ISTANBUL TECHNICAL UNIVERSITY

ELECTRICAL – ELECTRONICS ENGINEERING FACULTY

DESIGN AND IMPLEMENTATION OF A SECURE BLUETOOTH LOW ENERGY COMMUNICATION

BSc Thesis by

Bahadır GÜN

Department: Electronics and Communication Engineering

Programme: Electronics Engineering

Supervisor: Assoc. Prof. Dr. Sıddıka Berna ÖRS YALÇIN

MAY 2015

ACKNOWLEDGEMENT

First, I would like to sincerely thank my supervisor, Assoc. Prof. Dr. Sıddıka Berna Örs Yalçın, for her guidance and patience throughout this project. Her unselfish efforts have helped me significantly and I am deeply grateful.

Also, I would like to thank all my friends for their support. Their company has always kept me entertained and motivated in times of struggle.

Finally, I want to express my endless gratitude and appreciation to my family, who have supported my decisions and guided me with their experience. Without their support, I would not be in this point of my life.

Bahadır GÜN

MAY 2015

ÖZET

Basit ve küçük boyuttaki elektronik cihazların etkinlik alanları hızla artmaktadır. Özellikle *Nesnelerin İnterneti (Internet of Things - IoT),* akıllı binalar veya giyilebilir elektronik cihazlar gibi kavramlarla beraber bulunduğu ortamdan veya kullanıcıdan verilerin toplanıp, işlenmek üzere akıllı telefon veya bilgisayar gibi daha yetenekli cihazlara gönderilmesi yaygınlaşmaktadır. Veri toplayan cihazlarda iletişim protokolü olarak düşük güç tüketimi sebebiyle *Düşük Enerjili Bluetooth (Bluetooth Low Energy - BLE)* protokolü yaygınca kullanılmaktadır fakat BLE protokolünde iki cihazın eşleşme sürecinde bir güvenlik açığı bulunmaktadır. İletişim kanalını dinleyebilen pasif bir dinleyici paylaşılan anahtarı elde edip yapılan şifreli veri transferini çözebilmektedir.

Kişisel bilgiler taşıyabileceği için toplanan verilerin güvenli bir şekilde iletilmeleri yüksek öneme sahiptir ve BLE protokolündeki güvenlik zaafını gidermek için *Eliptik Eğrili Diffie-Hellman (Elliptic Curve Diffie-Hellman - ECDH)* anahtar paylaşım protokolü iki BLE uyumlu cihaz üstünde gerçeklenmiştir. Merkezi cihaz bilgisayar aracılığıyla çalıştırılıp, kodları C# dilinde yazılırken, çevresel cihaz Arduino aracılığıyla çalıştırılıp, kodları C dilinde yazılmıştır.

Güvenli bir şekilde anahtar paylaşımı yapıldıktan sonra veri transferi, günümüze kadar işlevsel bir saldırı yapılamamış *Üstün Şifreleme Standardı (Advanced Encryption Standard - AES)* algoritması ile şifrelenmiştir.

SUMMARY

Usage areas of small, simple electronic devices are rapidly increasing with concepts such as *Internet of Things (IoT),* smart buildings and wearable electronics. These devices collect data from their environment or their user and generally send the data to more capable devices, such as smartphones or computers, to be processed. Due to its low power consumption, one of the most popular communication protocols for these devices is *Bluetooth Low Energy (BLE).* However, it contains a security vulnerability in its pairing process. A passive eavesdropper may obtain the shared key and use it to decrypt the communication.

Securely transmitting these types of data is of high importance, since it may contain personal, confidential information. In order to overcome this security vulnerability, *Elliptic Curve Diffie-Hellman (ECDH)* key exchange protocol is implemented on two BLE devices. The master device is run through the computer and its program is written in C#, whereas the slave device is run through Arduino and its program is written in C.

After securely sharing a key, the data transfer is encrypted with *Advanced Encryption Standard (AES)* algorithm, which until today, has no known practical attacks against it.

INDEX

1. INTRODUCTION

With the increasing popularity of concepts such as *Internet of Things (IoT)*, electronic devices are starting to play bigger roles in our lives. In many areas, small and simple sensors are collecting data and this data is usually sent to another, more capable electronic device, such as a smart phone or a computer. The collected data carries information about either an environment or about a person. It is important to secure this kind of confidential information while continuing to use these electronic devices to our benefit.

The main way to provide security is to use encryption in the sensors' communication. To encrypt a communication, the parties need to establish a key and an encryption method. Since the communication is not encrypted until after a key has been negotiated, the key cannot be shared directly and a secure key sharing protocol should be used. After a key has successfully been negotiated the parties should use a reliable encryption method to secure the communication between.

The security protocol of the communication should be chosen with respect to the communication method's properties. Most of the aforementioned simple sensors are designed with the goal of minimizing their area and their power consumption. Thus, the preferred communication method should also support these design goals. The *Bluetooth Low Energy (BLE, also known as Bluetooth Smart)* is a protocol which is developed specifically to be used in such scenarios. It is the inspected communication method in this thesis, as it consumes less power compared to the similar communication protocols both during transmission [1] and throughout a cyclic sleep scenario [2].

1.1. Motivation

BLE protocol is extensively used in low power communication in various devices and scenarios; however, it has one drawback regarding its security. During the pairing process, first a 128-bit Temporary Key (TK), known to both parties, is used to generate a 128-bit Short Term Key (STK). Then an encrypted connection is started using STK. Then through this encrypted connection a 128-bit Long Term Key (LTK) is established. Both parties store LTK and use it to generate session key. Once the devices are paired, the session key changes for every connection but it is generated by the previously established LTK [3].

BLE offers three pairing methods which are only different until STK generation: Just Works, Passkey Entry and Out of Band (OOB). In Just Works, TK is always zero. This method is meant for devices with limited input/output capabilities. In Passkey Entry, a 6-digit Personal Identification Number (PIN) padded with zero to 128-bit is used as TK. To use this method, the devices must at least have display on one side and a keyboard on the other. A 6-digit PIN is always generated randomly and shown on a display, which should be entered via keyboard to the other device. In OOB, devices distribute a key using a different communication method instead of BLE, such as Near Field Communication (NFC).

It is stated in the Bluetooth Specification Version 4.0 that *"None of the pairing methods provide protection against a passive eavesdropper during the pairing process as predictable or easily established values for TK are used."*. This vulnerability is explained in detail by **Ryan (2013)** that it is even possible to force paired devices to negotiate a key once more and obtain the security information [4]. He also states that despite the vulnerability of the pairing process, the session encryption of BLE is adequately strong, if a key can be securely established. He concludes by offering a secure key exchange protocol such as *Elliptic Curve Diffie-Hellman (ECDH)* to be added to the BLE protocol.

After the starting date of this project, September 2014, Bluetooth SIG has published Bluetooth Specification Version 4.2 in December 2014, which now has a secure pairing mode with ECDH [5]. Despite this progress, there are billions of BLE

devices which has been manufactured without a secure pairing option and this project can still be used with these devices.

We have searched for a way to implement pairing security for the devices with the outdated specifications. Exchanging a key with ECDH and using it as a OOB key, since only in OOB a full 128-bit TK can be used, could solve the problem; however, most of the devices are designed with a minimalistic approach and do not have OOB support. Thus, we implemented our ECDH scheme on both parties and used the generated key with 128-bit Advanced Encryption Standard (AES) to establish communication security.

1.1.1. Building as a Service (BaaS)

Though our project can be used in almost any context where BLE is the preferred method of communication, it was originally intended to be used in the European Union funded *Building as a Service (BaaS)* project. BaaS is simply explained in its website as *"Software platform for configuration, operation and maintenance of intelligent building infrastructures"* [6]. Secure BLE communication is currently to be used just in the evacuation system, though BaaS covers many aspects of intelligent commercial buildings. Each person in the building is planned to have a device with an active Radio Frequency Identification (RFID) tag and a BLE chip in slave role. An indoor localization algorithm runs on the device and calculates its position with the help of passive RFID tags placed inside the building. Then, the location information is transferred to a central processing unit (CPU) via BLE. Since the location information of a person is confidential, the communication should be secure. The security protocol is implemented in both the slave device and the CPU.

2. PRELIMINARY INFORMATION

In this section, protocols, standards and techniques which are used in our project are explained. The explanations contain information related to the project. For example;

a working implementation of AES algorithm is used but it is unrelated to our project how it was implemented; thus this information is not given.

2.1. Elliptic Curve Diffie-Hellman (ECDH)

Elliptic curve Diffie–Hellman is a key establishment protocol which enables two parties to securely negotiate a shared secret key over an insecure channel. This protocol differs from the Diffie-Hellman protocol by using addition on elliptic curves instead of using multiplication and modulo operation. The parties should agree on domain parameters (*q, FR, a, b*{*, SEED*}*, G, n, h*) beforehand, which enables them to generate keys that are members of an elliptic curve. These parameters must satisfy the specifications provided by the *National Institute of Standards and Technology (NIST)* [7]. With these parameters, addition can be done on elliptic curves but not subtraction, which prevents an eavesdropper to calculate the original data from a sum.

To start the negotiation, the parties, A and B, first generate a private key *d* which is a random integer in the range [1, n-1]. Then a public key $Q = (x_Q, y_Q) = dG$, by multiplying the parameter *G*, a special point on the elliptic curve, with the private key *d*. The multiplication is done by using addition on the elliptic curve *d* times. The public key *Q* is also a point on the elliptic curve. At the end of these phase, A has its public-private key pair of d_A, $Q_A=d_AG$ and B has d_B, $Q_B=d_BG$. Due to the properties of the elliptic curve, an eavesdropper cannot expose the private key from the public key even with the knowledge of the multiplicand *G* and the result.

After key pair generation, the parties send each other their public key. Upon receiving the other side's public key, each party confirms that it is a point on the elliptic curve and then multiplies it with their own private key to calculate the shared secret key. At the end of this phase, A has d_AQ_B and B has d_BQ_A. It can be seen that $d_AQ_B = d_Ad_BG = d_Bd_AG = d_BQ_A$ and both parties have calculated the same value, without exposing their private keys over the insecure channel. The calculated value is also a point on the elliptic curve and its x-coordinate value is used as the shared secret key.

2.2. Advanced Encryption Standard (AES)

A working implementation of AES is used directly in our project. The underlying operation of the algorithm is not in the context of our project; however, information about its operation can be found in detail in *Federal Information Processing Standards Publication 197 (FIPS-197)* published by NIST [8].

However, we are interested in the security of AES. Since the Rijndael algorithm is accepted as AES, there have been numerous attempts to "break" it. As of today, a practical attack against AES has not been discovered. One of the last published attacks could retrieve a 128-bit AES key in $2^{126.1}$ tries instead of 2^{128} tries (as in a brute-force attack) but the attack is still impractical [9].

2.3. Bluetooth Low Energy (BLE)

This communication protocol is first defined by the *Bluetooth Special Interest Group (SIG)* in the Bluetooth Specification Version 4.0 [3]. It uses the unlicensed 2.4 GHz band for radio communication. The available band is also divided into 40, 2 MHz apart, physical channels. Three of these channels are for advertising while the rest is used for data transfer. During communication, two devices change (hop) the channels they use regarding the connection parameters, such as hopping interval and hopping increment, they previously agreed on.

BLE protocol stack consists of many layers and a schematic can be seen in Figure 2.1. Though it is not in the figure, a physical layer (PHY) encapsulates the shown layers. PHY layer includes the specifications such as the operating band, the modulation technique, the bit rate etc. PHY layer interacts with the Generic Access Profile (GAP) layer, which specifies the connectivity parameters of a device. These parameters include antenna gain, the device's name and appearance, the content and the sending interval of advertising messages etc. Slave devices can advertise in three different types. A connection without pairing (bonding) or a paired connection may be requested. Also information carrying messages may be broadcasted that does not allow connection. After the connectivity is established with the GAP layer, interaction at application level can be initiated.

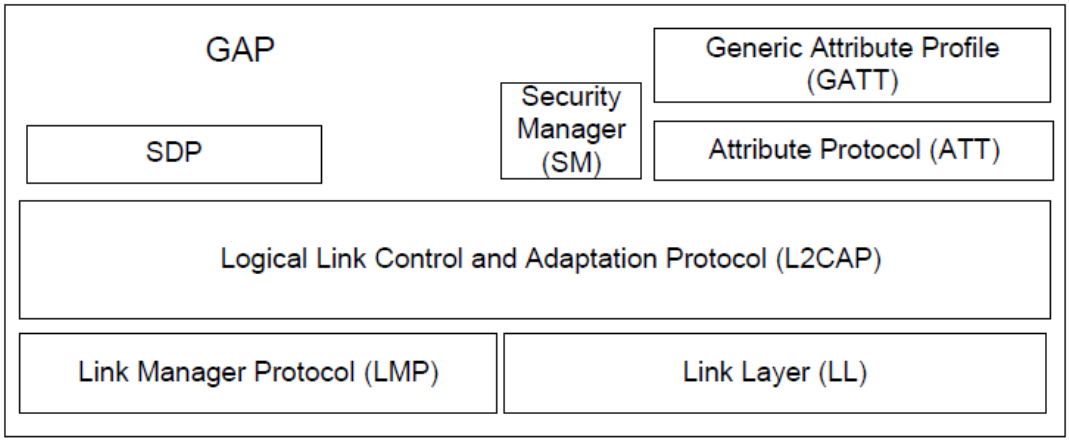

Figure 2.1 : Relationship of GAP with lower layers

GAP interacts with the Generic Attribute Profile (GATT) layer next. This layer contains attributes, which are discrete values with these three properties: an attribute type, an attribute handle and a set of permissions.

An attribute type is a 128-bit Universally Unique Identifier (UUID), which specifies the properties of the attribute. Bluetooth SIG has defined some attribute types, such as a heart rate monitor, but it is also possible for the user to define custom types. This helps interoperability between BLE devices.

An attribute handle is a 16-bit value, which can be used to reference a specific attribute. An application may have several heart rate monitors connected to it, which all have the same attribute type, and the user can reference a specific attribute by its handle.

The set of permissions specifies the access rights to an attribute. An attribute may be read only, write only or can only be read with a notification etc. For example the BLE device of a patient's heart rate monitor may have write only permission to an attribute and the BLE device of a doctor/nurse may have read only permission to the same attribute.

Applications can only send or receive data by modifying these attributes. An application may write a data into an attribute, then another device's application can read this attribute and complete a successful data transfer. A simple interaction schematic of layers can be seen in Figure 2.2.

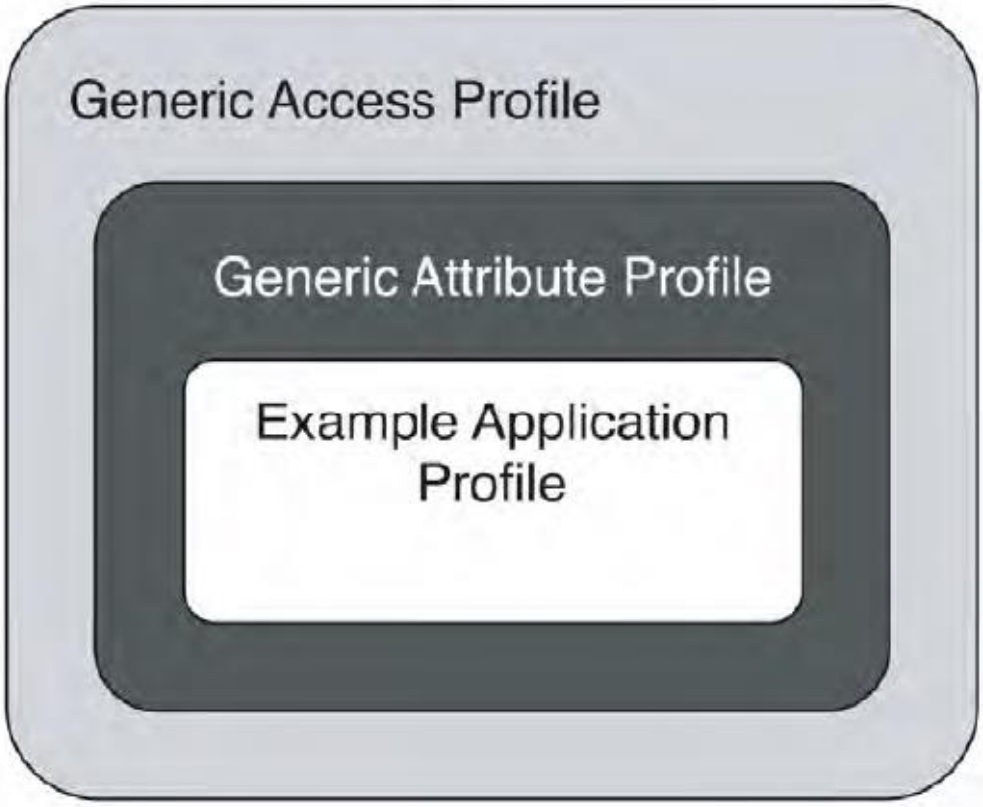

Figure 2.2 : Layer dependencies/interactions

2.4. C# and C/C++ Interoperability

In the final stages of our project, we used the Application Programmable Interface (API), given by our product's manufacturer, for the master role operation. This API is written in C# and manages the functionality of the master operation. However, all of our previously collected functions for ECDH scheme and AES encryption was written in C. These languages have some fundamental differences and some adjustments should be made to achieve interoperability.

In C, header files are used to call a function from a library; however in C#, header files are not used and libraries must be included as Dynamic Link Library (DLL) files.

Also, C# language is defined by Microsoft as a managed code, which its compiler manages its own memory deallocation (garbage collection) and does not allow the use of pointers. This way, the program will not have memory leaks or access

violations [10]. On the other hand, C uses pointers extensively and most of the functionality is achieved through pointer usage.

2.4.1. Changes on the C code

A C library can be converted easily to an unmanaged DLL file by using Microsoft Visual Studio (VS) with some small changes. Within VS, a new Visual C++ Win32 Console Application is created and the C codes are added to this project, both the header file and the C file. At the start of the header file *stdexcept* file is included. Then, every function prototype in the header file is wrapped with *extern "C"* keyword and before their return type declaration, *__declspec(dllexport)* keyword is added. The code should look like below [11].

```
#include <stdexcept>
using namespace std;

namespace MathFuncs
{
   extern "C" { __declspec(dllexport) double Add(double a, double
b); }
   extern "C" { __declspec(dllexport) double Subtract(double a,
double b); }
   extern "C" { __declspec(dllexport) double Multiply(double a,
double b); }
   extern "C" { __declspec(dllexport) double Divide(double a, double
b); }
}
```

Finally, the project is built with the */LD* option. With this option, if a DllMain function is not present in the project, the linker automatically inserts a DllMain function which returns TRUE [12]. Without a DllMain function, a DLL can be created without an error; however, when another program tries to call a function from this DLL, it cannot find an entry point (main function) and program crashes.

After the DLL is created, its exported functions can be checked by entering *dumpbin /exports "DLL_Name.dll"* command into the VS command prompt. The output of the DLL used in our project is given in Figure 2.3.

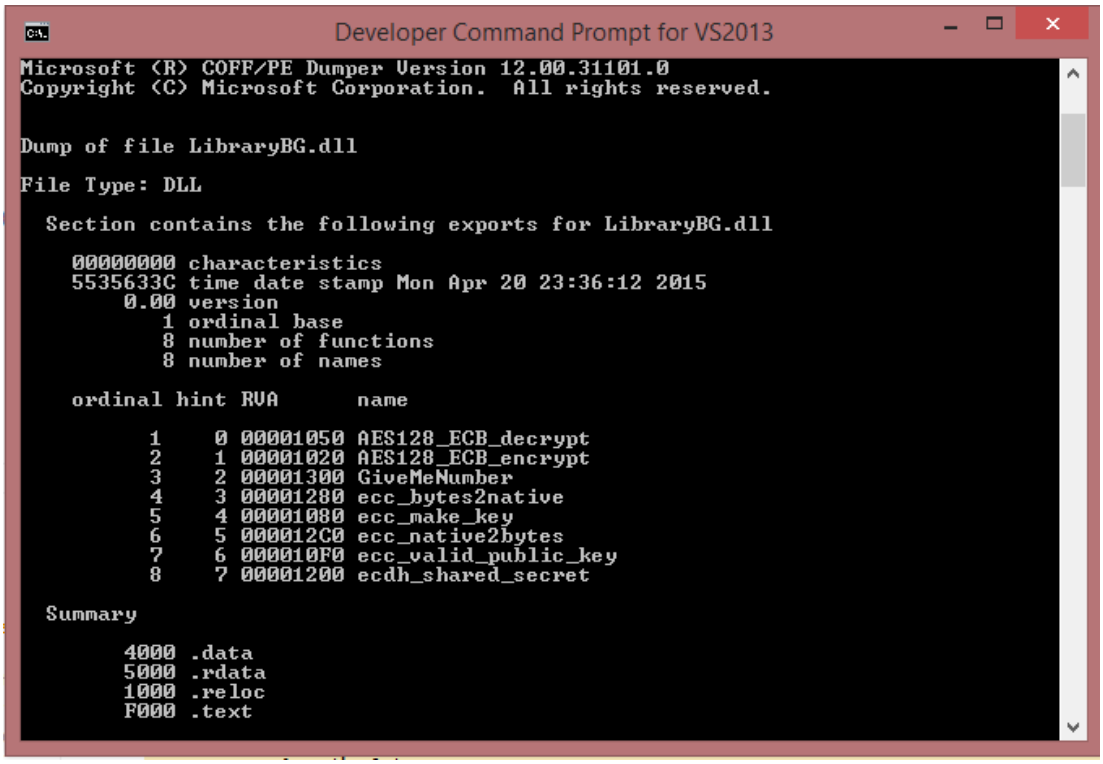

Figure 2.3 : Sample output of *dumpbin /exports* command

2.4.2. Changes on the C# code

If the exported C functions' arguments and return types do not have an unmanaged type such as a pointer or a struct, then by adding the command `[DllImport("lib.dll", CallingConvention = CallingConvention.Cdecl)]` and the *extern static* keywords before the function prototype. For example, the code below works perfectly fine.

```
using System; // Console
using System.Runtime.InteropServices; // DllImport

class App
{
    [DllImport("lib.dll", CallingConvention =
CallingConvention.Cdecl)]
    extern static int next(int n);

    static void Main()
    {
        Console.WriteLine(next(0));
    }
}
```

However, if the function requires interaction with an unmanaged type, then some additional work needs to be done.

2.4.2.1. Code Marshaling

Marshaling is the bridging operation between unmanaged types and managed types. .NET framework has a Marshal class, which contains useful functions (methods) to use in this process.

Some of the data types are common to managed code and unmanaged code. These types are called blittable types and are the following: signed and unsigned 8-bit, 16-bit, 32-bit, 64-bit integers; signed and unsigned pointers [13]. With these types, code marshaling is automatically done. For example a C function with the return type *uint8_t* can be used in C# with the return type *byte*. As for the pointer usage, the type *IntPtr* in C#, creates a pointer suitable to the operating system (OS). With the *IntPtr* usage, the memory increments should be carefully adjusted, since an increment on an *uint8_t* pointer in C is not equal to an increment on an *IntPtr* in a 64-bit OS.

Usage of *IntPtr* in C# is similar to the pointer usage in C. A block of memory can be allocated for *IntPtr* with the method *Marshal.AllocHGlobal* and after its usage this memory should be deallocated with the method *Marshal.FreeHGlobal*, since this type is not managed by the garbage collector of C# compiler [14].

However, if the pointers are used for passing arrays or blocks of data, C# arrays can be used instead of *IntPtr*. For example, our ECDH and AES functions work with 128-bit data blocks but they are written for 8-bit processors, so they use 8-bit data. These functions take *uint8_t* pointers as arguments but they only use them to store 128-bit data blocks. So if we have a *byte* array with 16 elements in C#, we can pass its address as an argument to these functions. The array sizes must be arranged properly, otherwise access violations may occur. Prototypes of a sample function in C and C# are given below.

```
extern "C" __declspec(dllexport) void AES128_ECB_encrypt(uint8_t*
input, uint8_t* key, uint8_t *output);
```

Function Prototype in C, with the added modifications

```
[DllImport(DllAddress, CallingConvention = CallingConvention.Cdecl)]
public static extern void AES128_ECB_encrypt(byte[] input, byte[]
key, byte[] output);
```

Its usage in C#

Unlike these blittable types, usage of structs needs some marshaling operations. C structs can be used as structs or classes in C#. We are only interested in C structs, which may contain several data types but no functions. Class-like structs containing functions can be used in C# with different techniques.

In C, the variables in a struct are laid out sequentially in the memory; however in C#, memory locations of different variables in a struct may be sequential or explicitly defined. For simplicity, we have used structs with sequential memory layouts. The compiler must be informed, that variables in the struct or class we create in C# are laid out in the memory sequentially, by adding a declarations before the struct. The compiler also needs to know how much memory the variables occupy and this must be declared to compiler before the variable declaration. A sample usage is shown below [15] [16].

```
typedef struct
{
      char data[15];
      int prob[15];
} LPRData;
```

Struct declaration in C

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi, Pack =
1)]
public struct LPRData
{
    // char[15]
    [MarshalAsAttribute(UnmanagedType.ByValTStr, SizeConst = 15)]
    public string data;

    // int[15]
    [MarshalAsAttribute(UnmanagedType.ByValArray, SizeConst = 15)]
    public int[] prob;
}
```

Its usage in C#

Structs are generally passed by reference, since they usually contain a large collection of data. Thus, a combination of the information should be applied to achieve this functionality. First, instead of using the C# struct's address, an *IntPtr* is used as an argument. To use an *IntPtr* as a pointer to a structure some functions of the Marshal class can be used. Since pointer usage is not allowed in C#, *IntPtr* does not point to the structure, a different memory block is allocated for it. What we do is,

copy the contents of the structure to the memory block of *IntPtr* before passing it to a function and then, copy the changed values of the memory block back to the structure after the function is returned. *Marshal.StructureToPtr* and *Marshal.PtrToStructer* methods are used for these operations. The copy operation is done with respect to the declarations we have made regarding the memory layout and the variables' memory occupation. Since *IntPtr* has a single block of memory allocated to itself, the struct's memory usage should be known to avoid mixing the variables' data.

3. IMPLEMENTATION

3.1. Used Equipment

Nordic Semiconductor's nRF8001 Development Kit (DK) is the chosen product for this project [17]. It contains several different modules and software to help the functionality of the nRF8001 chip. The chip itself is specifically designed for low-power peripheral (slave) role. Its features are summarized in Figure 3.1 [18].

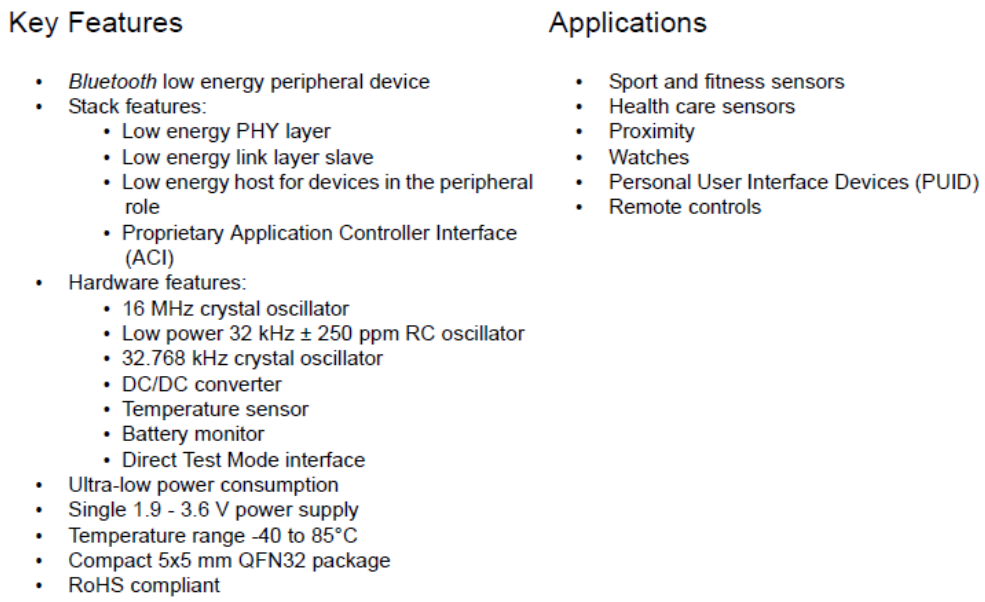

Figure 3.1 : nRF8001 summary of features

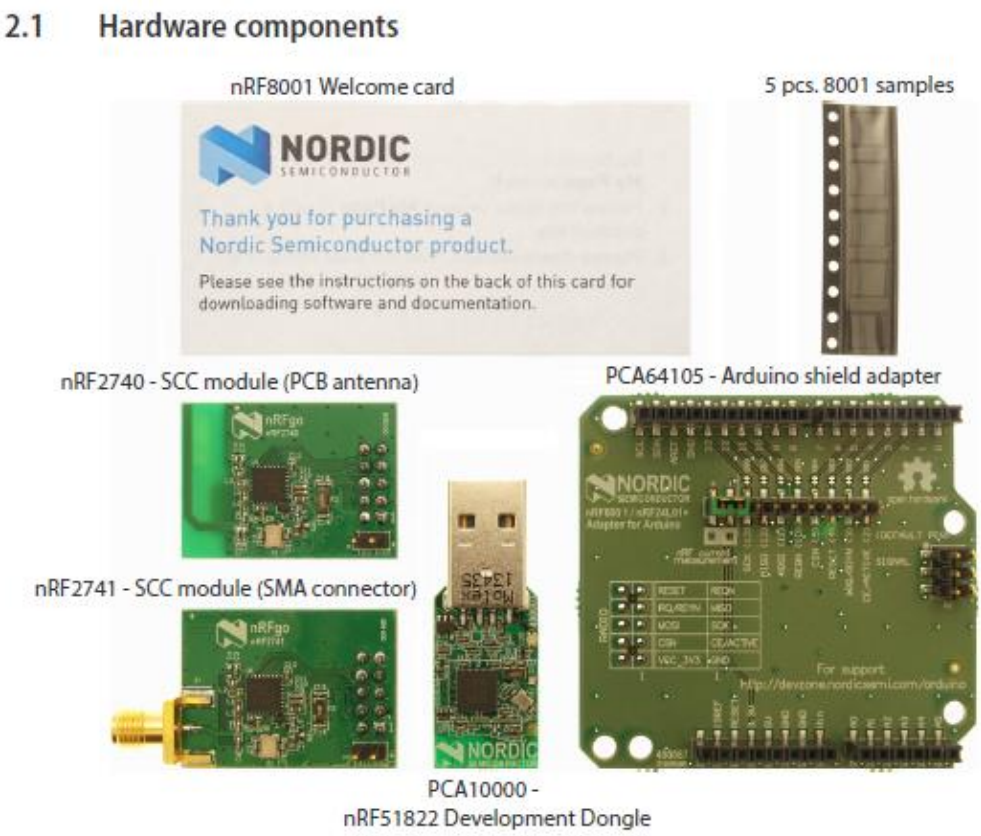

Figure 3.2 : nRF8001 DK Hardware components

The PCA10000 dongle (Master Emulator) can be used in the master role connected to a computer. It contains an nRF51822 chip, different from the other components since the nRF8001 chip only allows operation as a slave, which means the chip can only send advertisement packets and cannot connect to any advertising device. The Master Emulator can be used via the Master Control Panel program, with a Graphical User Interface (GUI) but limited functionality or via Master Emulator Application Programming Interface (API) codes of Nordic, written in C#.

3.1.1. Application Controller Interface (ACI)

nRF8001 chip requires an external microcontroller, referred as application controller by Nordic, to function. The interface between the application controller and nRF8001 is called Application Controller Interface (ACI) and its operation is briefly shown in Figure 3.3. The SDK for Arduino includes all of the supported ACI commands for nRF8001.

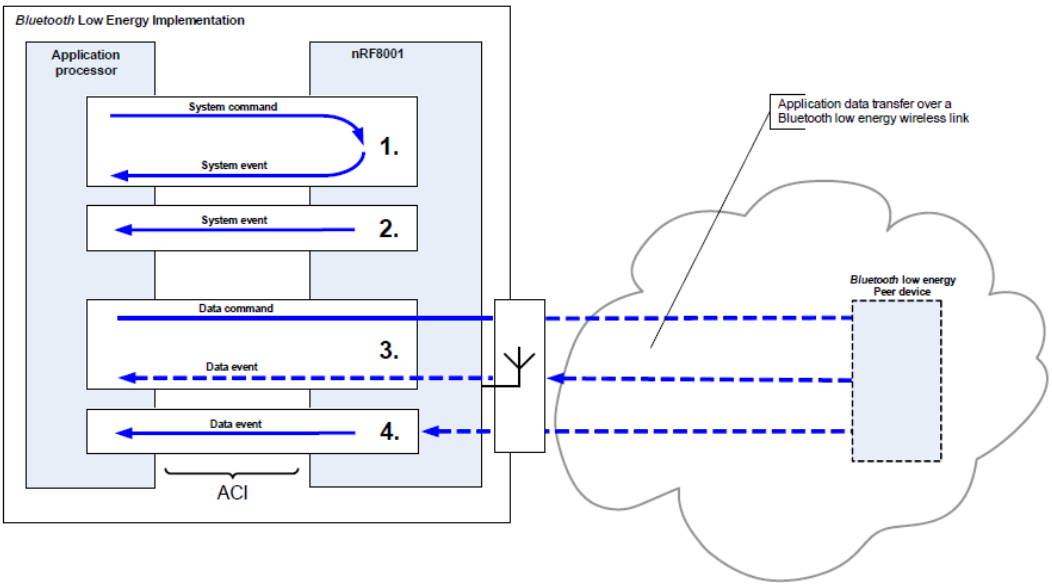

Figure 3.3 : ACI operating principle

A message sent through the ACI is called a packet. Every packet contains a two byte header, followed by a packet payload. The length of the packet payload may change for different ACI packets. The first byte of the header specifies the total packet length in bytes, excluding the length byte itself and the second byte specifies the unique opcode of the command/event. The messages sent to nRF8001 is called commands whereas the messages sent to the application controller is called events.

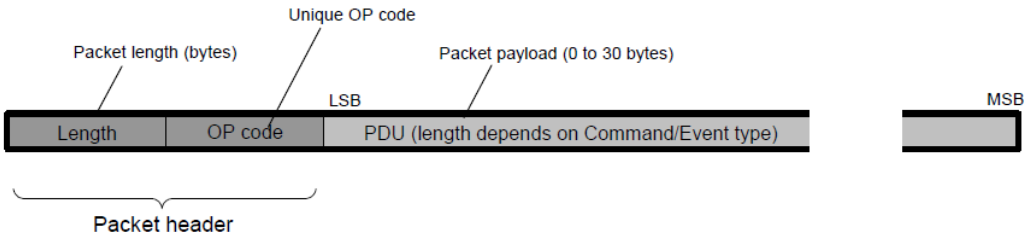

Figure 3.4 : ACI packet structure

3.1.2. Service Pipes

Services of the nRF8001 device are a collection of GATT attributes (characteristics) which are used together to achieve a functionality. Applications may contain many services and may also contain multiple instances of a service. Applications interface

with other devices through these characteristics. Nordic has defined the service pipes concept to use characteristics.

As it is explained before, a characteristic has a type, a handle and a set of permissions. For example, an application may have several characteristics of blood rate monitor type and through the usage of handle values, these characteristics are addressed. A characteristic may also have different permissions for different devices, for example a doctor may only read a heart rate monitor characteristic through his device and a patient's device may only write to the same heart rate monitor characteristic.

Service pipes are "channels" to these characteristics and they contain the characteristic's properties. A characteristic may have multiple service pipes. In the example above, the heart rate monitor characteristic has 2 pipes. One of the pipes is between the characteristic and the doctor's device and this pipe has read permission and the direction of data transfer is from the server to the doctor's device. The other pipe is between the characteristic and the patient's device and this pipe has write permission and the direction of data transfer is from the patient's device to the server.

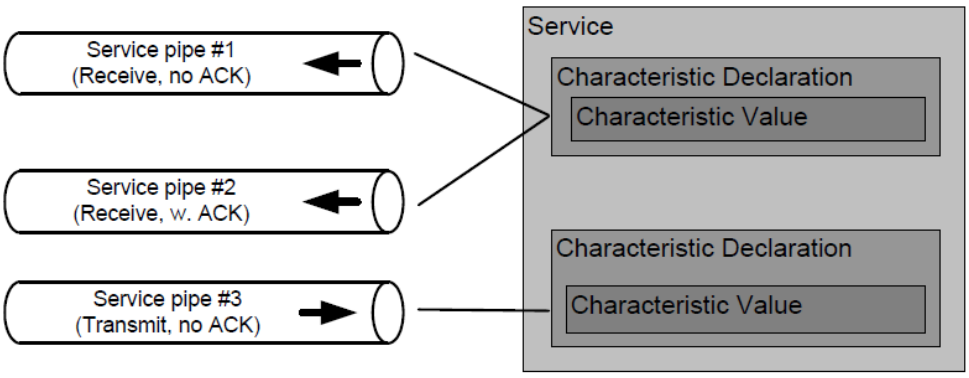

Figure 3.5 : Service pipes

Service pipes may have several properties (set of permissions). Some of these properties are given below:

Direction of data transfer: Data can either be received or transmitted within a service pipe. If bidirectional transfer is desired, two service pipes must be used. An example can be seen in Figure 3.7.

Server location: The value of the characteristic may be stored in either of the devices.

Acknowledgement: A device may request an acknowledgement after it transmits data or may send an acknowledgement after it receives data.

3.2. Software Environment

Since the devices operating in master and slave roles are different, programming environments for these devices also differ.

3.2.1. Nordic nRFGo Studio

This program allows the user to modify both nRF8001 and nRF51822. User can only modify the bootloader, the firmware or the program of the master emulator chip nRF51822. However, even the core Bluetooth functionality of the nRF8001 such as GATT services and GAP settings can be modified via nRFGo Studio. Main BLE functionality of an application is supplied by the GATT services.

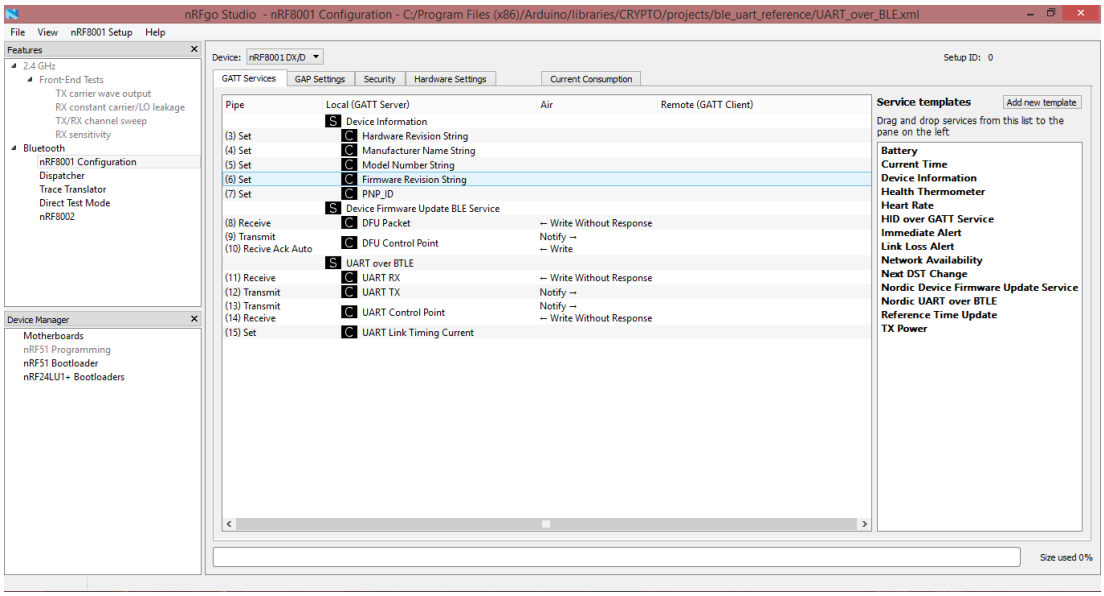

Figure 3.6 : nRFGo Studio GATT Services window

While GAP settings are mostly about connectivity, some functionality can be achieved by the adjustment of GAP settings. The contents of the three types of advertising messages can be seen within the GAP settings window in Figure 3.8. For example, a device may broadcast the temperature without connecting to a device.

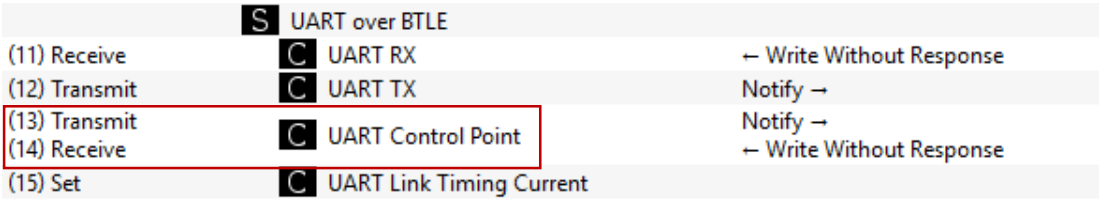

Figure 3.7 : Multiple service pipes of a characteristic

Also, a connecting device may or may not be informed of the services of the nRF8001. For example, nRF8001 may only inform the devices of its heart rate service if the devices pair (bond) for security measures.

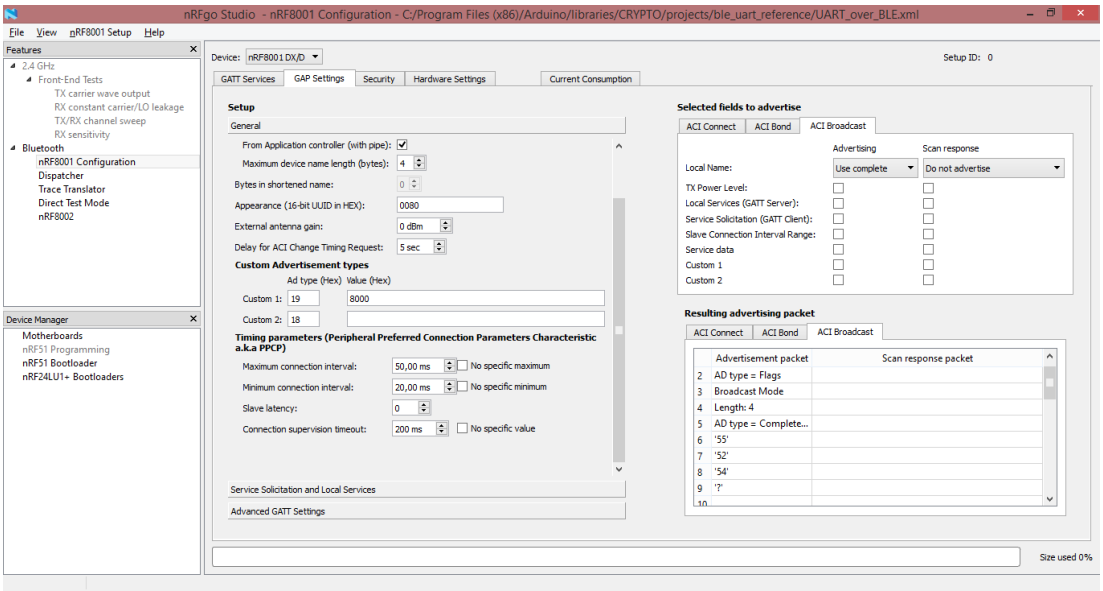

Figure 3.8 : GAP settings window

After the settings are adjusted, a header file and an optional C file may be generated by nRFGo Studio. The header file can be included in the application controller's program to modify the nRF8001's settings. These settings cannot be changed during run-time.

3.2.2. Nordic Master Control Panel (MCP)

This program is used to run the Master Emulator and monitor it during its operation. A firmware can be flashed to the Master Emulator with MCP also. Though the user may add GATT services, services are implemented on the nRF8001 in our project. After connecting to a device, characteristics implemented on the device can be seen

within MCP. User can also access the permitted service pipes, may write into them or read their contents.

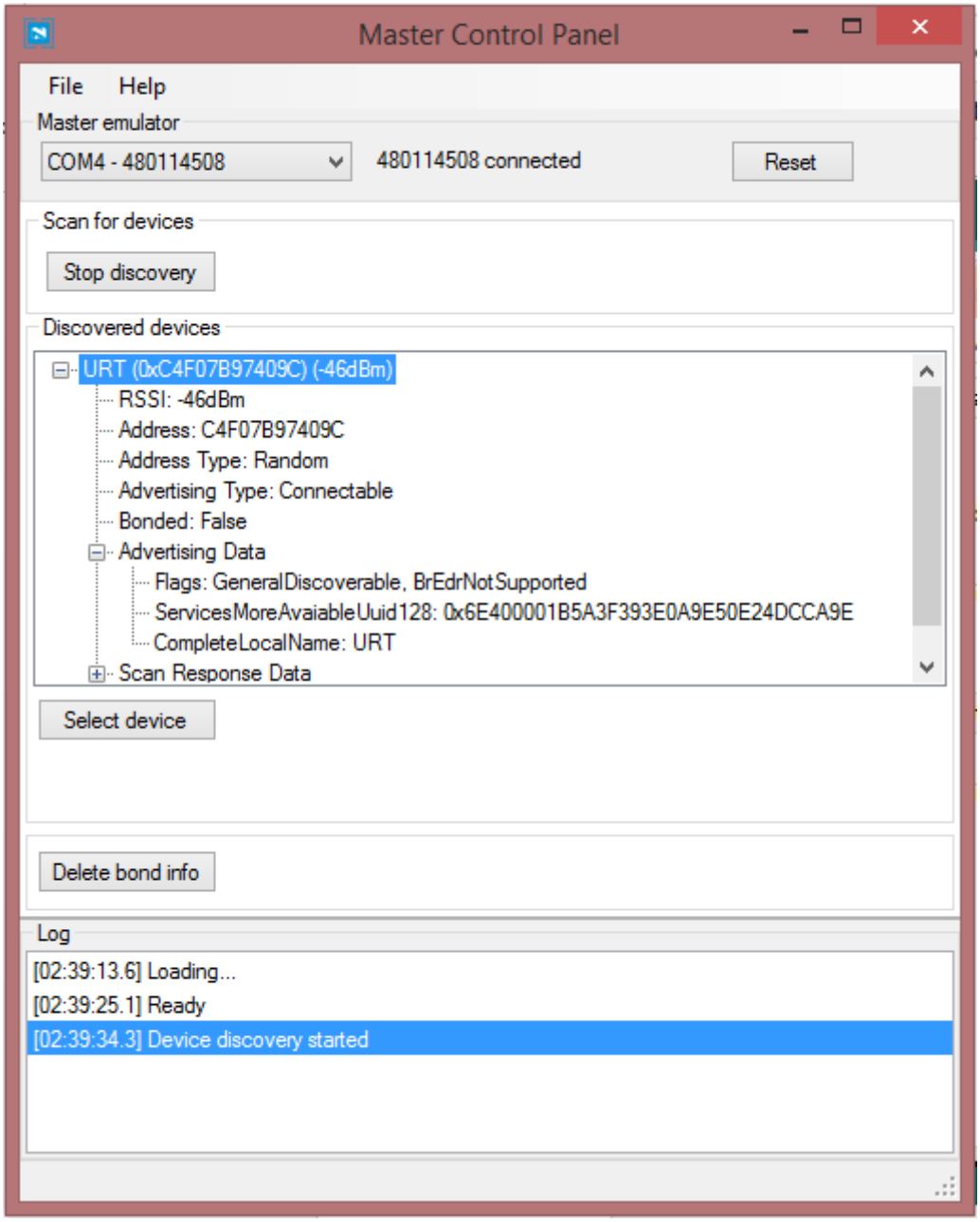

Figure 3.9 : MCP discovery window

Although MCP has an easy to use Graphical User Interface (GUI), it provides limited functionality. The user can only send data by manually entering and the received data cannot be processed. Due to these reasons, it is mainly used for connectivity confirmation in the early stages of the project.

3.2.3. Arduino Integrated Development Environment (IDE)

An Arduino Uno board is used as the application controller for the nRF8001 in our project. Nordic's SDK for Arduino contains many templates for different purposes and instead of writing a program from the start, we modified and used these templates in our project. The programs are compiled and uploaded to the board through the Arduino IDE.

The operation of nRF8001 can be monitored through the Arduino Serial Monitor (SM). By enabling the debug messages, the ACI command and event exchange between the application controller and the nRF8001 can also be seen. The contents of the ACI packets are displayed byte by byte in the SM. The debug messages are enabled by passing a *true* argument to the ACI initialization function.

```
lib_aci_init(&aci_state, true);
```

Examples of the debug messages in SM and monitoring the interaction will be given in further sections.

3.2.4. Microsoft Visual Studio (VS)

Visual Studio is used in our project to run the Nordic's Master Emulator API, written in C#. We have explained previously, how we integrated our C codes for ECDH scheme and AES encryption into the API. The implementation of our ECDH scheme and AES encryption are done in VS. Similar to the SDK for Arduino, template projects are given for the Master Emulator and we have added desired functionalities to these templates, instead of writing them from the start. Unlike the code interoperability part, the implementation can be done in any C# IDE, since exclusive features of VS are not used in this part of our project.

Detailed explanation of the master role operation will be given in further sections.

3.3. Testing the Functionality of nRF8001

Before starting to implement our security protocol, the correct operation of nRF8001 should be ensured. First, some changes need to be made on all of the examples in the Nordic's SDK for Arduino. If the Arduino shield in the Development Kit is used to

connect nRF8001 with Arduino, pin numbers need to be adjusted. The correct pin numbers are given below.

```
aci_state.aci_pins.reqn_pin   = SS; //SS for Nordic board, 9 for
REDBEARLAB_SHIELD_V1_1
  aci_state.aci_pins.rdyn_pin   = 3; //3 for Nordic board, 8 for
REDBEARLAB_SHIELD_V1_1
```

Although, the following commands are not in all of the examples, they also need to be changed.

Find: `static hal_aci_data_t setup_msgs[NB_SETUP_MESSAGES] PROGMEM = SETUP_MESSAGES_CONTENT;`

Replace: `static const hal_aci_data_t setup_msgs[NB_SETUP_MESSAGES] PROGMEM = SETUP_MESSAGES_CONTENT;`

Find: `aci_state.aci_setup_info.setup_msgs = setup_msgs;`

Replace: `aci_state.aci_setup_info.setup_msgs = (hal_aci_data_t*)setup_msgs;`

After these changes, the operation between nRF8001 and application controller is checked. To control data transfer between nRF8001 and the application controller, *ble_aci_transport_layer_verification* example is run. After nRF8001 setup, “Echo OK” message should be received from the Arduino SM periodically.

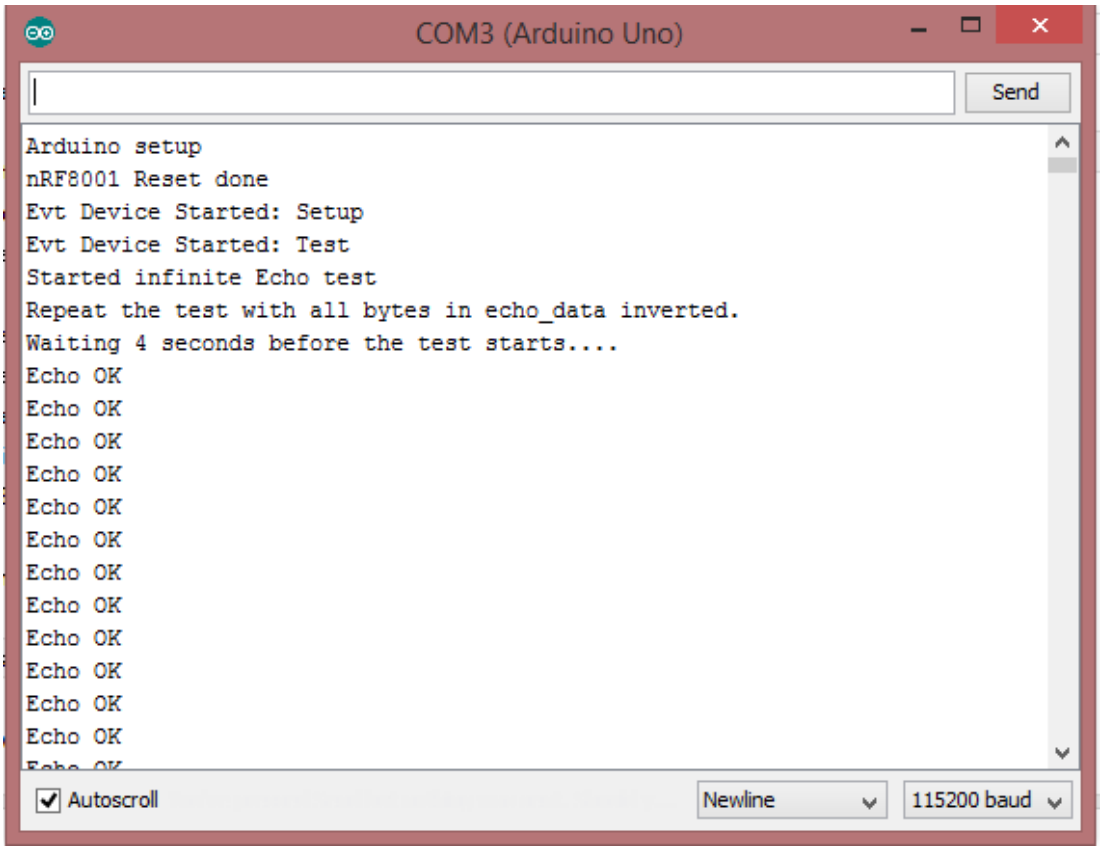

Figure 3.10 : Transport layer verification

3.4. Communication Between Arduino and Master Control Panel

After the operation of nRF8001 is confirmed, simple communication between two BLE devices are tested. Since the data will not be processed on the master side, MCP is used instead of Master Emulator API.

The master emulator dongle is plugged into the computer and MCP is run. Then, with the changes mentioned above, *ble_uart_project_template* example is run. User inputs entered into the textboxes in MCP or the Arduino SM are successfully sent to the other side. However, this template converts the received and transmitted data of the SM into an ASCII text. We want to transfer data without a conversion, so that part of the template should be adjusted. Since we have access to the received data before it is converted, that part will be modified when implementing our security protocol.

With the adjustments, it is made possible to send 8-bit data with hexadecimal notation. In Figure 3.10, the MCP and SM windows are shown. The blue frames

show the fields related with data transfer from MCP to SM and the red frames show the fields related with data transfer from SM to MCP. In a blue frame within the MCP, there is a characteristic called *UART RX*. As it was mentioned before, the current window of the MCP shows the services of the connected device and is called *Service Discovery* window. Values entered into the textbox of the MCP is sent to SM via the *UART RX* characteristic of the nRF8001.

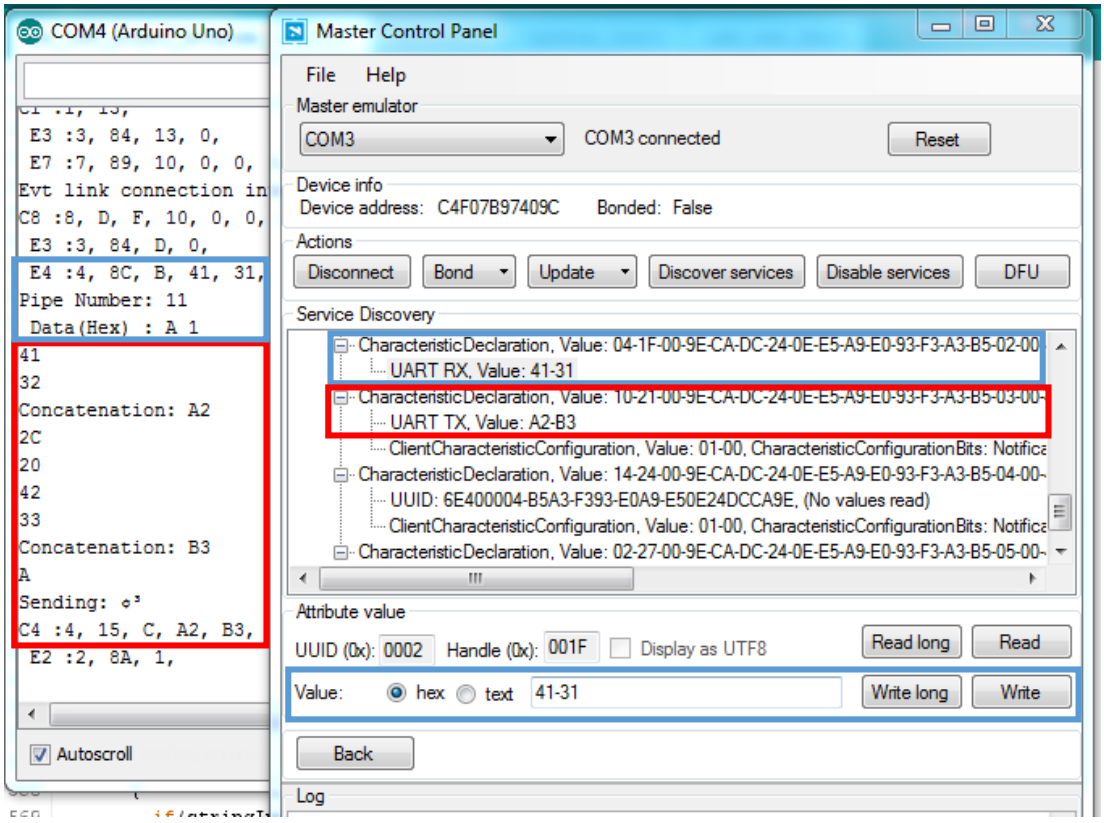

Figure 3.11 : Arduino and MCP communication

Debug messages of the nRF8001 are also enabled and the lines starting with C, show the ACI command packets where the lines starting with E show ACI event packets. It can be seen in the blue frame of the SM, that an ACI *event* packet is sent from the nRF8001 to the application controller after receiving the data. The event packet is 4-byte long (excluding the length byte at the beginning), has the opcode 8C (the code for *DataReceivedEvent*), followed by the byte B (hexadecimal notation of 11), which shows the service pipe number where this data is received. The remaining bytes represent the text sent from the MCP. It can be seen that the received message *41-31* is converted to ASCII and displayed as *A 1*.

As for the red frame of the SM, the input *A2, B3* is entered into the textbox of SM. The program processes the input character by character and the first two values in the red frame is the ASCII codes of *A* and *2*, respectively. These values are concatenated to *A2*. Then, values *2C* and *20* is seen in the frame, these values correspond to *comma (,)* and *space( )*, respectively and they are ignored by our program. The following values *42* and *33* correspond to *B* and *3*, respectively and they are also concatenated to *B3*. Finally, the value *A*, which corresponds to *newline*, is seen and after receiving the newline character, the program proceeds to send the concatenated values. Since the data is first transferred from the application controller to the nRF8001, an ACI *command* packet with 4-byte length (excluding the header byte), with the opcode 15 (the code for *SendData*), followed by the byte C (hexadecimal notation of 12), which shows the service pipe number where this data is going to be transmitted. It can be seen in the red frame of the MCP that the data *A2-B3* is received from the *UART TX* characteristic. Without any adjustments to the template, if *A2, B3* is entered into SM, the program sends *41-32-2C-20-42-33* to the MCP (newline character triggers the transmission).

3.5. Communication Between Arduino and Master Emulator Application Programmable Interface

After communication between two BLE devices are successfully established, we wanted to achieve the same success using the Master Emulator API instead of MCP.

Sample project template *nRFUart* is run in Visual Studio, while our modified *ble_uart_project_template* is still running in Arduino IDE. It was seen that *nRFUart* project also converts the data into ASCII before transmitting. At this phase, we have not made any adjustments. Only the communication functionality is checked.

In Figure 3.12, the windows of the API and the window of Arduino SM is shown. The red frames show data transfer from SM to API where the blue frames show data transfer from API to SM. It can be seen that data transfer occurs without a problem. After we have confirmed the operation on both platforms we wanted to use for the final phase, we started to implement our security protocol on both sides.

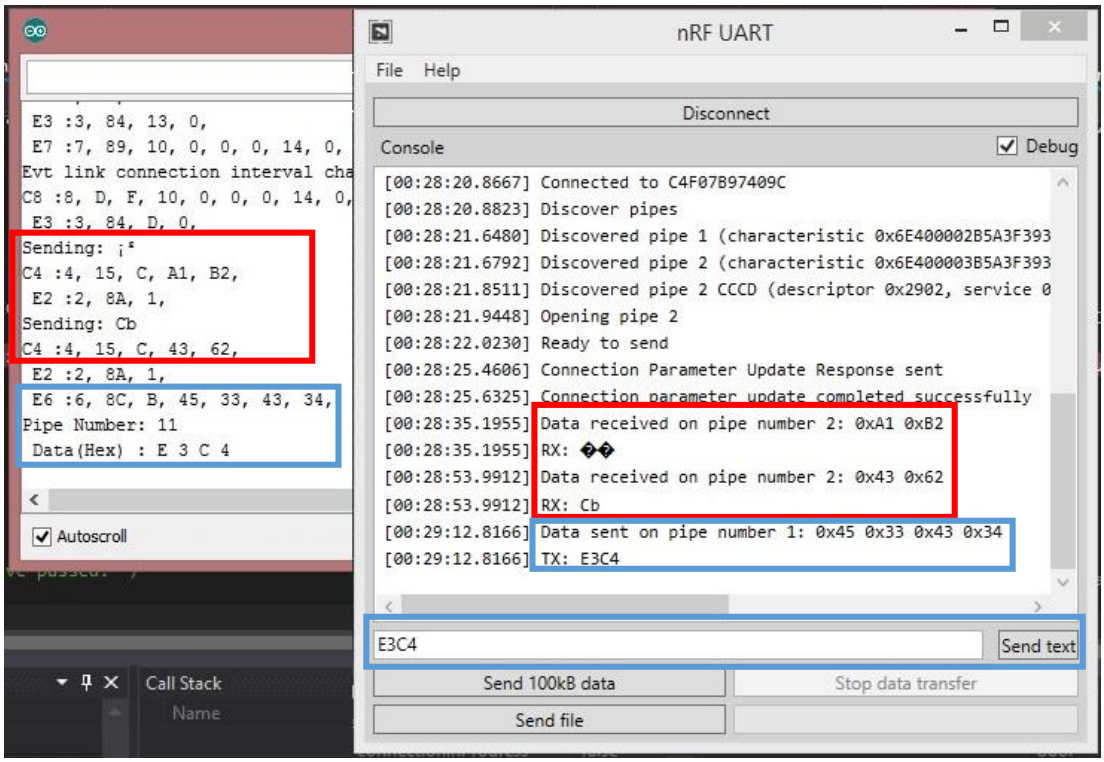

Figure 3.12 : Arduino and Master Emulator API communication

3.6. Secure Communication Between Arduino and Master Emulator Application Programmable Interface

First, we have designed an ECDH scheme for both sides. Since our AES functions run with 128-bit data blocks, we restricted the communication to 128-bit data blocks only. Larger data blocks are truncated while the smaller data blocks are padded. Our ECDH scheme is given in Figure 3.13 and Figure 3.14. The scheme starts automatically after connecting to an unpaired device.

We have defined three custom 16-byte messages. A *correct* message (confirmation) has the value *0x01* in all of its bytes, an *incorrect* message has the value *0x00* in all of its bytes and a *reset* message has the value *0xFF* in all of its bytes. After the devices are paired, the keys are saved and further connections with that device will not start the ECDH scheme. Keys are stored until the devices reset.

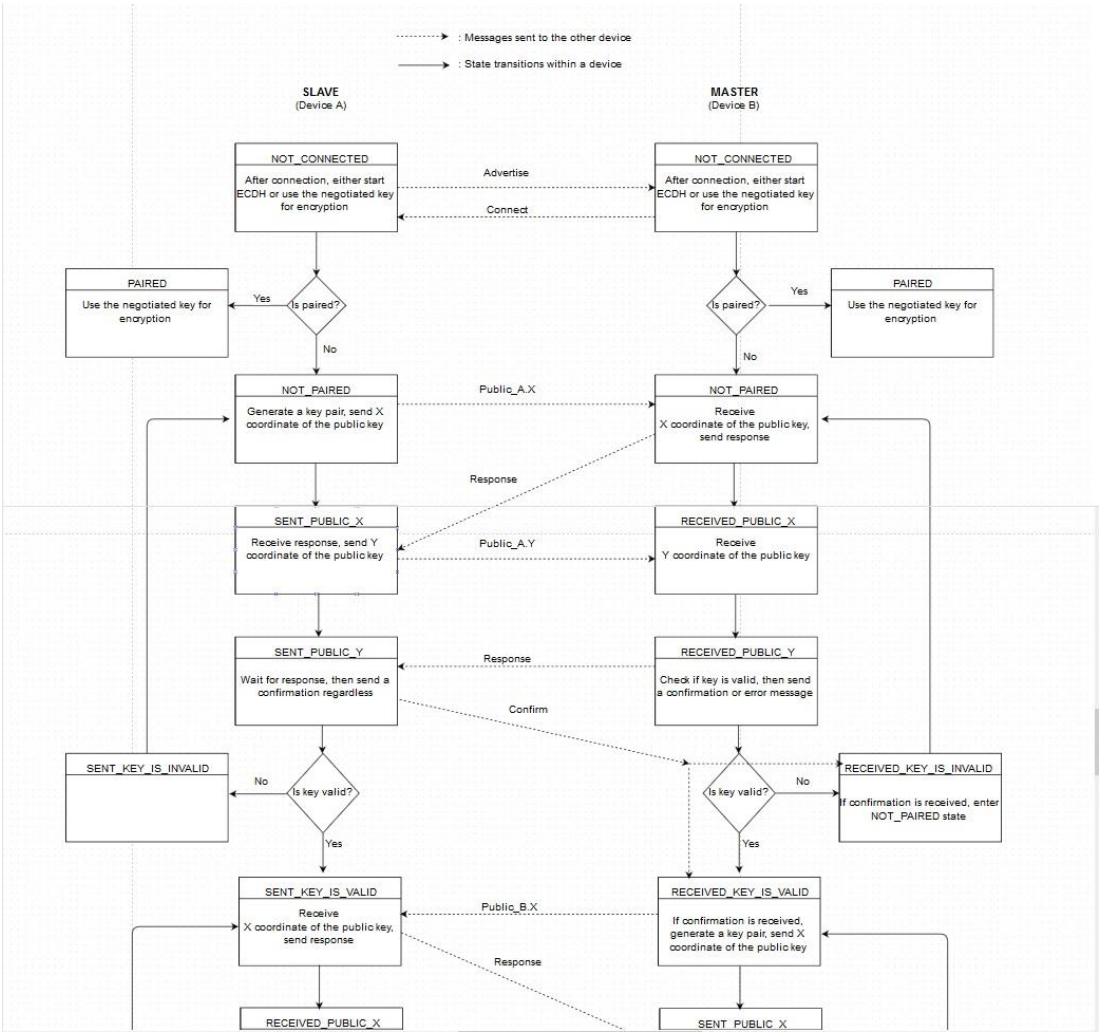

Figure 3.13 : ECDH scheme, part 1

Two user made classes are created in API. One is a static class called *UART_control* and includes the adjustment for sending hexadecimal data instead of an ASCII text. It also includes the 128-bit AES implementation in C [19]. The other class is called *ECDH_control* and contains the functions of ECDH implementation [20] as well as the supporting functions and variables for our ECDH scheme.

In the Arduino, the same ECDH implementation is used but an Assembly based 128-bit AES implementation is chosen to reduce memory usage and improve execution speed [21].

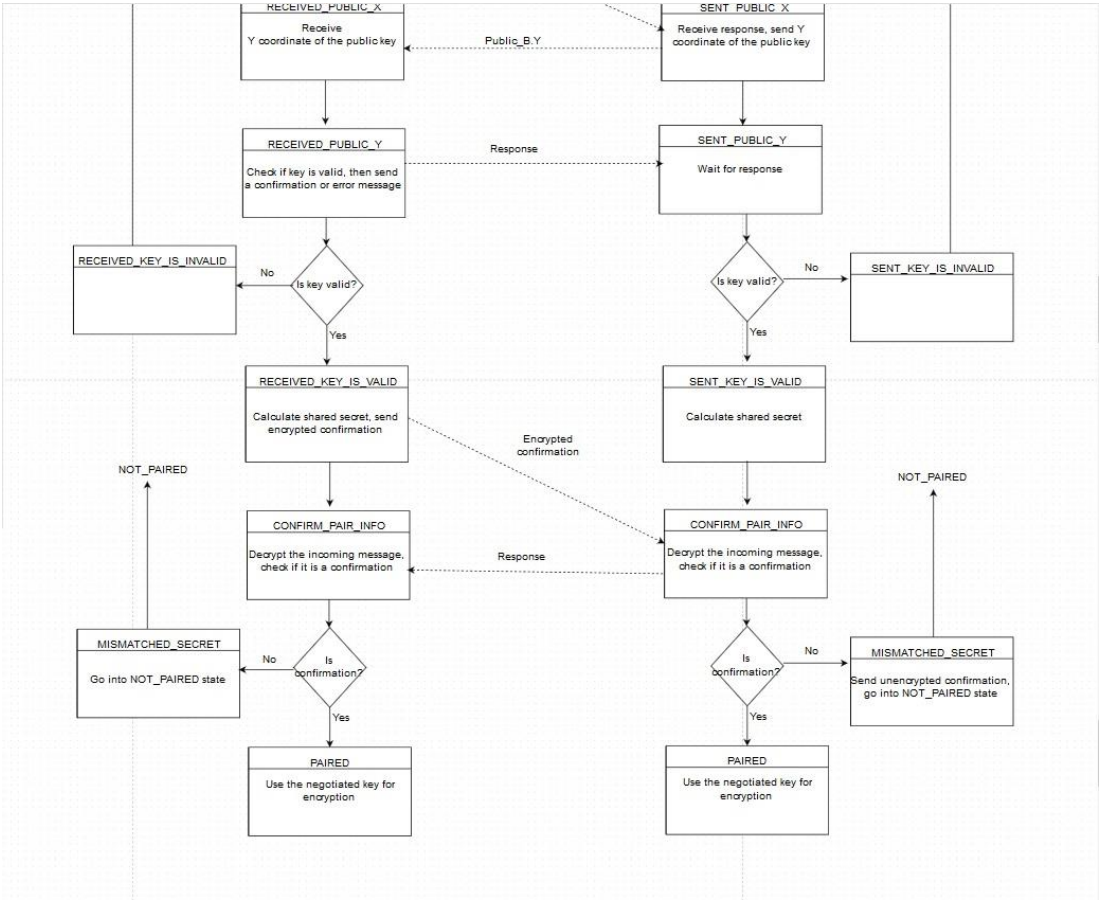

Figure 3.14 : ECDH scheme, part 2

In Figure 3.15 and Figure 3.16, communication after the pairing process is shown. It can be seen that the messages are always 16-byte long. Shorter messages are padded and longer messages are truncated.

The red frames show data transfer from SM to API where the blue frames show data transfer from API to SM. Each side sends a short and a long message and operation occurs without a problem.

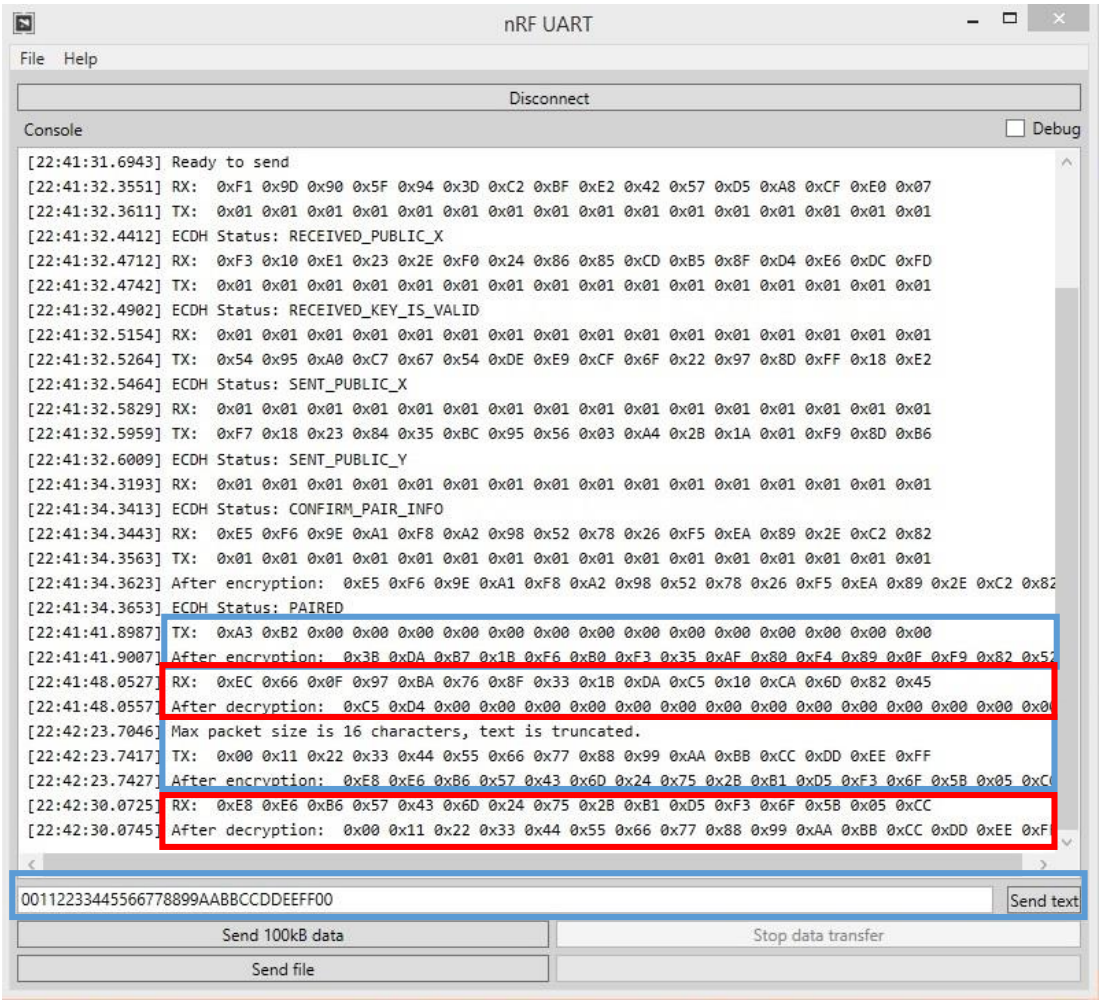

Figure 3.15 : Communication after pairing, API window

In Figure 3.17, it is shown that both sides can encrypt and decrypt messages in a new connection, if they are paired beforehand.

If a *reset* message is being sent from either side, it waits for confirmation to reset after sending it and upon receiving the reset request, other side sends a confirmation and goes into reset process. If the initiating side does not receive a confirmation message, it sends reset request for several times (the number of tries are left to user) and then resets. Reset operation erases the negotiated keys. A sample window after reset operation is shown in Figure 3.18.

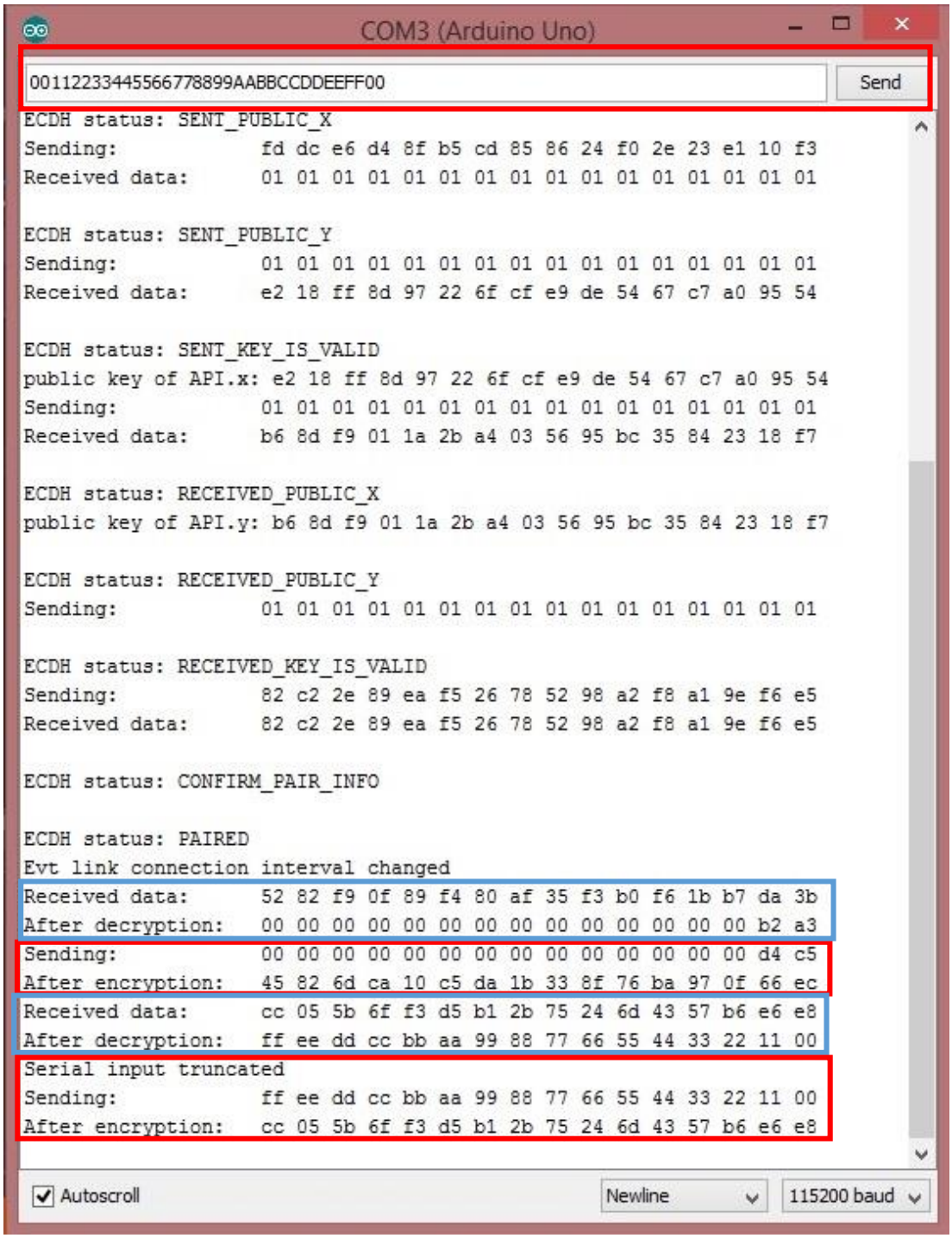

Figure 3.16 : Communication after pairing, SM window

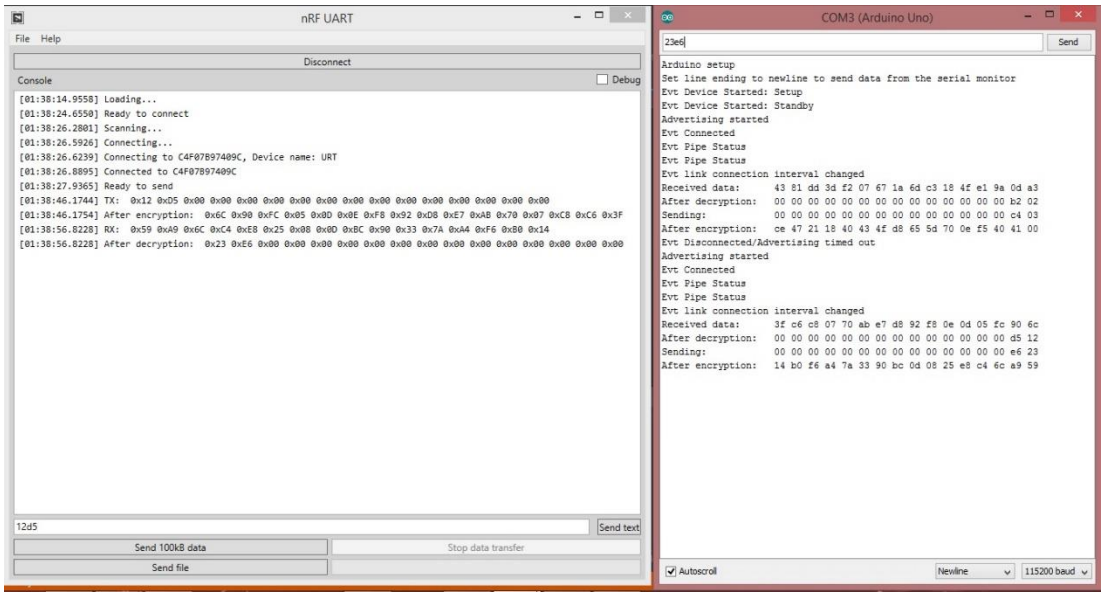

Figure 3.17 : Resuming secure operation after connection

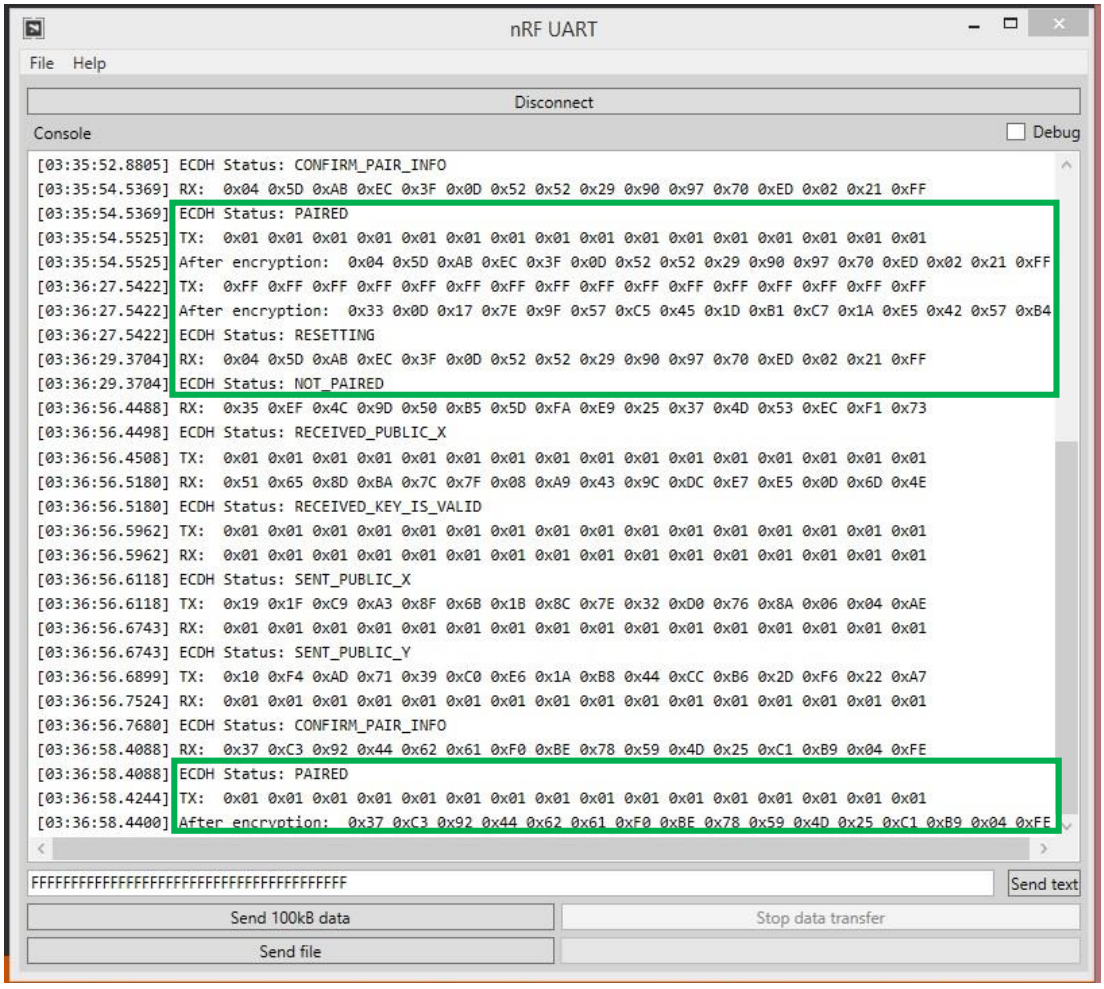

Figure 3.18 : Reset after pairing

4. FUTURE WORK

In this current state, our project has the adequate functionality for basic secure communication. However, there is still room for improvements and the some of them are discussed in this section.

4.1. Multiple Slave Device Connectivity

Our project currently supports a single slave device connection to the Master Emulator. Allowing multiple slave devices to pair is a very important aspect in most of the usage scenarios. This functionality can be implemented without much difficulty and is going to be added as soon as possible.

4.2. Support for Different Application Controllers

The slave side operation is currently supported only in Arduino boards. Since these boards are mainly for prototyping, additional support for microcontrollers suitable for real-world usage is also a needed aspect.

There are also plans to use a Field Programmable Gate Array (FPGA) in BaaS project's localization part as the computing unit of the active device. Since one of our target usage areas is to build a secure communication between this active device and a central processing unit, running our security protocol with an FPGA as the application controller is another future goal.

As an intermediate step towards FPGA usage, an FPGA prototyping board with similar functionality of Arduino can be used. An example for such a board is designed by Gadget Factory and called Papilio [22]. The similarity to Arduino may make the porting of the Nordic's BLE SDK easier.

4.3. Custom Message Design

In our ECDH scheme, we have defined three custom messages and the confirmation message is used extensively throughout the scheme. It may cause a security vulnerability and some information about the encryption key may be leaked by

tracking these confirmation messages. The custom messages and confirmation process could be designed to minimize the security vulnerability.

4.4. Session Encryption

Currently our protocol uses the same 128-bit key after it is negotiated. Using the same key over an extended period of time, especially with the usage of static messages such as confirmation, may also cause a security vulnerability. Thus, a different session encryption key may be generated by using the negotiated key as a base, similar to BLE's Long Term Key usage.

5. CONCLUSION

As the number of simple electronic devices and sensors increase in our lives, it is important to maintain the security of our personal information. BLE communication is used extensively with these devices as it consumes less power compared to similar protocols; however, it contains a security vulnerability in its pairing process. We have implemented a security protocol to negate that security vulnerability of the BLE protocol and with our updated security protocol, low power consumption of BLE communication can be used without hesitation to transfer confidential information.

REFERENCES

[1] Siekkinen, M. ; Hiienkari, M. ; Nurminen, J.K. ; Nieminen, J. (2012). *How low energy is bluetooth low energy? Comparative measurements with ZigBee/802.15.4*, Wireless Communications and Networking Conference Workshops (WCNCW), 2012 IEEE, p.236

[2] Dementyev, A. ; Hodges, S. ; Taylor, S. ; Smith, J. (2013). *Power consumption analysis of Bluetooth Low Energy, ZigBee and ANT sensor nodes in a cyclic sleep scenario*, Wireless Symposium (IWS), 2013 IEEE International

[3] Bluetooth SIG (2010). Bluetooth Specification Version 4.0

[4] Ryan, A. (2013). *Bluetooth: With Low Energy comes Low Security*, USENIX Workshop on Offensive Technologies 2013

[5] Bluetooth SIG (2014). Bluetooth Specification Version 4.2

[6] Building as a Service (n.d.) Retrieved May 8, 2015, from http://baas-itea2.eu/cms/

[7] Barker, E.; Chen, L.; Roginsky, A.; Smid, M. (2013). *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*, NIST Special Publication 800-56A, Revision 2

[8] National Institute of Standards and Technology. (2001). *Advanced Encryption Standard*, Federal Information Processing Standards Publication 197

[9] AES Encryption isn't Cracked. (2011). Retrieved May 8, 2015, from https://blog.agilebits.com/2011/08/18/aes-encryption-isnt-cracked/

[10] Interoperability (C# Programming Guide). (n.d.). Retrieved May 8, 2015, from https://msdn.microsoft.com/en-us/library/ms173184.aspx

[11] Use C codes in a C# project -- unmanaged C solution. (2013). Retrieved May 8, 2015, from https://drthitirat.wordpress.com/2013/05/30/combine-gui-of-c-with-c-codes/

[12] /MD, /MT, /LD (Use Run-Time Library). (n.d.). Retrieved May 8, 2015, from https://msdn.microsoft.com/en-us/library/2kzt1wy3.aspx

[13] Blittable and Non-Blittable Types. (n.d.). Retrieved May 9, 2015, from https://msdn.microsoft.com/en-us/library/75dwhxf7(v=vs.110).aspx

[14] Marshal Class. (n.d.). Retrieved May 9, 2015, from https://msdn.microsoft.com/en-us/library/system.runtime.interopservices.marshal(v=vs.100).aspx

[15] Marshal C struct array into C#. (n.d.). Retrieved May 9, 2015, from http://stackoverflow.com/questions/188299/marshal-c-struct-array-into-c-sharp

[16] Elsheimy, M. (n.d.). *Marshaling with C# – Chapter 1: Introducing Marshaling.* Retrieved May 9, 2015, from http://www.codeproject.com/Articles/66245/Marshaling-with-Csharp-Chapter-1-Introducing-Marsh.aspx

[17] Nordic Semiconductor (2014). nRF8001 Development Kit User Guide v2.0

[18] Nordic Semiconductor (2013). *nRF8001 Product Specification 1.2*

[19] Kokke/tiny-AES128-C. (n.d.). Retrieved May 10, 2015, from https://github.com/kokke/tiny-AES128-C

[20] Ryan, M. (n.d.). ISECPartners/nano-ecc. Retrieved May 10, 2015, from https://github.com/iSECPartners/nano-ecc

[21] DavyLandman/AESLib. (n.d.). Retrieved May 10, 2015, from https://github.com/DavyLandman/AESLib

[22] Papilio FPGA Platform. (n.d.). Retrieved May 10, 2015, from http://papilio.cc/

RESUME

Name and Surname: Bahadır GÜN

Birth Place and Date: Eskişehir, 1993

High School: Eskişehir Anadolu Lisesi, 2007-2011

BSc: Istanbul Technical University, Electronics and Communication Engineering, 2011-2015