**Internet of Things Systems Security**

**System Specification**

**UFCF8P-15-M**

**Luke Murray (19041865)**

**and**

**Jacob Williams (15008632)**

Table of Contents

[Hardware and Functionality 3](#__RefHeading___Toc288_3912960734)

[Design 4](#__RefHeading___Toc166_3912960734)

[Layman's Run Time 4](#__RefHeading___Toc168_3912960734)

[Saltshaking 5](#__RefHeading___Toc170_3912960734)

[Pin Code Entry 6](#__RefHeading___Toc172_3912960734)

[Rotary Dialer 6](#__RefHeading___Toc174_3912960734)

[Command Definition 7](#__RefHeading___Toc176_3912960734)

[DPK Generation 8](#__RefHeading___Toc178_3912960734)

[Encryption / Decryption 9](#__RefHeading___Toc180_3912960734)

[Circuit 10](#__RefHeading___Toc187_1889991527)

[References 11](#__RefHeading___Toc182_3912960734)

## Hardware and Functionality

The specification for this coursework proposes the possibility of using different devices from the previously used Microbit, such as the Raspberry Pi or an Arduino. However, since thus far the Microbit has been utilized primarily it would make sense to continue using the Microbit. This also has the added bonus of experience working with an unconventional IoT device that has little documentation, giving a relatively fresh experience of IoT programming.

The second consideration will be which communication method to use with the Microbits. The choices being Radio or Bluetooth Low Energy. If using the Radio functionality it is possible to continue using the already familiar DAL runtime, however if one wishes to use BLE one would have to switch to using ARM Mbed. Both functionalities are provided by Lancaster University.

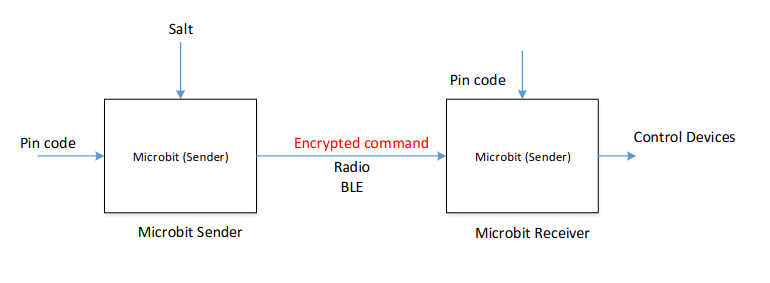
The Radio functionality is extremely simplistic, it works as one might expect. A device acts as the transmitter, another device idles using the DAL listen function to receive the communication and then one programs what happens as a result of what has been received. Lancaster University say that the Radio functionality is designed with “privacy in mind”, given that “there is nothing inherent to the protocol which can be used to identify you or your microbit”. However privacy doesn’t necessarily ensure security, given the open nature of the radio communication an attacker could technically intercept communications and even masquerade as either the sender or receiver if they so desired.

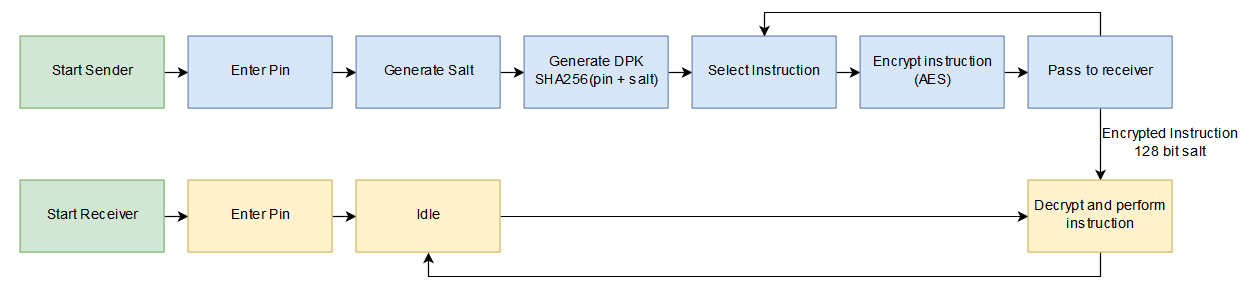
The Bluetooth functionality provides various security functions, including being able to pair with passkeys and including different protected communications. It also has whitelisting and the links between devices are encrypted by default. The draw back of BLE is that one would have to change the runtime and also develop for multiple different devices. As a result this method would be considered more difficult in the context of this project.

The method deigned to be used was Radio, its functionality sounds interesting and finding ways to incorporate security into a messaging functionality that is no way by default secure is an interesting prospect.

# Design

## Layman's Run Time

There is a example runtime laid out in the coursework specification, this being:  
The runtime being that the microbits are started, pins are entered on both microbits. The sender then generates a salt, which is combines with the pin code and encrypts in SHA256, giving us a 64 byte data protection key to use with the AES encryption later.

Above is a flow representation of the two roles the microbit can operate under, in theory one implementation can be created to perform both roles, where one simply chooses the desired role at the beginning of the runtime. The sender should be programmed to work like a hub, where it distributes commands to the recievers within range of it. The reciever should be designed in such a way that the number of recievers can be dynamic, anything running the reciever side of the program in range of the corresponding sender with the same pin should be able to receive the instructions.

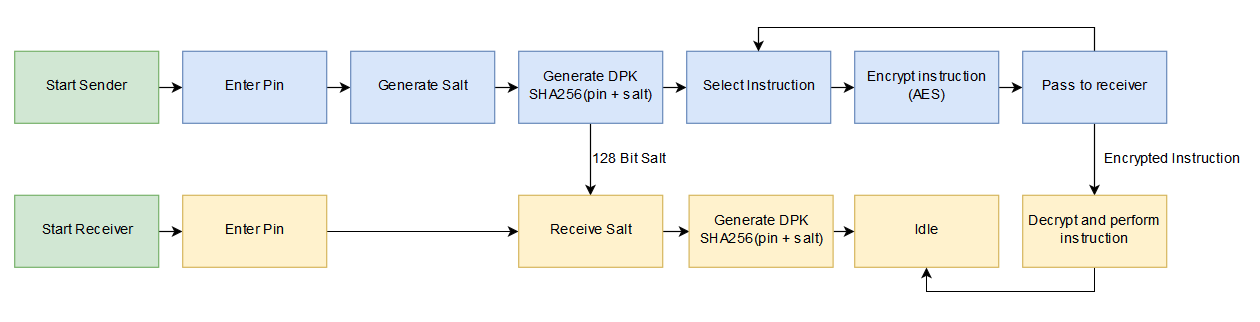
There is however a key flaw with the example runtime that stops it from being the basis of this project. The design implies that the salt is passed every time the sender sends an instruction which seems like a huge security flaw, as possible intruders intercepting the messages would see the repitition and possibly be able to breach the method of passing instructions.

Counteracting this would be a big undertaking, but an idea of how to reduce the potential entry of attack has been formulated, nicknamed “saltshaking”.

### Saltshaking

The concept of handshaking is an old and established protocol for linking two devices, effectively working by the sender sending an identifier, the receiver returning a confirmation of the identifier, and then the sender sending a confirmation of receiving the confirmation. This works to establish the connection and not require the passing of any extra identifiers after handshaking.

The Saltshaking implementation works under a similar basis. The sender distributes the Salt to the receivers before sending any instructions, so that when it comes to actually sending instructions there is no need to pass the salt multiple times.

In theory this implementation will reduce the potential of intercepting the salt and therefore potentially being able to breach the communication channel of the two Microbits, as all messages occurring after the Saltshake will be encrypted using the resultant DPK.

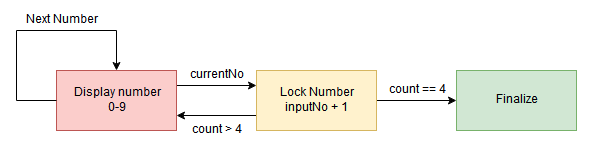
This method doesn’t counteract other possible security risks occurring from the factor of using the Microbit radio communication, such as intruders being able to masquerade as sender. But this method does reduce the potential effectiveness of an interception attack given that in essence the attacker has one chance to do it.

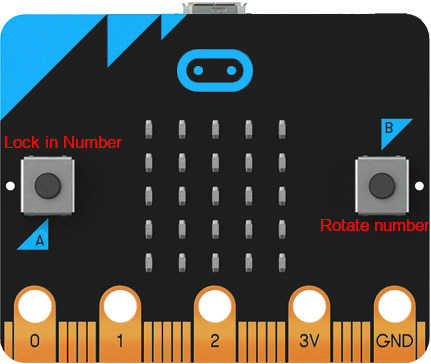
This method also creates another issue, the Saltshake can only in theory occur once, else it defeats its entire purpose. Because of this, if one unit in the cluster of devices loses its DPK (losing power, being manually reset) then the entire cluster will need to be re-initiated.

## Pin Code Entry

The initial problem of pin code entry boiled down to hardware, the external functionality of the Microbit. The Microbit has the ability to be used with other external peripherals thanks to its 40 general purpose input output headers (GPIO headers) and the breadboard supplied with it. It would be entirely possible to purchase an external membrane keypad and use a lookup matrix to get the desired pincode. Whilst this would certainly be effective, it was deemed necessary to explore methods of entering the pin code without resorting to extra purchases, and as such experimenting with the functionality the Microbit inherently had began.

### Rotary Dialer

A now niche relic of the past, the rotary dialer is a way of inputting phone numbers into your handset by rotating the central wheel to the desired number and waiting until it reset before entering the next. This implementation idea was spurred by this concept, what method can you use to enter a pin code when you have two buttons by default? Simply, one “rotates” the numerical value and the other locks it in. The reason this is reminiscent of the rotary dialer is that they both utilize the functionality that to access the higher numbers you have to bypass the lower ones first. It works on a ‘0-9’ basis, once it reaches 9, pressing next number reverts it to 0.



## **Command Definition**

## DPK Generation

Generating the DPK makes use of the pin code whose creation is detailed in previous sections.

### Salt Generation

The salt generation is simplistic yet effective. A previous task involved to utilization of a Random Number Generator, the goal of the task to implement the ten tests to verify the random nature of the values created. This random number generator paired with a lookup table containing characters deemed legal for use in salting would effectively create for us a random salt. Our definition of legal characters is broad, and mostly excludes special characters and punctuation characters, leaving the available character list 62 long consisting of “A-Z, a-z, 0-9”.

The random number generator itself utilizes the stdlib rand() function, which has often been criticized for its quality compared to heavier cryptographically secure random number generators, but we believe it more than effectively performs the task it is needed to do in this implementation. Furthermore, the pairing of the salt with SHA256 hashing increases the effectiveness of masking the pin and creating a unique key as even a change to a single bit can change the format of an entire SHA256 string, meaning that even if the random number generator only generates two different characters one time from the last, it would still be enough to alter the entire layout of the SHA256 string.

The salt should be 128 bits long, 64 bytes, 32 characters.

### Hashing

The chosen hashing algorithm for this was SHA256, a lightweight and effective hashing algorithm. There is much debate as to the security of SHA256, whilst it hasn’t been cracked its nature as a lightweight and easy to compute hashing function leaves it open and vulnerable to brute force attacks, hence why it is often advised when using SHA256 to also utilize a hashing function, as adding a component of randomness to the hashing can create a very difficult to force hash. SHA256 provides good defense against collision attacks, as it is nearly impossible to find two exactly similar SHA256 hashes, a hash can be made entirely different by changing a singular bit, as proven by an bug encountered and detailed in the next section. SHA256 is however considered to be poor at defending against length extension attacks. The output of SHA256 is a string of characters 256 bits / 32 Bytes / 64 characters long, which conveniently is the key size for AES256 encryption, it would also be possible to use substrings of the DPK for lower forms of AES encryption, but given the convenient interplay between the two functions it would make sense to utilize it to its fullest degree.

### SHA256 Variable Encoding Bug

An interesting bug was encountered during the creation of the DPK functionality. On the sender side, the pin was entered, salt generated, and the DPK sucessfully created by hashing the two in SHA256. On the receiver side the pin was entered, the salt was received, the DPK was generated with SHA256. However, the two hashes were different. The difference was not minor either, such a minor difference could be put down to the radio transmission happening too quickly and losing characters during the process, but the hashes were entirely different. Numerous tests were conducted, the SHA256 function was being passed the exact same string of pin and salt, but the two exactly the same inputs were producing two entirely different outputs.

The cause was a fairly strange one, simply the receiver had to utilize a transition variable, a ‘temp’ if you will. This transition variable was the Microbit uBit ‘managedString’, which is used to store the incoming radio transmission for processing, the value of this transition variable was then stored in the array for the salt on the receiving end. For some reason manageString changed the character value functionally but not fundamentally, a change at the bit level that was inconsequential at any other level. When comparing two characters to each other, one c char and the other uBit ManagedString it would return that they are the same character, but then when hashing them the two would return different hash values.

The first solution was to utilize the ManagedString in the creation of the salt on the sender side, which worked, the two salts were now using the exact same encoding and therefore would produce the same hash output. However this was deemed not to be really solving the issue, but rather expanding it to the point where it was no longer affecting this particular functionality, essentially turning a bug into a feature. What concreted the need for another solution was the possible impact this encoding error could have on the future AES implementation. Instead the solution found was to interpret the received character before storing it in the array using a lookup table of all the characters, this solution was derived from the fact mentioned earlier that when comparing the two characters of the different types they would still be deemed that same character. Using this method, the two hashes now match.

### Structure

|  |  |  |
| --- | --- | --- |
| Name | Length | Example |
| Pin | 16 bits / 2 Bytes / 4 Characters | 1111 |
| Salt | 128 bits / 16 Bytes / 32 Characters | 22222222222222222222222222222222 |
| PinSalt | 144 Bits / 18 Bytes / 36 Characters | 111122222222222222222222222222222222 |
| SHA256 Hash | 256 Bits / 32 Bytes / 64 Characters | dd9daf5d2d835295ded2a43b27c7e78a0bea5b208190757cf3b78dcc51d74a39 |

## Encryption / Decryption

## Circuit

# References

Lancaster University (2016) *micro:bit Runtime.* Available from: <https://lancaster-university.github.io/microbit-docs/> [Accessed 17 December, 2019]

**System specification (10%)**

**(For group – all group member can use the same system specification)**

This document describes how the system will be built. It must therefore contain sufficient and appropriate detail. It can also be used to divide work up among the team members and to schedule different pieces of activity. A good specification:

* uses appropriate descriptive methods such as diagrams, sketches, storyboards, data dictionaries, etc.
* is internally consistent – ie information in one part does not contradict what is said in another
* contains sufficient detail for the system your group implemented, such as your solutions, protocols, state diagram, etc.