ELEC 458 – EMBEDDED SYSTEMS

PROJECT 1 - REMOTE KEYLESS SYSTEM

AZİZ CAN AKKAYA 171024005
BERK SARI 141024068

Contents

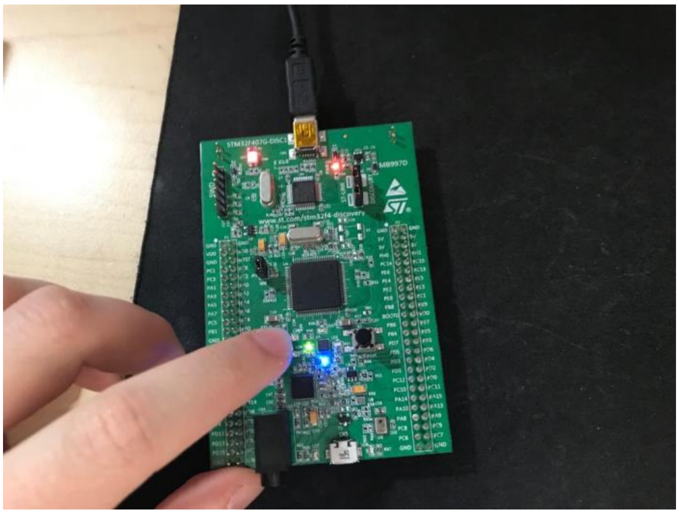
1 Intorduction

1.1 Briefing

Document Version : 1.3_b

Document Publised Date : 10/4/2020

Project Version : v1.1_f



1.2 Objective

This project is developed on a STM32F407VG Discovery Kit. The ARM Cortex-M4 architecture gained functionality with “template.s” file developed with Assembly Language. The aim of the Project is to understand and gain a breif knowledge about ARM Cortex-M4 and it’s functionality.

1.3 Specifications

1.3.1 Requirements

In this system, Manchester Encoding and Rolling Code will be used. With Rolling code, each time a different unique number will be generated. The generated code will be packed with destination adress, source adress and identifier then encrypted with encryption key. After that the packed data frame will be encoded with Manchester Encoding and will be send. This system will work on STM32F407 Discovery 1 and coded with assembly. The board requires 5V input and it's provided from USB Mini Type B port.

- The system will light up all the LEDs at startup and when reset button pressed.
- The transmission is done by using Manchester Endcoding and whole data frame will be encrypted.
- The system will generate Rolling Code from a hardcoded number that we predefined in software.
- The system will light up two of it's LEDs upon button press and after that Rolling Code's last 4 bits will shown on LEDs
- The Rolling Code and other needed data frames packed before the data transmission.

1.3.2 Project Tasks

Aziz Can AKKAYA	-Calculating Rolling Code -Manchester Encoding -Data Frame Transmission
Berk SARI	-Calculating Rolling Code -Showing last 4 bits of Rolling Code on LEDs -Data Frame packing -Encryption of Data Frame -Manchester Encoding

2 Technical Aspects

2.1 Materials

This project needed a last longing battery, so we decided to use STM32F407VG ARM Cortex-M4 microcontroller. It has fast execution and respond with a low possibility of data hazard errors.

As for the software we decide to use STM32Cude IDE 1.3.0. The reason behind is rising popularity of the IDE and it has same components as Keil (the most popular IDE currently) but a different and easier approach to UI.

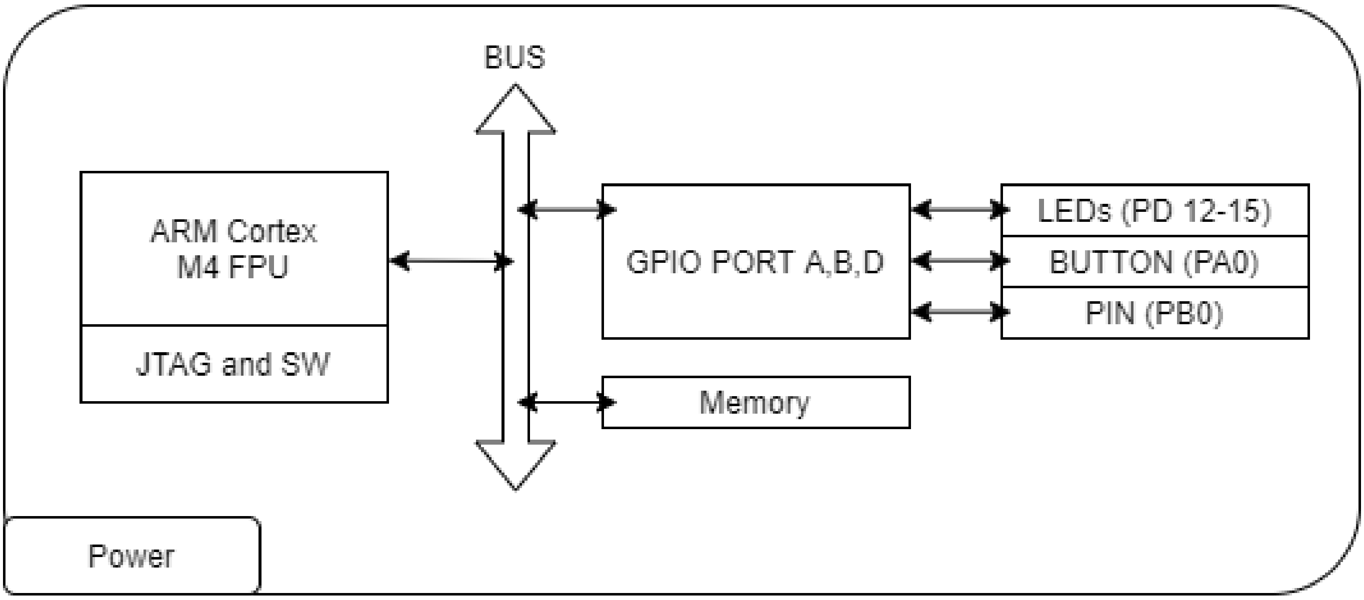
2.2 Method

After reset and the check of the button input (If it is pressed). To set the Transmission Data Frame we will need two components which are Encryption Key and Data frame. After these two data frame EXORed the data we will get is the Transmission Data Frame (TDF).

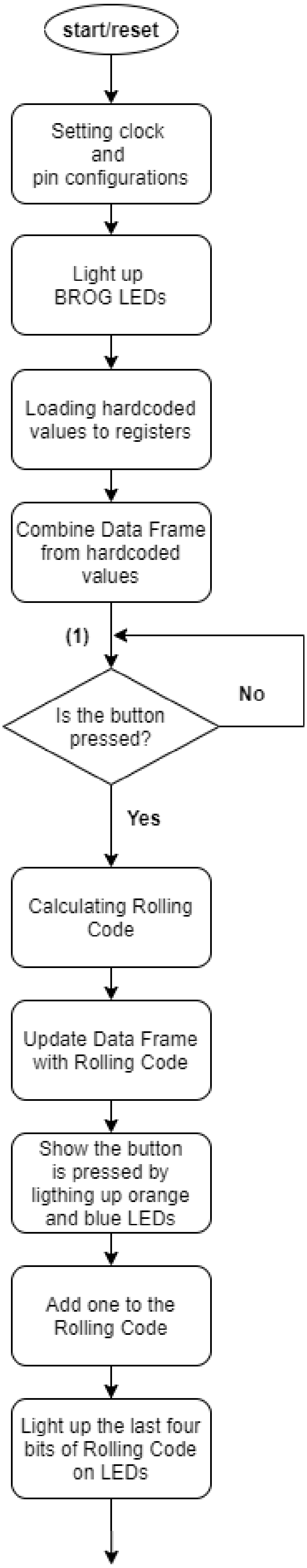
To encrypt the TDF we will use Manchester Encoding, which consists on taking the frame and the double the bits with 01's taking 0's and 10's taking their places. With that down we will have 64 bits frame, which is impossible to transmit with 32 bit registers. To prevent segmentation faults we will put the data frame in 32 step loop. The loops process is to transmit 0x01 for 0's and 0x10 for 1's. That way we will transmit the TDF bits one by one.

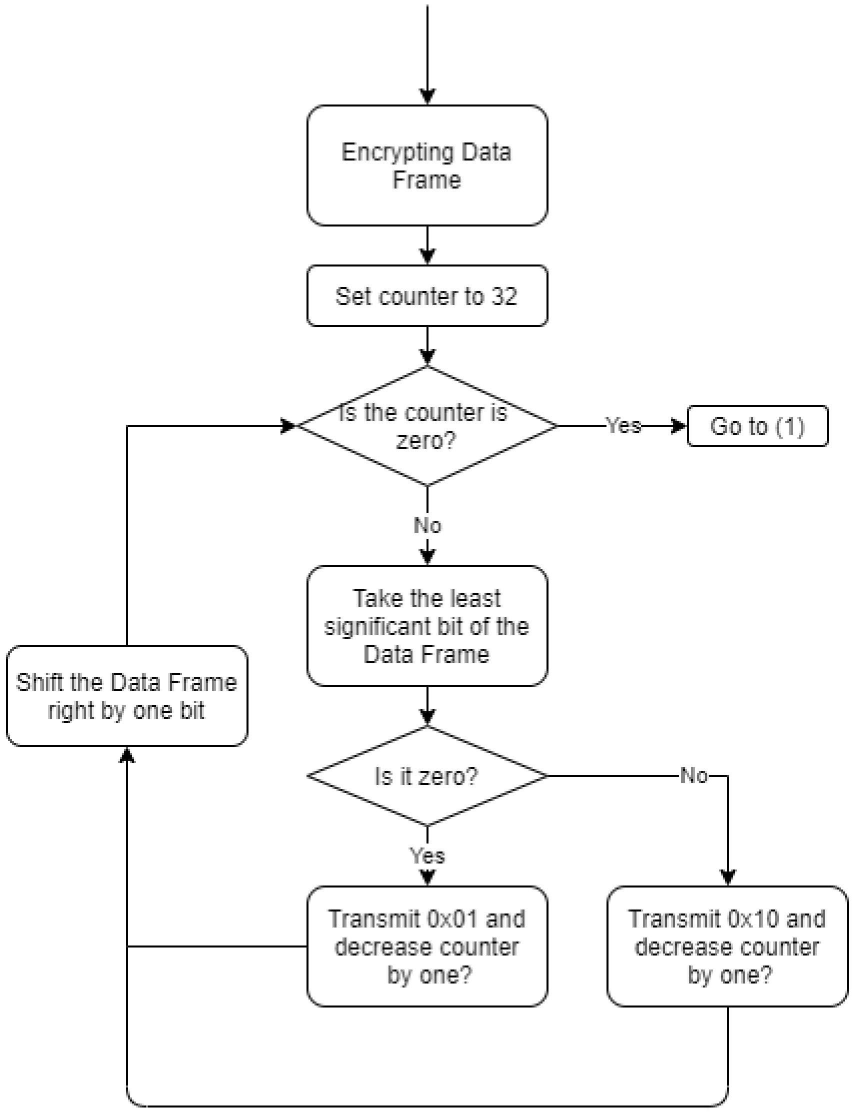
2.3 Diagrams

2.3.1 Hardware Block Diagram



2.3.2 Software Block Diagram

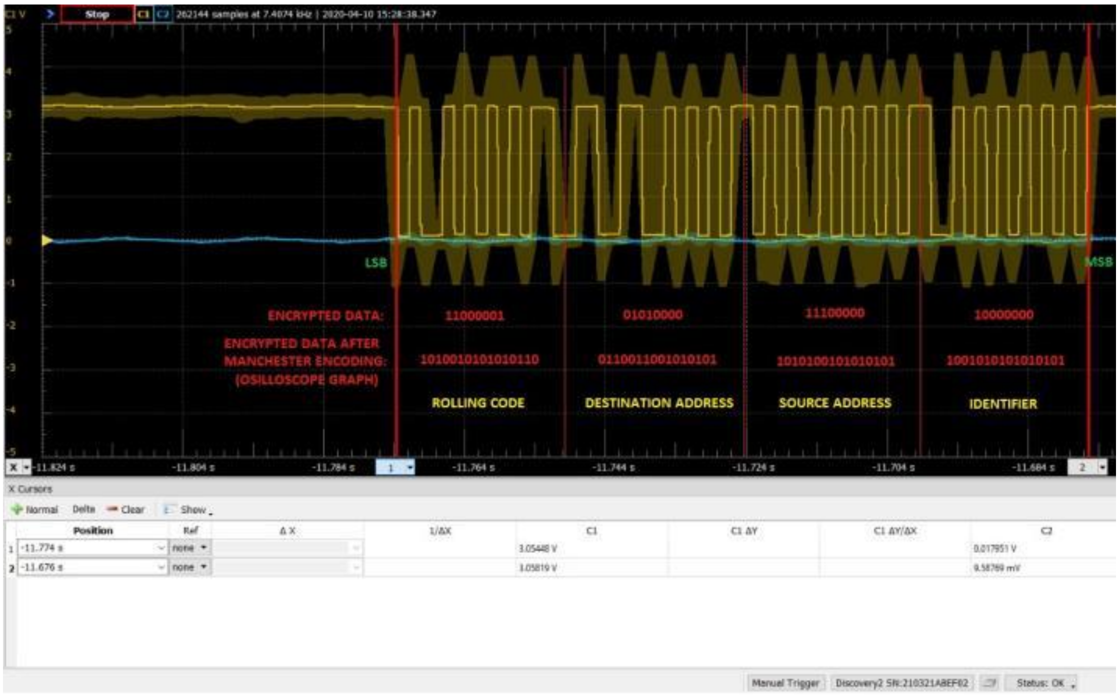




Encrypting Data Frame
Set counter to 32
Is the counter is zero?
Yes
Go to (1)
No
Take the least significant bit of the Data Frame
Shift the Data Frame right by one bit
Is it zero?
No
Yes
Transmit 0x01 and decrease counter by one?
Transmit 0x10 and decrease counter by one?

3 Output

Name	Value	Description
General Registers		General Purpose and ...
r0	10000001110000101010000011 (Binary)	Encrypted Data Frame
r1	17238528	
r2	16777216	
r3	458752	
r4	2560	
r5	0	
r6	201	
r7	32	
r8	1010000000000000 (Binary)	Last four bits of Rolling Code for to light up LEDs
r9	256	
r10	1001011 (Binary)	Next Rolling Code
r11	0	
r12	10000001110000101001001010 (Binary)	Packed Data Frame
sp	0x20020000	
lr	-1	
pc	0x8000112 <ME_Bit_Seperator+2>	
xpsr	553648128	



4 Conclusion

4.1 Design Overview

This Project divided in to two parts; Rolling Code and Manchester Encoding. First step was doing research about the parts and learning how they work. After enough data was collected, main logic and algorithms were created simultaneously.

In this Project, the most challenging part was Manchester Encoding and its transmission. Because Manchester encoding doubles the bits of data frame, we couldn't be able to transmit our data frame directly from a register. The solution we came up with is, doing Manchester Encoding bit by bit and sending that bit immediately after doing Manchester Encoding. With Manchester Encoding, our data frame becomes 64 bits from 32 bits, we have to send 64 bits from a pin and we have to do it under 100ms. So we calculated how long a bit can stay on the pin and we it to suspend the system. When the system performed Manchester Encoding on each bit from data frame and sent it to a pin, a counter triggered before next bits process starts. Counter value starts at 32 and when it reaches to 0, the system knows that Mnachester Encoding performed on all the bits from data frame and transmission is done.

4.2 Discussion

While working on this Project, we learned;

- How to define and use register adresses of peripherals and their ports.
- The use of ARM Assembly Debug and reading data driectly from registers.
- Arithmatic operations and how to manipulate data in ARM Assembly.
- When using delay function which makes use of cycles from MCU, how to calculate time.
- How to read products datasheet more efficiently.

This Project took around 30 hours to complete in total. While working on this project, because of our inexperience on ARM Assembly we used registers very poorly and not taking notes which register contains what kind of data. This caused us some problems. We had to go back and rearrange registers and fix things regularly. It would be much easier if we planned the use of registers for important datas in the start.

5 Appendix

```
// STM32F4 Discovery - Assembly template
// Turns on an LED attached to GPIOD Pin 12
// We need to enable the clock for GPIOD and set up pin 12 as output.

// Start with enabling thumb 32 mode since Cortex-M4 do not work with arm mode
// Unified syntax is used to enable good of the both words...

// Make sure to run arm-none-eabi-objdump.exe -d prj1.elf to check if
// the assembler used proper instructions. (Like ADDS)

.thumb
.syntax unified
//.arch armv7e-m

///////////////////////////////////////////////////////////////////////////////
// Definitions
///////////////////////////////////////////////////////////////////////////////
// Definitions section. Define all the registers and
// constants here for code readability.

// Constants
.equ    LEDDELAY,          2000000      // Counter value for 0.5 sec delay (4 cycles)
.equ    BITDELAY,          6250        // Time between each bit in
                                       // transmission (4 cycles)

.equ    RC_start,          0x00000049  // Initial value of RC in hexa (73)
.equ    mod_num,           0x00000100  // 256 in hexa
.equ    identifier,        0x01000000
.equ    source_adr,        0x00070000  // Grouo ID 7
.equ    dest_adr,          0x00000A00
.equ    Encrypt_Key,       0x000000C9  // ((Sum of our uni IDs) mod256)=201



// Register Addresses
// You can find the base addresses for all peripherals
// from Memory Map section

// RM0090 on page 64. Then the offsets can be found on
// their relevant sections.


// RCC   base address is 0x40023800
//   AHB1ENR register offset is 0x30
.equ     RCC_AHB1ENR,   0x40023830 //RCC AHB1 peripheral clock reg(page 180)

// GPIOD base address is 0x40020C00
//   MODER register offset is 0x00
//   ODR   register offset is 0x14
.equ     GPIOD_MODER,   0x40020C00 // GPIOD port mode register (page 281)
.equ     GPIOD_ODR,     0x40020C14 // GPIOD output data register (page 283)
.equ     GPIOA_MODER,   0x40020000 // GPIOA port mode register
.equ     GPIOA_IDR,     0x40020010 // GPIOA input data register
.equ     GPIOB_MODER,   0x40020400 // GPIOB    port mode register
.equ     GPIOB_ODR,     0x40020414 // GPIOB    output data register
```

```
// Start of text section
.section .text
////////////////////////////////////////////////////////////////////////////
// Vectors
////////////////////////////////////////////////////////////////////////////
// Vector table start
// Add all other processor specific exceptions/interrupts in order here
  .long    __StackTop        // Top of the stack. from linker script
  .long    _start +1         // reset location, +1 for thumb mode

////////////////////////////////////////////////////////////////////////////
// Main code starts from here
////////////////////////////////////////////////////////////////////////////

_start:
  // Enable GPIOA GPIOB & GPIOD Peripheral Clock (bit 0, 1 & 3 in AHB1ENR register)
  ldr r6, = RCC_AHB1ENR      // Load peripheral clock reg address to r6
  ldr r5, [r6]               // Read its content to r5
  orr r5, 0x0000000B         // Set bit 0,1 & 3 to enable GPIOA GPIOB & GPIOD clock
  str r5, [r6]               // Store result in peripheral clock register

  // Make GPIOD Pin 0,12,13,14,15 as output pin
  ldr r6, = GPIOD_MODER      // Load GPIOD MODER register address to r6
  ldr r5, [r6]               // Read its content to r5
  and r5, 0x00FFFFFF         // Clear bits 24, 31 for P12,13,14,15
  orr r5, 0x55000000         // Write 01 to bits 24-31 for P12
  str r5, [r6]               // Store result in GPIOD MODER register

  //GPIOA ( button )
  ldr r6, = GPIOA_MODER      // Load GPIOA MODER register to r6
  ldr r5, [r6]               // Read its content to r5
  and r5, 0xFFFFFFFC         // Clear bits
  orr r5, 0x0000000C         //
  str r5, [r6]               // Store result in GPIOA MODER register

  //GPIOB ( Pin )
  ldr r6, = GPIOB_MODER      // Load GPIOB MODER register to r6
  ldr r5, [r6]               // Read its content to r5
  and r5, 0xFFFFFFFE         // Cleart bits
  orr r5, 0x00000001         // Write 01 to bits 0 & 1
  str r5, [r6]               // Store result in GPIOB MODER register


////////////////////////////////////////////////////////////////////////////
//       @@       START        @@ //
////////////////////////////////////////////////////////////////////////////

  //@ Lights all LEDs for 1 sec at startup
X0:                          // LEDs ON
  ldr r6, = GPIOD_ODR
  ldr r5, [r6]
  orr r5, 0xF000
  str r5, [r6]
  ldr r7, =LEDDELAY
```

```
DELAY1:                    // Decides how long LEDs will be lit
  cbz r7, Y0               // If r7's data is 0, goes to 'Y0'
  subs r7, r7, #1          // Decreases r7 by 1 and writes to r7
  b DELAY1                 // Goes back to 'DELAY1'

Y0:                        // LEDs OFF
  and r5, 0x000
  str r5, [r6]
  ldr r7, =LEDDELAY

DELAY2:
  cbz r7, Prep             // If r7's data is 0, goes to 'Prep'
  subs r7, r7, #1          // Decreases r7 by 1 and writes to r7
  b DELAY2                 // Goes back to 'DELAY2'

/////////////////////////////////////////////////////////////////////////////

Prep:                // Constant values loaded to registers
  ldr r2, =identifier
  ldr r3, =source_adr
  ldr r4, =dest_adr
  and r1, 0x00       // Clearing r1 which will hold whole data frame

  orr r5, r2, r3     // Combining r2 and r3's data
  orr r1, r4, r5     // Data Frame packing, R1 -> identifier &
                     // source_adr & dest_adr & 00


  ldr r10, =RC_start // RC initial value loaded from constant to r10
  ldr r9, =mod_num   // mod 256's value loaded from constant to r9

/////////////////////////////////////////////////////////////////////////////

Button:              // Checks button press
  and r5, 0x00       // Clearing r5
  and r6, 0x00       // Clearing r6

  ldr r6, =GPIOA_IDR
  ldr r5, [r6]
  and r5, 0x1        // Clears r5 except 0th bit for reading button press
  cmp r5, #0         // r5's data compared with '0' bit
  beq Button         // if r5's data is equals to 0, go back to the 'Button'

Rolling_Code:        // Rolling code's mod process is done here
  and r5, 0x00       // Clearing r5
  and r6, 0x00       // Clearing r6

  // --TAKING RC’s MOD 256 STARTS HERE-- //

  udiv r5, r10, r9   // r10's data diveded by r9's data and
                     // result is written to r5

  mul r6, r5, r9     // r5's data multiplied by r9's data and
                     // result is written to r6

  subs r10, r10, r6  // r6'^s data substracted from r10 and
                     // result written to r10

                     // r10 = ( (RC) mod256 )
```

```
 // --TAKING RC’s MOD 256 ENDS HERE-- //

 and r12, #0         // Clearing r12
 orr r12, r1         // Data Frame(r1) transferred to r12
 orr r12, r10        // Data frame is packed, R12 -> identifier & source_adr & dest_adr & RC

/////////////////////////////////////////////////////////////////////////////

        // Lighs 2 LEDs(Orange & Blue) for 1 sec upon button press

X:                          // LEDs ON
 ldr r6, = GPIOD_ODR
 ldr r5, [r6]
 orr r5, 0xA000
 str r5, [r6]
 ldr r7, =LEDDELAY          // Calls LEDDELAY constant and loads it to r7

DELAY3:                     // Decides how long LEDs will be lit
 cbz r7, Y                  // If r7's data is 0, goes to 'Y'
 subs r7, r7, #1            // Decreases r7 by 1 and writes to r7
 b DELAY3                   // Goes back to 'DELAY3'

Y:                          // LEDs OFF
 and r5, 0x00
 str r5, [r6]
 ldr r7, =LEDDELAY          // Calls LEDDELAY constant and loads it to r7

DELAY4:                     // Decides how long system will be suspended
 cbz r7, RC_LED             // If r7's data is 0, goes to 'RC_LED'
 subs r7, r7, #1            // Decreases r7 by 1 and writes to r7
 b DELAY4                   // Goes back to 'DELAY4'

/////////////////////////////////////////////////////////////////////////////

RC_LED:                     // RC's last 4 bit is taken for displaying on LEDs
 and r8, 0x00               // Clearing r8
 orr r8, r10                // r10's data(RC) transferred to r8
 add r10, #1                // r10's data(RC) increased by 1
 lsl r8, #12                // r8's data shifted to left by 12 bits

X1:                         // LEDs ON
 ldr r6, =GPIOD_ODR
 ldr r5, [r6]
 orr r5, r8                 // Data input for LEDs (RC's last 4 bit is in r8)
 str r5, [r6]
 ldr r7, =LEDDELAY          // Calls LEDDELAY constant and loads it to r7

DELAY5:                     // Decides how long LEDs will be lit
 cbz r7, Y1                 // If r7's data is 0, goes to 'Y1'
 subs r7, r7, #1            // Decreases r7 by 1 and writes to r7
 b DELAY5                   // Goes back to 'DELAY5'

Y1:                         // LEDs OFF
 and r5, 0x00
 str r5, [r6]
 ldr r7, =LEDDELAY          // Calls LEDDELAY constant and loads it to r7
```

```
//////////////////////////////////////////////////////////////////////////////

Encrypt:                    //Encryption of Data Frame is done in here
  and r0, #0                // Clearing r0
  and r6, 0x00              // Clearing r6
  ldr r6, =Encrypt_Key      // Calls Encryption Key constant and
                            // loads it to r6
  eor r0, r12, r6           // Data frame is XORed with hardcoded encryption
                            // key and written on r0
  movs r7, #32              //Decimal number 32 loaded to r7,
                            // this'll be bit counter in M.E.

//////////////////////////////////////////////////////////////////////////////

ME_Bit_Seperator:
  cbz r7, Fin                             // When counter reaches 0, all the data frame is sent
  and r11, r0, #1                   // All data is deleted except for 0th bit and it's written to r11

  cmp r11, #1                       // Compares r11's data wit '1'
  beq ME_one                        // Goes M.E. if 0th bit is 1
  bne ME_zero                       // Goes M.E. if 0th bit is 0

//////////////////////////////////////////////////////////////////////////////

ME_one:
  ldr r6, = GPIOB_ODR
  ldr r5, [r6]
  and r5, 0xFFFFFFFE
  orr r5, 0x01              // bit '1' (high) sent to PB0 pin
  str r5, [r6]

  ldr r6, =BITDELAY         // Calls BITDELAY constant and loads it to r6

DELAY6:                     // Decides how long bit will shown on pin
  cbz r6, ME_one_2          // If r6's data is 0, goes to 'ME_one_2'
  subs r6, r6, #1           // Decreases r6 by 1 and writes to r6
  b DELAY6                  // Goes back to 'DELAY6'

ME_one_2:
  ldr r6, = GPIOB_ODR
  ldr r5, [r6]
  and r5, 0xFFFFFFFE
  orr r5, 0x00              // bit '0' (high) sent to PB0 pin
  str r5, [r6]

  lsr r0, #1                // Shifts Whole data frame to right by 1
  subs r7, r7, #1           // Decreases bit counter by 1

  ldr r6, =BITDELAY         // Calls BITDELAY constant and loads it to r6

DELAY7:                     // Decides how long bit will shown on pin
  cbz r6, Bridge            // If r6's data is 0, goes to 'Bridge'
  subs r6, r6, #1           // Decreases r6 by 1 and writes to r6
  b DELAY7

//////////////////////////////////////////////////////////////////////////////
```

```


ME_zero:
  ldr r6, = GPIOB_ODR
  ldr r5, [r6]
  and r5, 0xFFFFFFFE
  orr r5, 0x00                // bit '0' (low) sent to PB0 pin
  str r5, [r6]

  ldr r6, =BITDELAY           // Calls BITDELAY constant and loads it to r6

DELAY8:                       // Decides how long bit will shown on pin
  cbz r6, ME_zero_2           // If r6's data is 0, goes to 'ME_zero_2'
  subs r6, r6, #1             // Decreases r6 by 1 and writes to r6
  b DELAY8                    // Goes back to 'DELAY7'

ME_zero_2:
  ldr r6, = GPIOB_ODR
  ldr r5, [r6]
  and r5, 0xFFFFFFFE
  orr r5, 0x01                // bit '1' (high) sent to PB0 pin
  str r5, [r6]

  lsr r0, #1                  // Shifts Whole data frame to right by 1
  subs r7, r7, #1             // Decreases bit counter by 1

  ldr r6, =BITDELAY           // Calls BITDELAY constant and loads it to r6

DELAY9:                       // Decides how long bit will shown on pin
  cbz r6, Bridge              // If r6's data is 0, goes to 'Bridge'
  subs r6, r6, #1             // Decreases r6 by 1 and writes to r6
  b DELAY9

//////////////////////////////////////////////////////////////////////////////

Bridge:
  b ME_Bit_Seperator

Fin:
  b Button                    // Job is done, go back to the 'Button'
```

6 References

- STM32F407 Reference Manual