Warsztat IoT, 10.3.2016

Adafruit Windows iot Starter KIT, Azure Iot Hub

Tomasz Kopacz

2016

Table of Contents

[Wymagania wstępne 2](#_Toc445384139)

[Przegląd Windows for IoT – UI i procedury postępowania (kolejność ważna!) 3](#_Toc445384140)

[Połączenia 6](#_Toc445384141)

[Wyjścia Raspberry PI 2 / 3 7](#_Toc445384142)

[Komentarz do połączeń 8](#_Toc445384143)

[Kod 9](#_Toc445384144)

[Test 1 LED 9](#_Toc445384145)

[Istotne fragmenty kodu w MainPage.cs 9](#_Toc445384146)

[Kod 10](#_Toc445384147)

[Test 2 BMP280 10](#_Toc445384148)

[Istotne fragmenty kodu w MainPage.cs 10](#_Toc445384149)

[Kod 11](#_Toc445384150)

[Test 3 MPC3008 11](#_Toc445384151)

[Istotne fragmenty kodu w MainPage.cs 11](#_Toc445384152)

[Kod 12](#_Toc445384153)

[Test 4 TCS34725 12](#_Toc445384154)

[Istotne fragmenty kodu w MainPage.cs 12](#_Toc445384155)

[Azure - założenie IoT Hub i okolic 13](#_Toc445384156)

[Rejestracja urządzenia – DeviceExplorer 20](#_Toc445384157)

[Azure IoT Hub – czy to w ogóle działa? 21](#_Toc445384158)

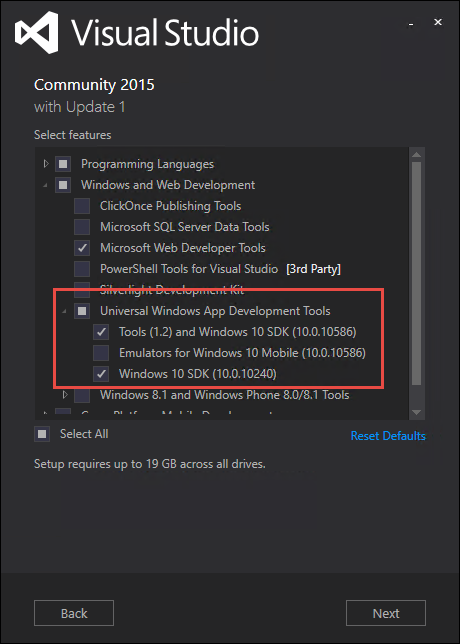
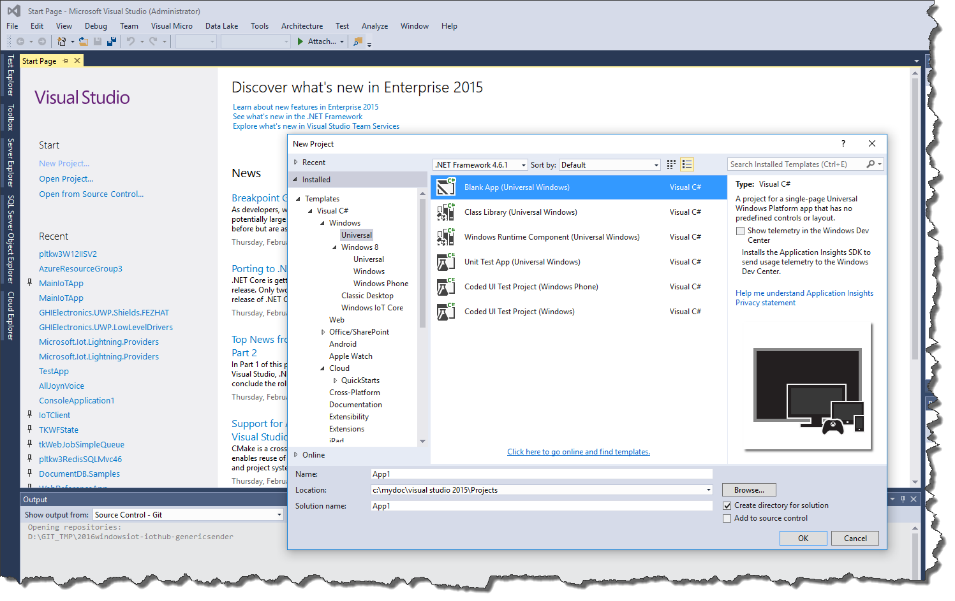
[Aplikacja UWP do wysyłania komunikatów do IoT Hub 23](#_Toc445384159)

[Dodanie do Stream Analytics dodatkowego wyjścia - centralnego „Event Hub” 30](#_Toc445384160)

[Aplikacja UWP do wysyłania komunikatów do IoT Hub – MQTT 32](#_Toc445384161)

[Własna aplikacja do odbierania i przetwarzania wiadomości z IoT Hub 36](#_Toc445384162)

# Wymagania wstępne

1. Testowane na komputerze z Windows 10, build 10586 – natomiast technicznie powinno to zadziałać od Windows 7 w górę.
2. Zainstalowane Visual Studio 2015 z Update 1 (stan na 9.3.2016), wraz z narzędziami do Uniwersal Windows Platform – wersje 10.0.10586 oraz 10.0.10240.  
   
3. Sprawdzenie:
   1. Założyć nowy projekt UWP  
      
   2. Jeżeli jest to Windows 10, projekt można uruchomić. Jeżeli starszy system – wystarczy zobaczyć czy projekt się prawidłowo wczyta i skompiluje
   3. (W razie potrzeby włączy
4. Opcjonalnie: (Windows IoT Core Project Templates) [https://visualstudiogallery.msdn.microsoft.com/55b357e1-a533-43ad-82a5-a88ac4b01dec](https://na01.safelinks.protection.outlook.com/?url=https%3a%2f%2fvisualstudiogallery.msdn.microsoft.com%2f55b357e1-a533-43ad-82a5-a88ac4b01dec&data=01%7c01%7clukapi%40064d.mgd.microsoft.com%7c57516dd0c9264e55743908d331a66b48%7c72f988bf86f141af91ab2d7cd011db47%7c1&sdata=ch4VKyNE41JtTtYhMNow0V%2fMlK7xS%2fIjPZzClNpbrOY%3d)
5. Z [http://ms-iot.github.io/content/en-US/Downloads.htm](https://na01.safelinks.protection.outlook.com/?url=http%3a%2f%2fms-iot.github.io%2fcontent%2fen-US%2fDownloads.htm&data=01%7c01%7clukapi%40064d.mgd.microsoft.com%7c57516dd0c9264e55743908d331a66b48%7c72f988bf86f141af91ab2d7cd011db47%7c1&sdata=oAgmq9TL2OJoqkEQBkpBiqpBhh3cht58b5QLCyeunME%3d) Windows 10 IoT Core dla Raspberry PI 2 ([http://go.microsoft.com/fwlink/?LinkId=691711](https://na01.safelinks.protection.outlook.com/?url=http%3a%2f%2fgo.microsoft.com%2ffwlink%2f%3fLinkId%3d691711&data=01%7c01%7clukapi%40064d.mgd.microsoft.com%7c57516dd0c9264e55743908d331a66b48%7c72f988bf86f141af91ab2d7cd011db47%7c1&sdata=Yo1aPcw5kV%2ba1e%2fY561eVUWyl3bO6mtpljdE06J1OcE%3d) ) – do przygotowania nowego obrazu

# Pobranie materiałów z repozytorium

## Główny warsztat:

git clone <https://github.com/tkopacz/2016windowsiot-iothub-workshopV1>

Zawartość:

* **Doc** – dokumenty i prezentacje
* **Test1\_BlinkingLed** – test, czy podłączona prawidłowo diodę (powinna mrugać)
* **Test2\_BMP280** – test, czy podłączony prawidłowo układ BMP280 do szyny I2C (mierzy temperatura, ciśnienie). Wyniki wyprowadzane w zakładce output w Visual Studio. Po dotknięciu układu palcem (lub – chuchnięciu), temperatura powinna wzrosnąć
* ms-iot – skopiowane elemety z <https://github.com/ms-iot/BusProviders.git>; potrzebne do obsługi przetwornika analogowo-cyfrowego – MCP3008.
* **Test3\_MCP3008** – test, czy podłączony jest prawidłowo układ MCP3008, przez szynę SPI. Obsługuje fotorezystor i dwa potencjometry. Wyniki wyprowadza w zakładce output w Visual Studio (w trybie dla debuggera)
* **Test4\_TCS34725** – test, czy podłączony prawidłowo układ TCS34725 do szyny I2C (odczytuje kolor podstawionego obiektu). Wyniki wyprowadzane w zakładce output w Visual Studio.
* **MainIoTApp** – główna aplikacja wysyłająca zdarzenia do Azure i odbierająca polecenia. Wymaga podania kluczy do IoT Hub i rejestracji konkretnego urządzenia.
* **MainIoTAppMqtt** – główna aplikacja wysyłająca zdarzenia do Azure (używa MQTT). Wymaga podania kluczy do IoT Hub.
* **Schematics** – schematy połączeń
* **Tools** – skopiowane narzędzia – DeviceExplorer i Service Bus Exlorer.

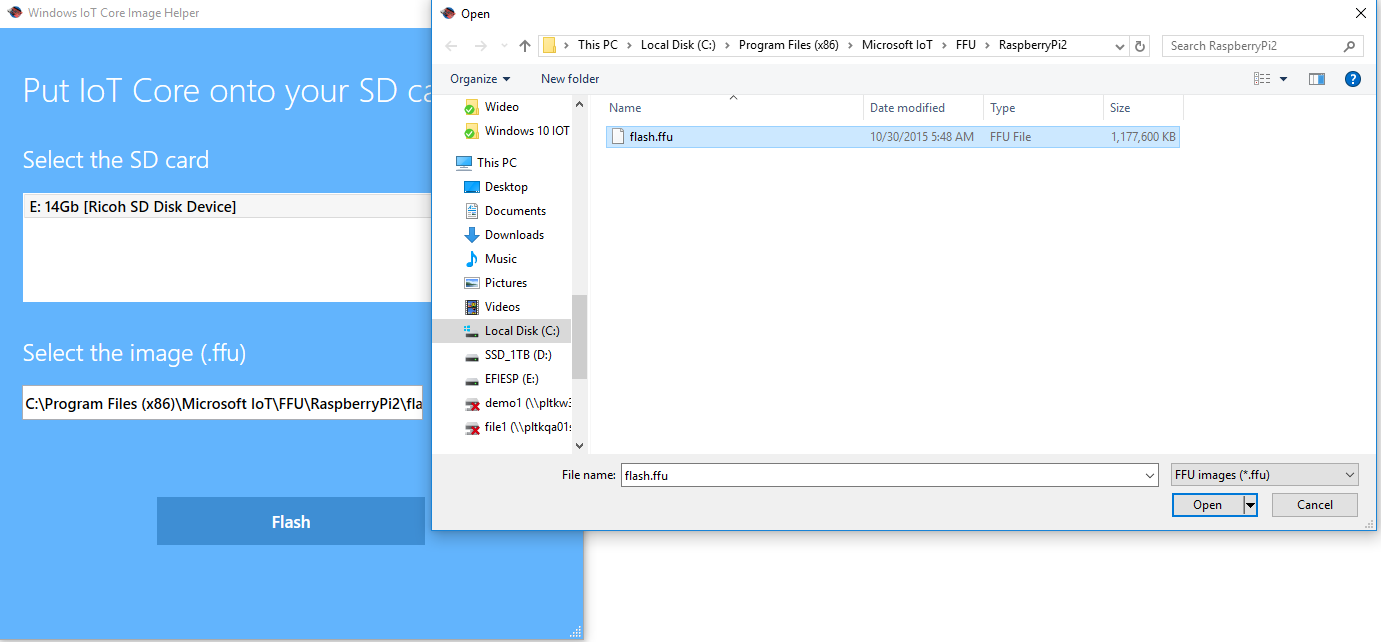
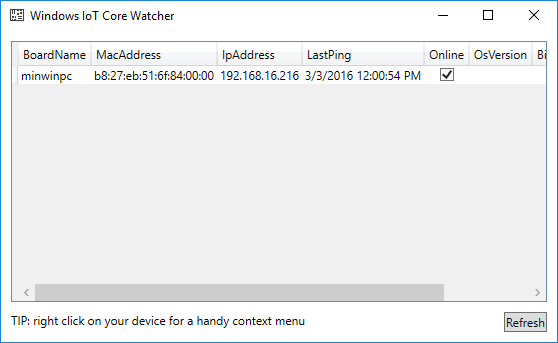
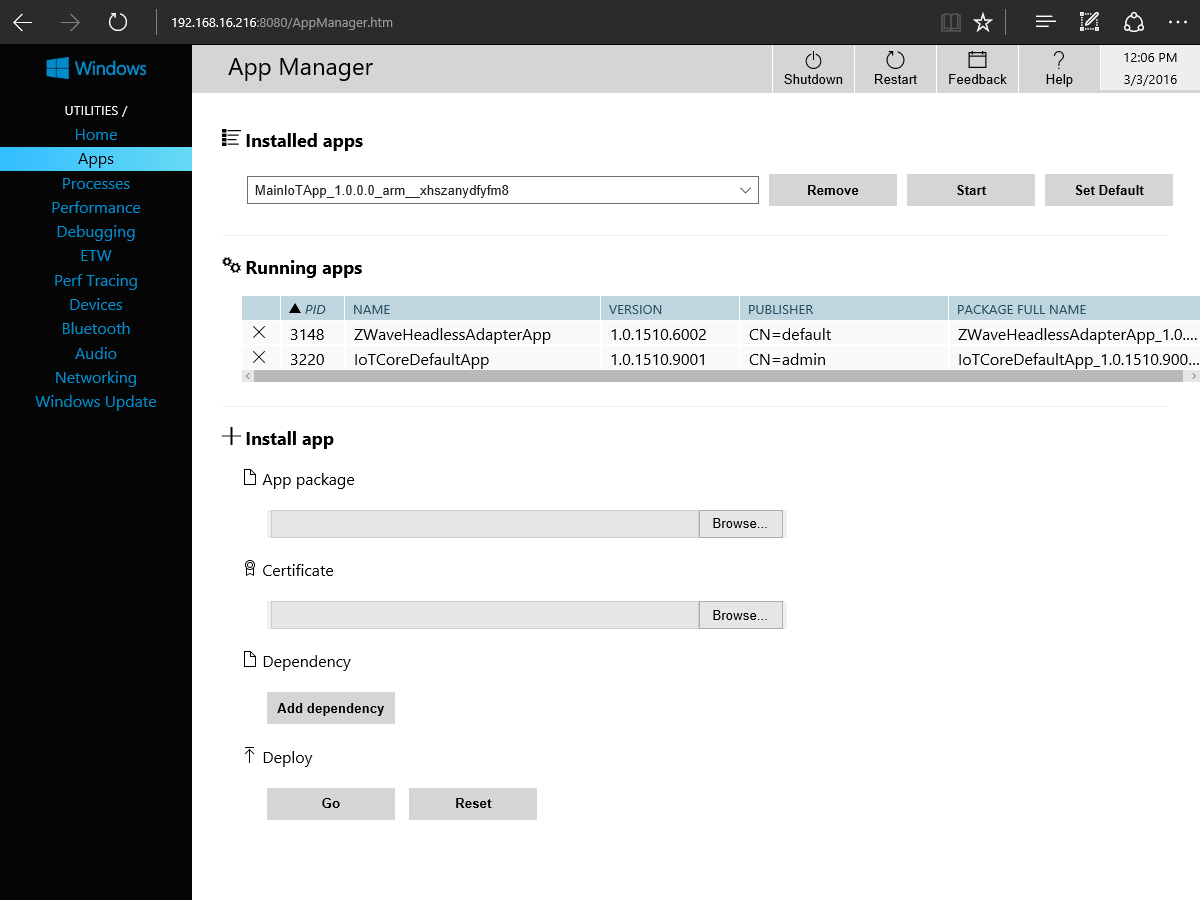
## Pomocnicze elementy związane z testowaniem IoT Hub:

git clone <https://github.com/tkopacz/2016windowsiot-iothub-genericsender>

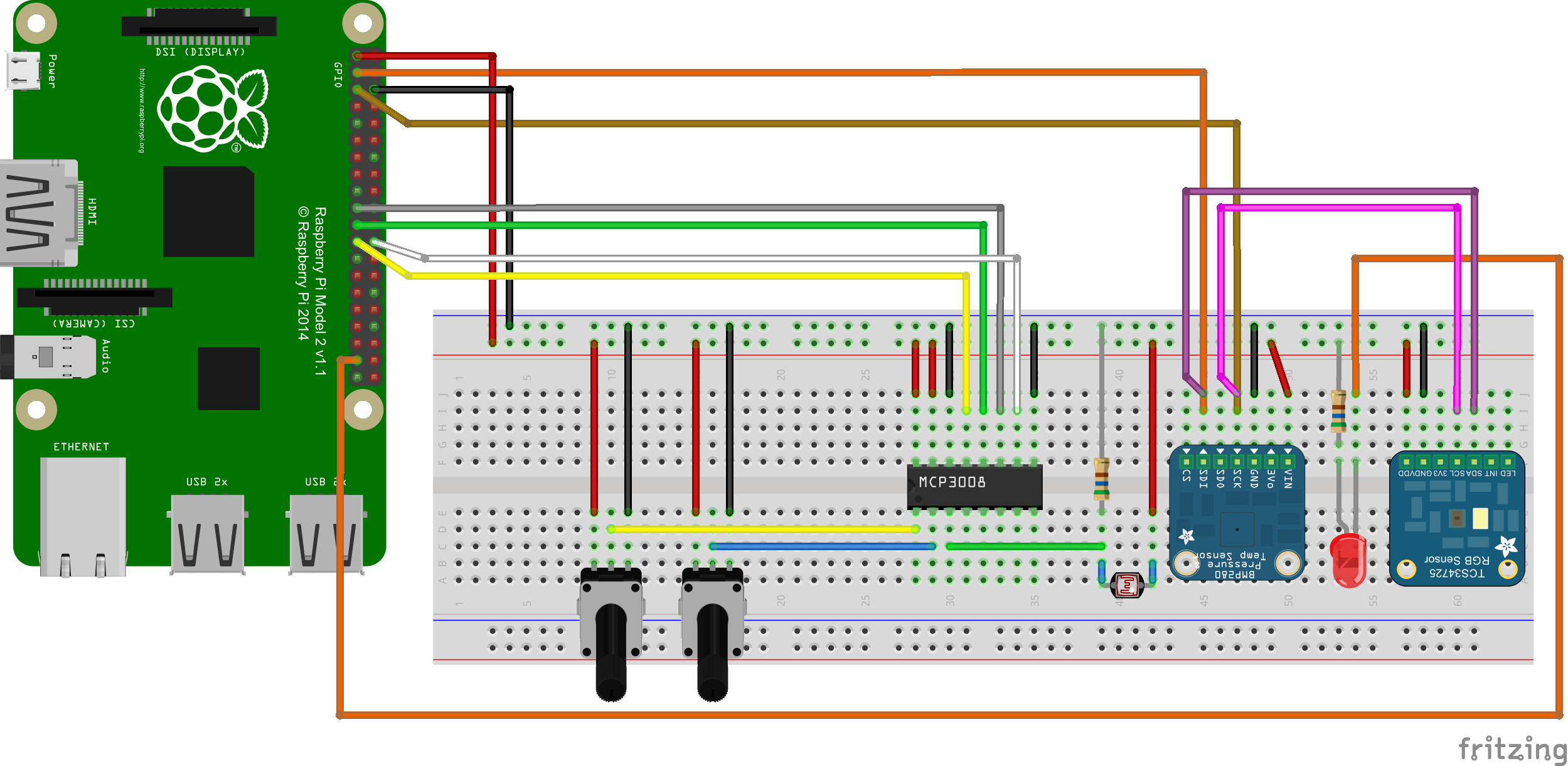
Zawartość:

* **DeviceExplorer** – kod źródłowy DeviceExlorer
* **READY\_TO\_RUN** – skompilowany DeviceExplorer
* **SamplePCClient** – przykładowi klienci IoT Hub niewymagający żadnego urządzenia sprzętowego (należy podać własne klucze do IoT Hub)

# Przegląd Windows for IoT – UI i procedury postępowania (kolejność ważna!)

1. Przygotować kartę microSD  
   
2. PO KOLEI WŁĄCZAĆ URZĄDZENIA – w kolejności wskazanej przez prowadzącego.
   1. **Jedynym jednoznacznym wyróżnikiem jest MAC Address – no i trzeba swój zapisać na kartce!**
   2. **Jak są monitory, na ekranie każdy uczestnik widzi adres IP swojego RPi2 – więc nie jest to niezbędne.**
3. Zmienić hasło na jakieś „własne” (i go nie zapomnieć – albo idź do punktu 1)
4. Zmienić nazwę urządzenia na D<numer stanowiska>
5. Obserwowanie WSZYSTKICH urządzeń z Windows IoT.   
   
6. Wejście na udział sieciowy //<nazwa urządzenia>/c$
7. Wejście na stronę WWW do zarządzania Windows IoT (http://IP:8080)  
   
8. Chwilę posłuchać co można zrobić na portalu.

# Połączenia



# RPi2 Pin HeaderWyjścia Raspberry PI 2 / 3

# Komentarz do połączeń

Numery znajdują się na boku płytki prototypowej (mała czcionka, kolor niebieski). Urządzenia należy połączyć w sposób pokazany na diagramie. Dodatkowo poniżej główne numery „pinów”. Dodatkowo zaznaczone są wyjścia na RPi2 które będą używane w naszym przykładzie.

1. Potencjometr 1
   1. 9,10,11
2. Potencjometr 2
   1. 15,16,17
3. MCP3008
   1. 28,29,30,31,32,33,34,35
4. Foto
   1. 39, 42 (obojętne w którą stronę)
5. BMP280
   1. 44,45,46,47,48,49,50
6. Dioda
   1. 53 (dłuższa noga), 54 (krótsza) (uwaga! On - Low, Off - High)
7. Rezystor1
   1. 39, do - (niebieskie)
8. Rezystor2
   1. 53, do + (czerwone)
9. Podłączyć zasilania i masy
10. Podłączyć inne sygnały
11. Sprawdzić podłączenia. Szczególną uwagę należy poświęcić zasilaniu (czerwony przewód, +) oraz masy (niebieski, -) przed wszystkim!

**JAK SIĘ ROBI CIEPŁE I CUCHNIE – SZYBKO WYŁĄCZYĆ ZASILANIE (szansa uszkodzeń mała - podłączamy wszystko pod 3.3V)**

# Kod

## Test 1 LED

1. Nowy projekt UWP
2. Dodać IoT Extension
3. Podłączona dioda do 37 (GPIO 26), za pośrednictwem płytki prototypowej.

### Istotne fragmenty kodu w MainPage.cs

public MainPage()

{

this.InitializeComponent();

setup();

}

DispatcherTimer m\_t;

GpioPin m\_blink;

GpioPinValue m\_blinkValue;

private async void setup()

{

var gpio = GpioController.GetDefault();

if (gpio!=null)

{

m\_blink = gpio.OpenPin(26); //See board

m\_blinkValue = GpioPinValue.High;

m\_blink.Write(m\_blinkValue);

m\_blink.SetDriveMode(GpioPinDriveMode.Output);

m\_t = new DispatcherTimer();

m\_t.Interval = TimeSpan.FromSeconds(1);

m\_t.Tick += M\_t\_Tick;

m\_t.Start();

}

}

private void M\_t\_Tick(object sender, object e)

{

if (m\_blinkValue == GpioPinValue.High) m\_blinkValue = GpioPinValue.Low; else m\_blinkValue = GpioPinValue.High;

m\_blink.Write(m\_blinkValue);

}

# Kod

## Test 2 BMP280

1. Nowy projekt UWP
2. Dodać IoT Extension
3. Podłączony BMP280, za pośrednictwem płytki prototypowej.
4. Pobrać plik z driverem – nieznacznie zmodyfikowany: <https://raw.githubusercontent.com/tkopacz/2016windowsiot-iothub-workshopV1/master/Test2_BMP280/Test2_BMP280/BMP280.cs>

### Istotne fragmenty kodu w MainPage.cs

public MainPage()

{

this.InitializeComponent();

setup();

}

DispatcherTimer m\_t;

BMP280 m\_bmp280;

private async void setup()

{

m\_bmp280 = new BMP280();

await m\_bmp280.Initialize();

m\_t = new DispatcherTimer();

m\_t.Interval = TimeSpan.FromSeconds(5);

m\_t.Tick += M\_t\_Tick;

m\_t.Start();

}

const float seaLevelPressure = 1013.25f;

private async void M\_t\_Tick(object sender, object e)

{

var altitude = await m\_bmp280.ReadAltitudeAsync(seaLevelPressure);

var pressure = await m\_bmp280.ReadPreasureAsync();

var temperature = await m\_bmp280.ReadTemperatureAsync();

Debug.WriteLine($"Alt:{altitude} m, Press:{pressure} Pa, Temp:{temperature} deg C");

}

# Kod

## Test 3 MPC3008

1. Nowy projekt UWP (Test3\_MCP3008)
2. Dodać IoT Extension
3. Podłączony BMP280, za pośrednictwem płytki prototypowej.
4. Pobrać ADC Providers z: <https://github.com/ms-iot/BusProviders.git>
5. Dodać projekt ms-iot\ADC\AdcMcp3008\AdcMcp3008.csproj do rozwiązania
6. Dodać referencję nowo dodanego projektu z poziomu Test3\_MCP3008

### Istotne fragmenty kodu w MainPage.cs

public MainPage()

{

this.InitializeComponent();

setup();

}

private AdcController m\_adc;

private AdcChannel[] m\_adcChannel;

DispatcherTimer m\_t;

private async void setup()

{

m\_adc = (await AdcController.GetControllersAsync(AdcMcp3008Provider.GetAdcProvider()))[0];

m\_adcChannel = new AdcChannel[m\_adc.ChannelCount];

for (int i = 0; i < m\_adc.ChannelCount; i++)

{

m\_adcChannel[i] = m\_adc.OpenChannel(i);

}

m\_t = new DispatcherTimer();

m\_t.Interval = TimeSpan.FromSeconds(5);

m\_t.Tick += M\_t\_Tick;

m\_t.Start();

}

private void M\_t\_Tick(object sender, object e)

{

for (int i = 0; i < m\_adc.ChannelCount; i++)

{

Debug.Write($"{i}:{m\_adcChannel[i].ReadRatio():F4}, ");

}

Debug.WriteLine("");

}

# Kod

## Test 4 TCS34725

1. Nowy projekt UWP (Test3\_MCP3008)
2. Dodać IoT Extension
3. Podłączony TCS34725, za pośrednictwem płytki prototypowej.
4. Pobrać lekko zmodyfikowaną bibliotekę: <https://raw.githubusercontent.com/tkopacz/2016windowsiot-iothub-workshopV1/master/Test4_TCS34725/Test4_TCS34725/TCS34725.cs>

### Istotne fragmenty kodu w MainPage.cs

public MainPage()

{

this.InitializeComponent();

setup();

}

TCS34725 m\_tcs;

DispatcherTimer m\_t;

private async void setup()

{

m\_tcs = new TCS34725();

await m\_tcs.Initialize();

m\_t = new DispatcherTimer();

m\_t.Interval = TimeSpan.FromSeconds(1);

m\_t.Tick += M\_t\_Tick;

m\_t.Start();

}

private async void M\_t\_Tick(object sender, object e)

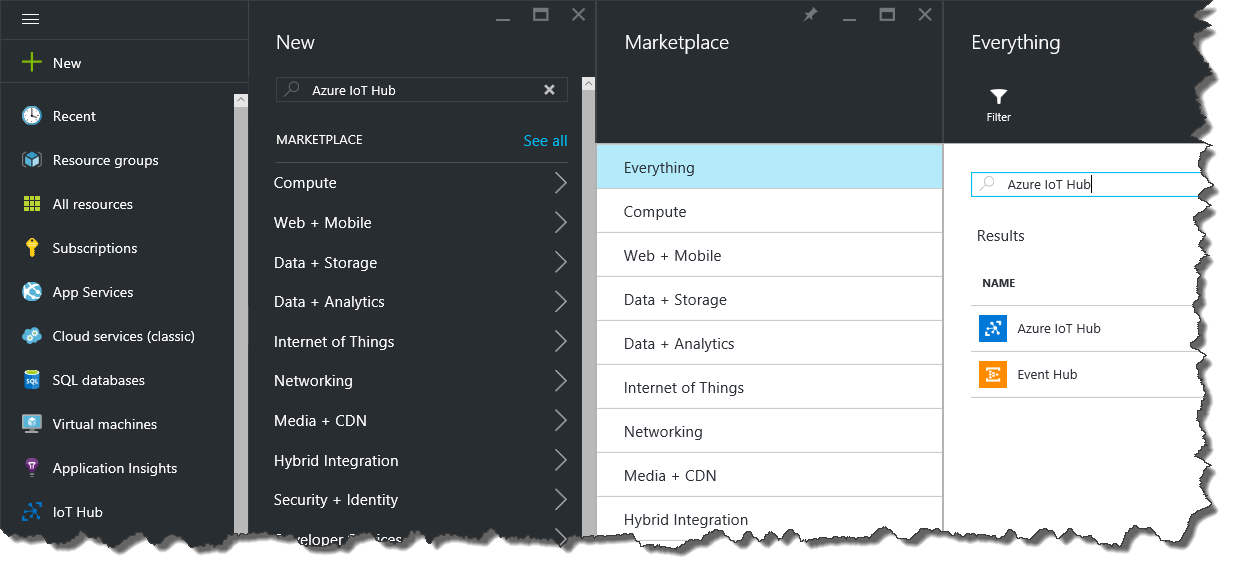
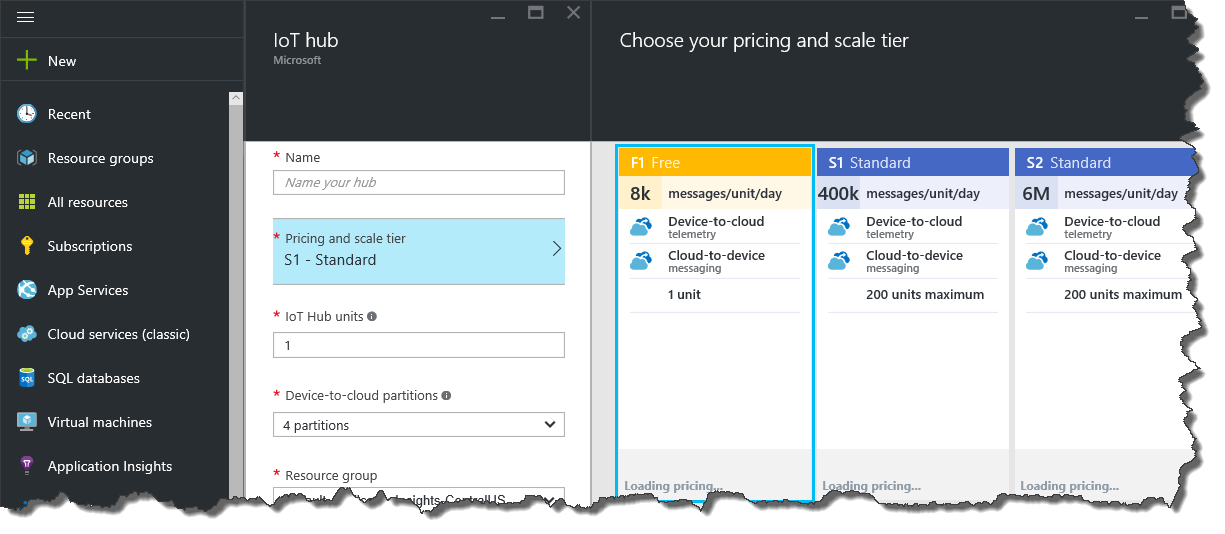
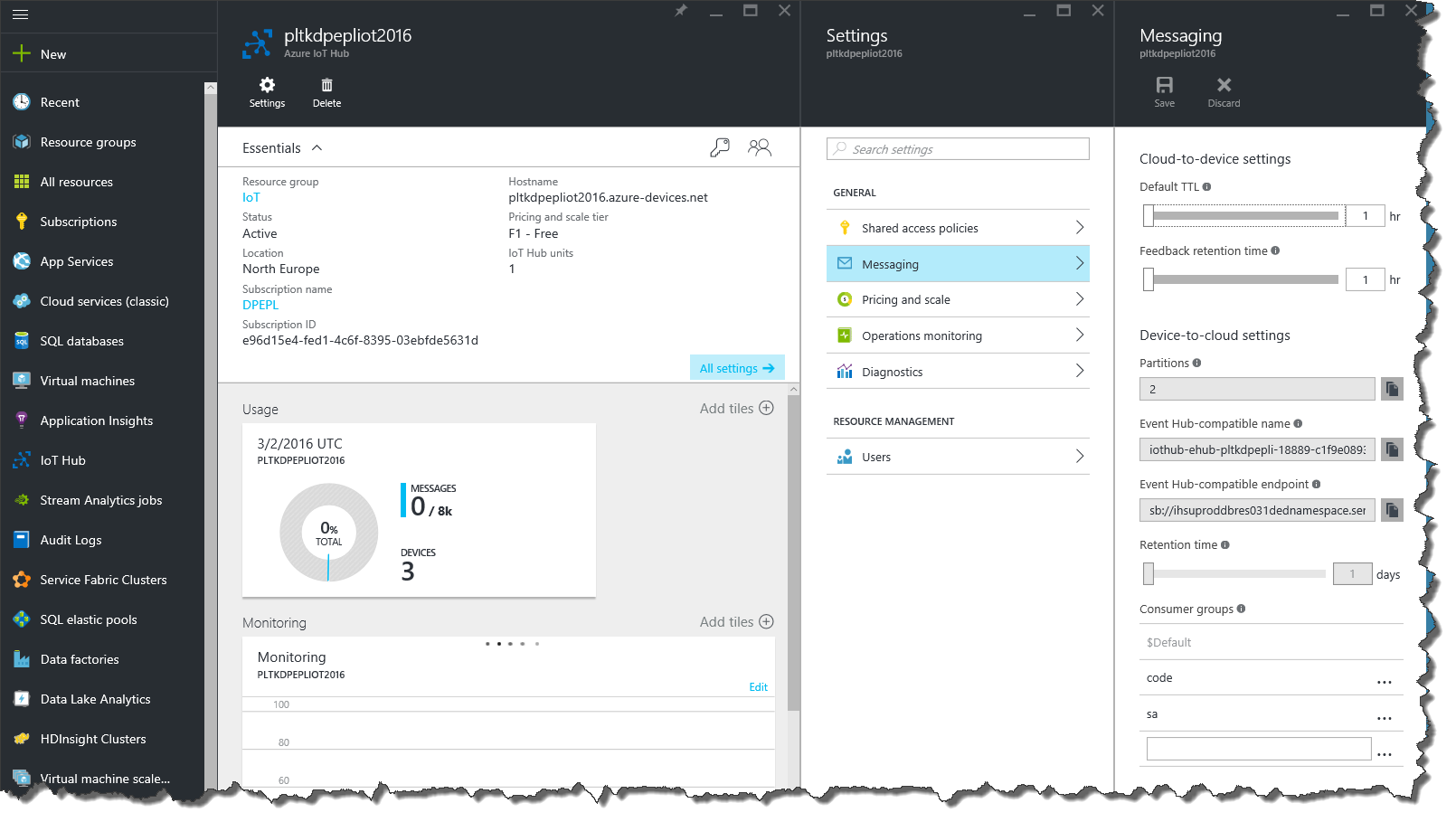
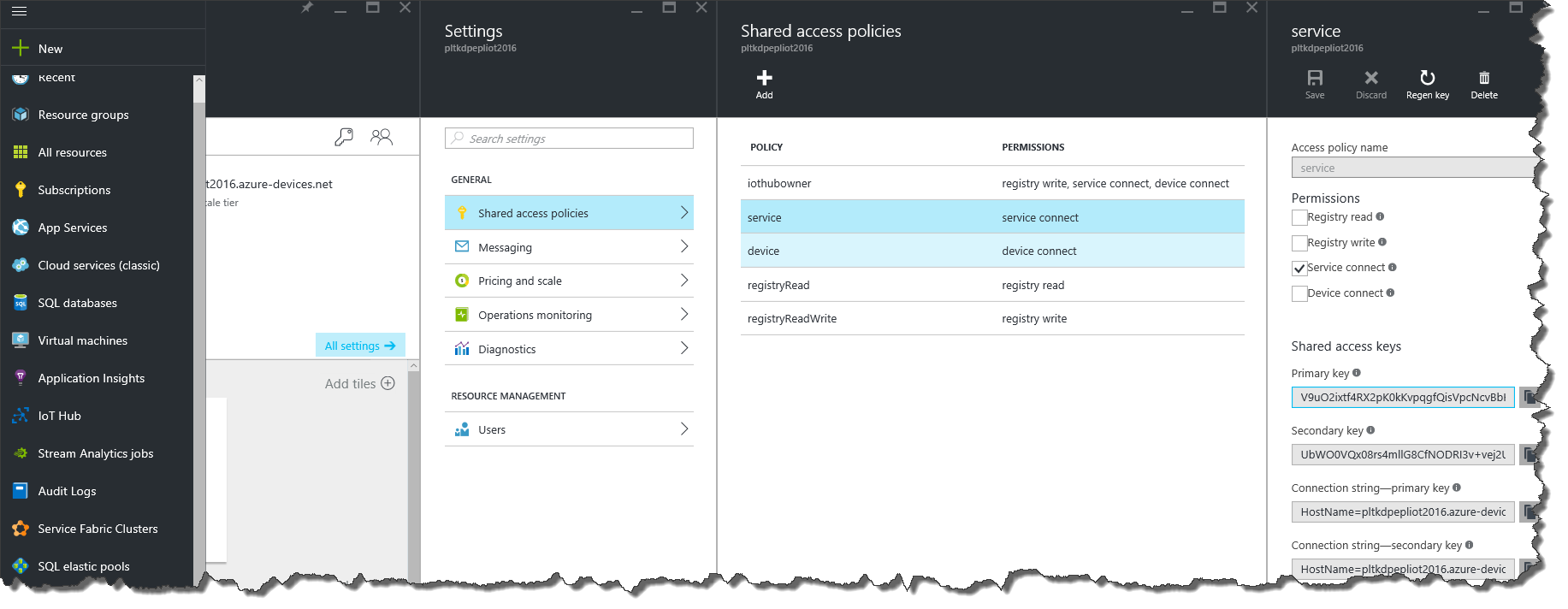
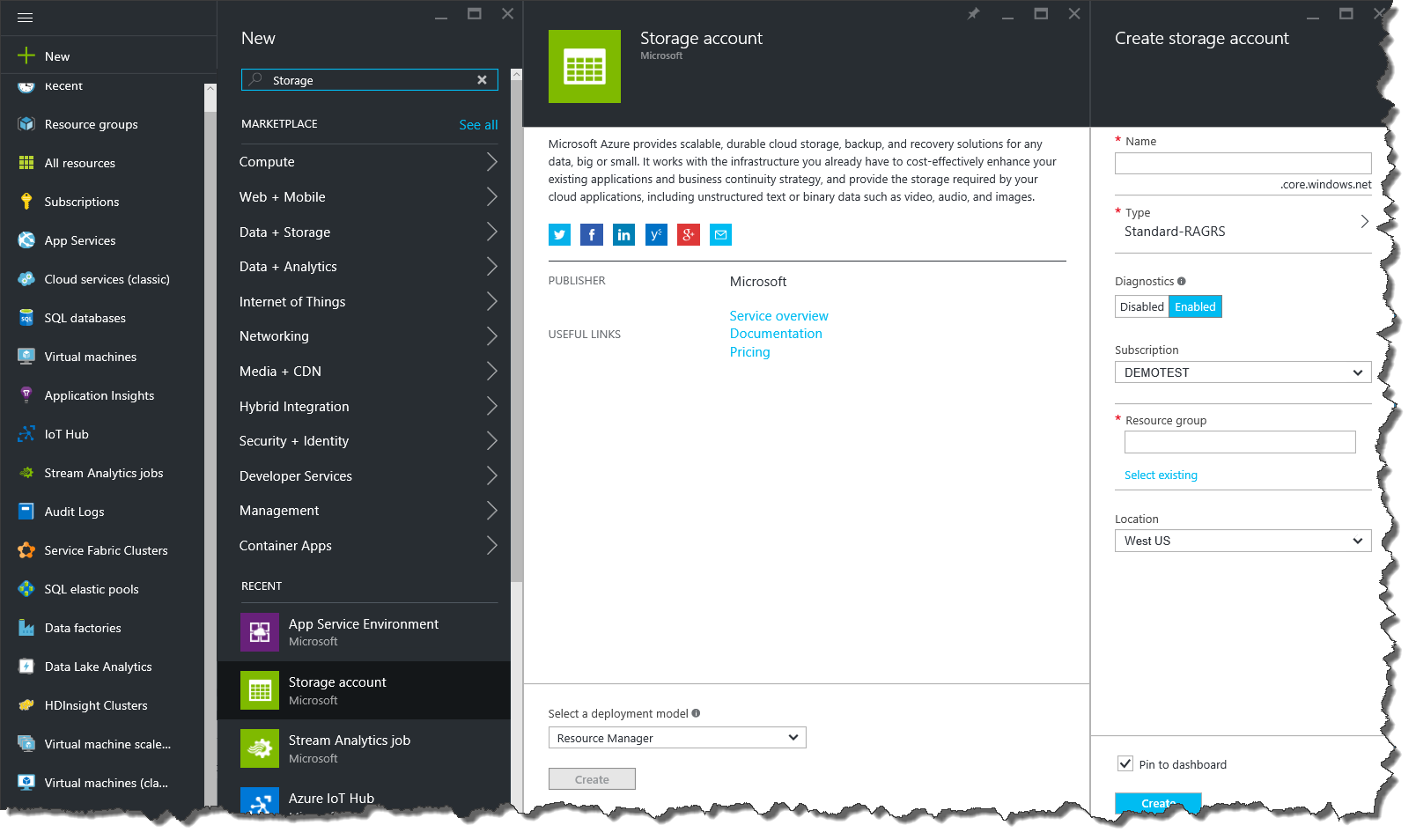
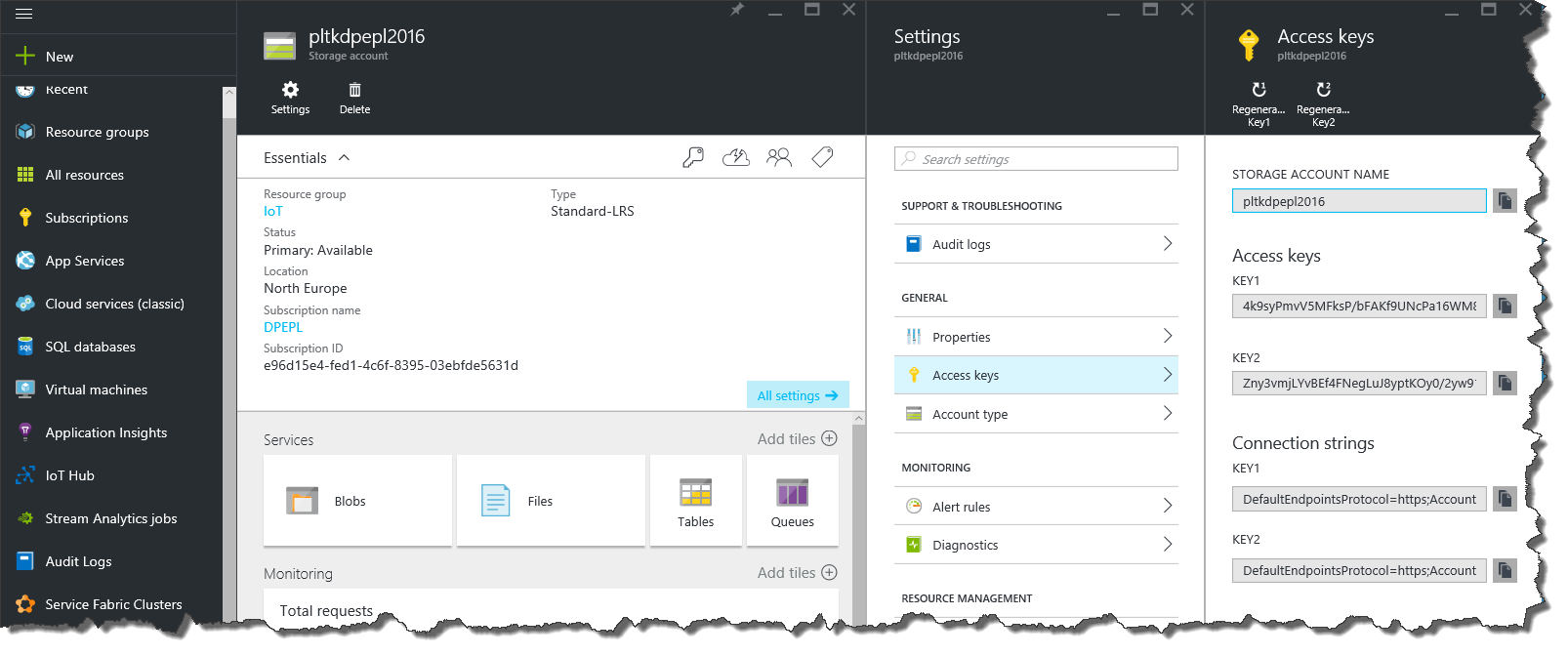
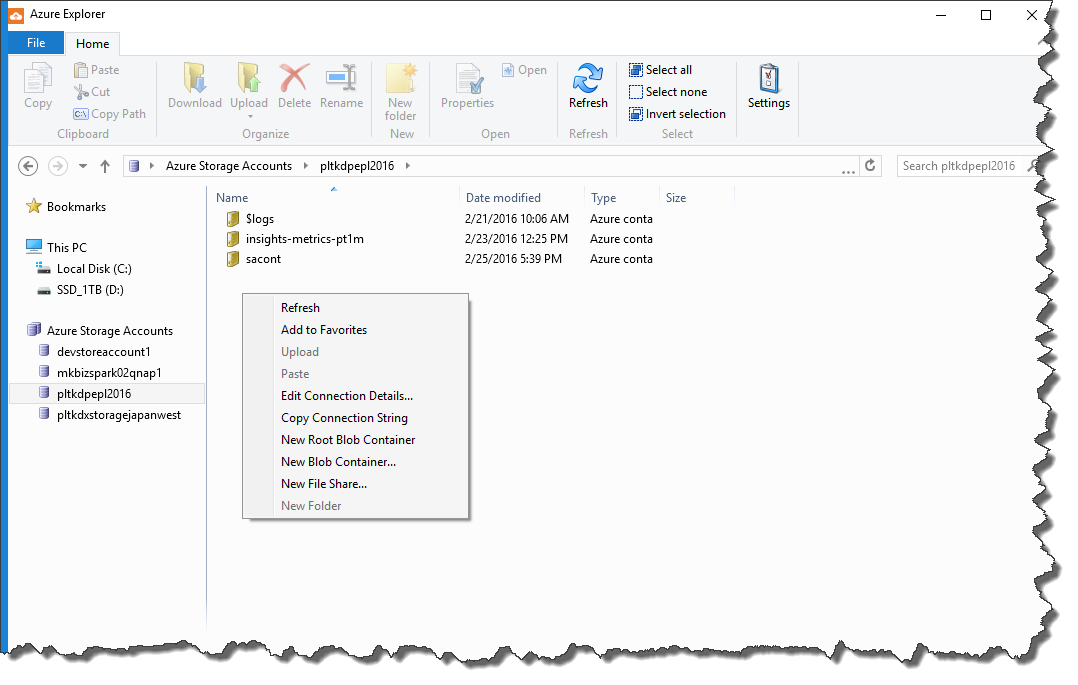
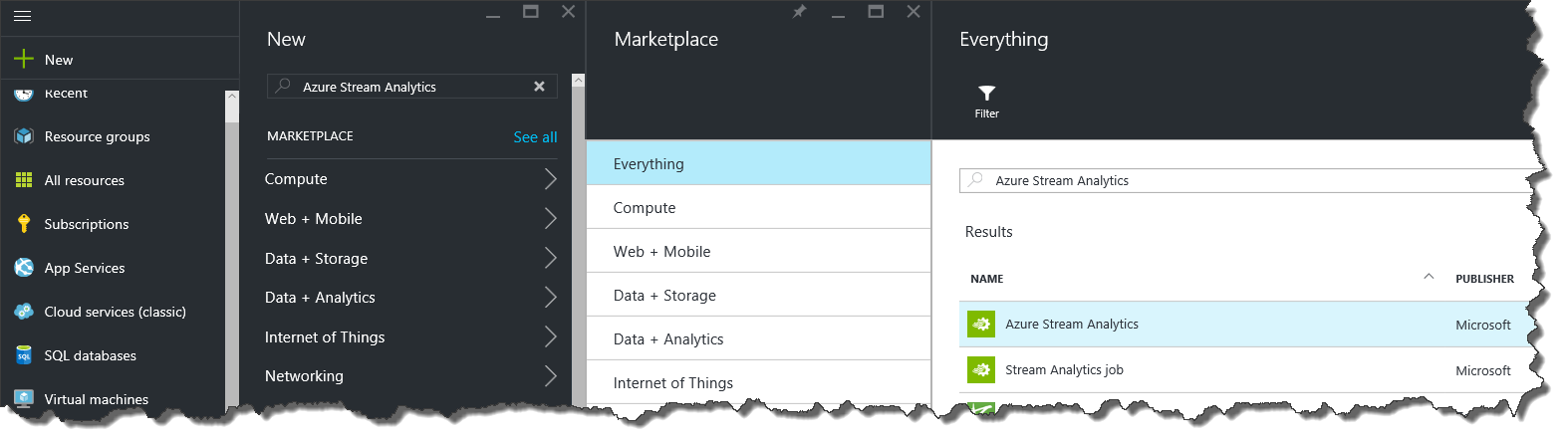
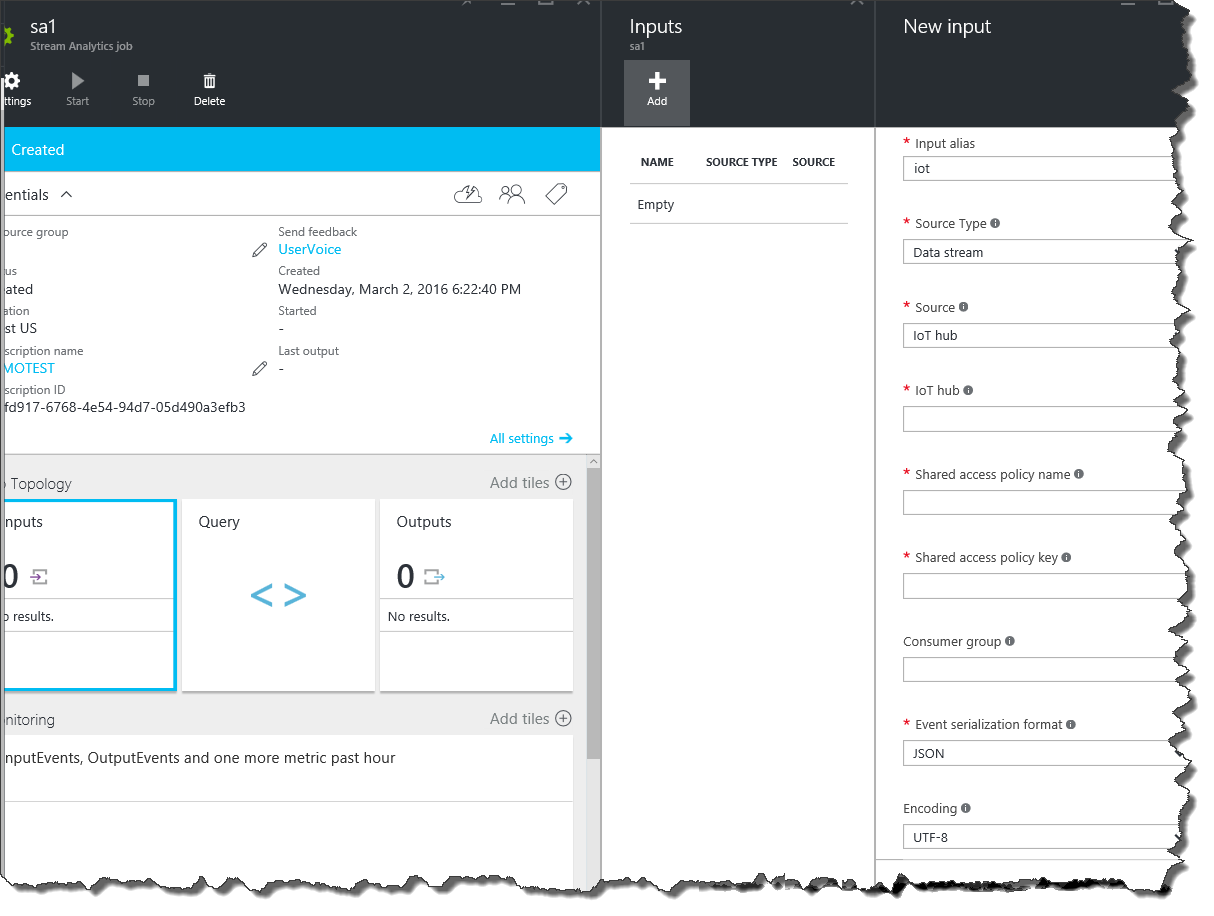
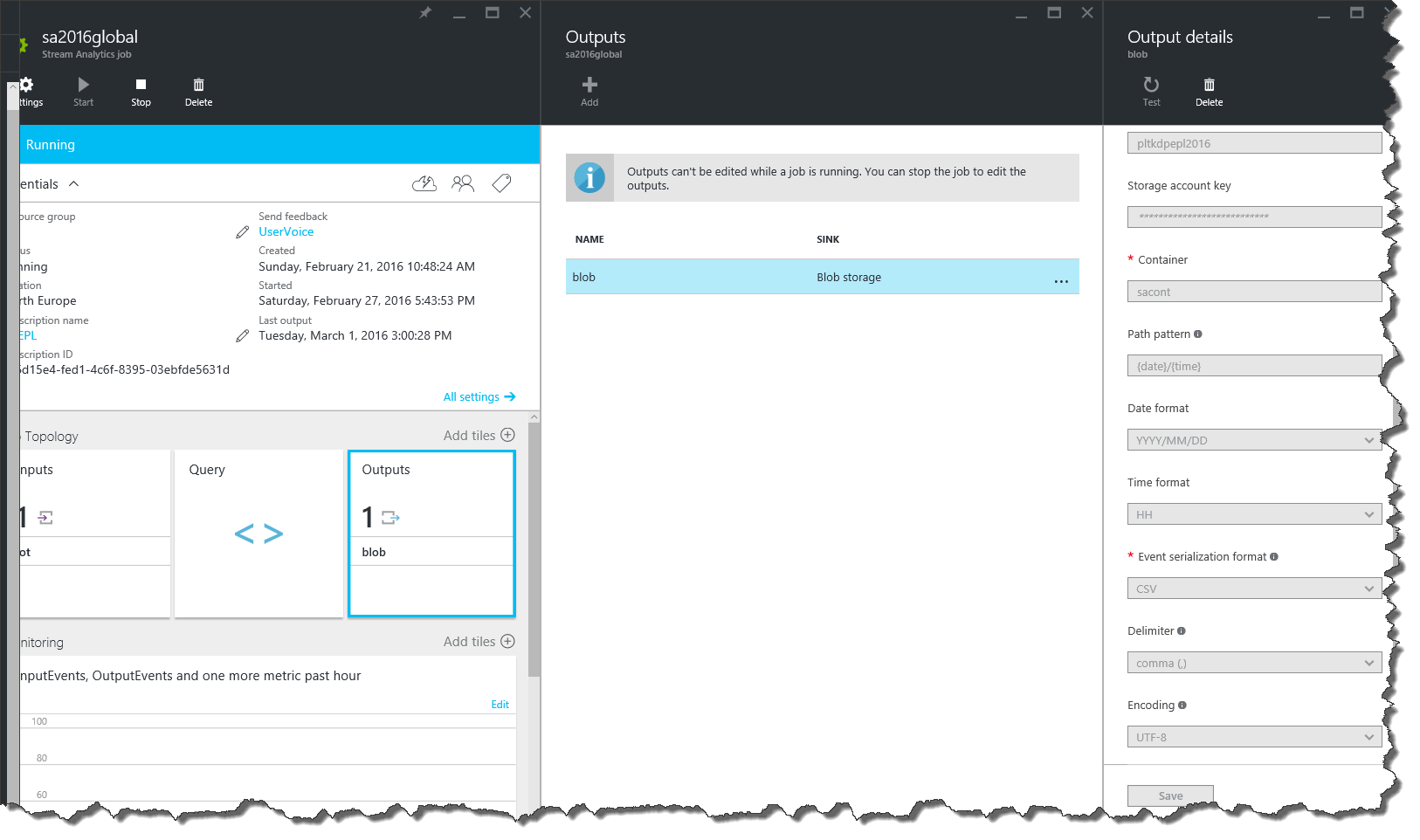
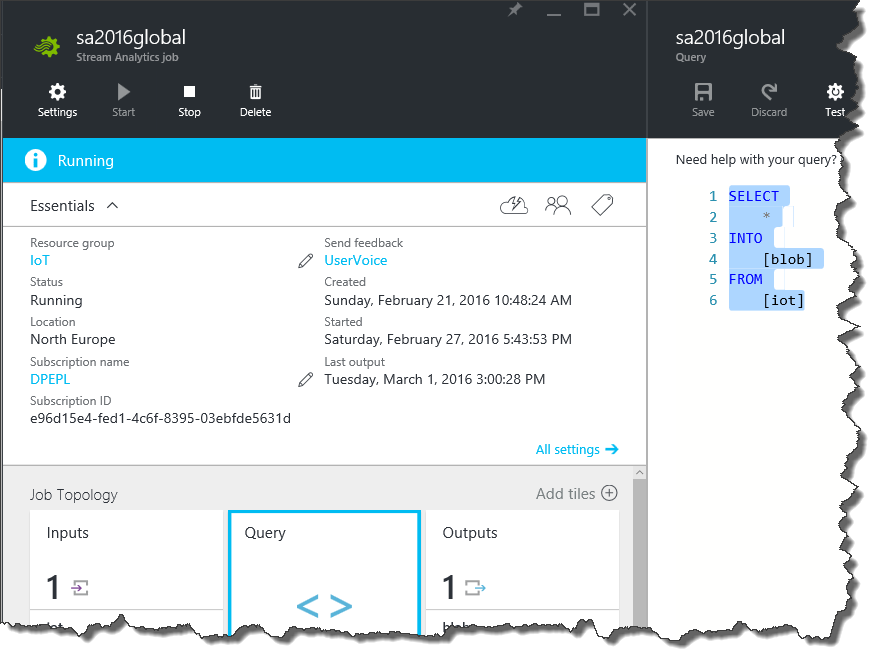
{

var color = await m\_tcs.GetRgbDataAsync();

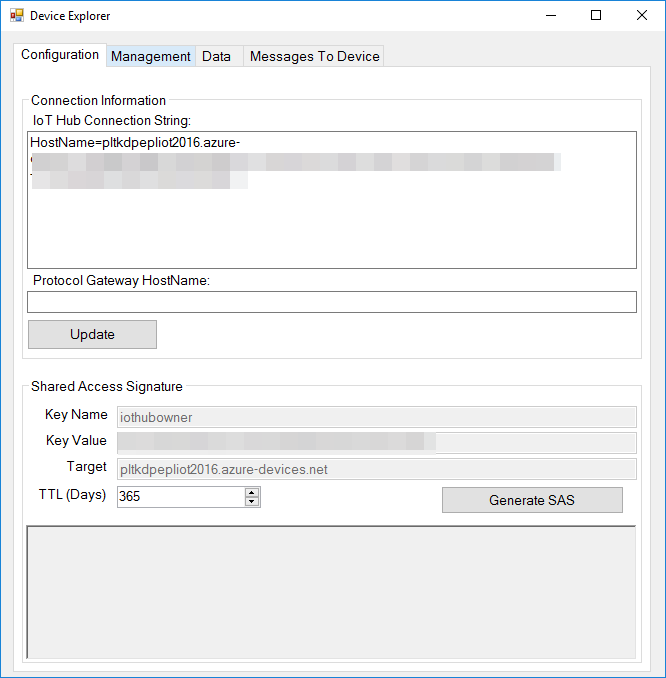
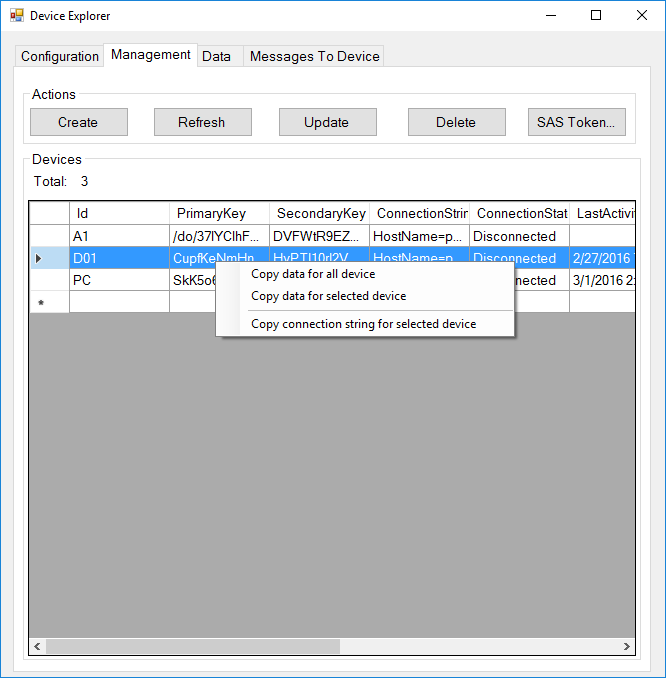
Debug.WriteLine($"{color.Red},{color.Green},{color.Blue}");

}

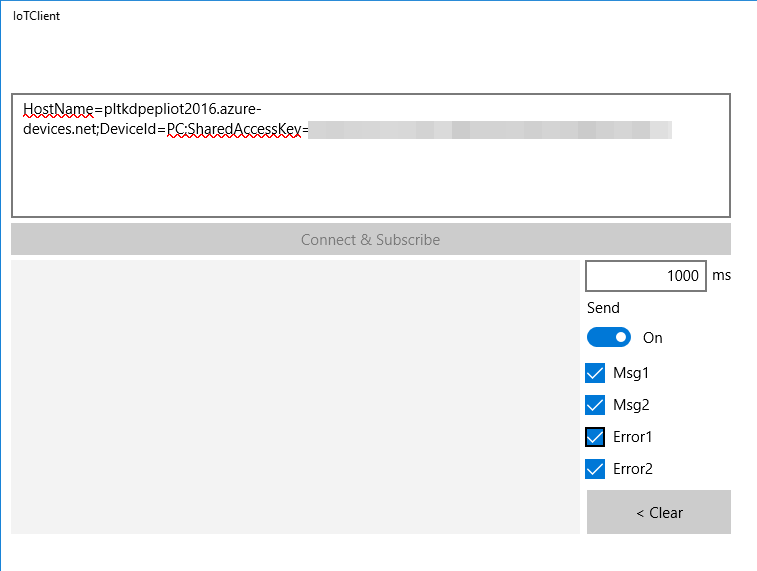
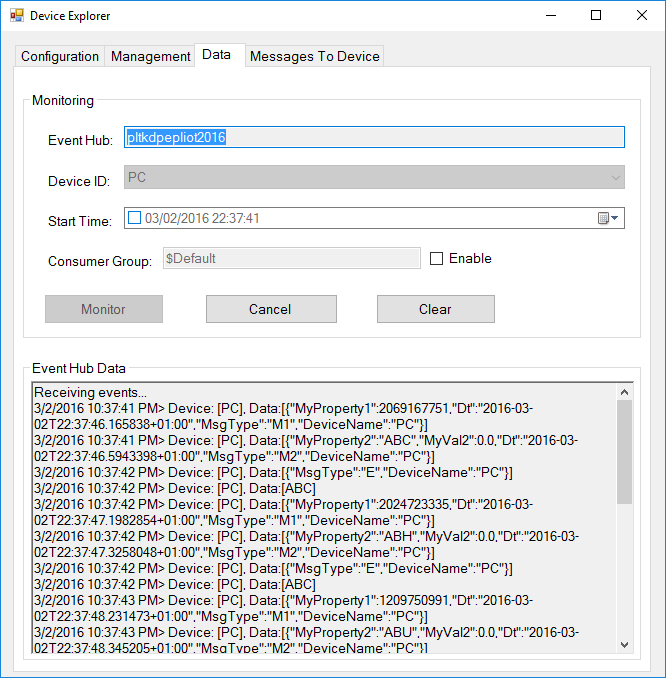
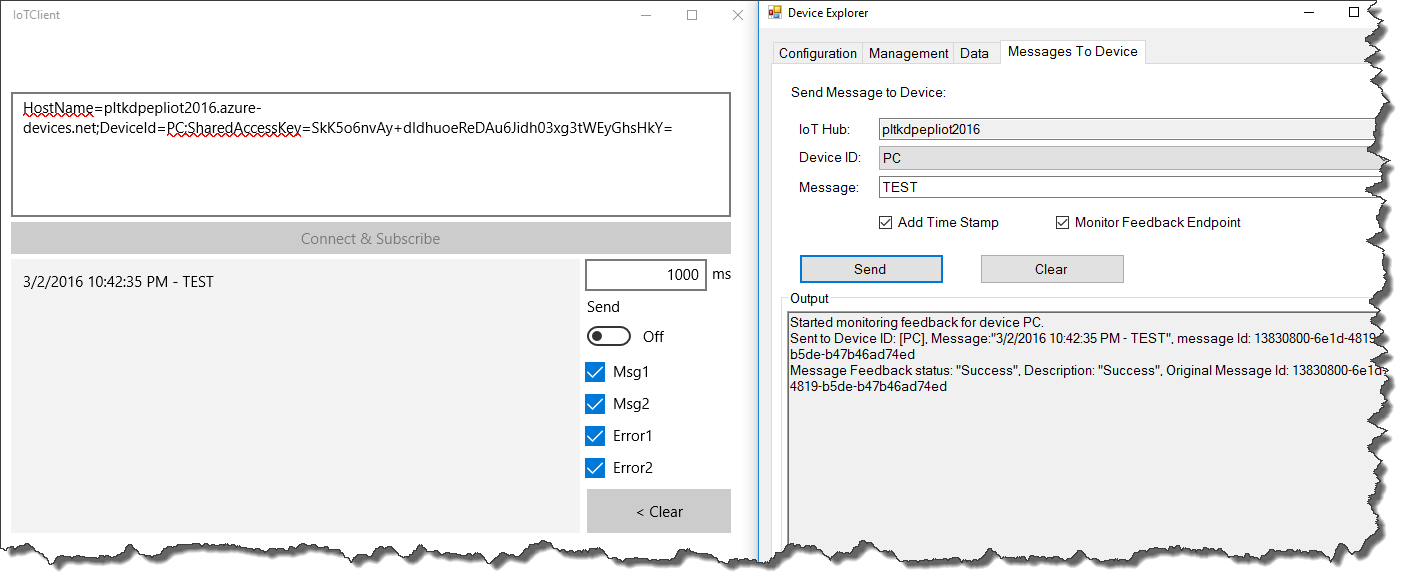
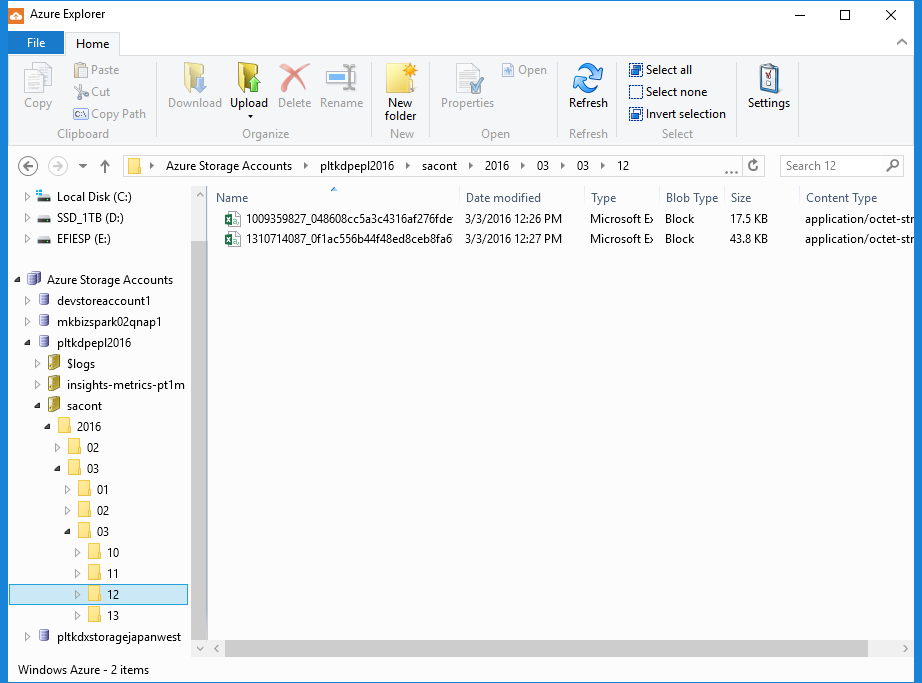
# Azure - założenie IoT Hub i okolic

1. <https://portal.azure.com>
2. New – wpisać *Azure IoT Hub* – Marketplace – wybrać *Azure IoT Hub*  
   
3. Kliknąc *Create*
4. Podać nazwę – sugerowane – iot*inicjałyDataEventu*
5. Wybrać wersję darmową IoT Hub (wystarczy do ćwiczeń).   
   
6. Po utworzeniu IoT Hub, dodać 2 Consumer Group: *sa* oraz *code*  
   
7. Skopiować na bok (przyda się potem):  
   Shared access policies : *iothubowner*, *service*, *device*  
   
8. Założyć konto Azure Storage (o nazwie st*inicjałyDataEventu*), może być deployment type Resource Manager  
   
9. Skopiować na bok: Storage Account Name i Account Keys – KEY1 (przyda się do exploratora i stream analytics)  
   
10. Podłączyć się Azure Explorer (lub innym narzędziem) do Storage:  
    
11. Utworzyć nowe *Azure Stream Analytics* (lub *Stream Analytics job*)  
    
12. Nazwa to na przykład: sa*inicjałyDataEventu*
13. Dodać źródło o nazwie iot, wskazujące na Shared Access Policy naszego IoT Hub. Należy podać też Consumer group (utworzone *sa*) oraz wybrać serializację JSON w UTF-8. Podajemy:  
    *IoT Hub*: nazwa huba (bez domeny)  
    *Shared access policy name*: service  
    *Shared access policy key*: <klucz skopiowany z konfiguracji IoT Hub dla service>  
    Consumer Group: **sa** (którą założyliśmy w IoT Hub)  
    
14. Dodać wyjście o nazwie blob wskazujące na utworzony kontener. Warto wybrać format CSV, z przecinkiem i w UTF-8. Można też określić ścieżkę (*Path pattern*): {date}/{time}  
    
15. Dodać kwerendę (Query): SELECT \* INTO [blob] FROM [iot]  
    
16. Koniec konfiguracji (na razie) Azure.

# Rejestracja urządzenia – DeviceExplorer

1. Git clone <https://github.com/tkopacz/2016windowsiot-iothub-workshopV1>
2. Uruchomić **Tools\READY\_TO\_RUN\DeviceExplorer.exe**
3. Zalogować się Connection string dla iothubowner  
   
4. Zarejestrować urządzenie o nazwie *D<numer stanowiska>* oraz PC
5. Skopiować Connection String dla obu urządzeń

# Azure IoT Hub – czy to w ogóle działa?

1. Git clone <https://github.com/tkopacz/2016windowsiot-iothub-genericsender>
2. Otworzyć projekt **SamplePCClient\IoTClient\IoTClient.sln**
3. Uruchomić SamplePCClient\IoTClient\IoTClient\IoTClient.csproj
4. Podać właściwy łańcuch połączeń (dla PC) – skopiowany przed chwilą po zarejestrowaniu urządzenia.  
   
5. Zobaczyć czy zdarzenia się odbierają w DeviceExplorer  
   
6. Przetestować wysyłanie wiadomości (zakładka Message To Device)  
   
7. Oczywiście – podejrzeć w debuggerze jak działa aplikacja.
8. Podejrzeć czy prawidłowo się generują blob-y (za pośrednictwem Stream Analytics Job). Dowolnym narzędziem (Visual Studio, Portal – tu – RedGate Azure Explorer, z Tools\setupAzureStorageExplorerCopy20160226.exe  
   

# Aplikacja UWP do wysyłania komunikatów do IoT Hub

1. Nowa aplikacja UWP
2. Dodać Windows Iot Extension
3. Dodać (NuGet): **Newtonsoft.Json**
4. Dodać (NuGet): **Microsoft.Azure.Devices.Client**
5. Dodać referencję z ADC Providers do ms-iot\ADC\AdcMcp3008\AdcMcp3008.csproj  
   Źródło: <https://github.com/ms-iot/BusProviders.git>
6. Pobrać plik z driverem dla BMP280 – nieznacznie zmodyfikowany (poprawić przestrzenie nazw itp.) <https://raw.githubusercontent.com/tkopacz/2016windowsiot-iothub-workshopV1/master/Test2_BMP280/Test2_BMP280/BMP280.cs>
7. Pobrać plik z driverem do TCS34725 (poprawić przestrzenie nazw itp.): <https://raw.githubusercontent.com/tkopacz/2016windowsiot-iothub-workshopV1/master/Test4_TCS34725/Test4_TCS34725/TCS34725.cs>
8. Dodać **Model.cs**, klasę opisującą komunikat wysyłany do IoT Hub

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MainIoTApp

{

public class MIoTBase

{

public DateTime Dt { get; set; }

public string MsgType { get; set; }

public string DeviceName { get; set; }

}

public class MSPI:MIoTBase

{

public double Potentiometer1 { get; set; }

public double Potentiometer2 { get; set; }

public double Light { get; set; }

}

public class MAll : MSPI

{

public double ADC3 { get; internal set; }

public double ADC4 { get; internal set; }

public double ADC5 { get; internal set; }

public double ADC6 { get; internal set; }

public double ADC7 { get; internal set; }

public float Altitude { get; internal set; }

public string ColorName { get; internal set; }

public ColorData ColorRaw { get; internal set; }

public RgbData ColorRgb { get; internal set; }

public float Pressure { get; internal set; }

public float Temperature { get; internal set; }

}

}

1. W pliku MainPage.xaml dodać “UI”. Składa się z:   
   TextBox **txtSPI** – określa jak często będzie odpytywany przetwornik ADC i wysyłany „mały” komunikat)  
   TextBox **txtAll** – określa jak często będzie odpytywany cały układ i wysyłany „duży” komunikat)  
   ToggleSwitch **tgSend** – włącza i wyłącza wysyłanie)  
   TextBlock **txtState** – pokazuje na ekranie stan działania (ostatnio wysłany komunikat, wyjątek itp.)  
     
   <Page

x:Class="MainIoTApp.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="using:MainIoTApp"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

mc:Ignorable="d">

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

<TextBlock x:Name="txtState" HorizontalAlignment="Left" Margin="0,133,0,0" TextWrapping="Wrap" Text="TextBlock" VerticalAlignment="Top" Height="200" Width="700" FontSize="13.333"/>

<TextBox x:Name="txtSPI" HorizontalAlignment="Left" Margin="266,73,0,0" TextWrapping="Wrap" Text="1000" VerticalAlignment="Top" FontSize="32" Width="166" TextChanged="txtSPI\_TextChanged" IsEnabled="False"/>

<ToggleSwitch x:Name="tgSend" Header="Send" HorizontalAlignment="Left" Margin="10,59,0,0" VerticalAlignment="Top" FontSize="21.333" Toggled="tgSend\_Toggled" IsEnabled="False" />

<TextBox x:Name="txtAll" HorizontalAlignment="Left" Margin="534,73,0,0" TextWrapping="Wrap" Text="5000" VerticalAlignment="Top" FontSize="32" Width="166" TextChanged="txtAll\_TextChanged" IsEnabled="False"/>

<TextBlock x:Name="textBlock" HorizontalAlignment="Left" Margin="474,77,0,0" TextWrapping="Wrap" Text="All :" VerticalAlignment="Top" FontSize="32"/>

<TextBlock x:Name="textBlock\_Copy" HorizontalAlignment="Left" Margin="198,77,0,0" TextWrapping="Wrap" Text="SPI :" VerticalAlignment="Top" FontSize="32"/>

</Grid>

</Page>

1. Podać parametry z łańcuchami połączeń do IoT Hub (partial class dla MainPage):  
   **DeviceId** – nazwa urządzenia (musi pasować) do SAS dla MQTT  
   **TKConnectionString** – łańcuch połączeń do IoT Hub  
   **TKConnectionMqtt** – nazwa DNS wskazująca na IoT Hub  
   **TKConnectionMqttPassword** – SAS wygenerowany dla DANEGO urządzenia używając DeviceExplorer  
   public sealed partial class MainPage : Page

{

const string DeviceId = "D01";

const string TKConnectionString = "HostName=pltkdpepliot2016.azure-devices.net;DeviceId=D01;SharedAccessKey=<własne>";

const string TKConnectionMqtt = "pltkdpepliot2016.azure-devices.net";

const string TKConnectionMqttUsername = TKConnectionMqtt + "/" + DeviceId;

const string TKConnectionMqttPassword = "SharedAccessSignature sr=pltkdpepliot2016.azure-devices.net&sig=….3d&se=1488102293&skn=device";

//const string TKConnectionMqttPassword = "SharedAccessSignature sr=pltkdpepliot2016.azure-devices.net&sig=…&se=1488062147&skn=iothubowner";

const string TKMqttTopicSend = "devices/"+ DeviceId + "/messages/events";

const string TKMqttTopicReceive = "devices/" + DeviceId + "/messages/devicebound";

}

1. Do pliku MainPage.xaml.cs i klasy MainPage dodać zmienne pomocnicze:  
   Uwaga! **MaxMsgCount** określa maksymalną liczbę komunikatów (małych i dużych) wysyłanych w danej sesji. Pozwala to uniknąć za szybkiego wyczerpania puli darmowych komunikatów w IoT Hub.   
      
    DispatcherTimer m\_t;

DispatcherTimer m\_tSPI;

GpioPin m\_blink;

GpioPinValue m\_blinkValue;

const float seaLevelPressure = 1013.25f;

BMP280 m\_bmp280;

AdcController m\_adc;

AdcChannel[] m\_adcChannel;

TCS34725 m\_tcs;

DeviceClient m\_clt;

MSPI m\_mSPI;

int m\_msgSpiCount = 0;

int m\_msgCount = 0;

int MaxMsgCount = 1000;

1. Inicjalizacja urządzeń:  
     
    public MainPage()

{

this.InitializeComponent();

setup();

}

private async void setup()

{

try

{

//0.IoTHub client

m\_clt = DeviceClient.CreateFromConnectionString(TKConnectionString, TransportType.Http1);

await m\_clt.SendEventAsync(new Message(new byte[] { 1, 2, 3 }));

Task.Run(() => ReceiveDataFromAzure()); //Loop.

//0. Cache for message

m\_mSPI = new MSPI();

m\_mSPI.DeviceName = DeviceId;

m\_mSPI.MsgType = "SPI";

//1. LED

var gpio = GpioController.GetDefault();

if (gpio != null)

{

m\_blink = gpio.OpenPin(26); //See board, connected to pin 19,

m\_blinkValue = GpioPinValue.High;

m\_blink.Write(m\_blinkValue);

m\_blink.SetDriveMode(GpioPinDriveMode.Output);

}

//2. BMP280

m\_bmp280 = new BMP280();

await m\_bmp280.Initialize();

//3. ADC

m\_adc = (await AdcController.GetControllersAsync(AdcMcp3008Provider.GetAdcProvider()))[0];

m\_adcChannel = new AdcChannel[m\_adc.ChannelCount];

for (int i = 0; i < m\_adc.ChannelCount; i++)

{

m\_adcChannel[i] = m\_adc.OpenChannel(i);

}

//4. TCS34725

m\_tcs = new TCS34725();

await m\_tcs.Initialize();

m\_t = new DispatcherTimer();

m\_t.Interval = TimeSpan.FromMilliseconds(5000);

m\_t.Tick += M\_t\_Tick;

m\_tSPI = new DispatcherTimer();

m\_tSPI.Interval = TimeSpan.FromMilliseconds(1000); //Caution - we have 8 000 messages / day. So - LIMIT RATE

m\_tSPI.Tick += M\_tSPI\_Tick;

//Can enable UI

txtAll.IsEnabled = txtSPI.IsEnabled = tgSend.IsEnabled = true;

tgSend.IsOn = true;

}

catch (Exception ex)

{

Debug.WriteLine(ex.ToString());

txtState.Text = ex.ToString();

}

}

1. Obsługa zegara wysyłającego stan przetwornika SPI  
   private async void M\_tSPI\_Tick(object sender, object e)

{

if (m\_msgSpiCount >= MaxMsgCount && MaxMsgCount != -1)

{

//No more than MaxMsgCount messages / run

m\_tSPI.Stop(); return;

}

m\_mSPI.Potentiometer1 = m\_adcChannel[0].ReadRatio();

m\_mSPI.Potentiometer2 = m\_adcChannel[1].ReadRatio();

m\_mSPI.Light = m\_adcChannel[2].ReadRatio();

m\_mSPI.Dt = DateTime.UtcNow;

var obj = JsonConvert.SerializeObject(m\_mSPI);

try

{

if (m\_clt!=null)

{

await m\_clt.SendEventAsync(new Message(System.Text.Encoding.UTF8.GetBytes(obj)));

m\_msgSpiCount++;

}

}

catch (Exception ex)

{

txtState.Text = ex.ToString();

}

}

1. Obsługa zegara wysyłającego komplet stanu odczytanego z dołączonych urządzeń  
    private async void M\_t\_Tick(object sender, object e)

{

if (m\_msgCount >= MaxMsgCount && MaxMsgCount != -1)

{

//No more than MaxMsgCount messages / run

m\_t.Stop(); return;

}

MAll m = new MAll();

m.DeviceName = DeviceId;

m.MsgType = "ALL";

m.Altitude = await m\_bmp280.ReadAltitudeAsync(seaLevelPressure);

m.Pressure = await m\_bmp280.ReadPreasureAsync();

m.Temperature = await m\_bmp280.ReadTemperatureAsync();

m.Potentiometer1 = m\_adcChannel[0].ReadRatio();

m.Potentiometer2 = m\_adcChannel[1].ReadRatio();

m.Light = m\_adcChannel[2].ReadRatio();

m.ADC3 = m\_adcChannel[3].ReadRatio();

m.ADC4 = m\_adcChannel[3].ReadRatio();

m.ADC5 = m\_adcChannel[3].ReadRatio();

m.ADC6 = m\_adcChannel[3].ReadRatio();

m.ADC7 = m\_adcChannel[3].ReadRatio();

m.ColorRgb = await m\_tcs.GetRgbDataAsync();

m.ColorRaw = await m\_tcs.GetRawDataAsync();

m.ColorName = await m\_tcs.GetClosestColorAsync();

m.Dt = DateTime.UtcNow;

var obj = JsonConvert.SerializeObject(m);

try

{ if (m\_clt != null)

{

await m\_clt.SendEventAsync(new Message(System.Text.Encoding.UTF8.GetBytes(obj)));

m\_msgCount++;

}

txtState.Text = obj + $", MSG:{m\_msgCount}, MSGSPI:{m\_msgSpiCount}";

}

catch (Exception ex)

{

txtState.Text = ex.ToString();

}

}

1. Obsługa UI (przyciski, zmiana częstotliwości działania timera itp.)  
    private void tgSend\_Toggled(object sender, RoutedEventArgs e)

{

if (tgSend.IsOn)

{

m\_t.Start();

m\_tSPI.Start();

}

else

{

m\_t.Stop();

m\_t.Stop();

}

}

private void txtSPI\_TextChanged(object sender, TextChangedEventArgs e)

{

double val;

if (double.TryParse(txtSPI.Text, out val) && val > 0)

{

m\_tSPI.Interval = TimeSpan.FromMilliseconds(val);

}

}

private void txtAll\_TextChanged(object sender, TextChangedEventArgs e)

{

double val;

if (double.TryParse(txtAll.Text, out val) && val > 0)

{

m\_t.Interval = TimeSpan.FromMilliseconds(val);

}

}

1. Obsługa zdarzeń (poleceń) odbieranych z chmury  
    public async Task ReceiveDataFromAzure()

{

Message receivedMessage;

string messageData;

if (m\_clt != null)

{

while (true)

{

try

{

receivedMessage = await m\_clt.ReceiveAsync();

if (receivedMessage != null)

{

messageData = System.Text.Encoding.ASCII.GetString(receivedMessage.GetBytes());

//Wykonanie polecenia

if (messageData.Length >= 2)

{

await this.Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>

{

double val;

switch (messageData[0])

{

case 'L':

//Light

if (messageData[1] == '0')

m\_blinkValue = GpioPinValue.High;

else

m\_blinkValue = GpioPinValue.Low;

m\_blink.Write(m\_blinkValue);

break;

case 'A':

//All messages - interval

if (double.TryParse(messageData.Substring(1), out val))

{

txtAll.Text = val.ToString();

}

break;

case 'S':

//SPI messages - interval

if (double.TryParse(messageData.Substring(1), out val))

{

txtSPI.Text = val.ToString();

}

break;

case 'O':

//On / Off

if (messageData[1] == '0')

tgSend.IsOn = false;

else

tgSend.IsOn = true;

break;

default:

break;

}

});

}

//await m\_clt.RejectAsync(receivedMessage);

//await m\_clt.AbandonAsync(receivedMessage); - reject, will be redelivered

//Confirm

await m\_clt.CompleteAsync(receivedMessage); //potwierdza odebranie

}

}

catch (Exception ex)

{

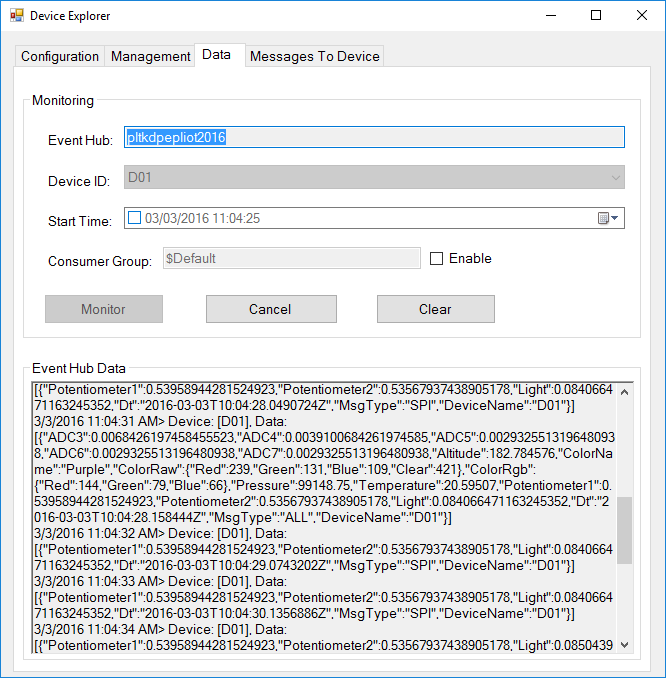
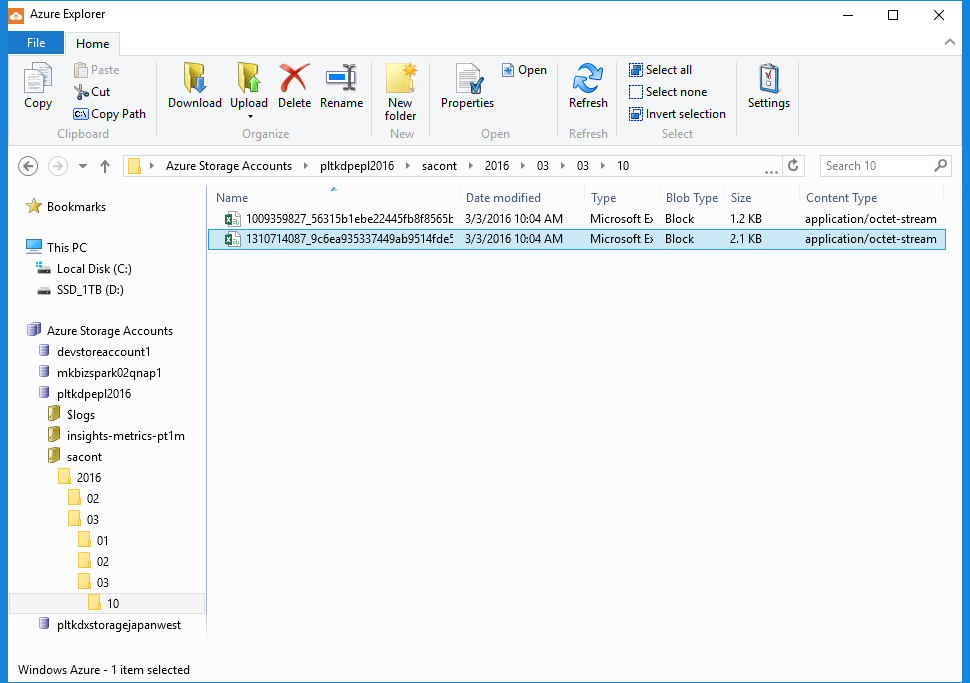
txtState.Text = ex.ToString();

}

}

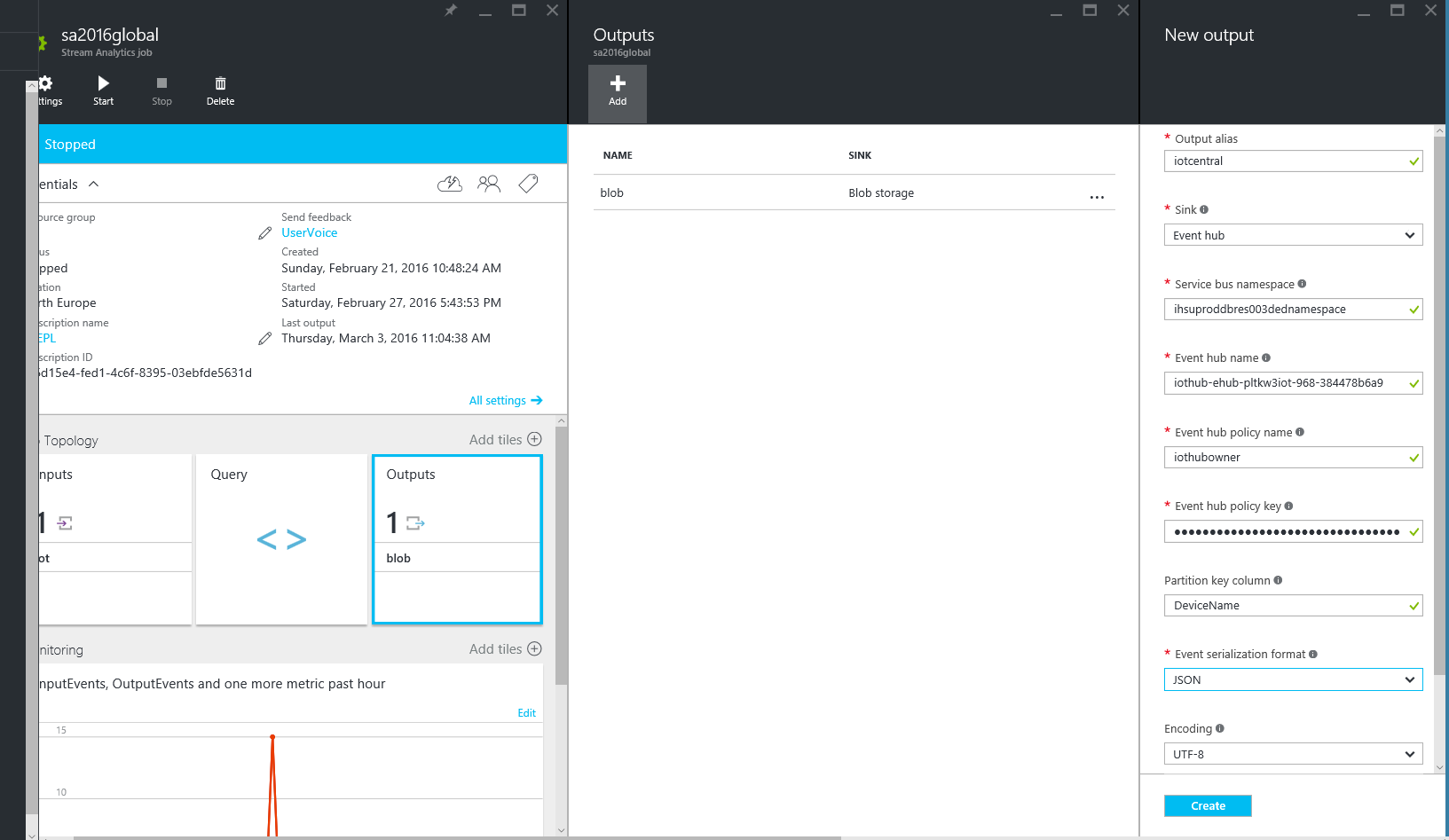
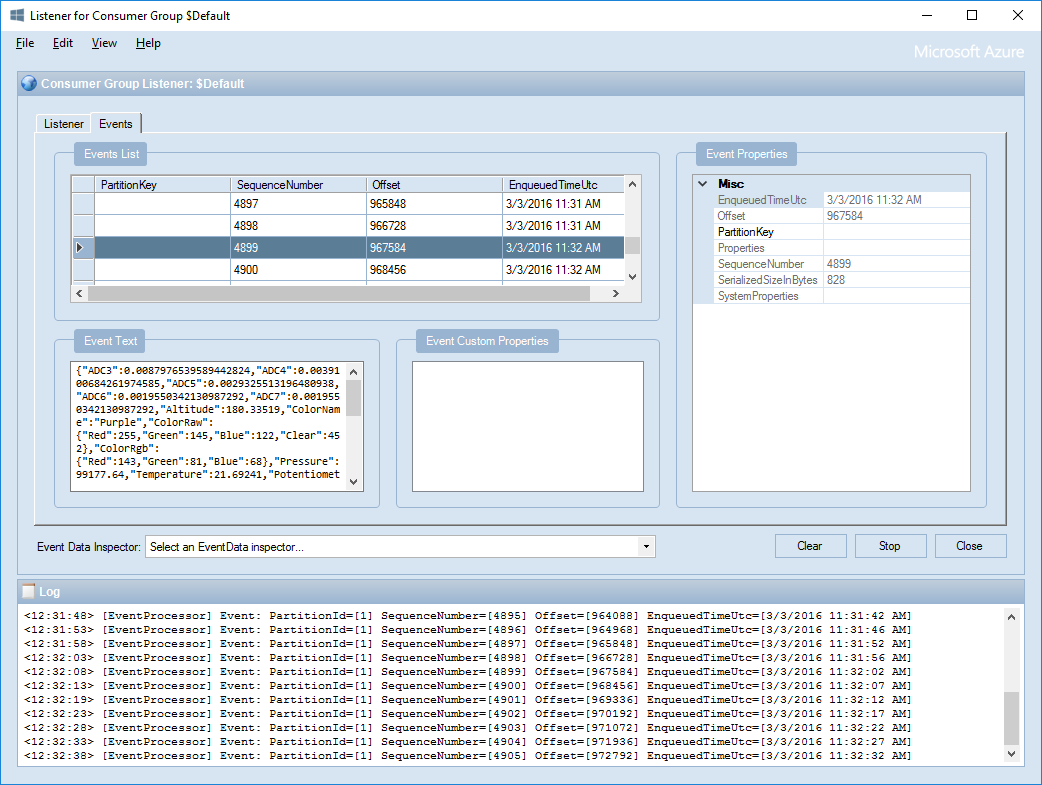
}

}

1. Uruchomić aplikację
2. Podejrzeć zdarzenia dla danego urządzenia używając DeviceExplorer  
   
3. Zobaczyć że zdarzenia zostały przetworzone przez Stream Analytics  
   
4. Sprawdzić że działa wysyłanie komunikatów do urządzenia i np. włączenie / wyłączenie wysyłania

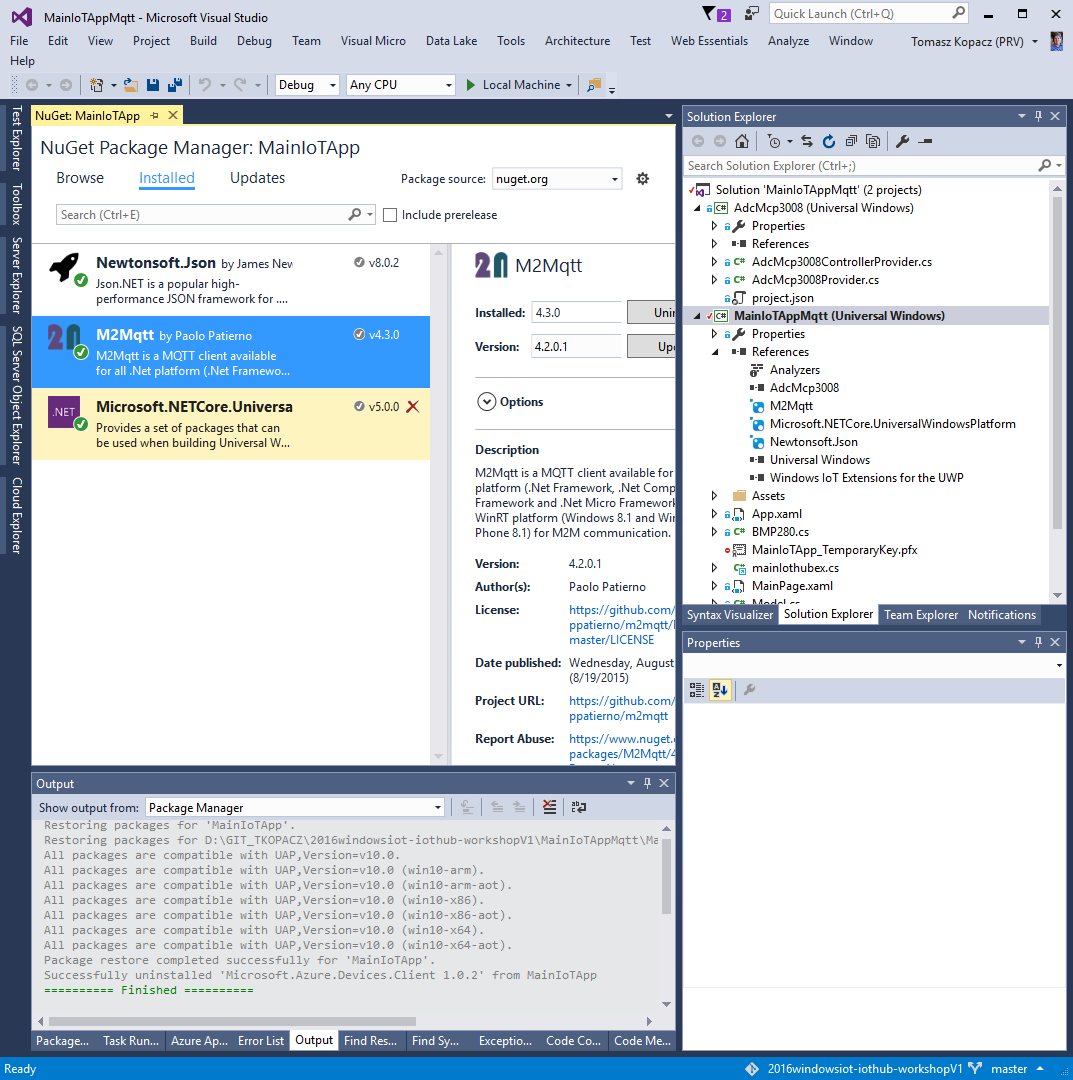
# Dodanie do Stream Analytics dodatkowego wyjścia - centralnego „Event Hub”

Cel – zebrać w jednym miejscu informacje ze wszystkich urządzeń.

1. Zatrzymać Stream Analytics Job
2. Dodać dodatkowe wyjście o nazwie iotcentral, typu Event Hub  
   Używamy API Event Hub. Parametry podane są w innym dokumencie!  
   
3. Zmodyfikować kwerendę, tak by były 2 wyrażenia (wysyłamy do drugiego IoT tylko wiadomości ALL):  
   SELECT \* INTO [blob] FROM [iot]  
   **SELECT \* INTO [iotcentral] FROM [iot] where MsgType='ALL'**
4. Uruchomić kwerendę (i wysyłanie zdarzeń). Po pewnym czasie prowadzący powinien widzieć w swoim Event Hub zdarzenia z sali.  
   
5. Dygresja: można by użyć IoT Hub – pod warunkiem ponownej rejestracji urządzeń itp.

# Aplikacja UWP do wysyłania komunikatów do IoT Hub – MQTT

Uwaga! Na UWP albo używamy IoT Hub albo biblioteki MQTT – nie można tego połączyć

1. Proszę skopiować do innego foldera całe rozwiązanie które powstało w ramach [Aplikacja UWP do wysyłania komunikatów do IoT Hub](#_Aplikacja_UWP_do)
2. Usunąć referencję (NuGet) do **Microsoft.Azure.Devices.Client**
3. Dodać referencję (NuGet) do **M2Mqtt** (biblioteka dla Mqtt). Referencje powinny wyglądać jak:  
   
4. Usunąć odwołania do Azure IoT Hub SDK – m\_clt, Message  
   ~~DeviceClient m\_clt;~~  
     
   ~~using Microsoft.Azure.Devices.Client;~~  
   …
5. Dodać odpowiednie parametry połączeń do MQTT(partial class dla MainPage). Więcej informacji: <https://azure.microsoft.com/en-us/documentation/articles/iot-hub-mqtt-support/>   
   Podać należy:  
   **TKMqttDeviceId** – nazwa urządzenia (musi pasować) do SAS dla MQTT  
   **TKConnectionString** – łańcuch połączeń do IoT Hub – tu nie jest potrzebny  
   **TKConnectionMqtt** – nazwa DNS wskazująca na IoT Hub  
   **TKConnectionMqttPassword** – SAS wygenerowany dla DANEGO urządzenia używając DeviceExplorer.

/// <summary>

/// Additional information - passwords etc

/// Enter correct values!:

/// </summary>

public sealed partial class MainPage : Page

{

const string TKMqttDeviceId = "D01";

const string TKConnectionMqtt = "pltkdpepliot2016.azure-devices.net";

const string TKConnectionMqttUsername = "pltkdpepliot2016.azure-devices.net/" + TKMqttDeviceId;

//SAS:

const string TKConnectionMqttPassword = "SharedAccessSignature sr=pltkdpepliot2016.azure-devices.net&sig=<enter>";

const string TKMqttTopicSend = "devices/" + TKMqttDeviceId + "/messages/events";

}

1. Zmienić sposób inicjowania komunikacji w setup:  
   private async void setup()

{

try

{

**//0. MQQT for IoT Hub, uPLibrary.Networking.M2Mqtt**

**m\_mqtt = new MqttClient(TKConnectionMqtt, 8883, true, MqttSslProtocols.TLSv1\_2);**

**m\_mqtt.Connect(DeviceId, TKConnectionMqttUsername, TKConnectionMqttPassword);**

**if (m\_mqtt.IsConnected == false) throw new ArgumentException("Bad username/password for MQTT");**

//0. Cache for message

m\_mSPI = new MSPI();

m\_mSPI.DeviceName = DeviceId;

m\_mSPI.MsgType = "SPI";

1. Zmienić sposób wysyłania komunikatów z ADC:  
     
   private async void M\_tSPI\_Tick(object sender, object e)

{

if (m\_msgSpiCount >= MaxMsgCount && MaxMsgCount != -1)

{

//No more than MaxMsgCount messages / run

m\_tSPI.Stop(); return;

}

m\_mSPI.Potentiometer1 = m\_adcChannel[0].ReadRatio();

m\_mSPI.Potentiometer2 = m\_adcChannel[1].ReadRatio();

m\_mSPI.Light = m\_adcChannel[2].ReadRatio();

m\_mSPI.Dt = DateTime.UtcNow;

var obj = JsonConvert.SerializeObject(m\_mSPI);

try

{

**if (m\_mqtt != null)**

**{**

**//Publish to internal queue; then - background process send messages to IoT Hub**

**m\_mqtt.Publish(TKMqttTopicSend, System.Text.Encoding.UTF8.GetBytes(obj));**

**m\_msgSpiCount++;**

**}**

}

catch (Exception ex)

{

txtState.Text = ex.ToString();

}

}

1. Zmienić sposób wysyłania pozostałych komunikatów:  
     
    private async void M\_t\_Tick(object sender, object e)

{

if (m\_msgCount >= MaxMsgCount && MaxMsgCount != -1)

{

//No more than MaxMsgCount messages / run

m\_t.Stop(); return;

}

MAll m = new MAll();

m.DeviceName = DeviceId;

m.MsgType = "ALL";

m.Altitude = await m\_bmp280.ReadAltitudeAsync(seaLevelPressure);

m.Pressure = await m\_bmp280.ReadPreasureAsync();

m.Temperature = await m\_bmp280.ReadTemperatureAsync();

m.Potentiometer1 = m\_adcChannel[0].ReadRatio();

m.Potentiometer2 = m\_adcChannel[1].ReadRatio();

m.Light = m\_adcChannel[2].ReadRatio();

m.ADC3 = m\_adcChannel[3].ReadRatio();

m.ADC4 = m\_adcChannel[3].ReadRatio();

m.ADC5 = m\_adcChannel[3].ReadRatio();

m.ADC6 = m\_adcChannel[3].ReadRatio();

m.ADC7 = m\_adcChannel[3].ReadRatio();

m.ColorRgb = await m\_tcs.GetRgbDataAsync();

m.ColorRaw = await m\_tcs.GetRawDataAsync();

m.ColorName = await m\_tcs.GetClosestColorAsync();

m.Dt = DateTime.UtcNow;

var obj = JsonConvert.SerializeObject(m);

try

{

**if (m\_mqtt != null)**

**{**

**//Publish to internal queue; then - background process send messages to IoT Hub**

**m\_mqtt.Publish(TKMqttTopicSend, System.Text.Encoding.UTF8.GetBytes(obj));**

**m\_msgSpiCount++;**

**}**

txtState.Text = obj + $", MSG:{m\_msgCount}, MSGSPI:{m\_msgSpiCount}";

}

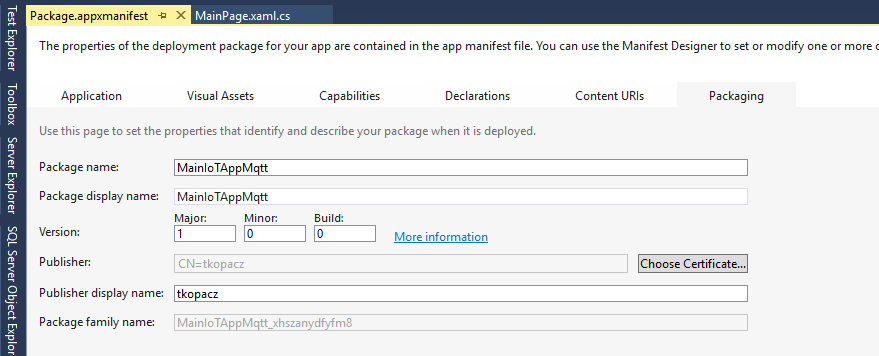
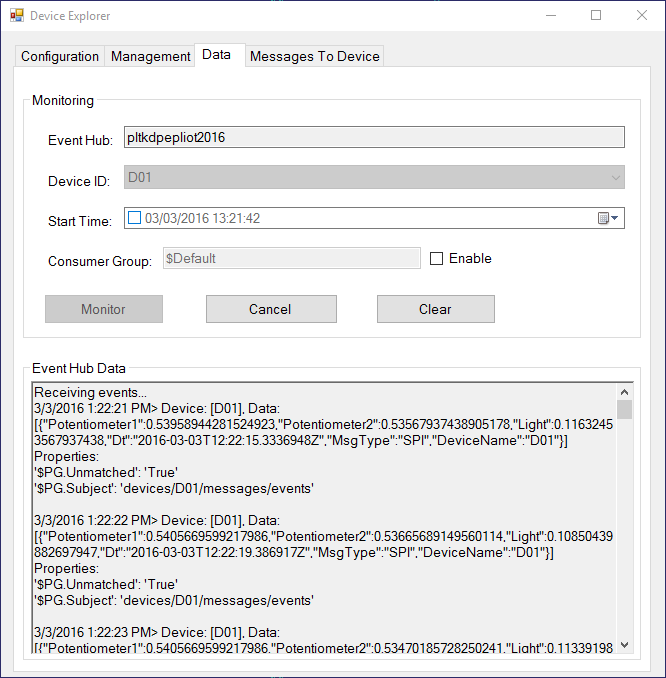
catch (Exception ex)

{

txtState.Text = ex.ToString();

}

}

1. Usunąć metodę public async Task ReceiveDataFromAzure()
2. Dla wygody, można zmienić nazwę pakietu (łatwiej wtedy znaleźć go w liście procesów do uruchomienia na RPi2.  
   
3. Wgrać rozwiązanie na RPi2
4. Zobaczyć, w DeviceExplorer / Stream Analytics, jak wysyłane są wiadomości  
   
5. Proszę zauważyć, że tym razem – dodatkowo są pewne właściwości (wynikające z działania i implementacji protokołu MQTT

# Własna aplikacja do odbierania i przetwarzania wiadomości z IoT Hub

1. Dodać nową aplikację konsolową ConsoleMonitor.
2. Dodac referencję (NuGet) do Microsoft.Azure.Devices, Microsoft.Azure.Amqp, Microsoft.AspNet.WebApi.Client
3. W funkcji Main dodać proste pobieranie parametrów z linii poleceń:  
     
    Console.WriteLine("Usage: \r\n" +  
    "ConsoleMonitor <iotHubOwner> [<consumerGroup>] [<device>] ");

string connection = args[0];

string consumerGroupName = "$Default";

if (args.Length>1) consumerGroupName = args[1];

string deviceName = "";

if (args.Length>2) deviceName = args[2];

1. W funkcji Main dodać nawiązywanie połączeń z IoT Hub:  
     
    EventHubClient eventHubClient = null;  
    EventHubReceiver eventHubReceiver = null;  
    eventHubClient = EventHubClient.CreateFromConnectionString(connection, "messages/events");
2. Pobranie informacji (między innymi) o ilości partycji:  
   var ri = eventHubClient.GetRuntimeInformation();
3. Jeżeli jako parametr została przekazana nazwa urządzenia, monitorujemy tylko tą partycję z IoT Hub/Event Hub do której komunikaty są wysyłane (ResolveToPartition zwraca nazwę partycji dla danej nazwy urządzenia):

if (deviceName != "")

{

string partition = EventHubPartitionKeyResolver.ResolveToPartition(deviceName, ri.PartitionCount);

eventHubReceiver = eventHubClient.GetConsumerGroup(consumerGroupName).CreateReceiver(partition, DateTime.Now);

Task.Run(() => eventLoop(eventHubReceiver));  
}

1. W przeciwnym wypadku, startujemy oddzielne wątki do monitorowania każdej partycji. Uwaga! Zwykle to będą oddzielne węzły klastra, oddzielne „spouts” ze Storm – albo – aktorzy z Service Fabric. Zakładamy też, że interesują nas zdarzenia od „teraz”.  
     
   else {

EventHubReceiver[] eventHubReceivers = new EventHubReceiver[ri.PartitionCount];

int i = 0;

foreach (var partition in ri.PartitionIds) {

eventHubReceivers[i] = eventHubClient.GetConsumerGroup(consumerGroupName).CreateReceiver(  
 partition, DateTime.Now);  
 //Task.Run(() => eventLoop(eventHubReceivers[i])); <- very common bug!

var r = eventHubReceivers[i];

Task.Run(() => eventLoop(r));

i++;

}  
 }

1. Ponieważ aplikacja uruchamia wątki w tle, trzeba jeszcze dodać:  
   Console.ReadLine();

(na końcu funkcji Main)

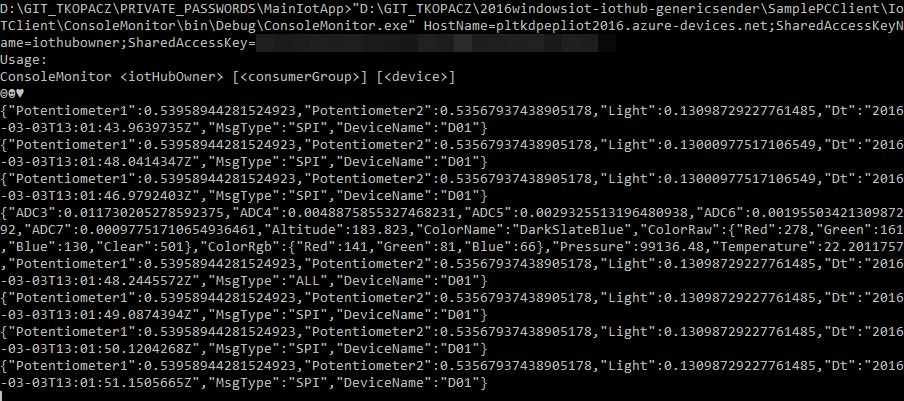
1. Procedura przetwarzania wiadomości z danej partycji jest dosyć oczywista:  
     
   private static async Task eventLoop(EventHubReceiver eventHubReceiver) {  
   while (true) {  
    var edata = await eventHubReceiver.ReceiveAsync();  
    if (edata != null) {  
    var data = Encoding.UTF8.GetString(edata.GetBytes());

Console.WriteLine(data);

}

}

}

1. Proszę uruchomić plik używając linii poleceń. Na przykład:  
   "D:\GIT\_TKOPACZ\2016windowsiot-iothub-genericsender\SamplePCClient\IoTClient\ConsoleMonitor\bin\Debug\ConsoleMonitor.exe" HostName=pltkdpepliot2016.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=klucz(własny)
2. Proszę uruchomić aplikację wysyłającą sygnały z RPi.
3. Po chwili powinniśmy zobaczyć ciąg zarejestrowanych zdarzeń  
   
4. Pełny kod rozwiązania:  
     
   using Microsoft.Azure.Devices.Common;

using Microsoft.ServiceBus.Messaging;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleMonitor

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Usage: \r\n" +

"ConsoleMonitor <iotHubOwner> [<consumerGroup>] [<device>] ");

string connection = args[0];

string consumerGroupName = "$Default";

if (args.Length>1) consumerGroupName = args[1];

string deviceName = "";

if (args.Length>2) deviceName = args[2];

EventHubClient eventHubClient = null;

EventHubReceiver eventHubReceiver = null;

eventHubClient = EventHubClient.CreateFromConnectionString(connection, "messages/events");

var ri = eventHubClient.GetRuntimeInformation();

if (deviceName != "")

{

string partition = EventHubPartitionKeyResolver.ResolveToPartition(deviceName, ri.PartitionCount);

eventHubReceiver = eventHubClient.GetConsumerGroup(consumerGroupName).CreateReceiver(partition, DateTime.Now);

Task.Run(() => eventLoop(eventHubReceiver));

} else

{

EventHubReceiver[] eventHubReceivers = new EventHubReceiver[ri.PartitionCount];

int i = 0;

foreach (var partition in ri.PartitionIds)

{

eventHubReceivers[i] = eventHubClient.GetConsumerGroup(consumerGroupName).CreateReceiver(partition, DateTime.Now);

//Task.Run(() => eventLoop(eventHubReceivers[i])); <- very common bug!

var r = eventHubReceivers[i];

Task.Run(() => eventLoop(r));

i++;

}

}

Console.ReadLine();

}

private static async Task eventLoop(EventHubReceiver eventHubReceiver)

{

while (true)

{

var edata = await eventHubReceiver.ReceiveAsync();

if (edata != null)

{

var data = Encoding.UTF8.GetString(edata.GetBytes());

Console.WriteLine(data);

}

}

}

}

}