OAMK
OULU UNIVERSITY OF APPLIED SCIENCES

Bereket Godebo

INTERFACING ARDUINO IN THE UNIVERSAL WINDOWS PLATFORM

INTERFACING ARDUINO IN THE UNIVERSAL WINDOWS PLATFORM

Bereket Godebo
Bachelor’s Thesis
Fall 2016
Degree Program in Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Embedded Systems

Author: Bereket Godebo
Title of the Bachelor's thesis: Interfacing Arduino in the Universal Windows Platform
Supervisors: Eino Niemi, Pekka Alaluukas
Term and year of completion: Fall 2016 Number of pages: 48 + 1 appendix

The objective of this Bachelor's thesis was to connect Raspberry Pi 2 running Windows 10 IoT Core via a USB to Arduino. Moreover, the aim was to delegate Arduino the work of controlling sensors, and to send data to Raspberry PI 2 whenever needed. This project was part of a Home Automation project created by my Instructor Pekka Alaluukas as a hobby and the devices used in this project were provided by the School of Engineering.

In the process of implementation, the project work was divided into tasks such as, developing a serial module on Raspberry Pi 2, developing its counterpart on Arduino and developing a protocol to be used for synchronizing the communication. And then it was proceeded incrementally by adding a functionality as required on the working modules.

Future improvements can be made in areas such as creating a Windows runtime component that connects to Arduino, supporting a cancelation of asynchronous operations and throwing an exception for an error that occurs in the chain of task instead of showing it on the screen.

Keywords: IoT, UWP, C++/CX, Asynchronous, RPi2, Arduino, SQLite3

PREFACE

This thesis project was conducted at Oulu University of Applied Sciences, School of Engineering campus and the devices needed were provided by the school.

I would like to thank Riitta Rontu, Head of the Department, for facilitating the thesis work and for her sincere support for me to complete the degree program in time.

I would also like to thank my supervisor Eino Niemi for his willingness to be flexible, as I have done this project during the summer break, and for being supportive.

I would not have done this thesis project if it was not for Pekka Alaluukas, I Thank him for the good ideas.

Last but not least, I would like to thank my wife for giving me such a wonderful daughter who has been an inspiration every day. And also my families who have continued to believe in me during the long years of studying in Finland.

Oulu, 16.08.2016

Bereket Hizkeal Godebo

TABLE OF CONTENTS

ABSTRACT 3

PREFACE 4

TABLE OF CONTENTS 5

LIST OF FIGURES AND TABLES 6

ABBREVIATIONS 7

1 INTRODUCTION 8

2 BACKGROUND 10

- 2.1 Universal Windows Platform 10
- 2.2 Asynchronous Model in C++/CX 11
- 2.3 Namespaces for Connecting 12
- 2.4 Serial On Arduino 13
- 2.5 AVR-Toolchain 14
- 2.6 Firmata Protocol 15
- 2.7 Custom Protocol 16

3 DEVICES 17

- 3.1 Raspberry Pi 17
- 3.2 Arduino 18
- 3.3 Sensor-AM2301 19

4 IMPLEMENTATION 21

- 4.1 Chain of Tasks 21
- 4.2 Get Arduino 22
- 4.3 Processing Input On Arduino 25
- 4.4 DHT-Lib 25
- 4.5 Process Input On RPi2 28
- 4.6 Save into SQLite 31

5 CONCLUSION 33

REFERENCES 34

LIST OF FIGURES AND TABLES

FIGURE 1. functional block diagram ---8

FIGURE 2. rpi2 model b with a 900mhz quad-core arm cortex-a7 cpu and 1gb ram (11)---16

FIGURE 3. arduino uno r3 based on atmega328p-------------------------------------17

FIGURE 4. am2301 also called dht21. -source datasheet---------------------------18

FIGURE 5. am2301 pin diagram and pin description -source datasheet---------18

FIGURE 6. am2301 data transmission format -source datasheet------------------19

FIGURE 7. am2301 1-Wire timing diagram -source datasheet---------------------19

FIGURE 8. chain of tasks in the program--20

FIGURE 9. arduino device property in device manager-----------------------------21

FIGURE 10. sqlite for uwp installation---30

FIGURE 11. adding reference for sqlite library---31

TABLE 1. firmata protocol description (10).--15

TABLE 2. custom protocol description---16

TABLE 3. am2301 timing description -source datasheet----------------------------20

ABBREVIATIONS

RPi2:	Raspberry Pi 2
APIs:	Application Programming Interfaces
UWP:	Universal Windows Platform
WinRT:	Windows Runtime
USB:	Universal Serial Bus
USART:	Universal Synchronous Asynchronous Receive Trans-mit
USB CDC:	USB Communication Device Class
SDKs:	Software Development Kits
STA:	Single-Threaded Apartment
TTL:	Transistor-Transistor-Logic
VID:	Vendor ID
PID:	Product ID

1 INTRODUCTION

The purpose of this Bachelor's thesis was to communicate from a Windows 10 host to a microcontroller based prototyping platform, Arduino Uno and be able to receive digital data from a sensor attached to it. This project was part of a home automation system, which e.g. makes it easy for a user to monitor remotely Humidity and Temperature values from a mobile device.

The original requirements of this thesis project were given by the author's instructor Pekka Alaluukas and designed by Eino Niemi, the supervisor together with the author.

The Arduino was to be connected to a RPi2 running Windows 10 IoT Core via a USB Connection. In addition, the sensor was to be connected to a digital pin in Arduino and to send serial data using a 1-Wire protocol.

In the process of implementation, first the APIs provided by the Windows 10 IoT Core to connect to a device attached via a USB were studied. Since a USB is an industry standard for communicating over a serial bus, the Arduino programming language was studied to determine what functions are provided by the platform that expose the serial port as a USB. Moreover, as Arduino is a microcontroller based platform as such serial communication module is a built-in module. Finally, to have a full control of the transmission of data from the sensor to the Windows 10 host, the available Arduino library that provided an API to receive sensor data was studied.

This thesis project was carried out by exploring the technical details of the existing library, the Firmata protocol implementation by Microsoft, which abstracts the use of a microcontroller for Windows developers. The effort was to increase the author's competence in the hybrid field that combines the experience of embedded system development and that of Internet Service development.

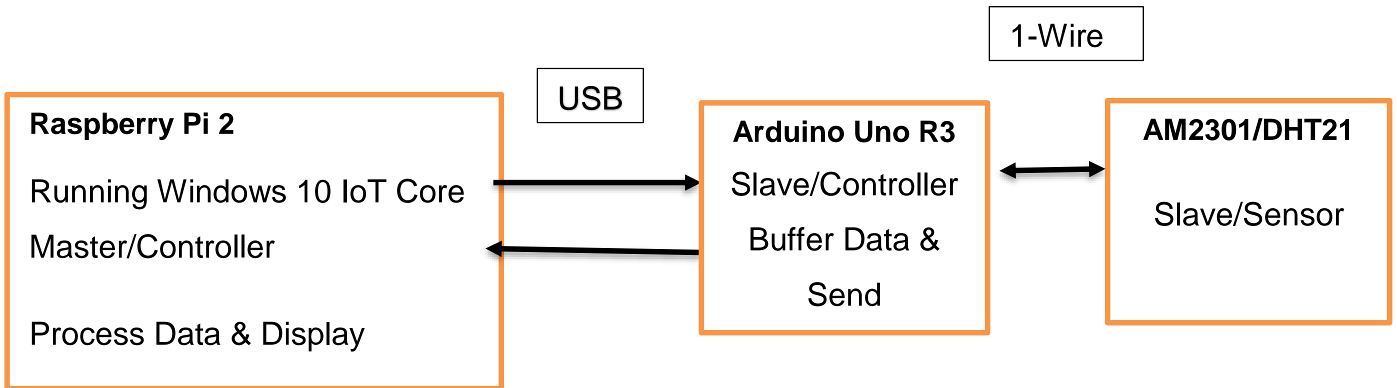

FIGURE 1. functional block diagram

2 BACKGROUND

The connection between the RPi2 and the Arduino comprises on the Arduino side a UART module which is exposed as a virtual Com port. This abstraction is implemented by using an internal USB-to-Serial bridge module (1). While on the RPi2 side, the new extension to the Windows Runtime, the Universal Windows Platform, provides an API to be able to find Arduino or a similar device with its Vendor ID and Product ID and then get a reference to it. Finally, it helps configure the connection parameters to fine tune the flow control as desired.

Therefore, this chapter briefly presents the various technologies that were discovered after a careful consideration of different sources relevant to the subject of the thesis.

2.1 Universal Windows Platform

UWP is the result of a unified core. Microsoft has undergone a major transformation that changed the way developers write programs, and use the development tools, and most importantly the way they interact with the underlying runtime environment. The result is the convergence of a program model, which in effect created a platform where the operating system maintains a set of APIs across all sets of hardware platforms while still providing flexibility by way of extensions on a specific set of hardware platforms called Device Family (2).

UWP is then the combination of APIs that are available in all device families and the extension SDKs that are available to a specific device family. Moreover, by developing a Universal Windows Platform apps, it is possible to target a specific device family or a universal device family, which contains all device families.

In this project, the RPi2 runs Windows 10 IoT Core, and as explained above, it contains APIs that are available on a universal device family and also the IoT extension SDKs that are used for the IoT specific functionality. Since the APIs

that are used in the implementation of this project take a significant amount of time, the UWP's Asynchronous model was used.

2.2 Asynchronous Model in C++/CX

An asynchronous model is the way the underlying runtime abstracts the execution of a routine that takes time to complete, to run at the same time with the thread of an execution that invokes it. This facilitates mainly the responsiveness of the UI thread while waiting for an operation to complete.

The UI of a UWP app runs in a Single-Threaded Apartment (3). This approach eliminates the need to lock a resource that is shared between threads, for the purpose of synchronization. The use of an asynchronous model, the implementation of which is the ***task*** class in the concurrency namespace, makes it possible to create a task from within the STA. And upon completion, the continuation of it is guaranteed to run in the STA resulting in a simplified access to the UI elements (3). Of course, the task object is created using the UWP's methods that return an asynchronous operation or action. The return type of these methods implement the interfaces ***IAsyncOperation<TResult>*** and ***IAsyncAction***, and their variations which support a progress report. Therefore, an operation returns a result whereas an action returns void (4.)

In this project a ***task*** object was created using a ***create_task*** Function from the same namespace, for its ease of use, since it allows the use of an ***auto*** keyword while creating tasks (5). The ***task*** class has a method ***task::then***, which can be called on the parent task object, and which returns a continuation task that is executed after the completion of the parent task.

2.3 Namespaces for Connecting

The ***Windows.Devices.SerialCommunication*** namespace defines a WinRT class which can be used to communicate with USB devices. Currently, the namespace supports USB devices within the communication device class -USB CDC (1).

USB is an industry standard for a communication between computers and electronic devices. It is categorized in classes to allow the USB host to recognize the USB device without the need for vendor specific drivers (6.) Moreover, as the name implies, the communication device class is a class that supports communication and networking functions, and devices in this class include e.g. Com-port devices. Since Arduino Uno R3 used in this project has an internal USB to a Serial bridge chip that exposes the UART module as a virtual Com-Port, the WinRT class defined in the ***SerialCommunication*** namespace can be used to connect to Arduino.

The static methods from this namespace that are used in this project are: `SerialDevice::FromIdAsync(Platform::String^)` and `SerialDevice::GetDeviceSelectorFromUsbVidPid(unsigned int, unsigned int)`. The first method returns a serial device object given the path that specifies the location of the device within the system. The second one takes as an argument, the VID and PID of a device and returns a string called an Advanced Query Syntax, which is used as a search criterion to find a given device from all serial devices in the system (1.)

In order to access a serial device, a UWP app must specify in its package manifest the name of the capability it uses. This helps to identify the device and its purpose for its users (1.)

The APIs in the ***Windows.Devices.Enumeration*** and ***Windows.Strorage.Streams*** namespace are used as a support class in the process of connecting to the serial device. In the Enumeration namespace there is only one

static method that is needed for this project: `DeviceInformation::FindAllAsync(Platform::String^)`. This method returns a collection of objects that describe connected devices based on the given search criterion, which is described previously. The Streams namespace is used after the serial object is obtained, and the two classes which are needed from this namespace are: DataReader and DataWriter. A receiver and a transmitter object are created using these classes, they are initialized to the input stream and output stream of the obtained serial object, respectively.

2.4 Serial On Arduino

All Arduino boards have at least one Universal Synchronous Asynchronous Receive Transmit module which is exposed as a serial port. Moreover, the serial communication is accomplished through the digital pin 0 as RX and 1 as TX that use TTL-compatible logic levels as well as through USB by using a bridging chip which was described on page 10 (7.)

In the Arduino programming language, the two important functions are: ***setup()*** and ***loop()*** (8)***.*** The first one is analogous to a constructor for an object. It is only called once in the beginning or when the board is reset. The second one is where the main program logic is written and executed forever.

The built in ***Serial*** library is used to configure and connect to the serial port. This library provides several functions that help in connecting to the device and in sending information as desired. In this project, only four of the functions are used: begin(), available(), read() and write(). As Arduino is microcontroller based, configuring its serial module only requires calling the Serial.begin(baudrate) in the set-up() function.

The member function ***Serial.begin(long)*** which was mentioned above, uses a default value of 8-N1 for the connection parameters, that is data-bits:8, parity: None, stop-bits:1 respectively. The overload of this function takes an additional parameter to change the default values in the frames sent or received.

The second member function, ***Serial.available(),*** returns the number of bytes available for reading, in other words the number of bytes in the receive buffer that has not yet been read. Usually, this function is used together with ***Serial.read()*** to get more bytes.

Finally, when Arduino sends back data, it uses the following two functions ***Serial.write(unsigned char)*** and ***Serial.write(buf, len).*** The second one sends an array of bytes, buf and of size, len.

2.5 AVR-Toolchain

This topic is being addressed here to simplify the explanation of the DHT library used to read the digital sensor data. The library was modified in a way that gives more control on the transmission of data from the sensor via Arduino to RPi2.

The avr-toolchain is based on the GNU Compiler Collection, and as the name implies GCC is a collection of compilers for different languages. One important feature of GCC is that it can use other programs, in a way that chains the output of one into the other and creates a final output (9). The GNU Assembler and GNU Linker are the other programs that work with GCC, and are part of another open-source project called GNU Binutils. When both these open source projects are built to execute on a host system such as Linux, or Windows, and built to generate code for the avr-microcontroller target, then they are given a prefix 'avr-' and as such gcc becomes avr-gcc, GNU assembler becomes avr-as, and GNU linker become avr-ld. Accordingly, the Standard C library has a version of it called avr-libc, which includes many of the standard library functions as well as those that are specific to AVR. Therefore, among others, these open source programs form a chain called AVR-Toolchain (9.)

2.6 Firmata Protocol

The protocol uses an MIDI message format (10). However, the message data is interpreted in a way that facilitates the raw data transmission between a microcontroller and a host machine. Furthermore, the protocol has a 4-bit resolution for transmission of additional data that describe the port or pin number. Hence, it has the constrain on the support of a maximum of 16 analog pins and 16 digital ports (ports are a group of 8-bits in an 8-bit architecture, the total number of digital pins is 16*8= 128 pins)

TABLE 1. firmata protocol description (10).

Type	Command	MIDI Channel	First byte	Second byte
analog I/O message	0xE0	pin #	LSB(bits 0-6)	MSB(bits 7-13)
digital I/O message	0x90	port	LSB(bits 0-6)	MSB(bits 7-13)
report analog pin	0xC0	pin #	disable/enable(0/1)	-N/A-
report digital port	0xD0	port	disable/enable(0/1)	-N/A-

In this protocol, the first byte contains a command to be sent or received together with a pin number for analog I/O or a port number for digital I/O. For instance, when sending a ***report analog pin*** command: 0xC7, this means send the value of an analog pin number 7. This is not the whole protocol description but only the part that is relevant to this project.

2.7 Custom Protocol

This is the protocol that is used in this project. It is based on the Firmata protocol described earlier in this document (p.15). The main difference is, the number of data message bytes that are supported. Here, the number is not fixed to two bytes only, it goes up to 5 bytes. Furthermore, commands that start with REPORT are sent from the host machine and commands that start with DIGITAL or ANALOG are sent from the board as a response. Only response commands have additional data with the command. Moreover, the first nibble in the first byte sent is the command and the second nibble is the additional data.

TABLE 2. custom protocol description

Type	Command	Additional Data	Byte1	Byte2	Byte3	Byte4	Byte5
Report Digital Data	0XA0		-	-	-	-	-
Digital Data	0XB0	data-status(0-2)	byte0	byte1	byte2	byte3	byte4
Report Analog Data	0XC0		-	-	-	-	-
Analog Data	0XD0	pin #	byte0	byte1	-	-	-

For example, a Digital Data response command: 0xB2 is interpreted as a digital message and data with a status of 2, which is Time-out Error.

3 DEVICES

In this chapter devices that are used in this project and their capabilities pertaining to the project are discussed briefly.

3.1 Raspberry Pi

RPi2 is a series of low-cost single-board computers which have gained popularity among hobbyists. They are also used to simplify the teaching of Computer Science in schools and developing countries (11).

FIGURE 2. rpi2 model b with a 900mhz quad-core arm cortex-a7 cpu and 1gb ram (11)

The board features among others several communication interfaces, 4 USB ports, a Full HDMI port, and an Ethernet Port. Furthermore, it has enough horse power to run all of ARM based GNU/Linux distributions as well as Microsoft Windows 10 IoT Core, which is used in this project (12).

3.2 Arduino

Arduino is an open-source single-board microcontroller which is based on Wiring which is an open-source electronic prototyping platform. The wiring project was inspired by a previous similar project called Processing which is again an open-source computer programming language and an integrated Development Environment that promotes computer programming in a visual context for artists (13).

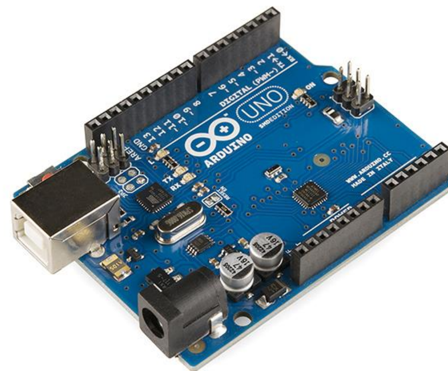

FIGURE 3. arduino uno r3 based on atmega328p (13)

Since the board is based on the ATmega328P avr-microcontroller from Atmel, the Arduino IDE uses GNU avr-toolchain to compile a code and to program the microcontroller. Hence the avr-libc libraries can be used while writing Arduino programs, also called sketches. Moreover, for those who want to have more access to the underlying microcontroller, avr-libc libraries provide the flexibility needed.

3.3 Sensor-AM2301

AM2301 is a digital humidity and temperature sensor. The sensor features an ultra-low power consumption, a capacitive humidity sensor, a standard digital single-bus output and a long transmission distance.

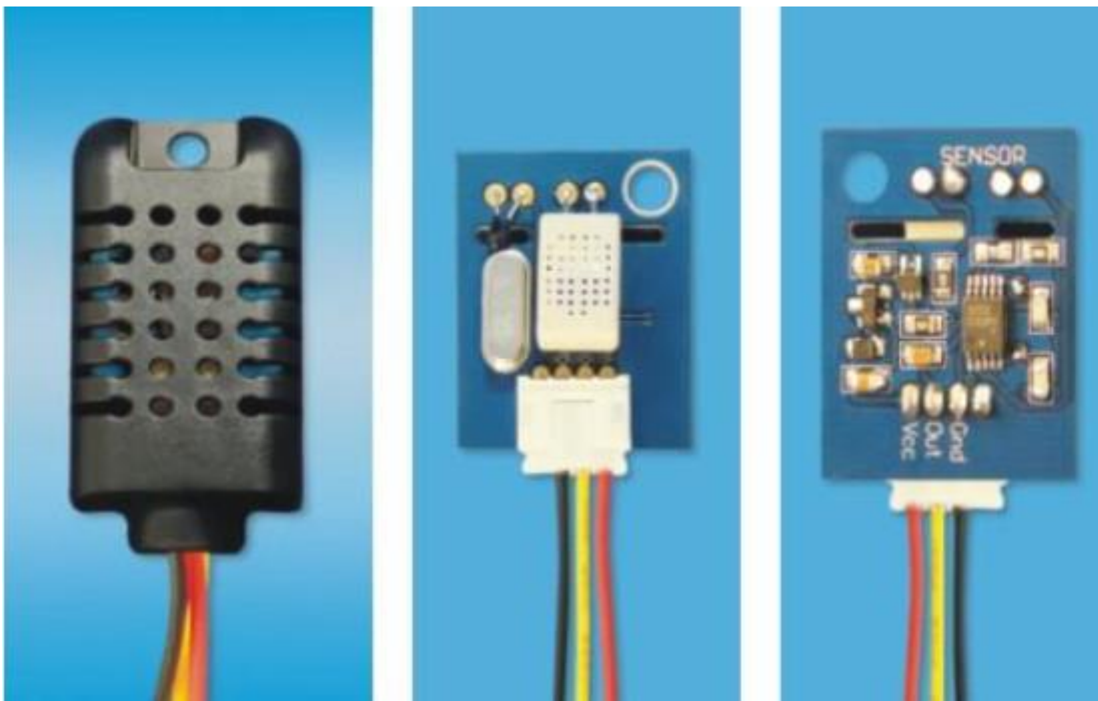

FIGURE 4. am2301 also called dht21. -source datasheet

Pin	Color	Name	Description
1	Red	VDD	Power (3.3V−5.2V)
2	Yellow	SDA	Serial data，Dual−port
3	Black	GND	Ground
4		NC	Empty

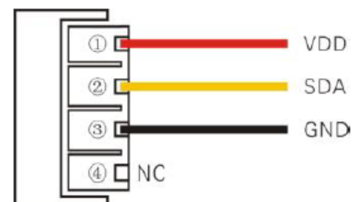

FIGURE 5. am2301 pin diagram and pin description -source datasheet

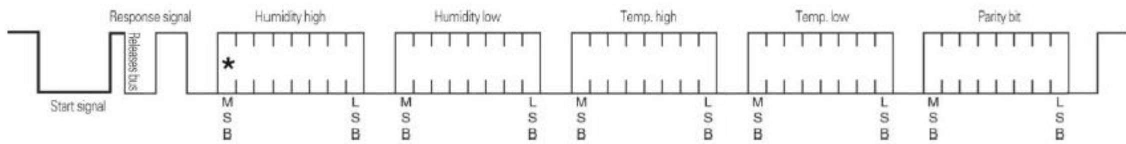

FIGURE 6. am2301 data transmission format -source datasheet

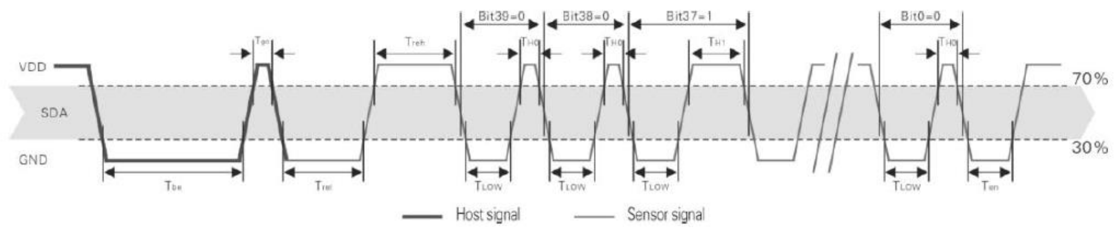

FIGURE 7. am2301 1-wire timing diagram -source datasheet

TABLE 3. am2301 timing description -source datasheet

Symbol	Parameter	min	typ	max	Unit
T_{be}	Host the start signal down time	0.8	1	20	mS
T_{go}	Bus master has released time	20	30	200	μS
T_{rel}	Response to low time	75	80	85	μS
T_{reh}	In response to high time	75	80	85	μS
T_{LOW}	Signal "0", "1" low time	48	50	55	μS
T_{H0}	Signal "0" high time	22	26	30	μS
T_{H1}	Signal "1" high time	68	70	75	μS
T_{en}	Sensor to release the bus time	45	50	55	μS

4 IMPLEMENTATION

In the process of implementation, the project work was divided into tasks such as, developing a serial communication on RPi2, and its counterpart on Arduino and developing a protocol to be used for synchronizing the communication. And it was proceeded incrementally by adding a functionality as required on the working modules.

4.1 Chain of Tasks

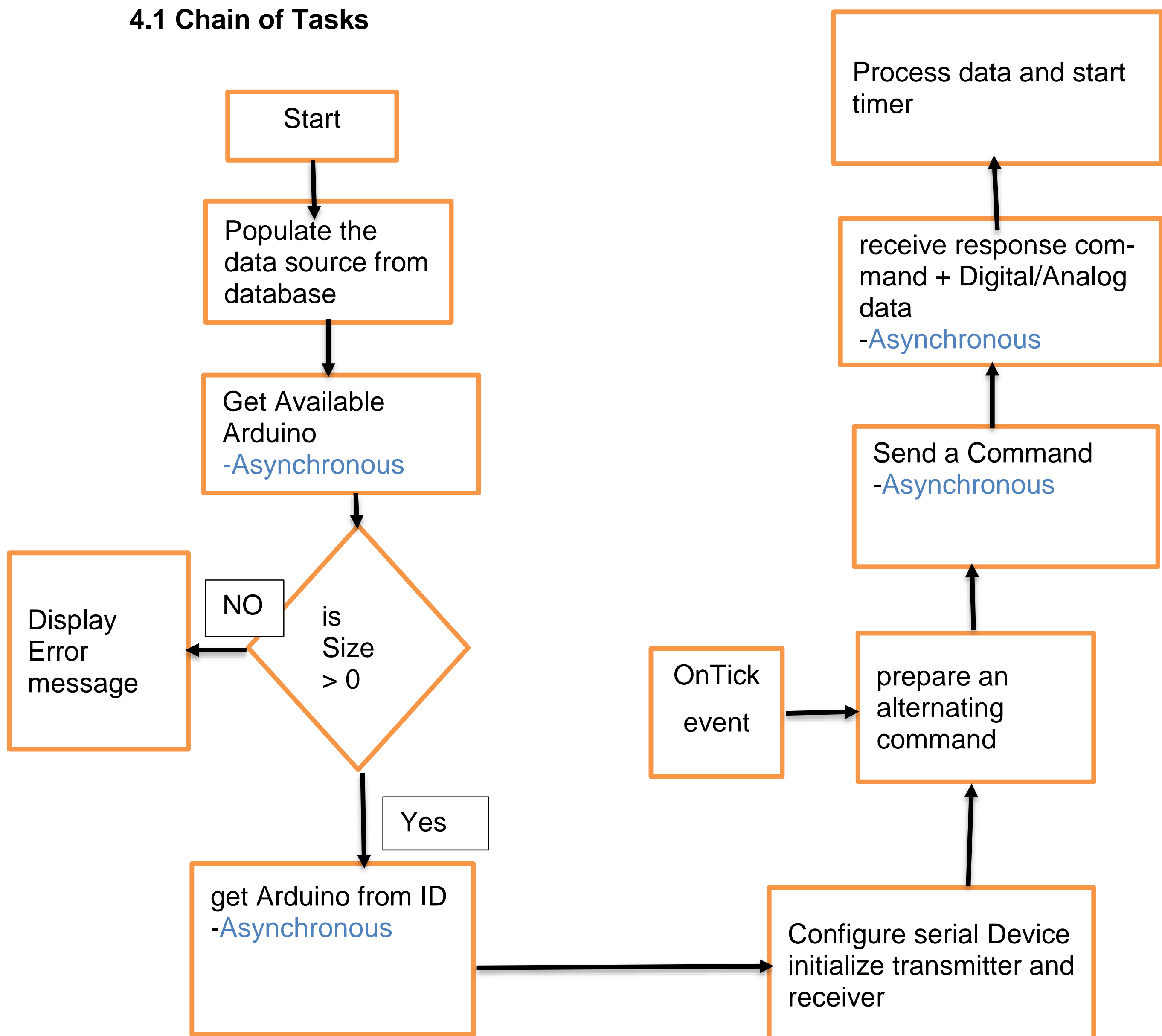

FIGURE 8. chain of tasks in the program

The diagram shown above is not a program flow chart, instead its shows how the chains of tasks are connected to each other.

4.2 Get Arduino

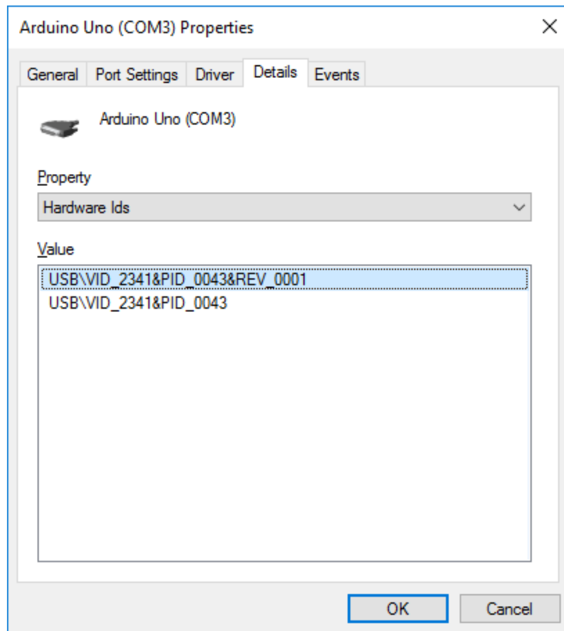

FIGURE 9. arduino device property in device manager

To find this property window, in Windows 10 machine, go to the Device manager and then open the Ports (Com & LPT) node and right click the device and then select properties.

Once the ***Hardware Ids*** is selected from the ***Property*** dropdown list, the vendor ID and product ID can be constructed from the string in the ***Value*** text box by prepending 0X to the number that follows VID and PID. This is shown in the code snippet below:

```
// arduino pid = 0043 and vid = 2341
static const unsigned int _pid = 0x0043;
static const unsigned int _vid = 0x2341;
```

Before calling the static ***FindAllAsync(String^)*** method in the ***Enumeration::DeviceInformation*** class, first, a string that contains a search criterion for finding Arduino must be created, using the static method ***GetDeviceSelectorFromUsbVidPid(unsigned int, unsigned int)*** in the ***SerialDevice*** class. This is show in the code snippet below:

```
IAsyncOperation<DeviceInformationCollection^>^
InterfacingArduino_UWP::MainPage::ListAvailableArduinosAsync(void)
{
    //Generate an Advanced Query Syntax (AQS) string that contains
    //search criteria for finding the device in the enumerated
    //device collection, then call GetDeviceSelectorFromUsbVidPid.
    auto aqs = SerialDevice::GetDeviceSelectorFromUsbVidPid(_vid, _pid);


    //Pass the retrieved string to FindAllAsync.
    //The call retrieves a DeviceInformationCollection object.
    return DeviceInformation::FindAllAsync(aqs);
}
```

The ***FindAllAsync(String^)*** method returns an ***IAsyncOperation<DeviceInformationCollection^>^*** object. Since this is an asynchronous operation, the task class is used to get the result, which is a ***DeviceInformationCollection*** object, upon completion.

In order to get the serial device, Arduino, four tasks are needed that run one after the other, forming a chain of tasks. The first one is created using a ***FindAllAsync(String^)*** method. The continuation task, which is created by calling a ***task::then*** method on the newly created task, takes as its argument, a lambda expression, the parameter of which is the return type of the antecedent task. This is shown in the code snippet below:

```
task<void> InterfacingArduino_UWP::MainPage::
GetAvailableArduinosAsync(void)
{
    return create_task(ListAvailableArduinosAsync()).
        then([this](DeviceInformationCollection ^arduinoCollection_)
    {
        // Store parameter as a member to ensure the duration of object allocation
        _arduinoCollection = arduinoCollection_;

        //check if there is any arduino connected
        if (!_arduinoCollection->Size)
        {
            AnalogPinValueTextBlock->Text =
                "Arduino is Not connected! Please Connect it and restart the application";
            CloseArduino();

        }
        else
        {
        //get the iterator object of type iterator<DeviceInformation>
        //for the collection and, since i am only interested
        //in the first connected arduino
            _deviceInfo = _arduinoCollection->First()->Current;

            GetArduinoFromIdAsync(_deviceInfo);//creates a new task
        }


    });
}
```

It should be noted that the antecedent task returns ***task<DeviceInformation-Collection^>*** and the result of this task is used in the continuation task that takes the lambda expression. Moreover, within the continuation, a new task is being created, to ensure that the next task, which needs a ***DeviceInformation*** object, is called only after the antecedent task is complete.

```
task<void> InterfacingArduino_UWP::MainPage::
GetArduinoFromIdAsync(DeviceInformation ^deviceInfo_)
{

    return create_task(SerialDevice::FromIdAsync(deviceInfo_->Id)).
        then([this](SerialDevice ^serial_device_)
    {

        try
        {
            // save the serial object to member variable
            _arduino = serial_device_;
```

As can be seen from the above code snippet, the serial device was ready to be configured with the default value 9600-8-N1.

4.3 Processing Input On Arduino

Within an infinite loop, Arduino was programmed to check if there were bytes available for reading and then responding with an appropriate command + data, based on the received command. The main loop is shown below:

```
void loop() {
  if(Serial.available() > 0)
  {
    unsigned char rxCommand = Serial.read();

    switch((rxCommand & 0xF0))
    {
      case REPORT_DIGITAL_DATA:
           sendDigitalSensorValue();
           break;

      case REPORT_ANALOG_DATA:
           sendAnalogPinValue();
           break;
    }
  }

}
```

4.4 DHT-Lib

This is a library by Rob Tillaart for receiving data from DHTxx devices - digital Temperature and Humidity sensors (14). It provides an interface for varieties of devices. The library used in this project, which was based on DHT21, had one function for reading and two public variables that held the Humidity and Temperature values. As shown in FIGURE 6, the device sends 5 bytes of data; The first two bytes represent the humidity value and the third and fourth bytes represent the Temperature values, and the last byte represents the parity bit.

As shown in the code snippet below, the library was modified to provide the whole buffer, the size of the buffer and error messages. In addition, the read logic was modified to match specifically the timing requirement of the AM2301.

```
int DHT::read21(uint8_t pin)
{
    //read values

    int rv = _readSensor(pin, DHTLIB_DHT_WAKEUP);

    if(rv != DHTLIB_OK)
    {
        /*
        humidity = DHTLIB_INVALID_VALUE;
        temperature = DHTLIB_INVALID_VALUE;*/
        return rv;
    }

    /*
    humidity = word(_bits[0], _bits[1]) * 0.1;
    temperature = word(_bits[2] & 0x7F, _bits[3]) * 0.1;

    if(_bits[2] & 0x80)// negative temperature
    {
        temperature = -temperature;
    }*/

    // Test checksum
    uint8_t sum = _bits[0] + _bits[1] + _bits[2] + _bits[3];
    if(_bits[4] != sum)
    {
        return DHTLIB_ERROR_CHECKSUM;
    }

    return DHTLIB_OK;
}
/*
    returns the const buffer, so as not to be modified
    modified by Bereket Godebo
*/
const uint8_t * DHT::getBuffer()
{
    return _bits;
}
/*
    returns the size of the buffer which is a constant 5
    modified by Bereket Godebo
*/
const size_t DHT::getBufferSize()
{
    return bufferSize;
}
```

In the above code snippet, the public function calls a private function, which contains the logic to read from the device. It was modified not to use the public variables but only to return the status. In addition, two more functions were added that returned the buffer and its size.

When sending the digital data, the response command DIGITAL_DATA and the data status constants were combined according to the status returned from the read21 function. This is shown in the code snippet below:

```
size_t bufSize = dht.getBufferSize();
int readStatus = dht.read21( DHT21_PIN);

switch(readStatus)
{
  case DHTLIB_OK:
      txCommand = (DIGITAL_DATA | DATA_OK);
      break;
  case DHTLIB_ERROR_CHECKSUM:
      txCommand = (DIGITAL_DATA | DATA_ERROR_CHECKSUM);
      break;
  case DHTLIB_ERROR_TIMEOUT:
      txCommand = (DIGITAL_DATA | DATA_ERROR_TIMEOUT);
      break;
  default:
      break;
}//switch
```

Next the command and the five data bytes were sent.

The analog message was used as an additional functionality to test the connection. ***analogRead()*** was the function that was used to get an analog value of a given pin. The function returned a value 0 to 1023, thus it was represented as Integer which was 16-bit in size. In the code snippet shown below, the first 10 bits were taken from the value returned by the function and put into two 8-bit variables. Then the response command ANALOG_DATA was combined with the analog pin number used.

```
analogPinValue = analogRead(analogPinA0);
unsigned char lowerByte = lowByte(analogPinValue);
unsigned char higherByte = highByte(analogPinValue);

// get the lower five bits of lowerByte
//dataMask = 0b00011111
unsigned char firstAnalogDataByte = lowerByte & dataMask;
// get the upper 3 bits of lowerByte
unsigned char threeDataBits = (lowerByte >> 5);
//create the second analog byte combining the remaining three bits
// in the lower byte with the two bits in the higher byte
unsigned char secondAnalogDataByte = (higherByte << 3) | threeDataBits;

//4 bits representing the analog pin number and a command to signal
//that analog message is being sent
txCommand =  (ANALOG_DATA | analogPinA0);
```

Next the command and the two bytes were sent.

4.5 Process Input On RPi2

After the serial device handle was obtained and configured, a data writer and a data reader object were created. It is shown in the code snippet below:

```
// setup our data reader for handling incoming data
_rx =
ref new Windows::Storage::Streams::DataReader(
                        _arduino->InputStream);
_rx->InputStreamOptions =
    Windows::Storage::Streams::InputStreamOptions::Partial;

// setup our data writer for handling outgoing data
_tx =
ref new Windows::Storage::Streams::DataWriter(
                        _arduino->OutputStream);

SendCommand();// a new task is create in this function
```

As can be seen above, a new task was created to send a command. It is important to keep the chain connected, thus a new task was created within a continuation task. While extending the chain, it has to be ensured that the chain is not broken.

As part of the design of the app, the method shown below was called repeatedly, in order to send alternating commands periodically

```
void InterfacingArduino_UWP::MainPage::SendCommand(void)
{
    if (_arduino != nullptr)
    {
            //The intetion is to toggle the command to send.
        _txCommand = (_txCommand == Command::REPORT_DIGITAL_DATA) ?
                                        Command::REPORT_ANALOG_DATA :
                                        Command::REPORT_DIGITAL_DATA;
        create_task(SendAsync(_txCommand));

    }

}
```

As can be seen above, a new task was created to send the command.

Here the command was sent and listening to a response from the Arduino was started, after a successful transmission of the command.

```
task<void> InterfacingArduino_UWP::MainPage::SendAsync(Command cmd_)
{
    uint8_t reportCommand = static_cast<uint8_t>(cmd_);
    _tx->WriteByte(reportCommand);
    return create_task(_tx->StoreAsync()).
        then([this](unsigned int bytesWritten)
    {
        if (bytesWritten > 0)
        {
            ReceiveCommand();
        }
    });
}
```

The following function was used just to check if Arduino was still connected.

```
void InterfacingArduino_UWP::MainPage::ReceiveCommand(void)
{
    if (_arduino != nullptr)
    {
        ReceiveAsync();
    }
}
```

Since the maximum number of bytes including the command, which were sent from the Arduino were 6, thus the buffer size was set to 6.

```
task<void> InterfacingArduino_UWP::MainPage::ReceiveAsync(void)
{
    unsigned int readBufferSize = 6;

    return create_task(_rx->LoadAsync(readBufferSize)).
        then([this](unsigned int bytesRead)
    {
        if (bytesRead > 0)
        {
            ProcessData();
        }

    });
}
```

It is assumed that Arduino always responds with an appropriate command + data response. As such there is no error messages shown here if there is no response.

After a command + data response was received, appropriate variables were initialized for data processing.

```
  //upper nibbles represent the command
  Command cmd = static_cast<Command>(data & 0xF0);


      switch (cmd)
      {
      case Command::DIGITAL_DATA:
          bytesRemaining = 5;
          //Lower nibbles represent the sensor data status
          dataStatus = static_cast<DigitalDataStatus>(data & 0x0F);
          break;
      case Command::ANALOG_DATA:
          bytesRemaining = 2;
          //Lower nibbles represent the analog pin number
          analogPinNumber = static_cast<uint8_t>(data & 0x0F);
          break;
      default:
          break;
      }

      while (bytesRemaining)
      {
          receivedData.push_back(_rx->ReadByte());
          bytesRemaining--;
      }
```

In the above code snippet, the remaining bytes were initialized, based on the command received.

When a DIGITAL_DATA response command was received, the remaining bytes were interpreted as shown in the code below.

```
humidity = ((receivedData.at(0) << 8) | receivedData.at(1)) * 0.1;
temperature = (((receivedData.at(2) & 0x7F) << 8) | receivedData.at(3) ) * 0.1;
```

Since the checksum had already been calculated, the fifth byte was ignored in the code snippet above. As shown in FIGURE 6, the most significant bytes were sent first, and Humidity and Temperature each represented a 16-bit value. Since the left most bit in the third byte represented a sign bit, the temperature value was calculated accordingly.

In the following code snippet, an analog value was reconstructed from the two bytes received.

```
rxAnalogData = (receivedData.at(0) | receivedData.at(1) << 5);
```

4.6 Save into SQLite

SQLite is an open-source embedded, serverless SQL database engine. It is implemented in C and it is a leading device side technology for a local data storage (15.)

In this project, SQLite Library for Universal Windows Platform apps was installed via Extensions and Updates for Visual Studio. This is shown in FIGURE 10.

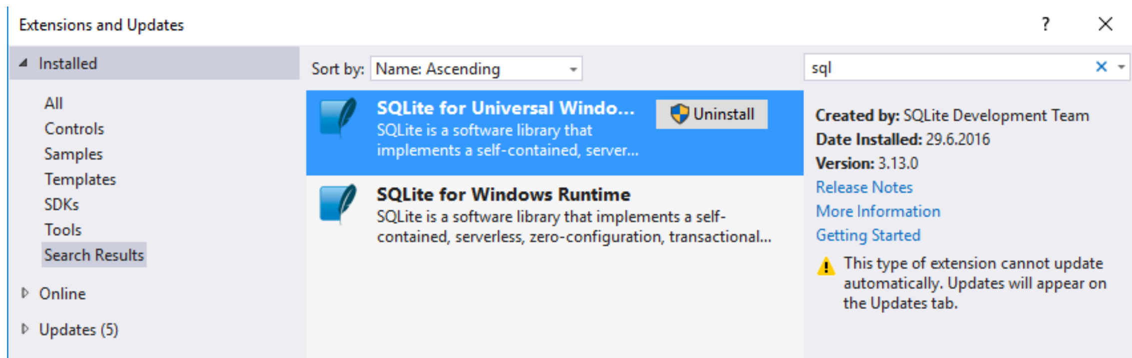

FIGURE 10. sqlite for uwp installation

And then the header `#include <sqlite3.h>` was included before using the library.

And also as shown in FIGURE 11 below, the reference to the library must be added into References in the project.

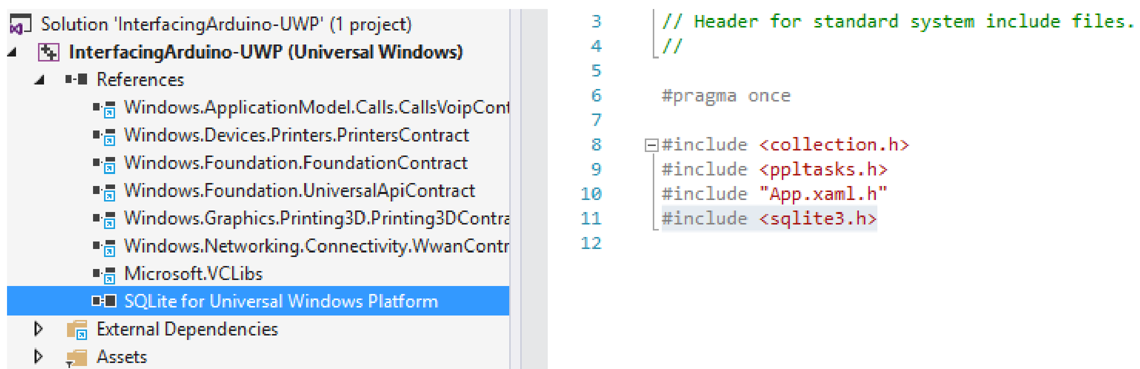

FIGURE 11. adding reference for sqlite library

5 CONCLUSION

The final product was deployed into the RPi2. It notified the user if either Arduino or the sensor was not attached. Moreover, it sent an alternating command to Arduino every 3 seconds and saved the data into the local database using SQLite. The commands were REPORT_DIGITAL_DATA and REPORT_ANALOG_DATA, and Arduino responded appropriately. The application on RPi2 showed the current data on the screen but the data from the database was shown only after restarting the application. This was because the aim of the project was to connect to Arduino and show the current data and save it into the database.

Accessing the serial device on the RPi2, required experience in Asynchronous programming in C++/CX. Therefore, referring deferent material on topics such as concurrency and threads was necessary. The shift from synchronous programming to asynchronous programming was not smooth. It required a lot of practice. On the other hand, the programming language used on RPi2, C++/CX, an extension to the standard C++, seemed at first as learning a whole new language but it turned out that there were a few more constructs to learn to comfortably write a code.

Future improvements can be made in areas such as creating a Windows runtime component that connects to Arduino, supporting the cancelation of asynchronous operations and throwing an exception for an error that occurs in the chain of task instead of showing it on the screen.

REFERENCES

1. Windows API Reference SerialCommunication namespace. 2016. Microsoft. Date of retrieval 13.08.2016

   https://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.serialcommunication.aspx

2. Guide to Universal Windows Platform (UWP) apps. 2016. Microsoft. Date of retrieval 11.08.2016
   https://msdn.microsoft.com/windows/uwp/get-started/universal-application-platform-guide
3. Asynchronous programming in C++. 2016. Microsoft. Date of retrieval 28.07.2016
   https://msdn.microsoft.com/en-us/windows/uwp/threading-async/asynchronous-programming-in-cpp-universal-windows-platform-apps
4. Windows Runtime APIs IAsyncInfo interface. 2016. Microsoft. Date of retrieval 13.08.2016
   https://msdn.microsoft.com/library/windows/apps/br206587
5. concurrency Namespace. create_task Function. Microsoft. 2016. Date of retrieval 13.08.2016
   https://msdn.microsoft.com/en-us/library/hh913025.aspx
6. Wikipedia. USB communications device class. 2016. Date of retrieval 13.08.2016
   https://en.wikipedia.org/wiki/USB
7. Arduino Language Reference. Serial. 2016. Date of retrieval 14.08.2016
   https://www.arduino.cc/en/Reference/Serial

8. Arduino. Language Reference. 2016. Date of retrieval 14.08.2016
   https://www.arduino.cc/en/Reference/HomePage
9. Standard C Library for AVR-GCC. Toolchain Overview. 2016. Date of retrieval 14.08.2016

http://www.nongnu.org/avr-libc/user-manual/overview.html

10. Firmata Protocol. V2.3ProtocolDetails. 2013. Date of retrieval 26.07.2016
http://firmata.org/wiki/Protocol
11. Wikipedia. Raspberry Pi. 2016. Date of retrieval 11.08.2016
https://en.wikipedia.org/wiki/Raspberry_Pi
12. Raspberry Pi Foundation. Raspberry Pi 2 Model B. 2015. Date of retrieval 11.08.2016
https://www.raspberrypi.org/products/raspberry-pi-2-model-b/
13. Wikipedia. Arduino. 2016. Date of retrieval 11.08.2016
https://en.wikipedia.org/wiki/Arduino
14. Arduino Playground. Class for DHTxx Sensors. 2015 Date of retrieval 14.08.2016
http://playground.arduino.cc/Main/DHTLib
15. Data access. 2016. Microsoft. Date of retrieval 28.07.2016
https://msdn.microsoft.com/en-us/windows/uwp/data-access/index

APPENDIX

```
1   /*
2    UserStory: =>In home the user can check data from RaspberryPi (GUI-Interface)
3               =>The data can be temperature, humidity, state of door (open/closed)
4               =>From Handy the user can check temperature,humidity,state of door
5
6     An implementation of a home automation system using UWP in C++/CX
7     This app runs on RaspberryPi2 and connects to
8             Arduino via a USB and gets Sensor data as well as analog pin value
9
10      By Bereket Godebo DIT8SN
11      OAMK-School of Engineering
12
13      MainPage.xaml.h
14      Implementation of the MainPage class.
15  */
16
17  #pragma once
18
19  #include "MainPage.g.h"
20
21  namespace InterfacingArduino_UWP
22  {
23
24      public ref class AirCondition sealed
25      {
26      private:
27          Platform::String ^_humidity;
28          Platform::String ^_temperature;
29          Platform::String ^_timeStamp;
30      public:
31          AirCondition(
32              Platform::String ^humidity_,
33              Platform::String ^temperature_,
34              Platform::String ^timeStamp_) :
35              _humidity(humidity_),
36              _temperature(temperature_),
37              _timeStamp(timeStamp_) {}
38
39          property Platform::String ^Humidity
40          {
41              Platform::String ^get() { return _humidity; }
42          }
43          property Platform::String ^Temperature
44          {
45              Platform::String ^get() { return _temperature; }
46          }
47          property Platform::String ^TimeStamp
48          {
49              Platform::String ^get() { return _timeStamp; }
50          }
51
52      };
53
54      public ref class Database sealed
55      {
56      private:
57          sqlite3 *_database;
58          sqlite3_stmt *_statement;
59          const char * _sqlCreate =
60          "CREATE TABLE IF NOT EXISTS h_t_table(humidity real, temperature real, time_stamp text);";
61          const char * _sqlInsert =
62              "INSERT INTO h_t_table VALUES(?, ?, ((SELECT time('now', 'localtime'))));";
63          const char * _sqlSelect =
64              "SELECT * FROM h_t_table;";
65          int _result;
```

```
    ~Database();
    Platform::String ^convertChar(char * originalChar_);

public:
    Database(Platform::String ^path_);
    void createTable(void);
    void selectFromTable(Windows::Foundation::Collections::IVector<AirCondition^>^ recordings_);
    void insertIntoTable(double h_, double t_);

    property int Result
    {
        int get() { return _result; }
    }


};

public ref class AirConditionViewModel sealed
{
private:
    Windows::Foundation::Collections::IVector<AirCondition^>^ _recordings;

public:
    AirConditionViewModel(Platform::String ^path_);

    property Windows::Foundation::Collections::IVector<AirCondition^>^ Recordings
    {
        Windows::Foundation::Collections::IVector<AirCondition^>^ get()
        {
            if (_recordings == nullptr)
            {
                _recordings = ref new Platform::Collections::Vector<AirCondition^>();
            }

            return _recordings;
        }
    }

};


public enum class Command
{
    REPORT_DIGITAL_DATA = 0xA0,
    DIGITAL_DATA = 0xB0,
    REPORT_ANALOG_DATA = 0xC0,
    ANALOG_DATA = 0xD0,
};

public enum class DigitalDataStatus
{
    DATA_OK,
    DATA_ERROR_CHECKSUM,
    DATA_ERROR_TIMEOUT,

};
/// <summary>
/// An empty page that can be used on its own or navigated to within a Frame.
/// </summary>
public ref class MainPage sealed
{
public:
    static Windows::Foundation::IAsyncOperation
        <Windows::Devices::Enumeration::DeviceInformationCollection ^>
        ^ListAvailableArduinosAsync(void);

```

```
133         MainPage();
134
135         property InterfacingArduino_UWP::AirConditionViewModel ^ViewModel
136         {
137             InterfacingArduino_UWP::AirConditionViewModel ^get()
138             {
139                 return _viewModel;
140             }
141         }
142
143
144     private:
145         // arduino pid = 0043 and vid = 2341
146         static const unsigned int _pid = 0x0043;
147         static const unsigned int _vid = 0x2341;
148         // used for alternatively sending Digital or Analog data request Command
149         InterfacingArduino_UWP::Command _txCommand;
150
151         Platform::String ^_path =
152             Windows::Storage::ApplicationData::Current->LocalFolder->Path + "\\test.db";//IAUWP00
153         AirConditionViewModel ^_viewModel;
154
155         Windows::Devices::Enumeration::DeviceInformationCollection ^_arduinoCollection;
156         Windows::Devices::Enumeration::DeviceInformation ^_deviceInfo;
157         Windows::Devices::SerialCommunication::SerialDevice ^_arduino;
158         Windows::Storage::Streams::DataWriter ^_tx;
159         Windows::Storage::Streams::DataReader ^_rx;
160
161         Windows::UI::Xaml::DispatcherTimer ^_timer;
162
163
164         void SendCommand(void);
165         void ReceiveCommand(void);
166         void ProcessData(void);
167         void StartTimer(void);
168         void CloseArduino(void);
169
170         Concurrency::task<void> GetAvailableArduinosAsync(void);
171         Concurrency::task<void> SendAsync(Command cmd_);
172         Concurrency::task<void> ReceiveAsync(void);
173         Concurrency::task<void> GetArduinoFromIdAsync(
174                         Windows::Devices::Enumeration::DeviceInformation ^deviceInfo_);
175
176
177         void OnLoaded(Platform::Object ^sender, Windows::UI::Xaml::RoutedEventArgs ^e);
178         void OnTick(Object^ sender, Object^ e);
179
180     };
181 }
182
```

```
1   /*
2     UserStory: =>In home the user can check data from RaspberryPi (GUI-Interface)
3                =>The data can be temperature, humidity, state of door (open/closed)
4                =>From Handy the user can check temperature,humidity,state of door
5
6      An implementation of a home automation system using UWP in C++/CX
7      This app runs on RaspberryPi2 and connects to
8               Arduino via a USB and gets Sensor data as well as analog pin value
9
10      By Bereket Godebo DIT8SN
11      OAMK-School of Engineering
12
13      MainPage.xaml.cpp
14      Implementation of the MainPage class.
15  */
16
17  #include "pch.h"
18  #include "MainPage.xaml.h"
19
20  using namespace InterfacingArduino_UWP;
21
22  using namespace Platform;
23  using namespace Windows::Foundation;
24  using namespace Windows::Foundation::Collections;
25  using namespace Windows::UI::Xaml;
26  using namespace Windows::UI::Xaml::Controls;
27  using namespace Windows::UI::Xaml::Controls::Primitives;
28  using namespace Windows::UI::Xaml::Data;
29  using namespace Windows::UI::Xaml::Input;
30  using namespace Windows::UI::Xaml::Media;
31  using namespace Windows::UI::Xaml::Navigation;
32
33  using namespace concurrency;
34  using namespace Windows::Devices::Enumeration;
35  using namespace Windows::Devices::SerialCommunication;
36
37  // The Blank Page item template is documented at
38  //http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409
39
40  IAsyncOperation<DeviceInformationCollection^>^
41  InterfacingArduino_UWP::MainPage::ListAvailableArduinosAsync(void)
42  {
43      //Generate an Advanced Query Syntax (AQS) string that contains
44      //search criteria for finding the device in the enumerated
45      //device collection, then call GetDeviceSelectorFromUsbVidPid.
46      auto aqs = SerialDevice::GetDeviceSelectorFromUsbVidPid(_vid, _pid);
47
48
49      //Pass the retrieved string to FindAllAsync.
50      //The call retrieves a DeviceInformationCollection object.
51      return DeviceInformation::FindAllAsync(aqs);
52  }
53
54
55
56  MainPage::MainPage()
57  {
58      InitializeComponent();
59      // populate the ListView data from local db
60      _viewModel = ref new InterfacingArduino_UWP::AirConditionViewModel(_path);
61
62      // so that connections initialize after the components are initialized
63      Loaded += ref new Windows::UI::Xaml::RoutedEventHandler(this,
64                              &InterfacingArduino_UWP::MainPage::OnLoaded);
65
66
67  }
```

```
68
69
70  void InterfacingArduino_UWP::MainPage::
71  OnLoaded(Platform::Object ^sender, Windows::UI::Xaml::RoutedEventArgs ^e)
72  {
73
74
75      GetAvailableArduinosAsync();
76      _timer = ref new Windows::UI::Xaml::DispatcherTimer();
77      _timer->Tick += ref new EventHandler<Object^>(this, &MainPage::OnTick);
78
79
80
81  }
82
83  task<void> InterfacingArduino_UWP::MainPage::
84  GetAvailableArduinosAsync(void)
85  {
86      return create_task(ListAvailableArduinosAsync()).
87          then([this](DeviceInformationCollection ^arduinoCollection_)
88      {
89          // Store parameter as a member to ensure the duration of object allocation
90          _arduinoCollection = arduinoCollection_;
91
92          //check if there is any arduino connected
93          if (!_arduinoCollection->Size)
94          {
95              AnalogPinValueTextBlock->Text =
96                  "Arduino is Not connected! Please Connect it and restart the application";
97              CloseArduino();
98
99          }
100         else
101         {
102         //get the iterator object of type iterator<DeviceInformation>
103         //for the collection and, since i am only interested
104         //in the first connected arduino
105             _deviceInfo = _arduinoCollection->First()->Current;
106
107             GetArduinoFromIdAsync(_deviceInfo);//creates a new task
108         }
109
110
111
112     });
113 }
114
115 task<void> InterfacingArduino_UWP::MainPage::
116 GetArduinoFromIdAsync(DeviceInformation ^deviceInfo_)
117 {
118
119     return create_task(SerialDevice::FromIdAsync(deviceInfo_->Id)).
120         then([this](SerialDevice ^serial_device_)
121     {
122
123         // save the serial object to member variable
124             _arduino = serial_device_;
125
126         if(_arduino != nullptr)
127         {
128             Windows::Foundation::TimeSpan _timeOut;
129             _timeOut.Duration = 10000000L;// = 1 second
130
```

```
131             // Configure serial settings
132             _arduino->WriteTimeout = _timeOut;
133             _arduino->ReadTimeout = _timeOut;
134             _arduino->BaudRate = 9600;
135             _arduino->Parity =
136             Windows::Devices::SerialCommunication::SerialParity::None;
137             _arduino->StopBits =
138             Windows::Devices::SerialCommunication::SerialStopBitCount::One;
139             _arduino->DataBits = 8;
140             _arduino->Handshake =
141             Windows::Devices::SerialCommunication::SerialHandshake::None;
142
143             // setup our data reader for handling incoming data
144             _rx =
145             ref new Windows::Storage::Streams::DataReader(
146                                     _arduino->InputStream);
147             _rx->InputStreamOptions =
148                 Windows::Storage::Streams::InputStreamOptions::Partial;
149
150             // setup our data writer for handling outgoing data
151             _tx =
152             ref new Windows::Storage::Streams::DataWriter(
153                                     _arduino->OutputStream);
154
155             SendCommand();// a new task is create in this function
156         }
157         else
158         {
159             //the variable _aruino can be null at this point
160             // if for example, the app does not have 'serialcommunication'
161             //capabilities specified in the app manifest
162             CloseArduino();
163         }
164
165     });
166 }
167
168 void InterfacingArduino_UWP::MainPage::SendCommand(void)
169 {
170     if (_arduino != nullptr)
171     {
172         //The intetion is to toggle the command to send.
173         _txCommand = (_txCommand == Command::REPORT_DIGITAL_DATA) ?
174                                         Command::REPORT_ANALOG_DATA :
175                                         Command::REPORT_DIGITAL_DATA;
176         create_task(SendAsync(_txCommand));
177
178     }
179
180 }
181
```

```
182  task<void> InterfacingArduino_UWP::MainPage::SendAsync(Command cmd_)
183  {
184      uint8_t reportCommand = static_cast<uint8_t>(cmd_);
185      _tx->WriteByte(reportCommand);
186      return create_task(_tx->StoreAsync()).
187          then([this](unsigned int bytesWritten)
188      {
189          if (bytesWritten > 0)
190          {
191              ReceiveCommand();
192          }
193      });
194  }
195
196  void InterfacingArduino_UWP::MainPage::ReceiveCommand(void)
197  {
198      if (_arduino != nullptr)
199      {
200          ReceiveAsync();
201      }
202  }
203
204
205  task<void> InterfacingArduino_UWP::MainPage::ReceiveAsync(void)
206  {
207      unsigned int readBufferSize = 6;
208
209      return create_task(_rx->LoadAsync(readBufferSize)).
210          then([this](unsigned int bytesRead)
211      {
212          if (bytesRead > 0)
213          {
214              ProcessData();
215          }
216
217      });
218  }
219
220  void InterfacingArduino_UWP::MainPage::ProcessData(void)
221  {
222      Database db(_path);
223      std::vector<uint8_t> receivedData;
224      std::size_t bytesRemaining = 0;
225      DigitalDataStatus dataStatus;
226      uint8_t analogPinNumber;
227      uint16_t rxAnalogData = 0;
228      uint8_t data = _rx->ReadByte();
229      double humidity;
230      double temperature;
231
232
233      //upper nibbles represent the command
234      Command cmd = static_cast<Command>(data & 0xF0);
235
236
237          switch (cmd)
238          {
239          case Command::DIGITAL_DATA:
240              bytesRemaining = 5;
241              //Lower nibbles represent the sensor data status
242              dataStatus = static_cast<DigitalDataStatus>(data & 0x0F);
243              break;
244          case Command::ANALOG_DATA:
245              bytesRemaining = 2;
246              //Lower nibbles represent the analog pin number
247              analogPinNumber = static_cast<uint8_t>(data & 0x0F);
248              break;
249          default:
250              break;
```

```
251             }
252
253             while (bytesRemaining)
254             {
255                 receivedData.push_back(_rx->ReadByte());
256                 bytesRemaining--;
257             }
258
259
260             if (cmd == Command::DIGITAL_DATA &&
261                     dataStatus == DigitalDataStatus::DATA_OK)
262             {
263                 humidity = ((receivedData.at(0) << 8) | receivedData.at(1)) * 0.1;
264                 temperature = (((receivedData.at(2) & 0x7F) << 8) | receivedData.at(3) ) * 0.1;
265
266                 if (receivedData.at(2) & 0x80)
267                 {
268                     temperature = -temperature;
269                 }
270
271                 CurrentHTValuetextBlock->Text =
272                     "Current H(%):  " +
273                     humidity +
274                     " And T(C):  " +
275                     temperature +
276                     "\n";
277
278                 //save into the db
279                 if (db.Result == SQLITE_OK)// check if connection is open
280                 {
281                     db.createTable();// create table if not exits
282                 }
283
284                 if (db.Result == SQLITE_DONE)// check if creating table was success
285                 {// save the data
286                     db.insertIntoTable(humidity, temperature);
287                 }
288             }
289             else if (cmd == Command::DIGITAL_DATA &&
290                     dataStatus == DigitalDataStatus::DATA_ERROR_TIMEOUT)
291             {
292                 CurrentHTValuetextBlock->Text = "Sensor Not connected!";
293             }
294             else if (cmd == Command::ANALOG_DATA)
295             {
296                 rxAnalogData = (receivedData.at(0) | receivedData.at(1) << 5);
297
298                 AnalogPinValueTextBlock->Text =
299                 "Analog Value at Pin number: " +
300                 analogPinNumber + " is " +
301                 rxAnalogData;
302             }
303
304             StartTimer();
305
306     }
307
```

```
308  void MainPage::StartTimer() {
309
310      TimeSpan ts;
311      ts.Duration = 30000000L;// in 3 seconds interval
312      _timer->Interval = ts;
313      _timer->Start();
314
315  }
316  //when timer overflows
317  void MainPage::OnTick(Object^ sender, Object^ e) {
318      //reset and send command again
319      _timer->Stop();
320      SendCommand();
321  }
322
323  void InterfacingArduino_UWP::MainPage::CloseArduino(void)
324  {
325      delete(_arduinoCollection);
326      _arduinoCollection = nullptr;
327
328      delete(_arduino);
329      _arduino = nullptr;
330
331      delete(_deviceInfo);
332      _deviceInfo = nullptr;
333
334      delete(_tx);
335      _tx = nullptr;
336
337      delete(_rx);
338      _rx = nullptr;
339
340      delete(_timer);
341      _timer = nullptr;
342
343  }
344
345  /*************************************************************
346  ****************************Support class definitions********
347  *************************************************************/
348
349
350  InterfacingArduino_UWP::AirConditionViewModel::
351  AirConditionViewModel(Platform::String ^ path_)
352  {
353      Database db(path_);
354      if (db.Result == SQLITE_OK)
355      {
356          db.createTable();
357      }
358
359      if (db.Result == SQLITE_DONE)//Succssfuly created table IF NOT EXISTS
360      {// then populate the collection if there is any
361          db.selectFromTable(Recordings);
362      }
363  }
364
```

```
365  InterfacingArduino_UWP::Database::~Database()
366  {
367      sqlite3_close(_database);
368  }
369
370  Platform::String ^ InterfacingArduino_UWP::Database::
371  convertChar(char * originalChar_)
372  {
373      std::size_t newsize = strlen(originalChar_) + 1;
374      wchar_t * wcstring = new wchar_t[newsize];
375      size_t convertedChars = 0;
376      mbstowcs_s(&convertedChars, wcstring, newsize, originalChar_, _TRUNCATE);
377      String^ str = ref new Platform::String(wcstring);
378      delete[] wcstring;
379      return str;
380  }
381
382  InterfacingArduino_UWP::Database::Database(Platform::String ^ path_)
383  {
384      //create path
385      std::string pathStr(path_->Begin(), path_->End());
386      const char * path = pathStr.c_str();
387      // open db
388      _result = sqlite3_open_v2(path,
389                                &_database,
390                                SQLITE_OPEN_READWRITE |
391                                SQLITE_OPEN_CREATE,
392                                NULL);
393  }
394
395  void InterfacingArduino_UWP::Database::createTable(void)
396  {
397      sqlite3_stmt *statement;
398
399      if (sqlite3_prepare_v2(_database,
400                                _sqlCreate,
401                                -1,
402                                &statement,
403                                0) == SQLITE_OK)
404      {
405          _result = sqlite3_step(statement);
406      }
407
408      sqlite3_finalize(statement);
409  }
410
411  void InterfacingArduino_UWP::Database::
412  selectFromTable(Windows::Foundation::Collections::IVector<AirCondition^>^ recordings_)
413  {
414      sqlite3_stmt *statement;
415      Platform::String ^humidity;
416      Platform::String ^temperature;
417      Platform::String ^timeStamp;
418      AirCondition ^data;
419
420      if (sqlite3_prepare_v2(_database,
421                             _sqlSelect,
422                             -1,
423                             &statement,
424                             0) == SQLITE_OK)
425      {
426
```

```
427             int cols = sqlite3_column_count(statement);
428             while (sqlite3_step(statement) == SQLITE_ROW)
429             {
430                 for (int col = 0; col < cols; col++)
431                 {
432                     char *ptr = (char *)sqlite3_column_text(statement, col);
433                     if (ptr)
434                     {
435                         switch (col)
436                         {
437                         case 0:
438                             humidity = convertChar(ptr);
439                             break;
440                         case 1:
441                             temperature = convertChar(ptr);
442                             break;
443                         case 2:
444                             timeStamp = convertChar(ptr);
445                             break;
446                         }
447                     }
448                 }// row complete
449 
450                 data = ref new AirCondition(humidity, temperature, timeStamp);
451                 recordings_->Append(data);
452             }//while
453             sqlite3_finalize(statement);
454         }
455     }
456 
457     void InterfacingArduino_UWP::Database::insertIntoTable(double h_, double t_)
458     {
459         sqlite3_stmt *statement;
460 
461         if (sqlite3_prepare_v2(_database, _sqlInsert, -1, &statement, 0) == SQLITE_OK)
462         {
463             //int sqlite3_bind_double(sqlite3_stmt*, int, double);
464             if (sqlite3_bind_double(statement, 1, h_) ==
465                         SQLITE_OK &&
466                         sqlite3_bind_double(statement, 2, t_) ==
467                         SQLITE_OK)
468             {
469                 _result = sqlite3_step(statement);
470 
471             }
472 
473         }
474 
475         sqlite3_finalize(statement);
476     }
477 
```

```
1   /*
2   UserStory: =>In home the user can check data from RaspberryPi (GUI-Interface)
3                =>The data can be temperature, humidity, state of door (open/closed)
4                =>From Handy the user can check temperature,humidity,state of door
5
6      Part of an implementation of a home automation system using UWP in C++/CX On the RaspberryPi side
7      This an Arduino program that connects via a USB to RaspberryPi 2
8      and sends Digital Sensor data as well as analog pin value
9
10      By Bereket Godebo DIT8SN
11      OAMK-School of Engineering
12  */
13  #include <dht.h>
14
15  #define DHT21_PIN 2
16
17
18
19  const unsigned char REPORT_DIGITAL_DATA = 0xA0;
20  const unsigned char DIGITAL_DATA = 0xB0;
21  const unsigned char REPORT_ANALOG_DATA = 0xC0;
22  const unsigned char ANALOG_DATA = 0xD0;
23  const unsigned char dataMask = 0x1F;
24
25
26
27
28  int analogPinValue = 0;
29  unsigned char txCommand = 0;
30
31  // digital data status to be sent along the command
32  const unsigned char DATA_OK             = ( 0 & 0x0F);
33  const unsigned char DATA_ERROR_CHECKSUM = ( 1 & 0x0F);
34  const unsigned char DATA_ERROR_TIMEOUT  = ( 2 & 0x0F);
35
36  // these values are to be sent with a resolution of 4-bits thus the mask=0x0F
37  const unsigned char analogPinA0 = ( 0 & 0x0F);
38
39  /* for future use, for now only analog pin A0 is used
40      const unsigned char analogPinA1 = ( 1 & 0x0F);
41      const unsigned char analogPinA2 = ( 2 & 0x0F);
42      const unsigned char analogPinA3 = ( 3 & 0x0F);
43      const unsigned char analogPinA4 = ( 4 & 0x0F);
44      const unsigned char analogPinA5 = ( 5 & 0x0F);
45  */
46
47  void sendAnalogPinValue(void);
48  void sendDigitalSensorValue(void);
49
50
51  DHT dht;
52
53  void setup() {
54    Serial.begin(9600);
55
56  }
57
```

```
58  void loop() {
59    if(Serial.available() > 0)
60    {
61      unsigned char rxCommand = Serial.read();
62
63      switch((rxCommand & 0xF0))
64      {
65        case REPORT_DIGITAL_DATA:
66             sendDigitalSensorValue();
67             break;
68
69        case REPORT_ANALOG_DATA:
70             sendAnalogPinValue();
71             break;
72      }
73    }
74
75  }
76
77  void sendDigitalSensorValue(void)
78  {
79      size_t bufSize = dht.getBufferSize();
80      int readStatus = dht.read21( DHT21_PIN);
81
82      switch(readStatus)
83      {
84        case DHTLIB_OK:
85            txCommand = (DIGITAL_DATA | DATA_OK);
86            break;
87        case DHTLIB_ERROR_CHECKSUM:
88            txCommand = (DIGITAL_DATA | DATA_ERROR_CHECKSUM);
89            break;
90        case DHTLIB_ERROR_TIMEOUT:
91            txCommand = (DIGITAL_DATA | DATA_ERROR_TIMEOUT);
92            break;
93        default:
94            break;
95      }//switch
96
97      Serial.write(txCommand);
98      Serial.write(dht.getBuffer(), bufSize);
99  }
100
101 void sendAnalogPinValue(void)
102 {
103     analogPinValue = analogRead(analogPinA0);
104     unsigned char lowerByte = lowByte(analogPinValue);
105     unsigned char higherByte = highByte(analogPinValue);
106
107     // get the lower five bits of lowerByte
108     //dataMask = 0b00011111
109     unsigned char firstAnalogDataByte = lowerByte & dataMask;
110     // get the upper 3 bits of lowerByte
111     unsigned char threeDataBits = (lowerByte >> 5);
112     //create the second analog byte combining the remaining three bits
113     // in the lower byte with the two bits in the higher byte
114     unsigned char secondAnalogDataByte = (higherByte << 3) | threeDataBits;
115
116     //3 bits representing the analog pin number and a command to signal
117     //that analog message is being sent
118     txCommand =  (ANALOG_DATA | analogPinA0);
119
120     //send the command and then the two byte data
121     Serial.write(txCommand);
122     Serial.write(firstAnalogDataByte);// a byte containing 5-bits of data
123     Serial.write(secondAnalogDataByte);// a byte containing 5-bits of data
124 }
```