**Preface**

This tutorial assumes some background knowledge of programming, but no prior experience with Raspbian (or any Linux distribution), GPIO microprocessors (Pi, Arduino, etc), or Python. References may be made to previous tutorials.

My main programming languages are Java, C#.NET, and VB.NET. You may notice some slight differences between my Python code and the standard conventions and formatting for Python. This should not affect your understanding of the code.

**Goal**

Prepare the Raspberry Pi 3 to upload a larger set of data, and set up a .NET application to save all of the data into a CSV file.

**Getting Started**

It’s nice to collect a lot of data on your Pi that you may need later, but what’s the point if you can’t transmit them in a timely and efficient manner? Let’s load up the module you’ve written in Python from the first two tutorials.

First, let’s do a little bit of cleanup. We’ve got a main method, but we don’t really use it to do anything except to instantiate our client. Immediately following the main() call, we have a While loop that seems to change every time we need to do something different. Time to reorganize!

The following code I’ll leave to you to decide how exactly you want to implement, so I won’t be providing a strict copy of what your code should be. However, here’s the structure I recommend following:

def sendButtonState():

global ...

# Collect state and send as key/value pair

def main():

global ..., sendButtonState

# Instantiation code...

while True:

time.sleep(15) # 15 second intervals

sendButtonState()

...

# Begin code execution

...

main()

# END OF MODULE

By creating a new method for checking button state, we can easily set our main() to do whatever we want simply by adding and removing method calls from within the While loop.

**Sending More Data**

Now that our code looks cleaner, we have another method to write. Our current key/value transmitting method only accepts a single key and a single value, which is wasteful if we need to send multiple pairs of data. Consider if you had three sensors that collected different types of data. Rather than turning everything into one long, specially formatted string that you’d need to parse later, it would be much more efficient to save this as a dictionary:

“Temperature:74.4248422;Humidity:32.661892;Pressure:101.3088224”  
vs  
Index 0: Temperature = 74.4248422  
Index 1: Humidity = 32.661892  
Index 2: Pressure = 101.3088224

Not only is this more readable, it also is more readily accessible and doesn’t require a parsing protocol. You simply need to retrieve the data, convert the string into a number (float or double in this case), and your values are ready to go.

It is worth noting that in our current implementation, perhaps it would be easier to compress the data on the Pi’s end into a CSV-acceptable format, and then save the received string directly to file. However, this means we’re not expecting to view or manipulate this data from within the application, and instead we will be importing the data into another program like Excel or a database for more work to be done.

In the case that you want to display real-time data as you get it, it would be cleaner and more versatile to use the key/value pair option that the IoT Hub client provides. So, let’s add some code to take advantage of this.

Again, we’re focusing on ease of use and not a quick-and-dirty implementation here – so we’ll have to create a new class as our message builder.

class DataBuilder:

m = 0

ref\_CLIENT = 0

def \_\_init\_\_(this, ref\_C):

this.ref\_CLIENT = ref\_C

this.m = IoTHubMessage(“DataBuilder message”)

def send(this):

this.ref\_CLIENT.send\_event\_async(

this.m, send\_confirm\_callback, “builder”)

def add(this, key, value):

this.m.properties().add(key, value)

**Notes:**

* *Don’t copy-paste code! Formatting differences may lead to unintended behavior.*
* The constructor method for Python classes is \_\_init\_\_, with two underscores per side.
* All class method arguments must start with a ‘self’ or ‘this’ reference. However, when passing arguments to the method, you skip this extra argument.
* All class variables must be accessed via the aforementioned ‘self’/’this’ reference.
* In send(this), “send\_confirm\_callback” is a global method, so no reference is needed.

How do we use this class? We’ll instantiate one each time we’re ready to send a message, collect all the data points from our various sensors, and then allow it to transmit the final message. The Python garbage collection will automatically take care of disposing the unneeded object afterwards.

def main():

...

while True:

time.sleep(15)

# Instantiate a builder, passing our CLIENT as arg

msgBuilder = DataBuilder(CLIENT)

# Add keys and values

msgBuilder.add(“buttonState”, str(readSwitch()))

# All done? Upload

msgBuilder.send()

readSwitch() actually returns an enumeration, which can be equivalent to GPIO.LOW or GPIO.HIGH; the actual value returned is 0 or 1, respectively. Since Python doesn’t have implicit int->string conversion, we need to force it a little bit here. Alternatively, you may choose to move the str() function into DataBuilder.add(), which would prevent this error from occurring at any point.

Beware that we’ve removed the sendButtonState() method call here, since we’re trying to reduce the number of message sent per interval. Remember that we do have a message limit, so sending more than one message per interval will cause us to exceed that limit. We can keep the method declaration, however, in case we ever need to trigger it manually.

Run your console application from *Tutorial 2*, and begin connecting to your IoT Hub. Wait for any additional data to be downloaded, if your Python code was running when the console wasn’t.

Once that’s ready, run your Python code. You should be able to see updates on your console application, stating that the data says “DataBuilder message”, with one key/value pair, which should say “buttonState” and either “0” or “1”.

If you want to, you can modify the msgBuilder.add() call to send “LOW”/“HIGH” rather than “0”/“1”.

**Creating a CSV Protocol**

Now that our DataBuilder class works and is capable of sending multiple messages, let’s start saving this data. Stop your Python module and the console application.

This tutorial will be taking you through the creation of a relatively flexible CSV-file creator. This will be able to run without needing to create a new file each time, appending new data to a single file. It will also be able to accept new data labels as well as missed labels. If you don’t think this is necessary, there are many sources online that describe the basic way to generate formatted CSV files.

In order to maintain the proper sequencing of labels, we need to keep a list of all known labels and use this list to compile our newly received data. Let’s go edit our code.

Recall that I use VB.NET for this tutorial; C# will be similar.

**Note:** For this part, the Java code structure will be pretty similar, if you prefer Java.

Let’s remove the current WriteLine statement from within the nested For loop, and make a call towards a currently non-existent method within SimpleEventProcessor called “NewDatapoint”.

Function ProcessEventsAsync(...) As Task Implements ...

For Each eventData In messages

...

For Each piece in eventData.Properties

NewDatapoint(piece.Key, piece.Value)

Next

Next

...

End Function

The idea is to allow the SimpleEventProcessor to detect missing labels and accordingly adjust the CSV output. The NewDatapoint() method will need to help do this. It will require a few variables, however:

Private KnownLabels As New List(Of String)

Private InsertData As New List(Of String)(0)

Function ProcessEventsAsync(...) As Task Implements ...

...

End Function

First, we search if the label is previously known (and thus has a pre-determined index). Then we add the new value to a list for compilation later.

Sub NewDatapoint(key As String, value As String)

If KnownLabels.Contains(key) Then

Dim index = KnownLabels.IndexOf(key)

InsertData.Insert(index, value)

Else

KnownLabels.Add(key)

InsertData.Add(value)

End If

End Sub

**Tip: ‘**Sub’ is equivalent to C#’s void; it does not return a value.

You may notice that this has cases where exceptions may be thrown at InsertData.Insert() – don’t worry, we’ll handle that now:

Sub RefreshDataList()

InsertData = New List(Of String)(KnownLabels.Count)

End Sub

This way, InsertData always is able to contain at least the number of known labels. If additional labels appear, NewDatapoint will be able to handle it. But let’s make sure we call this new method at the right time:

Function ProcessEventsAsync(...) As Task Implements ...

For Each eventData In messages

'Make sure the datalist is correctly sized and empty

RefreshDataList()

...

For Each piece in eventData.Properties

NewDatapoint(piece.Key, piece.Value)

Next

Next

...

End Function

Finally, we need to allow SimpleEventProcessor to write to a few files.

Sub WriteToDisk()

Dim FileLocation = My.Computer.FileSystem.CurrentDirectory

Dim CSVFile As String = FileLocation & "/IoTHubData.CSV"

Dim LabelFile As String = FileLocation & "/IoTHubLabels.txt"

'Generate new data line and label file

Dim Data As String = ""

Dim LabelFileText As String = ""

For i = 0 To KnownLabels.Count - 1

LabelFileText += KnownLabels(i) & vbCrLf

If Not IsNothing(InsertData(i)) AndAlso

InsertData(i) <> "" Then

Data += InsertData(i)

End If

Data += ","

Next

Data.TrimEnd(",")

Data += vbCrLf

'Append new data to existing file

IO.File.AppendAllText(CSVFile, Data)

LabelFileText.TrimEnd(vbCrLf)

'Overwrite the known labels file

IO.File.WriteAllText(LabelFile, LabelFileText)

End Sub

**Notes:**

* You may change the FileLocation to be whatever directory you choose, so long as the directory string *does* ***not*** end with a slash “/” character.
* VB.NET evaluates for loops as (i = 0; i <= MAX; i++), so it is necessary to reduce the count by 1 to match the indices.
* IsNothing(a) is equivalent to a == null.
* “<>” is equivalent to “!=”
* “&” is equivalent to obj.ToString + obj.ToString
* “vbCrLf” is a line-return character, similar to “\n”
* There are many methods available to write files; if you have another method that you prefer to use, feel free to replace the ones written above. Just make sure their behavior is in line with what you intend to do; i.e. overwriting vs appending.

That’s a long method, but it generates a CSV file with each sent set being compiled into a line, with null spaces where data wasn’t received, with a tolerance for adding additional sensor data after-the-fact. The CSV file can then be imported into Excel or other applications for further analysis or graphing. If needed, the data columns can be read from the individual lines of the labels file.

As the final touch, let’s also allow the application to load up the previously known labels so that the order remains the same between separate runs. To do this, let’s generate a constructor method.

Public Sub New()

Dim FileLocation = My.Computer.FileSystem.CurrentDirectory

Dim LabelFile As String = FileLocation & “/IoTHubLabels.txt”

Dim labels = IO.File.ReadAllLines(LabelFile)

KnownLabels = New List(Of String)(labels)

End Sub

Notice that we didn’t previously put FileLocation and LabelFile as class variables, so we need to rewrite them here. If you’d like, go ahead and reorganize them to be class variables to prevent problems in case you decide to rename or relocate the file later. For now, sticking with our current code will be fine.

But wait! On first run, you won’t have a file to read from, so the line “KnownLabels = New List” will throw an exception. No worries, let’s just quickly amend that with a line to create the file if it doesn’t exist.

Public Sub New()

Dim FileLocation = My.Computer.FileSystem.CurrentDirectory

Dim LabelFile As String = FileLocation & “/IoTHubLabels.txt”

If Not IO.File.Exists(LabelFile) Then

Dim f = IO.File.Create(LabelFile)

'Required call to close the filestream

f.Dispose()

End If

Dim labels = IO.File.ReadAllLines(LabelFile)

KnownLabels = New List(Of String)(labels)

'Ignore empty key on initial creation

If KnownLabels.Count = 1 AndAlso KnownLabels(0) = "" Then

KnownLabels.Remove(0)

End Sub

Finally, we need a call to WriteToDisk after all the events have been processed.

Function ProcessEventsAsync(...) As Task Implements ...

For Each eventData In messages

'Make sure the datalist is correctly sized and empty

RefreshDataList()

...

For Each piece in eventData.Properties

NewDatapoint(piece.Key, piece.Value)

Next

Next

WriteToDisk()

...

End Function

Run your console application and Python module! Let it grab a few datapoints, then check the file with an application like Notepad++. N++ will let you refresh the file’s content without having to close and reopen it, so you can see the new data as it gets written to file. Just click out of the window, then bring back focus to N++; a prompt will appear to reload the file.

**Conclusion**

Congrats! You’ve successfully created a console application in .NET that can download and view the real-time data being received by your IoT Hub, and created a CSV file that contains all of that data. The CSV protocol is also robust enough to expand the number of variables and skip invalid values, so you never have to manually adjust the file yourself.

**What Else Can I Try?**

You can begin looking for sensors that you want to install, and finding their respective Python libraries or reference documents. Test your newly-made application with lots of data, and find out when it would be a good time to reset the save file. Determine file sizes based on time run.