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
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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPi2</td>
<td>Raspberry Pi 2</td>
</tr>
<tr>
<td>APIs</td>
<td>Application Programming Interfaces</td>
</tr>
<tr>
<td>UWP</td>
<td>Universal Windows Platform</td>
</tr>
<tr>
<td>WinRT</td>
<td>Windows Runtime</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>USART</td>
<td>Universal Synchronous Asynchronous Receive Trans-</td>
</tr>
<tr>
<td></td>
<td>mit</td>
</tr>
<tr>
<td>USB CDC</td>
<td>USB Communication Device Class</td>
</tr>
<tr>
<td>SDKs</td>
<td>Software Development Kits</td>
</tr>
<tr>
<td>STA</td>
<td>Single-Threaded Apartment</td>
</tr>
<tr>
<td>TTL</td>
<td>Transistor-Transistor-Logic</td>
</tr>
<tr>
<td>VID</td>
<td>Vendor ID</td>
</tr>
<tr>
<td>PID</td>
<td>Product ID</td>
</tr>
</tbody>
</table>
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.Storage.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(baud-rate) 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).

<table>
<thead>
<tr>
<th>Type</th>
<th>Command</th>
<th>MIDI Channel</th>
<th>First byte</th>
<th>Second byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>analog I/O message</td>
<td>0xE0</td>
<td>pin #</td>
<td>LSB(0-6)</td>
<td>MSB(7-13)</td>
</tr>
<tr>
<td>digital I/O message</td>
<td>0x90</td>
<td>port</td>
<td>LSB(0-6)</td>
<td>MSB(7-13)</td>
</tr>
<tr>
<td>report analog pin</td>
<td>0xC0</td>
<td>pin #</td>
<td>disable/enable(0/1)</td>
<td>-N/A-</td>
</tr>
<tr>
<td>report digital port</td>
<td>0xD0</td>
<td>port</td>
<td>disable/enable(0/1)</td>
<td>-N/A-</td>
</tr>
</tbody>
</table>

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 RE-PORT 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Command</th>
<th>Additional Data</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
<th>Byte4</th>
<th>Byte5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report</td>
<td>0xA0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Digital</td>
<td>0xB0</td>
<td>data-status(0-2)</td>
<td>byte0</td>
<td>byte1</td>
<td>byte2</td>
<td>byte3</td>
<td>byte4</td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Report</td>
<td>0xC0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Analog</td>
<td>0xD0</td>
<td>pin #</td>
<td>byte0</td>
<td>byte1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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).

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

<table>
<thead>
<tr>
<th>Pin</th>
<th>Color</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Red</td>
<td>VDD</td>
<td>Power (3.3V–5.2V)</td>
</tr>
<tr>
<td>2</td>
<td>Yellow</td>
<td>SDA</td>
<td>Serial data, Dual–port</td>
</tr>
<tr>
<td>3</td>
<td>Black</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>4</td>
<td>NC</td>
<td>NC</td>
<td>Empty</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>min</th>
<th>typ</th>
<th>max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{bc}$</td>
<td>Host the start signal down time</td>
<td>0.8</td>
<td>1</td>
<td>20</td>
<td>ms</td>
</tr>
<tr>
<td>$T_{go}$</td>
<td>Bus master has released time</td>
<td>20</td>
<td>30</td>
<td>200</td>
<td>$\mu$s</td>
</tr>
<tr>
<td>$T_{rel}$</td>
<td>Response to low time</td>
<td>75</td>
<td>80</td>
<td>85</td>
<td>$\mu$s</td>
</tr>
<tr>
<td>$T_{reh}$</td>
<td>In response to high time</td>
<td>75</td>
<td>80</td>
<td>85</td>
<td>$\mu$s</td>
</tr>
<tr>
<td>$T_{low}$</td>
<td>Signal &quot;0&quot;, &quot;1&quot; low time</td>
<td>48</td>
<td>50</td>
<td>55</td>
<td>$\mu$s</td>
</tr>
<tr>
<td>$T_{h0}$</td>
<td>Signal &quot;0&quot; high time</td>
<td>22</td>
<td>26</td>
<td>30</td>
<td>$\mu$s</td>
</tr>
<tr>
<td>$T_{h1}$</td>
<td>Signal &quot;1&quot; high time</td>
<td>68</td>
<td>70</td>
<td>75</td>
<td>$\mu$s</td>
</tr>
<tr>
<td>$T_{en}$</td>
<td>Sensor to release the bus time</td>
<td>45</td>
<td>50</td>
<td>55</td>
<td>$\mu$s</td>
</tr>
</tbody>
</table>
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 it 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:

```c
// 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 `Enumera-
tion::DeviceInformation` class, first, a string that contains a search criterion for
finding Arduino must be created, using the static method `GetDeviceSele-
torFromUsbVidPid(unsigned int, unsigned int)` in the `SerialDevice` class.
This is show in the code snippet below:

```csharp
IAsyncOperation<DeviceInformationCollection^>^ InterfacingArduino_Wrk::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.
    return DeviceInformation::FindAllAsync(aqs);
}
```

The `FindAllAsync(String^)` method returns an `IAasyncOperation<DeviceIn-
f ormationCollection^>^` 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 af-
ter the other, forming a chain of tasks. The first one is created using a
`FindAllAsync(String^)` method. The continuation task, which is created by call-
ing 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:
It should be noted that the antecedent task returns `task<DeviceInformationCollection>` 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.

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:

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

    switch((rxCommand & 0x0F))
    {
      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.
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:

```c
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.
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:

```c
// 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.
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.

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

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 re-
sponse.

After a command + data response was received, appropriate variables were ini-
tialized for data processing.

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

  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<wint8_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.

```c
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.

```c
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.
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.
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


   https://msdn.microsoft.com/library/windows/apps/br206587


   https://en.wikipedia.org/wiki/USB


   http://firmata.org/wiki/Protocol


    https://en.wikipedia.org/wiki/Arduino

    http://playground.arduino.cc/Main/DHTLib

APPENDIX

/*
UserStory: =>In home the user can check data from RaspberryPi (GUI-Interface)
   =>The data can be temperature, humidity, state of door (open/closed)
   =>From Handy the user can check temperature,humidity,state of door

An implementation of a home automation system using UWP in C++/CX
This app runs on RaspberryPi2 and connects to Arduino via a USB and gets Sensor data as well as analog pin value
By Bereket Godeho DITBSN
OMIK-School of Engineering

MainPage.xaml.h
Implementation of the MainPage class.
*/

#pragma once
#include "MainPage.g.h"

namespace InterfacingArduino_UWP
{

    public ref class AirCondition sealed
    {
        private:
            Platform::String ^humidity;
            Platform::String ^temperature;
            Platform::String ^_timeStamp;
        public:
            AirCondition(
                Platform::String ^humidity_,
                Platform::String ^temperature_,
                Platform::String ^_timeStamp_)
            {
                _humidity(humidity_);
                _temperature(temperature_);
                _timeStamp(_timeStamp_);
            }

            property Platform::String ^Humidity
            {
                Platform::String ^get() { return _humidity; }
            }

            property Platform::String ^Temperature
            {
                Platform::String ^get() { return _temperature; }
            }

            property Platform::String ^TimeStap
            {
                Platform::String ^get() { return _timeStamp; }
            }
    }

    public ref class Database sealed
    {
        private:
            sqlite3 * _database;
            sqlite3_stmt * _statement;
            const char * _sqlCreate =
                "CREATE TABLE IF NOT EXISTS h_t_table(humidity real, temperature real, time_stamp text);"
            const char * _sqlInsert =
                "INSERT INTO h_t_table VALUES (?, ?, ((SELECT time('now', 'localtime'))));"
            const char * _sqlSelect =
                "SELECT * FROM h_t_table;"
            int _result;
        }
```csharp
<Database();
   Platform::String ^convertChar(char * originalChar_);

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

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

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

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

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

public enum class Command
{
   REPORT_DIGITAL_DATA = 0xA0,
   DIGITAL_DATA = 0xA9,
   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 Window::Foundation::IAsyncOperation
      <Window::Devices::Enumerations::DeviceInformationCollection ^>
      ^lstAvailableArduinoAsync(void);
MainPage();

property InterfacingArduino_WP::AirConditionViewModel ^viewModel
{
    InterfacingArduino_WP::AirConditionViewModel ^get()
    {
        return _viewModel;
    };
}

private:
    // arduino pid = 0043 and vid = 2341
    static const unsigned int _pid = 0x0043;
    static const unsigned int _vid = 0x2341;
    // used for alternatively sending Digital or Analog data request Command
    InterfacingArduino_WP::Command ^txCommand;

    Platform::String ^_path =
        Windows::Storage::ApplicationData::Current->LocalFolder->Path + "\test.db"; // IAUP00
    AirConditionViewModel ^_viewModel;

    Windows::Devices::Enumeration::DeviceInformationCollection ^arduinoCollection;
    Windows::Devices::Enumeration::DeviceInformation ^deviceInfo;
    Windows::Devices::SerialCommunication::SerialDevice ^arduino;
    Windows::Storage::Streams::DataWriter ^tx;
    Windows::Storage::Streams::DataReader ^rx;

    Windows::UI::Xml::XPath::DispatcherTimer ^_timer;

    void SendCommand(void);
    void ReceiveCommand(void);
    void ProcessData(void);
    void StartTimer(void);
    void CloseArduino(void);

    Concurrency::task<void> GetAvailableArduinosAsync(void);
    Concurrency::task<void> SendAsync(Command cmd_);  
    Concurrency::task<void> ReceiveAsync(void);
    Concurrency::task<void> GetArduinoFromIdAsync(
        Windows::Devices::Enumeration::DeviceInformation ^deviceInfo_);

    void OnLoaded(Platform::Object ^sender, Windows::UI::Xml::RoutedEventArgs ^e);
    void OnTick(Object^ sender, Object^ e);

    [};
Source Code

APPENDIX

```cpp
/*
UserStory: =>In home the user can check data from RaspberryPi (GUI-Interface)
=The data can be temperature, humidity, state of door (open/closed)
=From Handy the user can check temperature,humidity, state of door
An implementation of a home automation system using UWP in C++/CX
This app runs on RaspberryPi2 and connects to
Arduino via a USB and gets Sensor data as well as analog pin value

By Bereket Gedeno DITBSN
OAHK-School of Engineering

MainPage.xaml.cpp
Implementation of the MainPage class.
*/

#include "pch.h"
#include "MainPage.xaml.h"

using namespace InterfacingArduino_UWP;

using namespace Platform;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;

using namespace concurrency;
using namespace Windows::Devices::Enumeration;
using namespace Windows::Devices::SerialCommunication;

// The Blank Page item template is documented at
// http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

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 eqs = SerialDevice::GetDeviceSelectorFromUsbVidPid(_vid, _pid);
    // Pass the retrieved string to FindAllAsync.
    // The call retrieves a DeviceInformationCollection object.
    return DeviceInformation::FindAllAsync(eqs);
}

MainPage::MainPage()
{
    InitializeComponent();
    // populate the ListView data from local db
    _viewModel = ref new InterfacingArduino_UWP::AirConditionViewModel(_path);
    // so that connections initialize after the components are initialized
    Loaded += ref new Windows::UI::Xaml::RoutedEventArgsHandler(this, &InterfacingArduino_UWP::MainPage::OnLoaded);
}
```
```csharp
void InterfacingArduino_WNP::MainPage::
    OnNavigatedTo(Platform::Object^ sender, Windows::UI::Xaml::RoutedEventArgs^ e)
{
    GetAvailableArduinosAsync();
    _timer = ref new Windows::UI::Xaml::DispatcherTimer();
    _timer->Tick += ref new EventHandler<Object^>(this, &MainPage::OnTick);
}

task<void> InterfacingArduino_WNP::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
            }
        });
}

void InterfacingArduino_WNP::MainPage::
    GetArduinoFromIdAsync(DeviceInformation ^deviceInfo)
{
    return create_task(serialDevice::FromIdAsync(deviceInfo->Id)).
        then([this](SerialDevice *serial_device)
        {
            // save the serial object to member variable
            _arduino = serial_device;

            if (_arduino != nullptr)
            {
                Windows::Foundation::TimeSpan _timeOut;
                _timeOut.Duration = 1000000; // = 1 second
                // ... more code ...
            }
        });
}```
// Configure serial settings
_orgduino->WriteTimeout = _timeOut;
_orgduino->ReadTimeout = _timeOut;
_orgduino->BaudRate = 9600;
_orgduino->Parity =
    Windows::Devices::SerialCommunication::SerialParity::None;
_orgduino->StopBits =
    Windows::Devices::SerialCommunication::SerialStopBitCount::One;
_orgduino->DataBits = 8;
_orgduino->Handshake =
    Windows::Devices::SerialCommunication::SerialHandshake::None;

// 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
}
else
{
    // the variable _arduino can be null at this point
    // If for example, the app does not have 'serialcommunication'
    // capabilities specified in the app manifest
    CloseArduino();
}

});// void InterfacingArduino_UIP::MainPage::SendCommand(void)

if (_arduino != nullptr)
{
    // The intention 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));
}
```c
void InterfacingArduino_UDP::MainPage::SendAsync(Command cmd_)
{
    uint8_t reportCommand = static_cast<uint8_t>(cmd_);
    _tx->WriteByte(reportCommand);
    return create_task(_tx->StoreAsync());
}

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

void InterfacingArduino_UDP::MainPage::ReceiveAsync()
{
    unsigned int readBufferSize = 6;
    return create_task(_rx->LoadAsync(readBufferSize)).then([this](unsigned int bytesRead)
    {
        if (bytesRead > 0)
        {
            ProcessData();
        }
    });
}

void InterfacingArduino_UDP::MainPage::ProcessData()
{
    Database db(_path);
    std::vector<uint8_t> receivedData;
    std::size_t bytesRemaining = 0;
    DigitalDataStatus dataStatus;
    uint8_t analogPinNumber;
    uint16_t rxAnalogData = 0;
    uint8_t data = _rx->ReadByte();
    double humidity;
    double temperature;

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

    switch (cmd)
    {
    case Command::DIGITAL_DATA:
        bytesRemaining = 3;
        // Lower nibbles represent the sensor data status
        dataStatus = static_cast<DigitalDataStatus>(data & 0xF);
        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--;
}

if (cmd == Command::DIGITAL_DATA &&
    dataStatus == DigitalDataStatus::DATA_OK)
{
    humidity = ((receivedData.at(0) << 8) | receivedData.at(1)) / 10;
    temperature = ((receivedData.at(2) & 0x7F) << 8) | receivedData.at(3) / 10;

    if (receivedData.at(2) & 0x80)
    {
        temperature -= temperature;
    }

    CurrentHTValuetextBlock->Text =
        "Current H(%): " +
        humidity +
        " And T(°C): " +
        temperature +
        "\\n";

    // save into the db
    if (db.Result == SQLITE_OK) // check if connection is open
    {
        db.createTable(); // create table if not exits
    }

    if (db.Result == SQLITE_DONE) // check if creating table was success
    {//  save the data
        db.insertIntoTable(humidity, temperature);
    }

    } else if (cmd == Command::DIGITAL_DATA &&
    dataStatus == DigitalDataStatus::DATA_ERROR_TIMEOUT)
    {
    CurrentHTValuetextBlock->Text = "Sensor Not connected!";
    }
else if (cmd == Command::ANALOG_DATA)
{
    rXAnalogData = (receivedData.at(0) | receivedData.at(1) << 5);
    AnalogPinValueTextBlock->Text =
        "Analog Value at Pin number: " +
        analogPinNumber + " is " +
        rXAnalogData;
    }
StartTimer();}
```cpp
void MainPage::StartTimer() {
    TimeSpan ts;
    ts.Duration = 3000000000L; // in 3 seconds interval
    _timer->Interval = ts;
    _timer->Start();
}

// when timer overflows
void MainPage::OnTick(Object^ sender, Object^ e) {
    // reset and send command again
    _timer->Stop();
    SendCommand();
}

void InterfacingArduino_WNP::MainPage::CloseArduino(void)
{
    delete(_arduinoCollection);
    _arduinoCollection = nullptr;

    delete(_arduino);
    _arduino = nullptr;

    delete(_deviceInfo);
    _deviceInfo = nullptr;

    delete(_tx);
    _tx = nullptr;

    delete(_rx);
    _rx = nullptr;

    delete(_timer);
    _timer = nullptr;
}

//***********************************************************************************
//** Support class definitions**
//***********************************************************************************

InterfacingArduino_WNP::AirConditionViewModel::
AirConditionViewModel(Platform::String ^ path_)
{
    Database db(path_);
    if (db.Result == SQLITE_OK)
    {
        db.createTable();
    }

    if (db.Result == SQLITE_DONE)// Successfully created table IF NOT EXISTS
    {// then populate the collection IF there is any
        db.selectFromTable(Recordings);
    }
```

```c
365  InterfacingArduino_UPN::Database::Database()
366  {
367      sqlite3_close(_database);
368  }
369
370  Platform::String ^ InterfacingArduino_UPN::Database::
371      convertChar(char * originalChar_,
372      { std::size_t newsize = strlen(originalChar_) + 1;
373          wchar_t * wcstring = new wchar_t[newsize];
374          size_t convertedChars = 0;
375          mbstowcs_s(&convertedChars, wcstring, newsize, originalChar_, _TRUNCATE);
376          String^ str = ref new Platform::String(wcstring);
377          delete[] wcstring;
378          return str;
379      }
380  }
381
382  InterfacingArduino_UPN::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 do
388      _result = sqlite3_open_v2(path,
389          &_database,
390          SQLITE_OPEN_READWRITE |
391          SQLITE_OPEN_CREATE,
392          NULL);
393  }
394
395  void InterfacingArduino_UPN::Database::createTable(void)
396  {
397      sqlite3_stmt *statement;
398      if (sqlite3_prepare_v2(_database,
399          _sqlCreate,
400          -1,
401          &statement,
402          0) == SQLITE_OK)
403      {
404          _result = sqlite3_step(statement);
405      }
406      sqlite3_finalize(statement);
407  }
408
409  void InterfacingArduino_UPN::Database::
410      selectFromTable(Windows::Foundation::Collections::IVector<AirCondition>^ recordings_)
411  {
412      sqlite3_stmt *statement;
413      Platform::String ^humidity;
414      Platform::String ^temperature;
415      Platform::String ^timestamp;
416      AirCondition ^data;
417      if (sqlite3_prepare_v2(_database,
418          _sqlSelect,
419          -1,
420          &statement,
421          0) == SQLITE_OK)
422      {
423      }
424  ```
```c
int cols = sqlite3_column_count(statement);
while (sqlite3_step(statement) == SQLITE_ROW)
{
    for (int col = 0; col < cols; col++)
    {
        char *ptr = (char *)sqlite3_column_text(statement, col);
        if (ptr)
        {
            switch (col)
            {
            case 0:
                humidity = convertChar(ptr);
                break;
            case 1:
                temperature = convertChar(ptr);
                break;
            case 2:
                timeStamp = convertChar(ptr);
                break;
            }
        }
    } //row complete

    data = ref new AirCondition(humidity, temperature, timeStamp);
    recordings_ -> Append(data);
} //while
sqlite3_finalize(statement);

void InterfacingArduino_UWP::Database::insertIntoTable(double h_, double t_)
{
    sqlite3_stmt *statement;

    if (sqlite3_prepare_v2(_database, _sqlInsert, -1, &statement, 0) == SQLITE_OK)
    {
        //int sqlite3_bind_double(sqlite3_stmt*, int, double);
        if (sqlite3_bind_double(statement, 1, h_) ==
            SQLITE_OK &&
            sqlite3_bind_double(statement, 2, t_) ==
            SQLITE_OK)
        {
            _result = sqlite3_step(statement);
        }
    }
    sqlite3_finalize(statement);
```
```
UserStory: In home the user can check data from RaspberryPi (GUI-Interface)
   - The data can be temperature, humidity, state of door (open/closed)
   - From here the user can check temperature, humidity, state of door
   
   Part of an implementation of a home automation system using ULP in C++/CX on the RaspberryPi side
   This an Arduino program that connects via a USB to Raspberry Pi 2
   and sends Digital Sensor data as well as analog pin value

   By Berak Godebo DIT185N
   OAMK-School of Engineering

   */

#include <dht.h>

#define DHT21_PIN 2

const unsigned char REPORT_DIGITAL_DATA = 0xA0;
const unsigned char DIGITAL_DATA = 0xB8;
const unsigned char REPORT_ANALOG_DATA = 0xC0;
const unsigned char ANALOG_DATA = 0xDB;
const unsigned char datamask = 0xF;

int analogPinValue = 0;
unsigned char txCommand = 0;

// digital data status to be sent along the command
const unsigned char DATA_OK = (0 & 0x0F);
const unsigned char DATA_ERROR_CHECKSUM = (1 & 0x0F);
const unsigned char DATA_ERROR_TIMEOUT = (2 & 0x0F);

// these values are to be sent with a resolution of 4-bits thus the mask=0x0F
const unsigned char analogPinA0 = (0 & 0x0F);

/* for future use, for now only analog pin A0 is used
const unsigned char analogPinA1 = (1 & 0x0F);
const unsigned char analogPinA2 = (2 & 0x0F);
const unsigned char analogPinA3 = (3 & 0x0F);
const unsigned char analogPinA4 = (4 & 0x0F);
const unsigned char analogPinA5 = (5 & 0x0F);

*/

void sendAnalogPinValue(void);
void sendDigitalSensorValue(void);

DHT dht;

void setup() {
    Serial.begin(9600);
}
```
```c
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;
    }
  }
}

void sendDigitalSensorValue(void) {
  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
  Serial.write(txCommand);
  Serial.write(dht.getBuffer(), bufSize);
}

void sendAnalogPinValue(void) {
  analogPinValue = analogRead(analogPinA0);
  unsigned char lowerByte = lowByte(analogPinValue);
  unsigned char higherByte = highByte(analogPinValue);

  // get the lower five bits of lowerByte
  // dataMask = 0b000001111
  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;

  // 3 bits representing the analog pin number and a command to signal
  // that analog message is being sent
  txCommand = (ANALOG_DATA | analogPinA0);

  // send the command and then the two byte data
  Serial.write(txCommand);
  Serial.write(firstAnalogDataByte); // a byte containing 5-bits of data
  Serial.write(secondAnalogDataByte); // a byte containing 5-bits of data
```