# DCC Report - Key Technical Documentation

Zeddicaius Serjeant (1261476)

19 June 2020

## 1 Core Algorithms

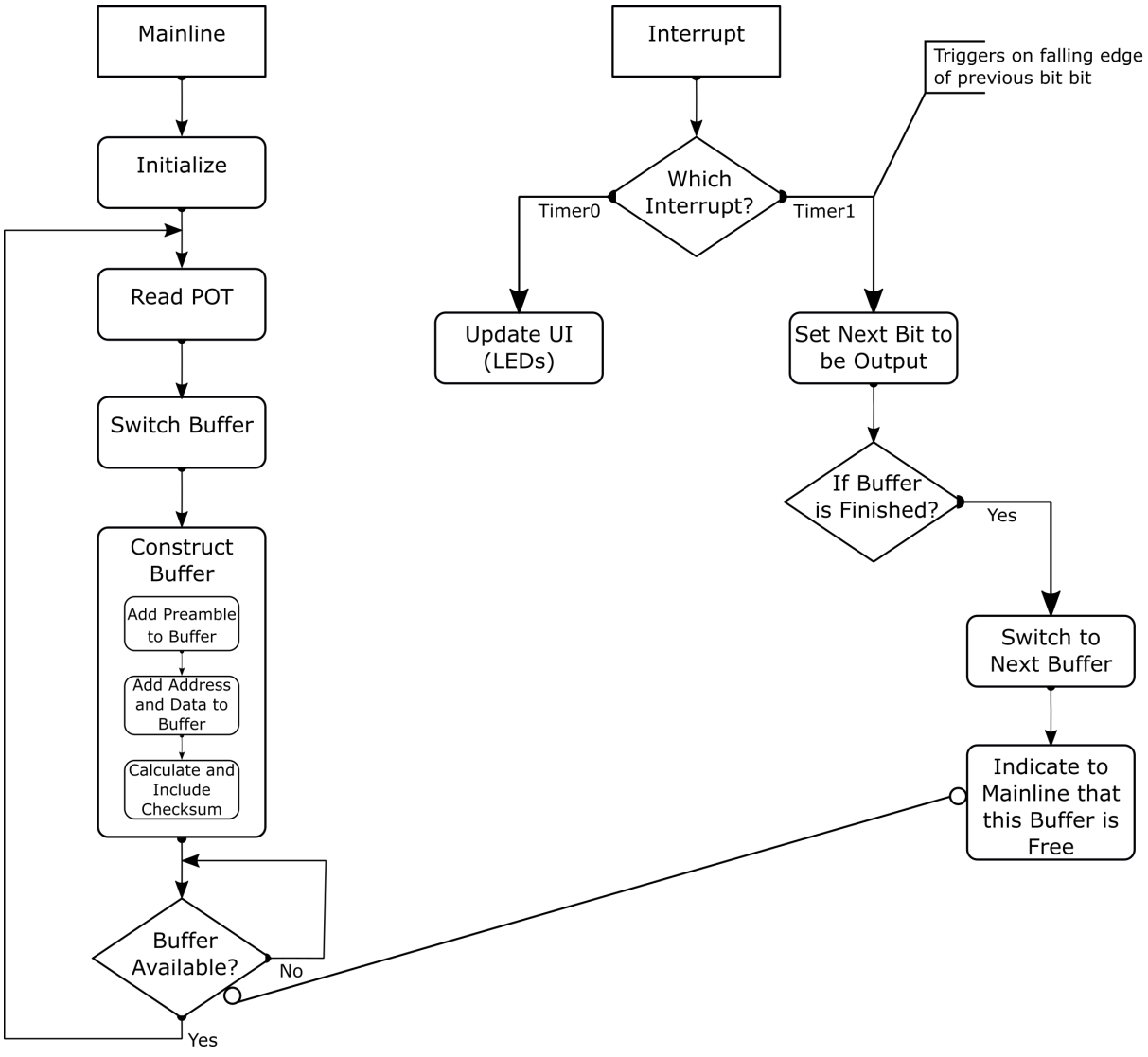


Figure 1: Flow Chart of Transmitter

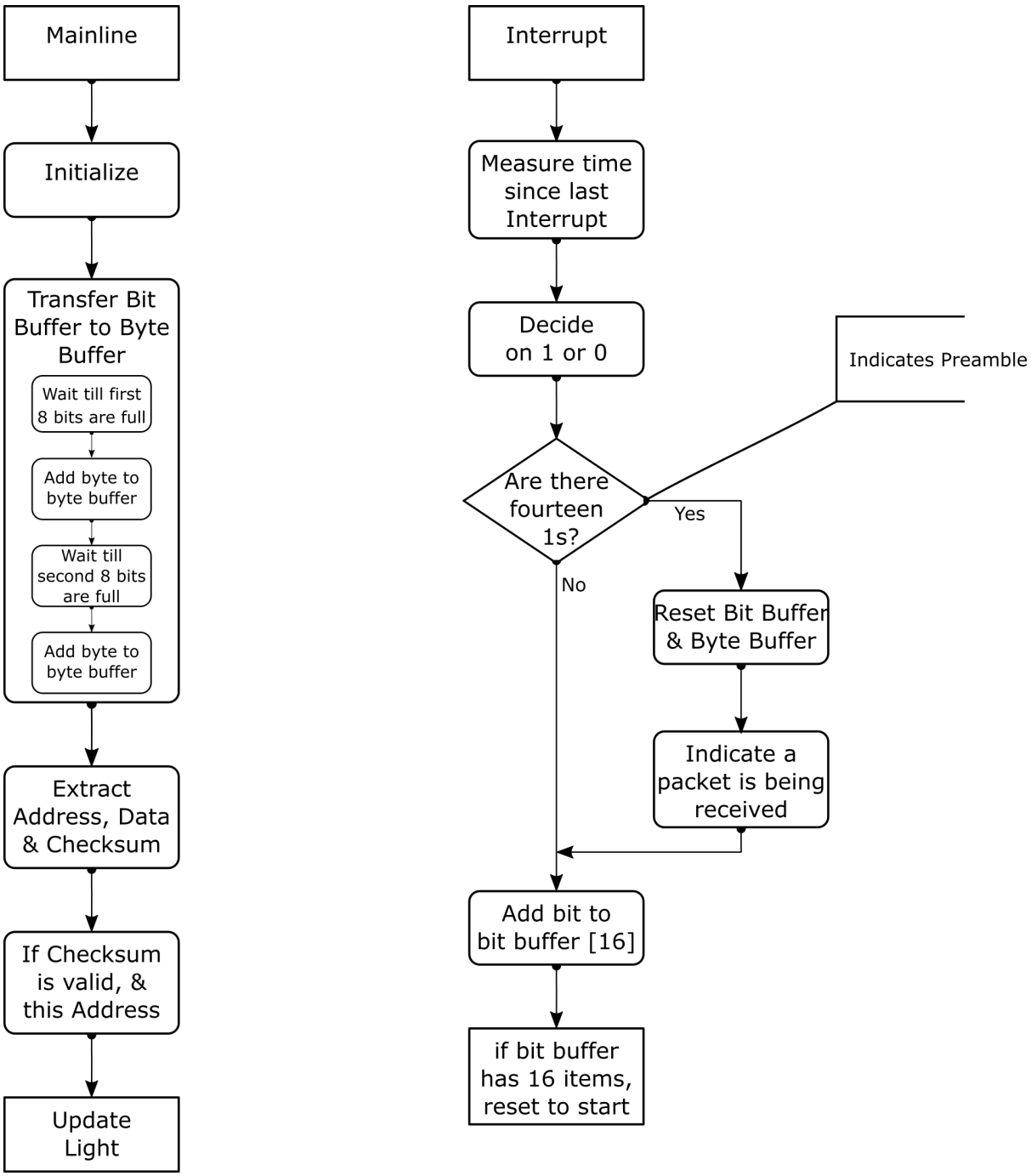


Figure 2: Flow Chart of Receiver

# 2 Circuit Diagrams

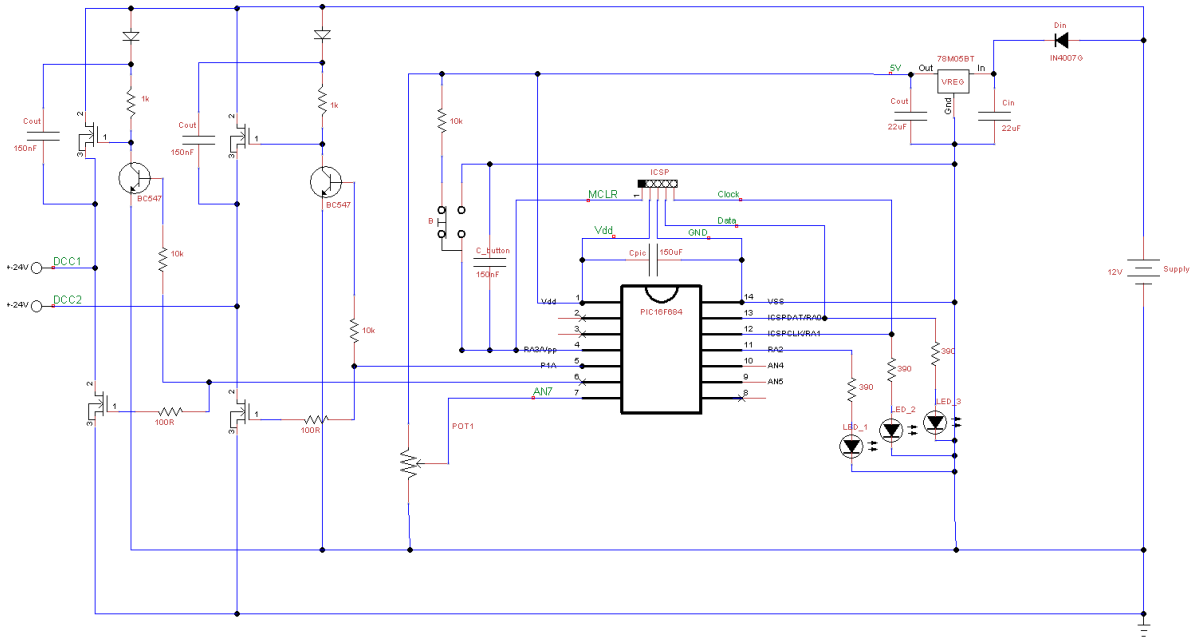


Figure 3: Circuit Diagram of Transmitter

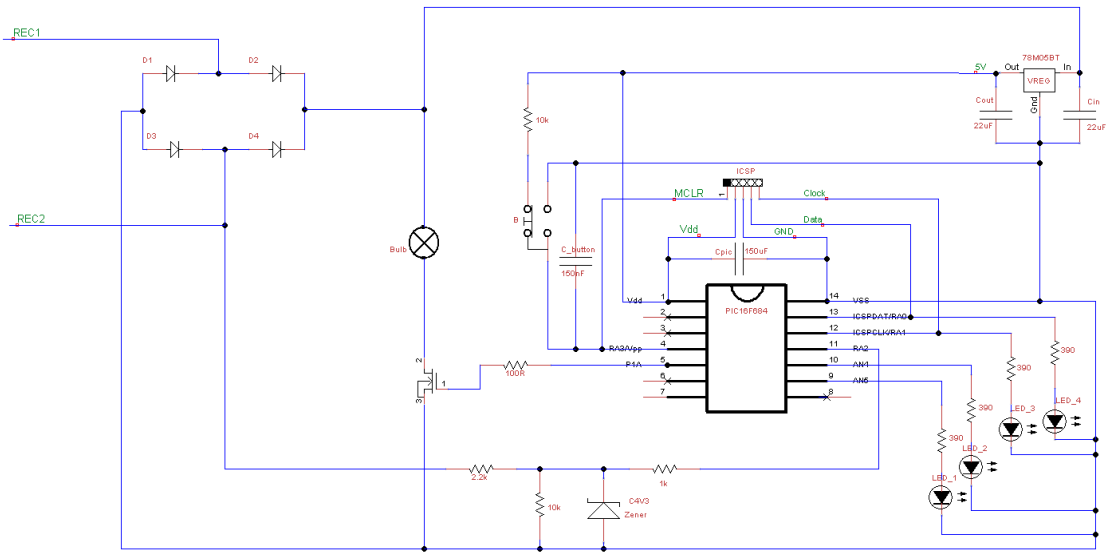


Figure 4: Circuit Diagram of Receiver

# 3 Timing

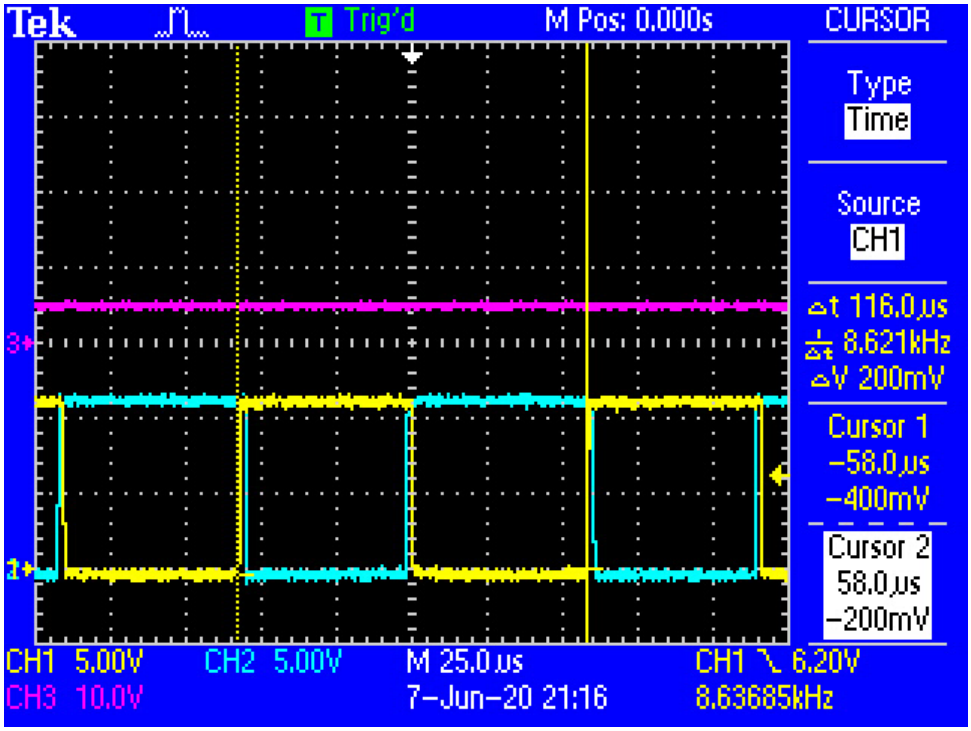


Figure 5: Timing of a ONE

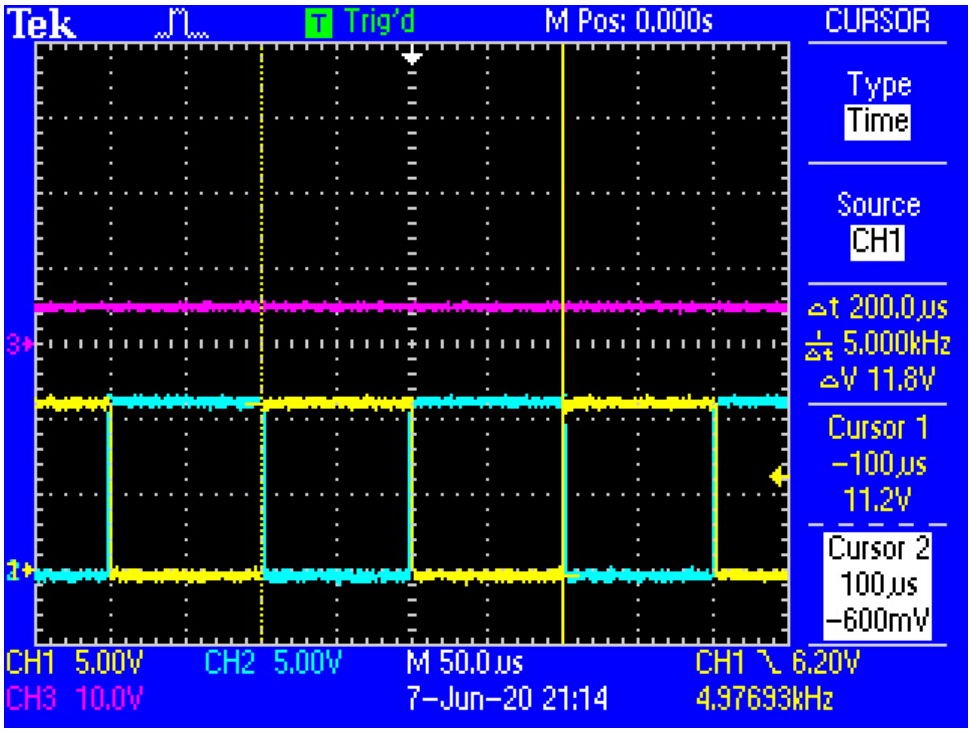


Figure 6: Timing of a ZERO

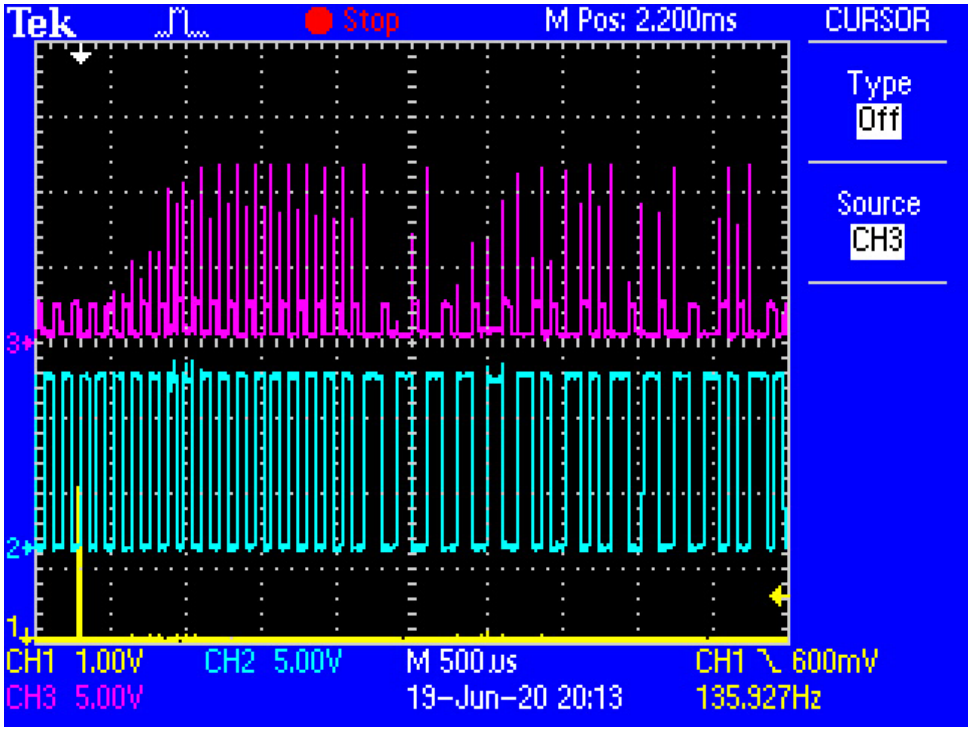


Figure 7: Preamble for each packet

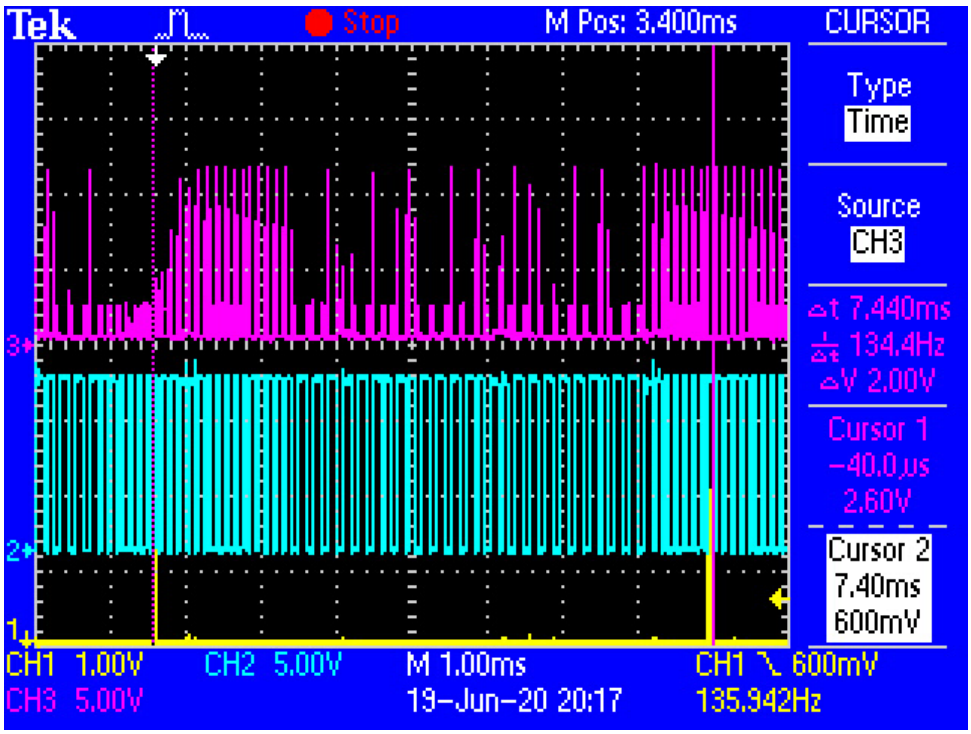


Figure 8: Length of average full packet

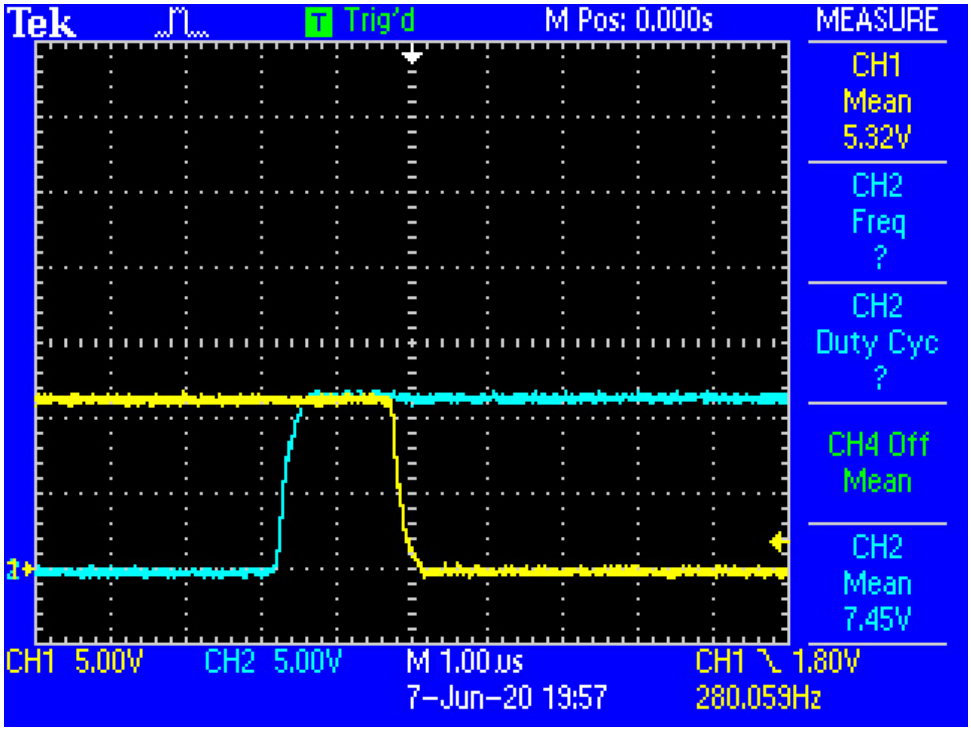


Figure 9: Deadband within PWM Timing of Transmitter

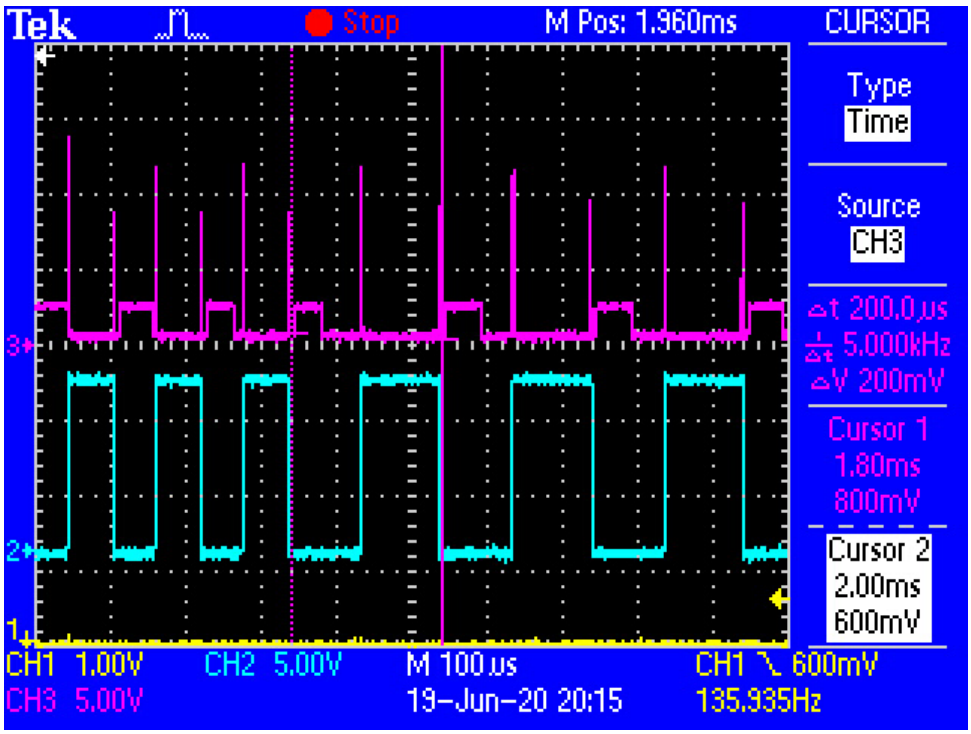


Figure 10: Time Spent in Interrupt when core is running at 8MHz - Receiver

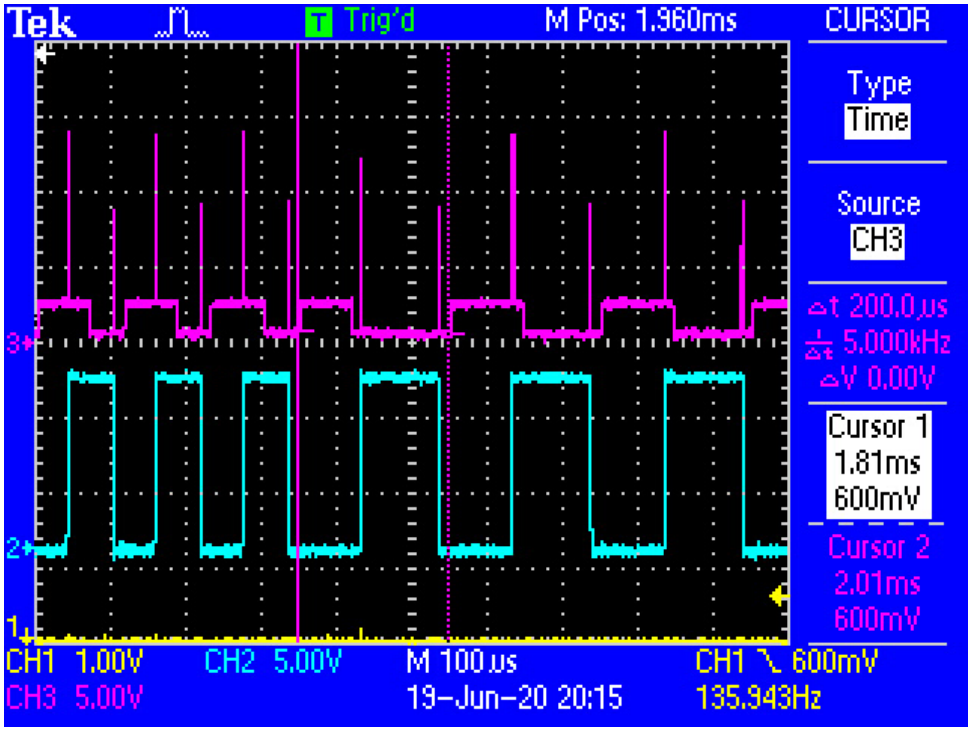


Figure 11: Time Spent in Interrupt when core is running at 4MHz - Receiver

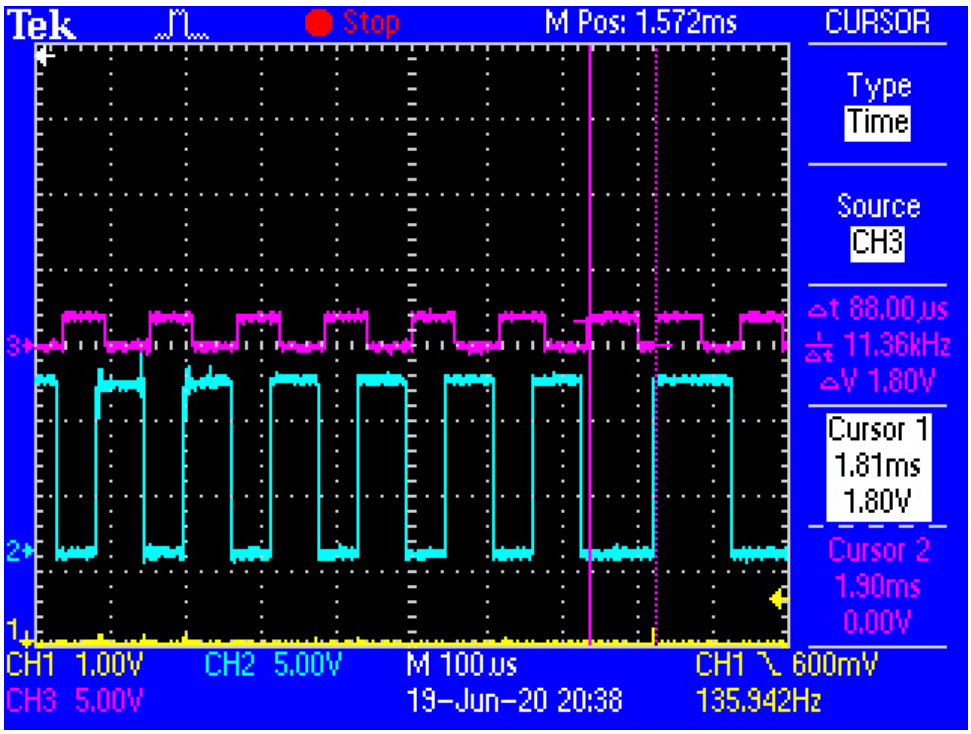


Figure 12: Time Spent in Interrupt when core is running at 4MHz - Transmit

# 4 Interface

Table 1: Transmission Button

| Button | State |
|---|---|
| Default | Transmit to First Device |
| First Press | Transmit Corrupt Packet |
| Second Press | Transmit to Different Device |
| Third Press | Move to Default State |

Table 2: Transmission LEDs

| LED | Meaning |
|---|---|
| LED_1 | Lights when a new packet is being transmitted (For Diagnostics) |
| LED_2 | Lights when Button has been pressed |
| LED_3 | Duty Cycle indicates current POT reading |

Table 3: Receiver Button

| Button | State |
|---|---|
| Default | Clockrate is 8MHz, bulb follows packet |
| First Press | Bulb forced to full brightness |
| Second Press | Clockrate is 4MHz, bulb follows packet |
| Third Press | Move to Default State |

Table 4: Receiver LEDs

| LED | Meaning |
|---|---|
| LED_1 | On when there is a bitstream (Receiver Interrupt cycle) |
| LED_2 | Indicates an error (Corrupt Packet with valid preamble) |
| LED_3 | Indicates that the received packet is for this device |
| LED_4 | Indicates a Packet with a valid Preamble has been detected |

# 5 Optimisations

Certain parts of the code have been modified for speed. As these may be confusing to a future engineer, this section gives a brief explanation.

## 5.1 Pregenerated Bit Masks

The PIC16F684 doesn't have hardware support for bitwise shifts of more than one. Instead, it generates a loop to accomplish this. This takes many cycles and so these shifts are computed at start up.

## 5.2 Double Buffer in Receiver

To enable a lower clock speed on the receiver side, the system avoids compressing bits into bytes within the interrupt. Instead, there is a two byte (16 bit) buffer that the interrupt continuously writes into, wrapping around using modulo. Code in the mainly detects when eight bits have been written, then it compresses them into a byte and places it in a second buffer while the interrupt writes the second byte. This repeats until the byte buffer is full before processing. This leads to much more memory used (16 bits are stored in 16 bytes) however it leads to much quicker interrupt.

## 5.3 Byte Buffer

Because of the way these optimisations interact with the DCC standard, the layout of the byte buffer is unusual and presented here to better understand code that extracts data from a packet.

Legend:

**A** : Address bit

**D** : Data bit

**E** : Checksum bit

Table 5: Byte Buffer Structure

| Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|
| `0 AAAAAAA` | `A 0 DDDDDD` | `DD 0 EEEEE` | `EEE 1 1111` |

## 6 Physical Layout

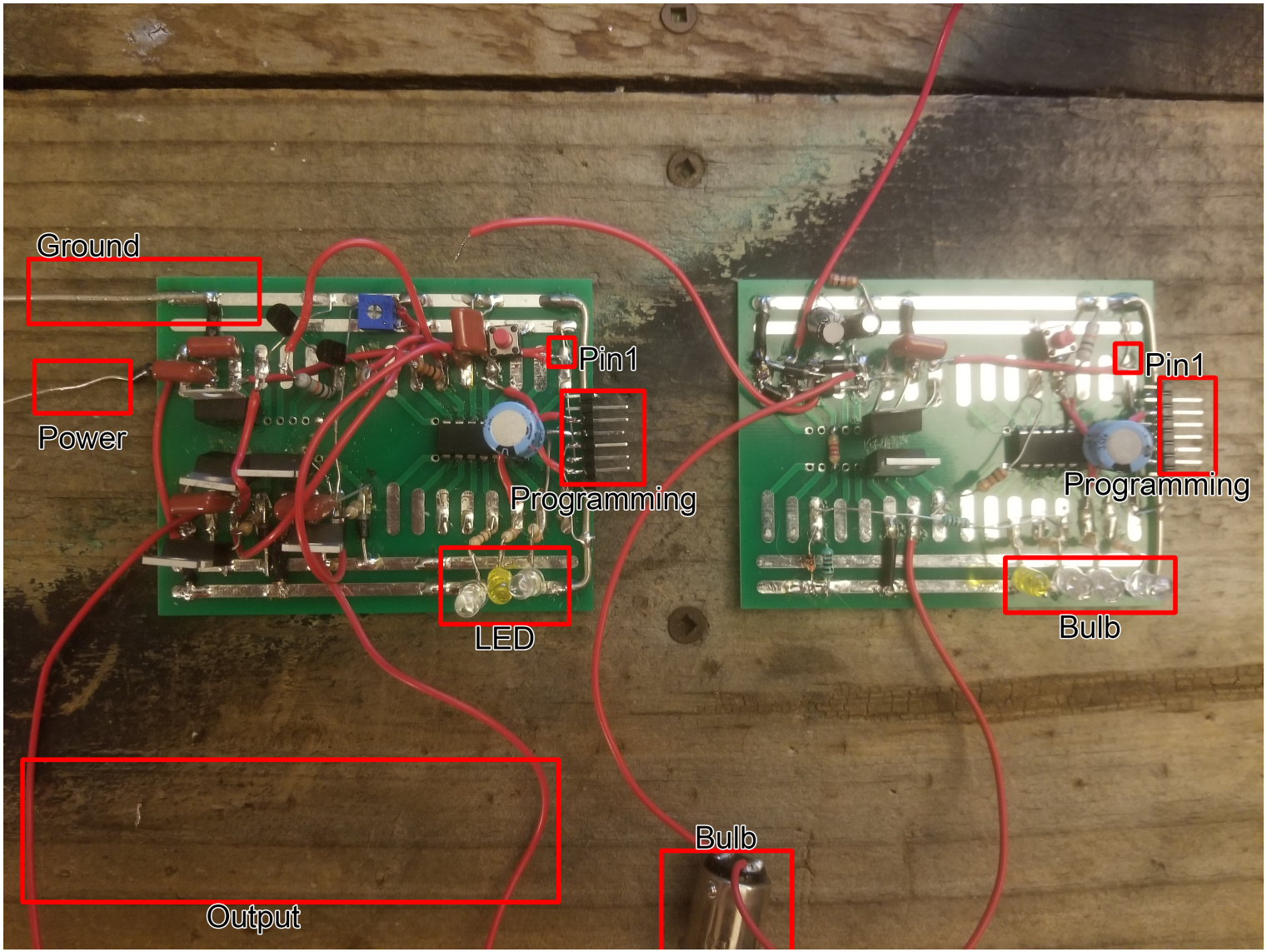


Figure 13: Circuit Diagram of Receiver

# 7 Code - Transmitter

## 7.1 head.h

```
#include <xc.h> // include processor files - each processor file is guarded.

enum FLAGS {OFF=0, OUTPUT=0, INTERNAL=0, CLEAR=0, DIGITAL=0, FALSE=0, LEFT=0, ANALOG=1, INPUT=1, ON=1, RIGHT=1, TRUE=1};
#define GLOBAL_INTERRUPTS GIE

//Pins

#define DAT_LED RA0
#define DAT_LED_PIN TRISA0
#define DAT_LED_TYPE ANS0
#define DAT_LED_BIT 0b000001

#define CLK_LED RA1
#define CLK_LED_PIN TRISA1
#define CLK_LED_TYPE ANS1
#define CLK_LED_BIT 0b000010

#define IND_LED RA2
#define IND_LED_PIN TRISA2
#define IND_LED_TYPE ANS2
#define IND_LED_BIT 0b000100


#define DCC1 RC4
#define DCC1_PIN TRISC4

#define DCC2 RC5
#define DCC2_PIN TRISC5


// button
#define BUTTON RA3
#define BUTTON_PIN TRISA3
#define BUTTON_INTERRUPT IOCA3
#define RA_INTERRUPT RAIE
#define BUTTON_INTERRUPT_FLAG RAIF

//POT
#define POT RC3
#define POT_PIN TRISC3
#define POT_TYPE ANS7

//TIMER0
#define PRESCALER PSA
#define TIMER0_COUNTER TMR0
#define TIMER0_CLOCK_SCOURCE T0CS
#define TIMER0_INTERRUPT T0IE
#define TIMER0_INTERRUPT_FLAG TMR0IF

//timer 1
#define TIMER1_H TMR1H
#define TIMER1_L TMR1L
```

```
#define TIMER1_INTERRUPT_FLAG TMR1IF
#define TIMER1_INTERRUPT TMR1IE
#define TIMER1 TMR1ON


//PWM
//parts are specified as an offset within a register
#define ECCP_CONTROL CCP1CON
// #define PWM_MODE P1M
#define PWM_MODE (unsigned char)6
// #define PWM_OUTPUT CCP1M
#define PWM_OUTPUT (unsigned char)0
#define PWM_DUTYCYCLE_MSB CCPR1L
// #define PWM_DUTYCYCLE_LSB DC1B
#define PWM_DUTYCYCLE_LSB (unsigned char)4
#define PWM_PERIOD PR2
#define PWM_CONTROL PWM1CON

//timer 2 (needed for PWM)
#define TIMER2 TMR2
#define TIMER2_CONTROL T2CON
#define TIMER2_ON 2
#define TIMER2_INTERRUPT_FLAG TMR2IF
#define TIMER_CLOCK_PRESCALE (unsigned char)0
#define TIMER_CLOCK_POSTSCALE (unsigned char)3
#define PRESCALE_1 (unsigned char)0b00
#define PRESCALE_4 (unsigned char)0b01
#define PRESCALE_16 (unsigned char)0b10

#define PERIPHAL_INTERRUPT PEIE
#define TIMER2_INTERRUPT TMR2IE

// #define TIMER2_PRESCALER T2CKPS
#define ACTIVE_HIGH_ACTIVE_HIGH (unsigned char)0b1100
#define ACTIVE_LOW_ACTIVE_LOW (unsigned char)0b1111
#define SINGLE_OUTPUT (unsigned char)0b00
#define HALF_BRIDGE (unsigned char)0b10

//ADC
#define ADC_VOLTAGE_REFERENCE VCFG
#define ADC_CLOCK_SOURCE2 ADCS2
#define ADC_CLOCK_SOURCE1 ADCS1
#define ADC_CLOCK_SOURCE0 ADCS0
#define ADC_CHANNEL2 CHS2
#define ADC_CHANNEL1 CHS1
#define ADC_CHANNEL0 CHS0
#define ADC_GODONE GO_DONE
#define ADC_OUTPUT_FORMAT ADFM
#define ADC_RESULT_HIGH ADRESH
#define ADC_RESULT_LOW ADRESL
#define ADC_INTERRUPT ADIE
#define ADC_INTERRUPT_FLAG ADIF
#define ADC_ON ADON
#define ADC_PAUSE asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;")
```

## 7.2 main.c

```
// CONFIG
#pragma config FOSC = INTOSCIO  // Oscillator Selection bits (INTOSCIO oscillator: I/O function on RA4/OSC2/CLKOUT pin, I/O function on RA5/OSC1/CLKIN)
#pragma config WDTE = OFF       // Watchdog Timer Enable bit (WDT disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable bit (PWRT disabled)
#pragma config MCLRE = OFF      // MCLR Pin Function Select bit (MCLR pin function is digital input, MCLR internally tied to VDD)
#pragma config CP = OFF         // Code Protection bit (Program memory code protection is disabled)
#pragma config CPD = OFF        // Data Code Protection bit (Data memory code protection is disabled)
#pragma config BOREN = OFF      // Brown Out Detect (BOR disabled)
#pragma config IESO = OFF       // Internal External Switchover bit (Internal External Switchover mode is disabled)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enabled bit (Fail-Safe Clock Monitor is disabled)

#include "head.h"

// for millisecond timer
#define TIMER0_INITIAL 132

// period of flashing LED [ms]
#define LED_PERIOD 255
unsigned short int led_duty_cycle_counter = 0;
unsigned short int led_duty_cycle = 0; // Duty cycle of LED as on_time[ms]
unsigned char led_state;
__bit reset; // to indicate if devices downstream should reset

//button
unsigned char button_state = 0; // default unpressed


// shadow register for PortA, so as to not suffer from read/modify/write errors
volatile union
{
    unsigned char byte;
    struct
    {
        unsigned RA0:1;
        unsigned RA1:1;
        unsigned RA2:1;
        unsigned RA3:1;
        unsigned RA4:1;
        unsigned RA5:1;
    } bits;
    struct
    {
        unsigned DAT_LED:1;
        unsigned CLK_LED:1;
        unsigned IND_LED:1;
        unsigned _:3;
    };
} PORTA_SH;

// DCC stuff
// to calibrate transmitor
#define ONE_BIT_DUTY 56
#define ONE_BIT_PERIOD 115
```

```
#define ZERO_BIT_DUTY 98
#define ZERO_BIT_PERIOD 200

volatile __bit packet_ready; // whether we have a packet ready to send
__bit buffer; // 0: buffer0, 1:buffer1
//buffers for DCC packets. There are 2: one for current transmission, one for the next
#define BUFFER_LENGTH 6
//0b11100101
unsigned char buffer0[BUFFER_LENGTH] = {0xFF, 0xFE, 0,0,0,0};
unsigned char buffer1[BUFFER_LENGTH] = {0xFF, 0xFE, 0,0,0,0};
unsigned char *next_buffer; // allows the system to point to the buffer that is ready for transmit
unsigned char MASKS[8]; // this contains the bit masks, rather than shifting each cycle.
//This is because there is no hardware support for multiple shifts, and it takes precious cycles.


void __interrupt() ISR()
{
    static unsigned char *current_buffer = buffer0;
    static unsigned char index_bit = 0;
    static unsigned char index_byte = 0;
    static unsigned char current_bit=1; // used to speed up the setting of the pwm period

    static unsigned char button_count = 255; // for debounce. I'm using polling, not interrupts, due this: http://www.ganssle.com/debouncing.htm
    static __bit button_change; // so holding the button doesn't break things

    //connected to the PWM, outputs datastream
    if (TIMER2_INTERRUPT_FLAG)
    {
        TIMER2_INTERRUPT_FLAG = CLEAR;

        //set pwm period for this cycle
        if (current_bit)
        {
            PWM_PERIOD = ONE_BIT_PERIOD;
        }
        else
        {
            PWM_PERIOD = ZERO_BIT_PERIOD;
            TIMER0_INTERRUPT = ON; // Turn this on, since a zero is long enough to allow other interrupts
        }

        // get next bit from buffer. It is counted using two chars, because shifting (index>>3) doesn't have hardware support, so takes precious cycles
        index_bit++;
        if (index_bit > 7)
        {
            index_bit = 0;
            index_byte++;
            if (index_byte == BUFFER_LENGTH) // check to see if we have output an entire buffer
            {
                PORTA = 0x4 | led_state; // indicate the packet, without breaking other LEDS
                index_byte = 0;
                if (packet_ready) // if there is a new packet that is ready to transmit, do that. Otherwise, retransmit this one
                {
                    current_buffer = next_buffer;
                    packet_ready = FALSE;
                }
```

```
            PORTA = led_state; // turn off IND_LED without affecting other LEDS
        }
    }

    current_bit = current_buffer[index_byte] & (MASKS[7-index_bit]); // fetch current bit, using pregenerated mask for speed

    if (current_bit) // if next bit is a one
    {
        PWM_DUTYCYCLE_MSB = ONE_BIT_DUTY; // the bit of the next cycle will be a one
    }
    else // if its a zero
    {
        PWM_DUTYCYCLE_MSB = ZERO_BIT_DUTY; // the bit of the next cycle will be a zero

    }
}

//millisecond interrupt for LED control
else if (TIMER0_INTERRUPT_FLAG) // if the timer0 interrupt flag was set (timer0 triggered)
{
    TIMER0_INTERRUPT_FLAG = CLEAR; // clear interrupt flag since we are dealing with it
    TIMER0_COUNTER = TIMER0_INITIAL + 2; // reset counter, but also add 2 since it takes 2 clock cycles to get going

    led_duty_cycle_counter++; // increment the led counter

    if (led_duty_cycle_counter >= led_duty_cycle) // toggle state
    {
        if (led_duty_cycle_counter >= LED_PERIOD) // restart cycle
        {
            led_duty_cycle_counter -= LED_PERIOD; //reset led counter safely
            // led_state = ON; // we are in the ON part of the duty cycle
        }
        else
        {
            led_state = OFF;
        }
    }
    else
    {
        led_state = ON; // within On part of duty cycle
    }

    //button press
    // debounce
    if (BUTTON && button_count==0)
    {
        button_count = 30;
    }
    else if (button_count > 1)
    {
        button_count--;

        if (reset) //flash reset led
        {
            CLK_LED = ON;
        }
```

```
        }
        else if (button_count == 1)
        {
            if (BUTTON)
            {
                if (button_change) // if the button was definitely pressed
                {
                    button_change = 0;
                    button_count = 255;
                    button_state++;
                    button_state = button_state%3; // three states

                    reset = TRUE; // upon a press, send a reset signal. button_state can be used to detect if this has occured

                }
            }
            else
            {
                reset = FALSE;
                button_change = 1;
                button_count = 0;
            }
        }
        TIMER0_INTERRUPT = OFF; // turn off until the pwm turns it back on when it has time

    }
}

void main()
{
    unsigned char target_address = 0b00000101; // the device we will be talking to
    unsigned char checksum;

    // initialize pins
    DAT_LED_TYPE = DIGITAL;
    DAT_LED_PIN = OUTPUT;
    CLK_LED_TYPE = DIGITAL;
    CLK_LED_PIN = OUTPUT;
    IND_LED_TYPE = DIGITAL;
    IND_LED_PIN = OUTPUT;

    DCC1_PIN = OUTPUT;
    DCC2_PIN = OUTPUT;

    // Set up timer0
    // calculate intial for accurate timing $ inital = TimerMax-((Delay*Fosc)/(Prescaler*4))
    TIMER0_COUNTER = TIMER0_INITIAL; // set counter
    TIMER0_CLOCK_SCOURCE = INTERNAL; // internal clock
    PRESCALER = 0; // enable prescaler for Timer0
    PS2=0; PS1=1; PS0=0; // Set prescaler to 1:8
    // TIMER0_INTERRUPT = ON; // enable timer0 interrupts

    //setup PWM and timer 2 for data out
    TIMER2_CONTROL = (ON<<TIMER2_ON) | (PRESCALE_1<<TIMER_CLOCK_PRESCALE); // Timer 2 register
    PWM_PERIOD = ONE_BIT_PERIOD;
    PWM_DUTYCYCLE_MSB = ONE_BIT_DUTY; //defaut to sending a DCC-1 bit
```

```
PWM_CONTROL = 0x03; // 4 instruction deadband
ECCP_CONTROL = (HALF_BRIDGE<<PWM_MODE) | (ACTIVE_HIGH_ACTIVE_HIGH<<PWM_OUTPUT); //PWM register set
PERIPHAL_INTERRUPT = ON;
TIMER2_INTERRUPT = ON;

//fill bit masks (pregenerated for speed)
for (char i=0; i<8; i++)
{
    MASKS[i] = 1<<i;
}

//Set up ADC
// POT_PIN = INPUT;
// POT_TYPE = ANALOG;
ADC_VOLTAGE_REFERENCE = INTERNAL;
ADC_CHANNEL2 = 1; ADC_CHANNEL1 = 1; ADC_CHANNEL0 = 1; // Set the channel to AN7 (where the POT is)
ADC_CLOCK_SOURCE2 = 0; ADC_CLOCK_SOURCE1 = 0; ADC_CLOCK_SOURCE0 = 1; // Set the clock rate of the ADC
ADC_OUTPUT_FORMAT = LEFT; // right Shifted ADC_RESULT_HIGH contains the first 2 bits
// ADC_INTERRUPT = OFF; // by default these aren't necessary
ADC_ON = ON; // turn it on

//turn on interrupts
GLOBAL_INTERRUPTS = ON;

while (1)
{
    if (!packet_ready) // if a new packet hasn't already been generated (both buffers are thus full/being processed)
    {
        if (buffer == 0) // if buffer0 is currently being transmitted
        {
            next_buffer = buffer1; // make buffer1 the next buffer
            buffer = 1;
        }
        else if (buffer == 1) // if buffer1 is currently being transmitted
        {
            next_buffer = buffer0;
            buffer = 0;
        }

        if (button_state == 0) // normal transmit mode
        {
            //fetch data
            ADC_GODONE = ON;
            while(ADC_GODONE);

            //output this value as an LED
            led_duty_cycle = ADC_RESULT_HIGH;

            // calculate checksum byte (xor of address and data)
            checksum = target_address ^ ADC_RESULT_HIGH;

            //fill buffer
            next_buffer[0] = 0xFF;
            next_buffer[1] = 0xFE; // preamble
            next_buffer[2] = target_address;
            next_buffer[3] = ADC_RESULT_HIGH>>1; // shifted because the protocol demands a zero at the front
```

```
            next_buffer[4] = (ADC_RESULT_HIGH<<7) | (checksum>>2); // contains last bit of data, a zero, then 6 bits of checksum
            next_buffer[5] = (checksum<<6) | 0x3F; // contains last 2 bits of checksum,
            //then the packet end bit (one) then a stream of ones to link into the next preamble
        }
        else if (button_state == 1) // blank state for demo
        {
            for (int i=0; i<BUFFER_LENGTH; i++)
            {
                next_buffer[i] = 0; // corrupt buffer for demonstration
            }

            led_duty_cycle = 0;
        }
        else if (button_state == 2) // output data to different address
        {
            ADC_GODONE = ON;
            while(ADC_GODONE);

            //output this value as an LED for testing
            led_duty_cycle = ADC_RESULT_HIGH;

            // calculate checksum byte (xor of address and data)
            checksum = target_address ^ ADC_RESULT_HIGH;

            next_buffer[0] = 0xFF;
            next_buffer[1] = 0xFE; // preamble
            next_buffer[2] = 0b00000001; // different address
            next_buffer[3] = ADC_RESULT_HIGH>>1; // shifted because the protocol demands a zero at the front
            next_buffer[4] = (ADC_RESULT_HIGH<<7) | (checksum>>2); // contains last bit of data, a zero, then 6 bits of checksum
            next_buffer[5] = (checksum<<6) | 0x3F; // contains last 2 bits of checksum,
            //then the packet end bit (one) then a stream of ones to link into the next preamble
        }
        packet_ready = TRUE; // let transmit know there is a packet ready
    }
  }
}
```

# 8 Code - Receiver

## 8.1 head.h

```
#include <xc.h> // include processor files - each processor file is guarded.

enum FLAGS {OFF=0, OUTPUT=0, INTERNAL=0, CLEAR=0, DIGITAL=0, FALSE=0, ANALOG=1, INPUT=1, ON=1, RIGHT=1, TRUE=1};
#define GLOBAL_INTERRUPTS GIE

//Pins

#define DAT_LED RA0
#define DAT_LED_PIN TRISA0
#define DAT_LED_TYPE ANS0
#define DAT_LED_BIT 0b000001

#define CLK_LED RA1
#define CLK_LED_PIN TRISA1
#define CLK_LED_TYPE ANS1
#define CLK_LED_BIT 0b000010

#define NXT_LED RC0
#define NXT_LED_PIN TRISC0
#define NXT_LED_TYPE ANS4

#define YEL_LED RC1
#define YEL_LED_PIN TRISC1
#define YEL_LED_TYPE ANS5

#define PWM RC5
#define PWM_PIN TRISC5

#define DATA RA2
#define DATA_PIN TRISA2
#define DATA_TYPE ANS2

#define EDGE_INTERRUPT INTE
#define EDGE_INTERRUPT_FLAG INTF
#define EDGE_DIRECTION INTEDG
#define RISING 1
#define FALLING 0



// button
#define BUTTON RA3
#define BUTTON_PIN TRISA3
#define BUTTON_INTERRUPT IOCA3
#define RA_INTERRUPT RAIE
#define BUTTON_INTERRUPT_FLAG RAIF

//POT
#define POT RC0
#define POT_PIN TRISC0
#define POT_TYPE ANS4
```

```
//TIMER0
#define PRESCALER PSA
#define TIMER0_COUNTER TMR0
#define TIMER0_CLOCK_SCOURCE T0CS
#define TIMER0_INTERRUPT T0IE
#define TIMER0_INTERRUPT_FLAG TMR0IF

//timer 1
#define TIMER1_H TMR1H
#define TIMER1_L TMR1L
#define TIMER1_INTERRUPT_FLAG TMR1IF
#define TIMER1_INTERRUPT TMR1IE
#define TIMER1 TMR1ON

//timer 2 (needed for PWM)
#define TIMER2 TMR2
#define TIMER2_CONTROL T2CON
#define TIMER2_ON 2
#define TIMER2_INTERRUPT_FLAG TMR2IF
#define TIMER_CLOCK_PRESCALE (unsigned char)0
#define TIMER_CLOCK_POSTSCALE (unsigned char)3
#define PRESCALE_1 (unsigned char)0b00
#define PRESCALE_4 (unsigned char)0b01
#define PRESCALE_16 (unsigned char)0b10


//PWM
//parts are specified as an offset within a register
#define ECCP_CONTROL CCP1CON
// #define PWM_MODE P1M
#define PWM_MODE (unsigned char)6
// #define PWM_OUTPUT CCP1M
#define PWM_OUTPUT (unsigned char)0
#define PWM_DUTYCYCLE_MSB CCPR1L
// #define PWM_DUTYCYCLE_LSB DC1B
#define PWM_DUTYCYCLE_LSB (unsigned char)4
#define PWM_PERIOD PR2
#define PWM_CONTROL PWM1CON



#define PERIPHAL_INTERRUPT PEIE
#define TIMER2_INTERRUPT TMR2IE

// #define TIMER2_PRESCALER T2CKPS
#define ACTIVE_HIGH_ACTIVE_HIGH (unsigned char)0b1100
#define ACTIVE_LOW_ACTIVE_LOW (unsigned char)0b1111
#define SINGLE_OUTPUT (unsigned char)0b00
#define HALF_BRIDGE (unsigned char)0b10

//ADC
#define ADC_VOLTAGE_REFERENCE VCFG
#define ADC_CLOCK_SOURCE2 ADCS2
#define ADC_CLOCK_SOURCE1 ADCS1
#define ADC_CLOCK_SOURCE0 ADCS0
#define ADC_CHANNEL2 CHS2
```

```
#define ADC_CHANNEL1 CHS1
#define ADC_CHANNEL0 CHS0
#define ADC_GODONE GO_DONE
#define ADC_OUTPUT_FORMAT ADFM
#define ADC_RESULT_HIGH ADRESH
#define ADC_RESULT_LOW ADRESL
#define ADC_INTERRUPT ADIE
#define ADC_INTERRUPT_FLAG ADIF
#define ADC_ON ADON
#define ADC_PAUSE asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;");asm("NOP;")
```

## 8.2 main.c

```
// CONFIG
#pragma config FOSC = INTOSCIO  // Oscillator Selection bits (INTOSCIO oscillator: I/O function on RA4/OSC2/CLKOUT pin, I/O function on RA5/OSC1/CLKIN)
#pragma config WDTE = OFF       // Watchdog Timer Enable bit (WDT disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable bit (PWRT disabled)
#pragma config MCLRE = OFF      // MCLR Pin Function Select bit (MCLR pin function is digital input, MCLR internally tied to VDD)
#pragma config CP = OFF         // Code Protection bit (Program memory code protection is disabled)
#pragma config CPD = OFF        // Data Code Protection bit (Data memory code protection is disabled)
#pragma config BOREN = OFF      // Brown Out Detect (BOR disabled)
#pragma config IESO = OFF       // Internal External Switchover bit (Internal External Switchover mode is disabled)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enabled bit (Fail-Safe Clock Monitor is disabled)

#include "head.h"

#define ADDRESS 0b00000101

//led stuff
#define LED_TIME 2
unsigned char packet_led = 0;//indicate to the user when a packet has been detected
unsigned char update_led = 0;//indicate to the user when a packet is for this device, and that this device has acted upon the information
                             //(bulb brightness adjusted)
unsigned char error_led = 0; //indicate when a packet is dropped due to some error

// initialize high to avoid post-programming/turning noise
static unsigned char button_count = 100; // for debounce. I'm using polling, not interrupts, due this: http://www.ganssle.com/debouncing.htm
static __bit button_change; // so holding the button doesn't break things



// shadow register for PortA, so as to not suffer from read/modify/write errors
volatile union
{
    unsigned char byte;
    struct
    {
        unsigned RA0:1;
        unsigned RA1:1;
        unsigned RA2:1;
        unsigned RA3:1;
        unsigned RA4:1;
        unsigned RA5:1;
    } bits;
    struct
    {
        unsigned DAT_LED:1;
        unsigned CLK_LED:1;
        unsigned IND_LED:1;
        unsigned _:3;
    };
} PORTA_SH;

union reading // 16 BIT representation, to save memory
{
    // arranged this way for memory locations to match
    struct
```

```
    {
        unsigned char low;
        unsigned char high;
    };

    unsigned short int byte; //reading1_array[1] # reading1_array[0]
} mem;

// DCC stuff
// to calibrate receiver (microseconds)
#define ONE_DURATION_MIN 104
#define ONE_DURATION_MAX 132

#define ZERO_DURATION_MIN 180
#define ZERO_DURATION_MAX 20000

#define REQUIRED_PREAMBLE 14
#define PACKET_LENGTH 44

volatile __bit packet_found;
volatile __bit error;
//buffers for DCC packets. There are 2: one for current transmission, one for the next
#define BUFFER_LENGTH 6
unsigned char buffer[BUFFER_LENGTH];
unsigned char buffer_index=0;
// unsigned char buffer1[BUFFER_LENGTH];
unsigned char bit_buffer[16]; // store bits before they are combined into a byte, for speed
unsigned char bit_index = 0;
__bit byte; // indicates whether we are in the first 8bits of the bit buffer, or last 8 bits.  This is so the check doesn't have to happen in the interrupt
__bit byte_ready;
unsigned char MASKS[8]; // this contains the bit masks, rather than shifting each cycle. This is because there is no hardware support for multiple shifts,
//                         and it takes precious cycles.
unsigned short int time=0; // to record the time between EDGE (INTF) interrupts, to distinguish bits

//button
unsigned char button_state = 0; // default unpressed

void __interrupt() ISR()
{

    YEL_LED = ON;
    static unsigned char preamble_count = 0;
    // static unsigned char *current_buffer = bit_buffer0;
    static unsigned char current_bit = 0;

    static unsigned short int last_time=0; // time of last interrupt

    if (EDGE_INTERRUPT_FLAG) // if an external edge interrupt is triggered
    {
        //represent time as 16 bit
        mem.low = TIMER1_L;
        mem.high = TIMER1_H;

        EDGE_INTERRUPT_FLAG = CLEAR;
```

```
        time = mem.byte - last_time;
        last_time = mem.byte;


        if (time < ONE_DURATION_MIN) // glitch, so search for new preamble
        {
            packet_found = FALSE;
            preamble_count = 0;
            buffer_index = 0;
        }
        else if (time < ONE_DURATION_MAX) // its a one
        {
            preamble_count++;
            current_bit = 1;
        }
        else if (time < ZERO_DURATION_MIN) // a one combined with a zero, we are triggering on the incorrect edge
        {

            EDGE_DIRECTION = ~EDGE_DIRECTION; // change the edge trigger, since that solves this issue
            error = TRUE;
            packet_found = FALSE;
            preamble_count = 0;
            buffer_index = 0;

        }
        else if (time < 400) // its a zero! (temp shorter time due to bugs...)
        {
            if (preamble_count >= REQUIRED_PREAMBLE) //  a valid packet beginning
            {
                packet_found = TRUE;
                buffer_index = 0;
                bit_index = 0;
            }
            preamble_count = 0;
            current_bit = 0;
        }
        else // massive glitch, or the tranmission has fucked up. For something like this to occur client side, multiple bits have to be missed
            //and it can be cleaned up in the encoder anyway
        {
            preamble_count = 0;
            packet_found = FALSE;
            buffer_index = 0;
        }


        bit_buffer[bit_index] = current_bit;
        bit_index++;
        bit_index = bit_index%16;
    }

    YEL_LED = OFF;
}

void main()
{
    unsigned char address, data, checksum;
```

```
OSCCON = 0b01110001; // 8MHz clock (by default, can toggle to 4MHz)

// set pins
DAT_LED_TYPE = DIGITAL;
DAT_LED_PIN = OUTPUT;
CLK_LED_TYPE = DIGITAL;
CLK_LED_PIN = OUTPUT;
NXT_LED_PIN = OUTPUT;
NXT_LED_TYPE = DIGITAL;
YEL_LED_PIN = OUTPUT;
YEL_LED_TYPE = DIGITAL;

BUTTON_PIN = INPUT;

PWM_PIN = OUTPUT;


//setup PWM and timer 2 for bulb
TIMER2_CONTROL = (ON<<TIMER2_ON) | (PRESCALE_16<<TIMER_CLOCK_PRESCALE); // Timer 2 register
PWM_PERIOD = 255;
PWM_DUTYCYCLE_MSB = 30;
ECCP_CONTROL = (SINGLE_OUTPUT<<PWM_MODE) | (ACTIVE_HIGH_ACTIVE_HIGH<<PWM_OUTPUT); //PWM register set


//fill bit masks. This is precomputed for performance
for (char i=0; i<8; i++)
{
    MASKS[i] = 1<<i;
}

//enable INT (edge triggers) for detecting the bitstream
DATA_TYPE = DIGITAL;
DATA_PIN = INPUT;
EDGE_DIRECTION = FALLING; // Trigger on rising edges
EDGE_INTERRUPT = ON; //enable the interrupt


//set up timer1 for timing the bits
TIMER1_H = 0x00; TIMER1_L = 0x00;
T1CKPS1 = 0; T1CKPS0 = 1; //TIMER1 prescaler
TIMER1 = ON; // turn the timer1 on, with a 1:2 prescale so with an 8MHz clock, it counts microseconds
// TIMER1_INTERRUPT = ON;
// PERIPHAL_INTERRUPT = ON;


//turn on interrupts
GLOBAL_INTERRUPTS = ON;


while (1)
{
    if (packet_found) // a packet will begin being written into bit_buffer (system found a preamble)
    {
        buffer[buffer_index] = 0; // clear the current byte
        if (byte == 0) // first byte of buffer is being recorded
```

```
    {
        while (1) // loop for this, for speed, by holding the instruction pointer hostage
        {
            if (bit_index > 7) // the first byte is full, we can move it
            {
                for (int i=0; i<8; i++)
                {
                    if (bit_buffer[i]) // if its a one, make it so, otherwise skip this bit (defaults to zero)
                    {
                        buffer[buffer_index] |= (MASKS[7-i]); // use the mask to set to 1
                    }
                }
                break; // leave loop
            }
        }
        buffer_index++;
        byte = 1;
    }
    else // second byte of the buffer is being recorded
    {
        while (1) // loop for this, for speed
        {
            if (bit_index < 7) // the second byte is full, we can move it
            {
                for (int i=8; i<16; i++)
                {
                    if (bit_buffer[i]) // if its a one, make it so, otherwise skip
                    {
                        buffer[buffer_index] |= (MASKS[7-(i-8)]);
                    }
                }
                break;
            }
        }
        buffer_index++;
        byte = 0;
    }
}

if (buffer_index == 4) // buffer full, begin decode of packet
{
    packet_led = LED_TIME; // indicate that the packet has been found

    // buffer is rearanged to remove the structure imposed by DCC and the receiver optimisation
    address = (buffer[0]<<1) | (buffer[1]>>7);
    data = (buffer[1]<<2) | (buffer[2]>>6);
    checksum = (buffer[2]<<3) | (buffer[3]>>5);

    if ((address ^ data) == checksum) // as per DCC spec: is this a valid packet?
    {
        if (address == ADDRESS) //if this packet was addressed to this receiver
        {
            update_led = LED_TIME; // indicate that a packet was aimed at this device

            if (button_state == 1) // if the button has been pressed
            {
```

```
                PWM_DUTYCYCLE_MSB = 255; // put the output full on
            }
            else // if the user has not requested dimming, then allow the transmitter to control it
            {
                PWM_DUTYCYCLE_MSB = data;
            }
        }
    }
    else // the packet was corrupted
    {
        error_led = LED_TIME; // indicate that the packet was corrupted
    }
    buffer_index = 0;
    packet_found = FALSE;
}
else //  no packet to decode, update other parts of system (this doesn't use a timer so the interrupt can be fast.)
{
    // this doesn't impact anything, since the timing isn't crucial, the LEDs only indicate things to the user

    //LED TIMING
    if (packet_led) // if we need to display whether a packet has been received
    {
        packet_led--;
        PORTA_SH.DAT_LED = ON;
    }
    else
    {
        PORTA_SH.DAT_LED = OFF;
    }

    if (update_led) // if we need to display whether a packet has been received
    {
        update_led--;
        PORTA_SH.CLK_LED = ON;
    }
    else
    {
        PORTA_SH.CLK_LED = OFF;
    }

    if (error_led) // if we need to display whether a packet has been received
    {
        error_led--;
        NXT_LED = ON;
    }
    else
    {
        NXT_LED = OFF;
    }

    // process button input. This is safer as a polling exercise anyway
    // debounce
    if (BUTTON && button_count==0)
    {
        button_count = 5;
    }
```

```
            else if (button_count > 1)
            {
                button_count--;
            }
            else if (button_count == 1)
            {
                if (BUTTON)
                {
                    if (button_change) // if the button has actually been pressed, change state
                    {
                        button_change = 0;
                        button_count = 100;
                        button_state++;

                        button_state = button_state%3; //three states

                        if (button_state == 0) // this state also forces a clockspeed update
                        {
                            OSCCON = 0b01110001; // 8MHz clock
                            T1CKPS1 = 0; T1CKPS0 = 1; //TIMER1 prescaler (1:2)
                        }
                        else if (button_state == 2) // clockspeed update
                        {
                            OSCCON = 0b01100001; // 4MHz clock
                            T1CKPS1 = 0; T1CKPS0 = 0; //TIMER1 prescaler (1:2)
                        }
                    }
                }
                else
                {
                    button_change = 1;
                    button_count = 0;
                }
            }
        }

        PORTA = PORTA_SH.byte; // update LEDs
    }
}
```