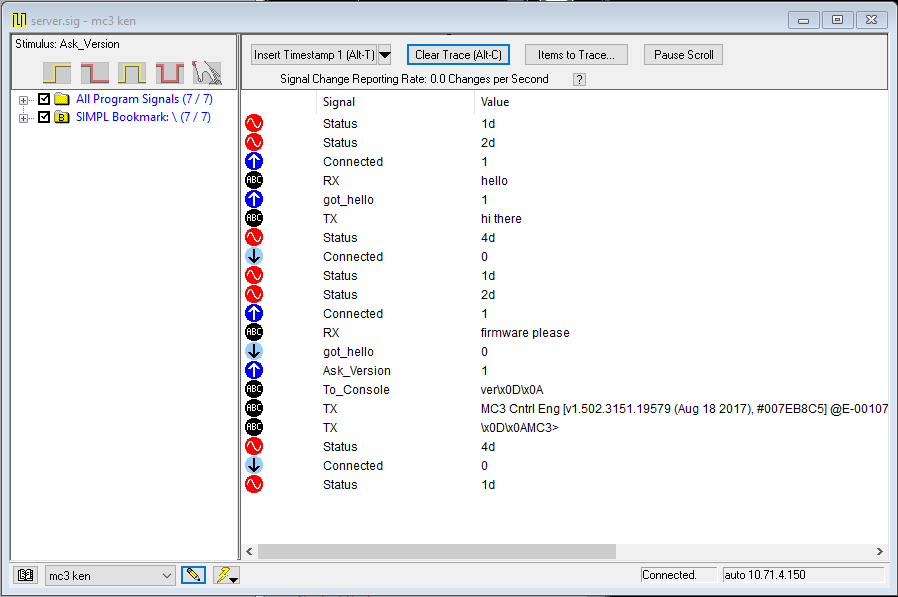


The program is triggered in debugger by setting the ‘start’ digital signal high. If there are no issues with the connection to the 2nd processor, the resulting debugger traces will show the following;

Client program;



Server program;



# Appendix A

## SIMPL# CODE

using System;

using System.Text;

using Crestron.SimplSharp;// For Basic SIMPL# Classes

using Crestron.SimplSharp.CrestronSockets;

namespace TCP\_Client

{

public class ConnectionClass

{

public ConnectionClass()

{

}

public string connectMe(string srvHost, ushort portNum, string query)

{

int bufferSize = 1000;

byte[] message = Encoding.ASCII.GetBytes(query);

TCPClient myTcpClient = new TCPClient(srvHost, portNum, bufferSize);

myTcpClient.ConnectToServer();

myTcpClient.SendData(message, message.Length);

int numBytes = myTcpClient.ReceiveData();

CrestronConsole.Print("numBytes = {0}", numBytes);

string messageBack = Encoding.ASCII.GetString(myTcpClient.IncomingDataBuffer, 0, numBytes);

myTcpClient.DisconnectFromServer();

return messageBack;

}

}

}

## SIMPL+ CODE

#SYMBOL\_NAME "SIMPL# TCP\_IP Direct Socket"

#DEFAULT\_VOLATILE

#ENABLE\_STACK\_CHECKING

#ENABLE\_TRACE

#print\_to\_trace

#user\_simplsharp\_library "TCP\_Client"

analog\_input port;

string\_input serverIP[15], inquiry[30];

string\_output response;

ConnectionClass myConn;

THREADSAFE CHANGE inquiry

{

string messageToSend[30];

while (1)

{

try

{

messageToSend = gather("\x0d", inquiry);

messageToSend = left(messageToSend, len(messageToSend)-1);

response = myConn.connectMe(serverIP, port, messageToSend);

}

catch

{

print("something went wrong");

}

}

}