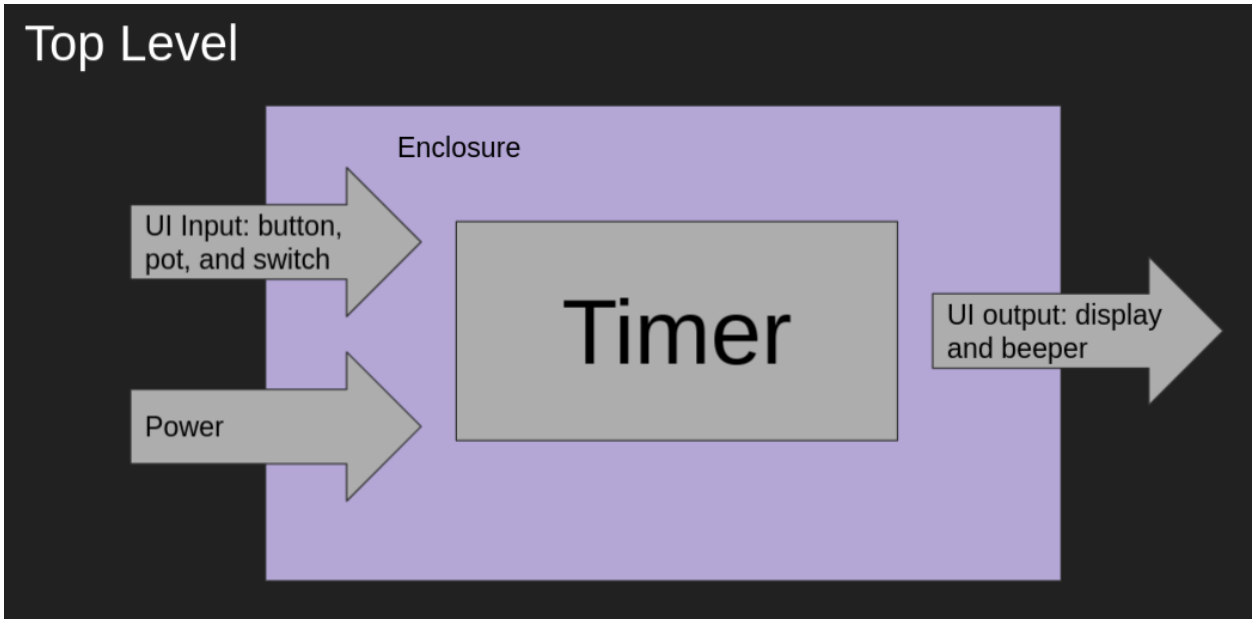
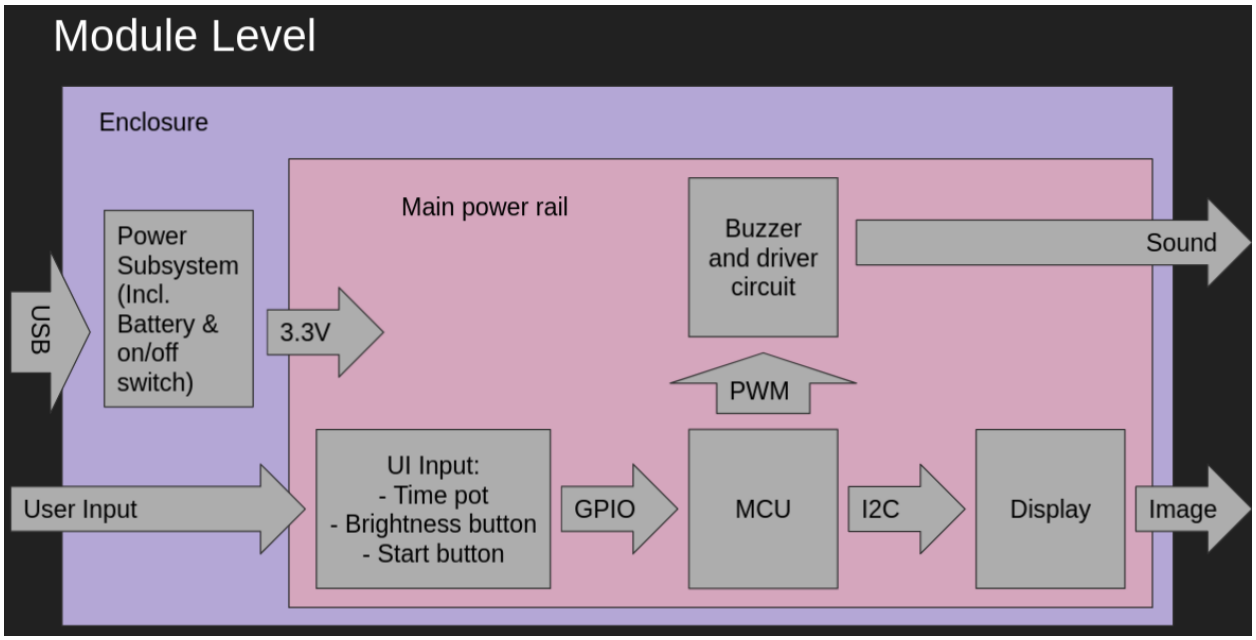
Timer(3) Technical Documentation

Anton Liakhovitch

Block Diagrams

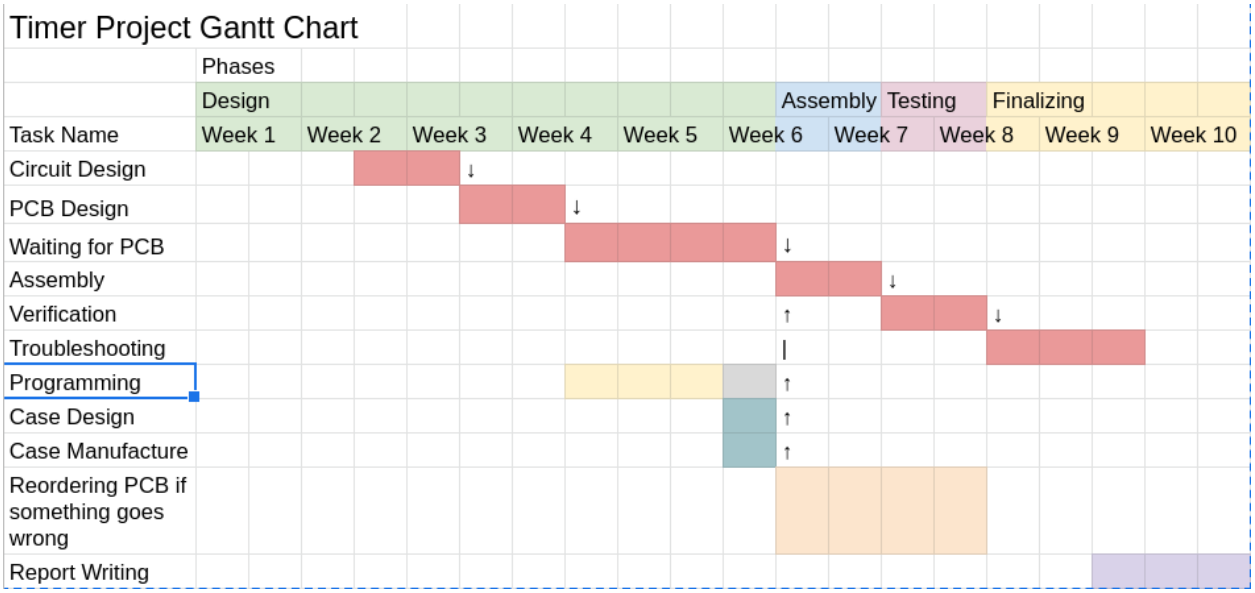




Interface Specifications

From	To	Description
External USB power	Power subsystem	5V DC, min 400mA (current draw may be revised)
Power subsystem	Power rail	3.3V DC, min 200mA (current draw may be revised)
Pot	MCU	3.3V Voltage divider on analog pin. Must discern at least 128 knob positions.
Start Button	MCU	GPIO input with internal pullup to 3.3V
Brightness Button	MCU	GPIO input with internal pullup to 3.3V
MCU	OLED display	I2C, 3v3 logic. Must display at least five alphanumeric characters
MCU	Buzzer circuit	3.3V PWM on two pins with inverse polarity, 400-1000 Hz, max 50mA

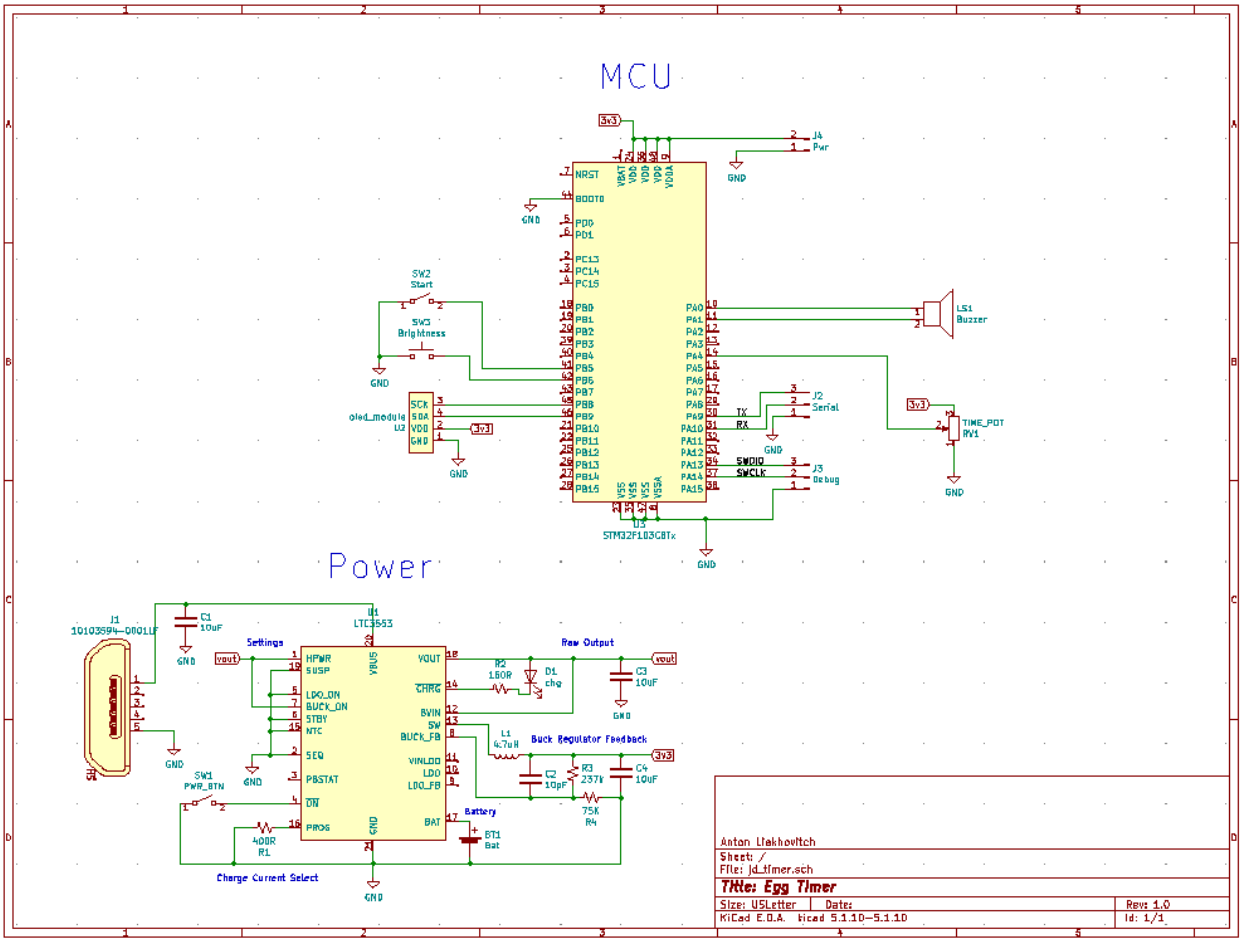
Gannt Chart


Timer Project Gantt Chart
Phases
Design
Assembly
Testing
Finalizing
Task Name
Week 1
Week 2
Week 3
Week 4
Week 5
Week 6
Week 7
Week 8
Week 9
Week 10
Circuit Design
PCB Design
Waiting for PCB
Assembly
Verification
Troubleshooting
Programming
Case Design
Case Manufacture
Reordering PCB if something goes wrong
Report Writing

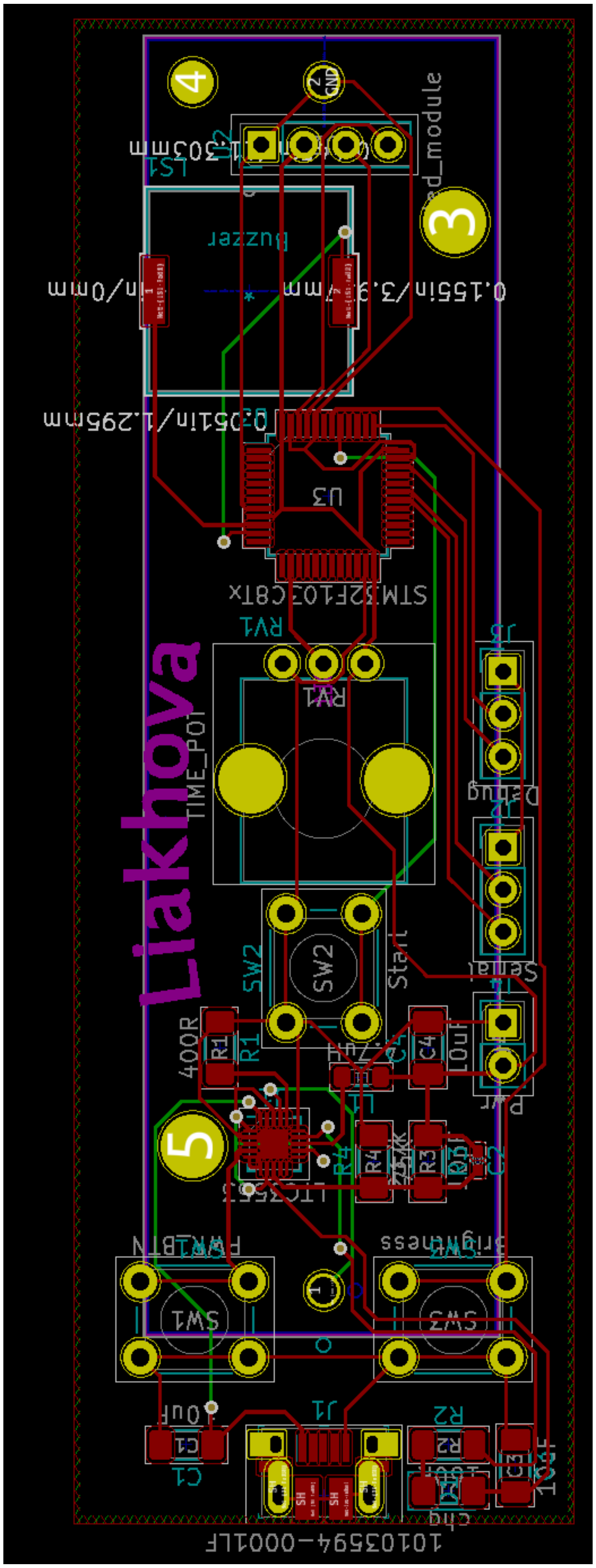
BOM

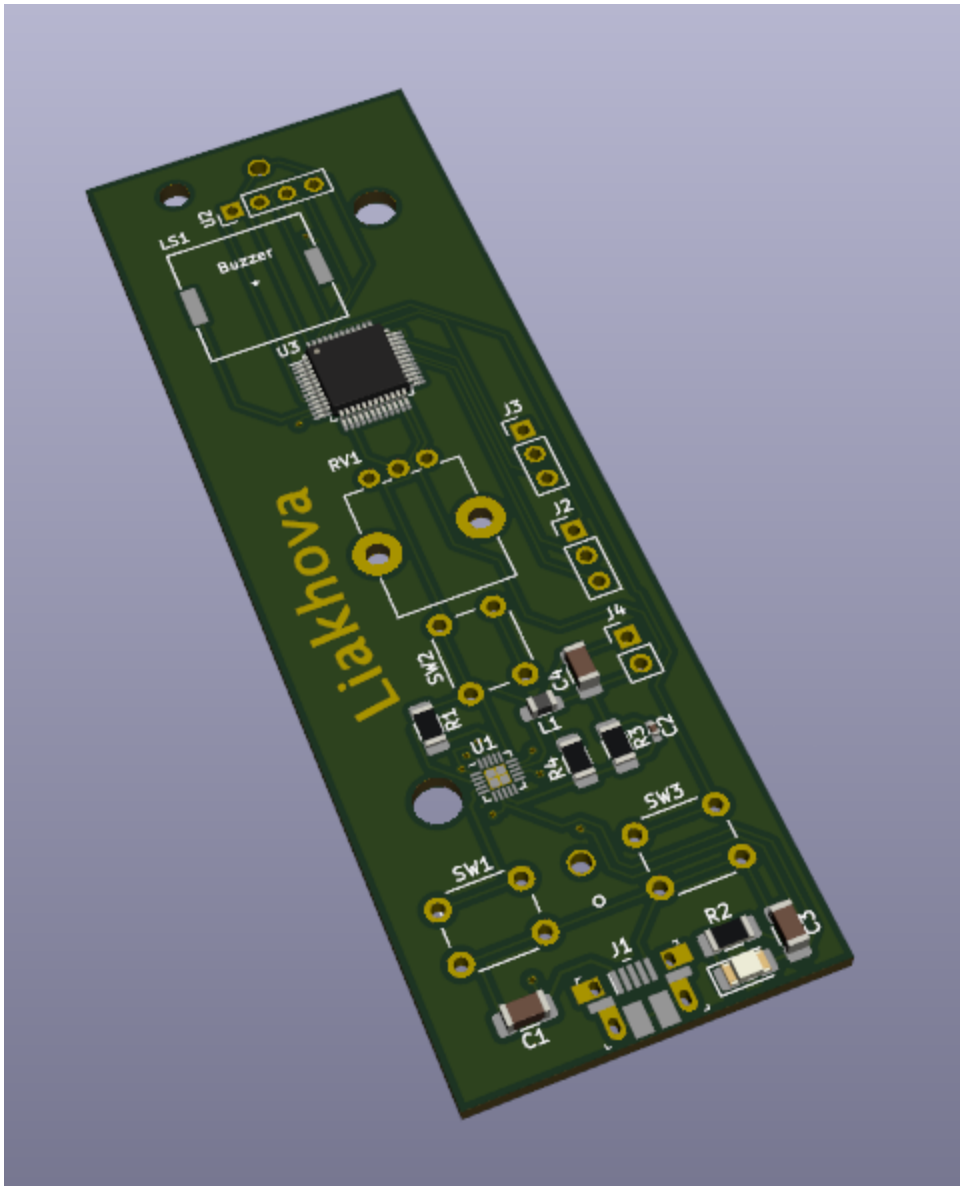
Manufacturer Part Number	Manufacturer	Unit Price	Description
STM32F103C8T6	STMicroelectronics	5.95000	IC MCU 32BIT 64KB FLASH 48LQFP
LTC3553EUD#PBF	Analog Devices Inc.	6.01000	IC USB POWER MANAGER 20UTQFN
10103594-0001LF	Amphenol ICC (FCI)	0.80000	CONN RCPT USB2.0 MICRO B SMD R/A
PRT-12895	SparkFun Electronics	5.95000	BATTERY LITHIUM 3.7V 2.6AH
RT1206BRD07400RL	Yageo	0.62000	RES SMD 400 OHM 0.1% 1/4W 1206
IN-S126ATR	Inolux	0.36000	LED RED CLEAR 1206 SMD
RT1206BRD07160RL	Yageo	0.62000	RES SMD 160 OHM 0.1% 1/4W 1206
CV201210-4R7K	Bourns Inc.	0.10000	FIXED IND 4.7UH 30MA 1 OHM SMD
1043	Keystone Electronics	3.14000	BATTERY HOLDER 18650 PC PIN
PKLCS1212E2000-R1	Murata Electronics	1.37000	AUDIO PIEZO TRANSDUCER 12.5V SMD
PTV09A-4020U-B103	Bourns Inc.	0.83000	POT 10K OHM 1/20W CARBON LINEAR
RT1206DRD07237KL	Yageo	0.29000	RES SMD 237K OHM 0.5% 1/4W 1206
RT1206DRE0775KL	Yageo	0.23000	RES SMD 75K OHM 0.5% 1/4W 1206
CL31B106KQHNFNE	Samsung Electro-Mechanics	0.12900	CAP CER 10UF 6.3V X7R 1206
CC0402KRNPO7BN100	Yageo	0.10000	CAP CER 10PF 16V NPO 0402

Schematic

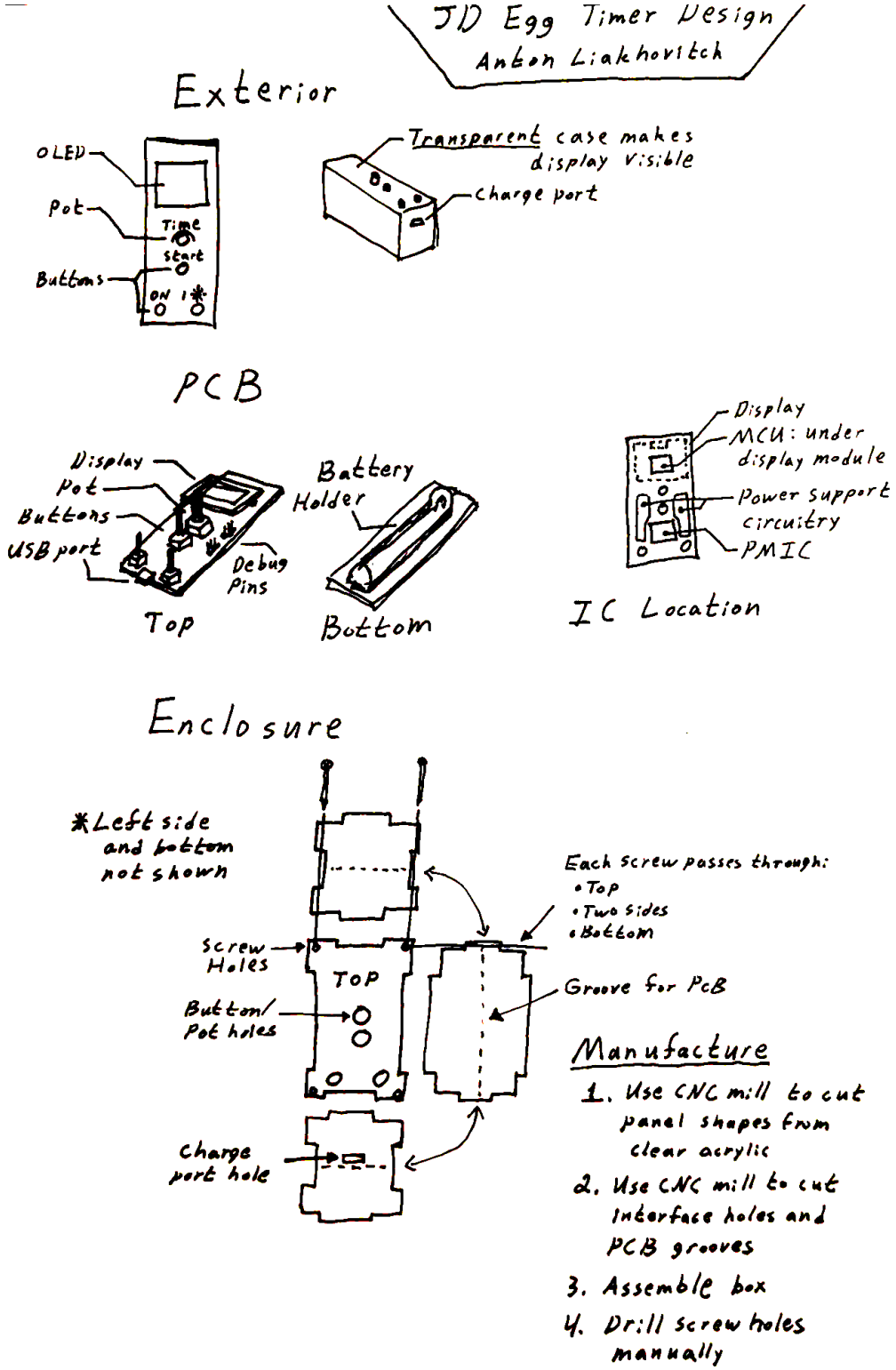

MCU
Power
STM32F103C8Tx
LTC3553
Anton Ilakhovitch
Sheet: /
File: jd_timer.sch
Title: Egg Timer
Size: USLetter
Date:
Rev: 1.0
KiCad E.D.A. kicad 5.1.10-5.1.10
Id: 1/1

PCB

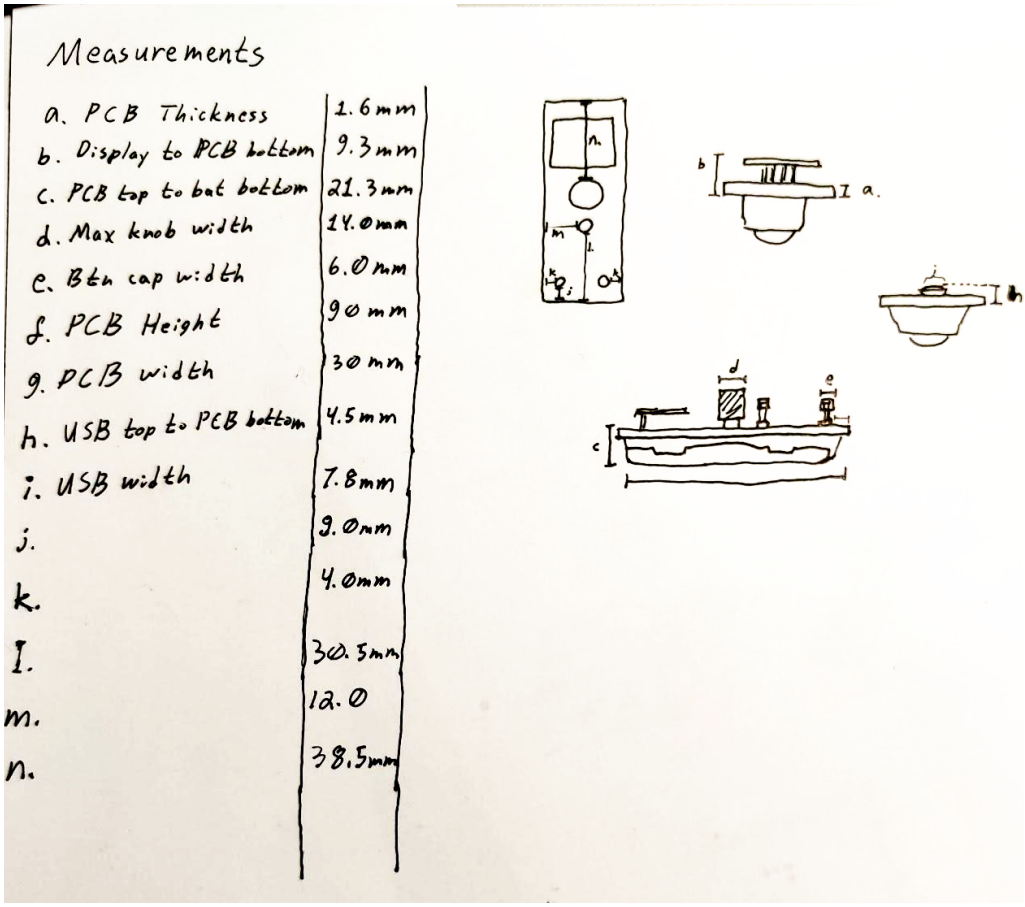

Liakhova
TIME_POT
RV1
SW2
Start
Serial
Debug
J5
R1
400R
R4
R3
C4
C2
J1
R2
C1
C3
3
4
5

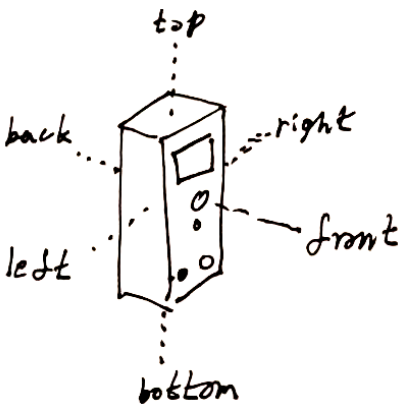

LS1
Buzzer
U3
RV1
J3
J2
J4
SW2
R1
C3
L1
U1
R4
R3
C2
SW3
SW1
R2
C1
J1
Liakhova

Exterior Sketch



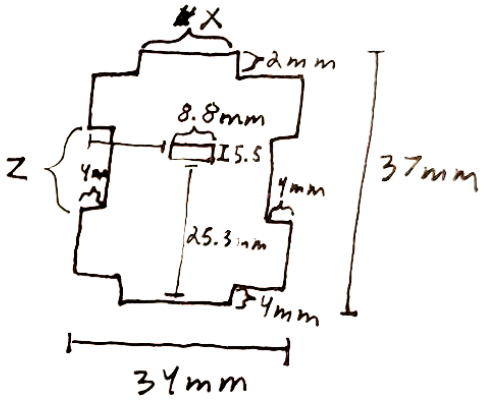
Design Plans





Top/bottom

thickness: 4mm



Height of enclosed space:

90mm (f.)

Width of enclosed space:

26mm (g. - 2m - 2m)

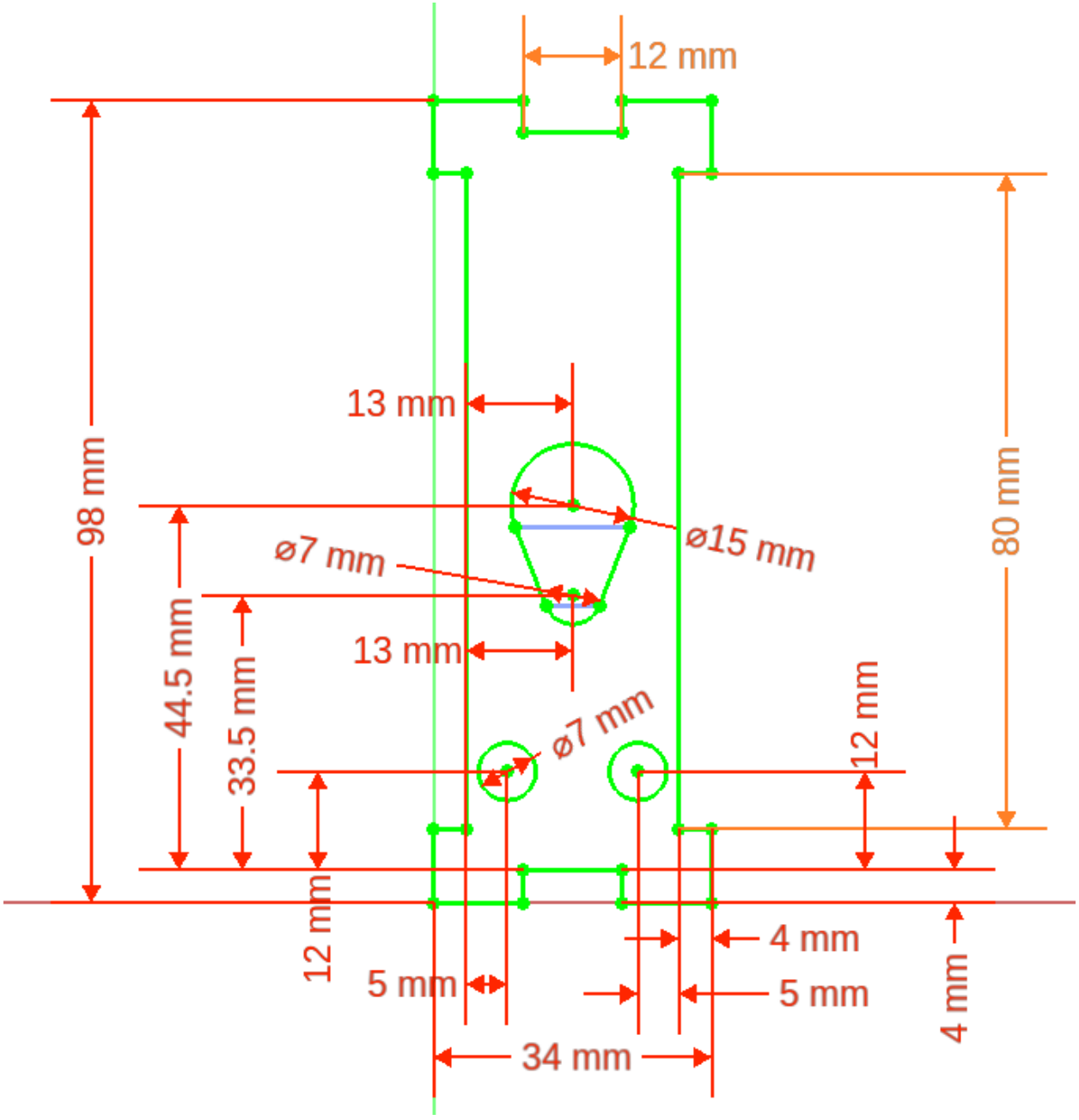
Depth of enclosed space:

31mm

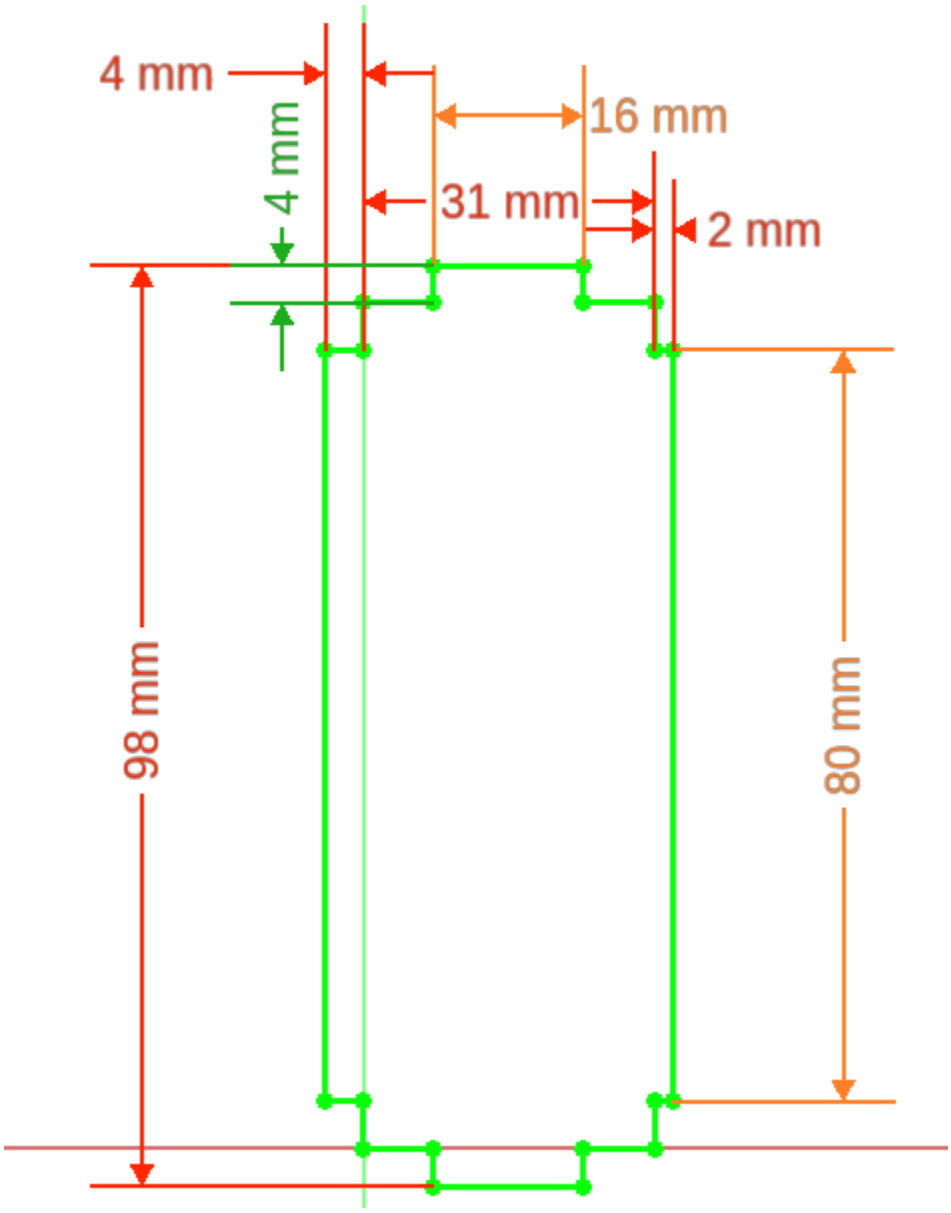
Mechanical Drawings

Front Panel (2mm Thick)

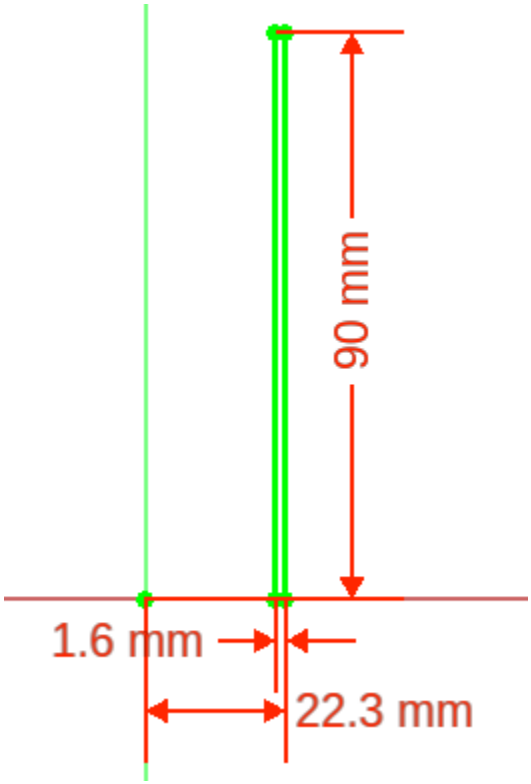
Note: bottom panel is the same, except it lacks holes and is 4mm thick



Right and Left Panel (4mm Thick)

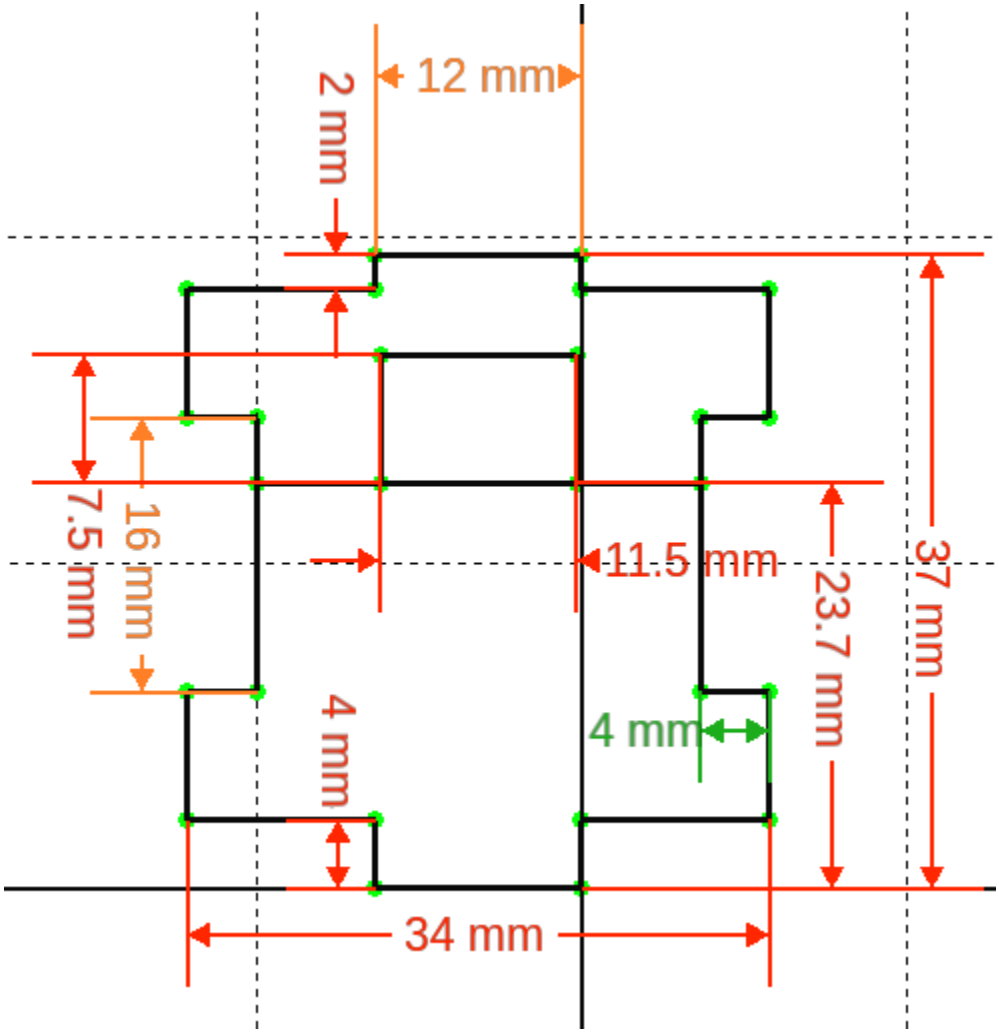


And the groove for the PCB:

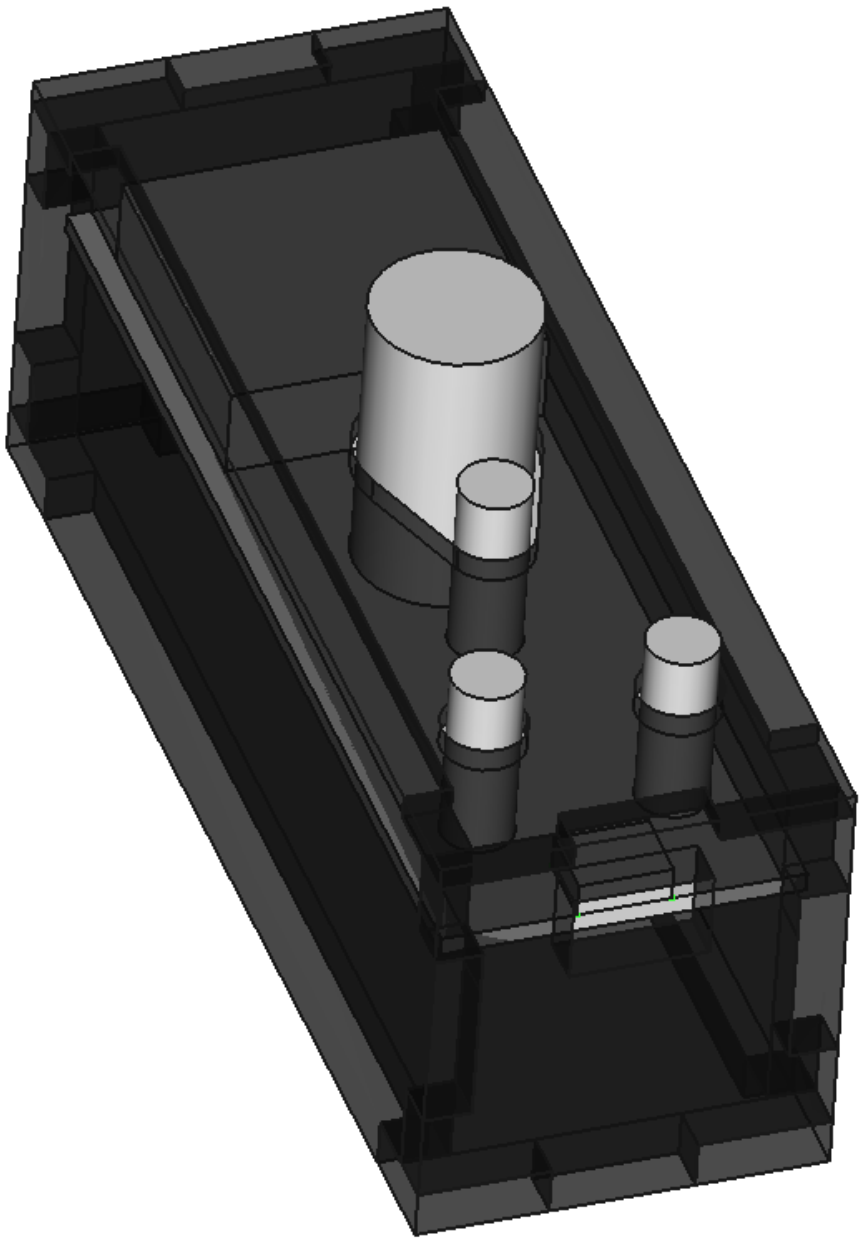


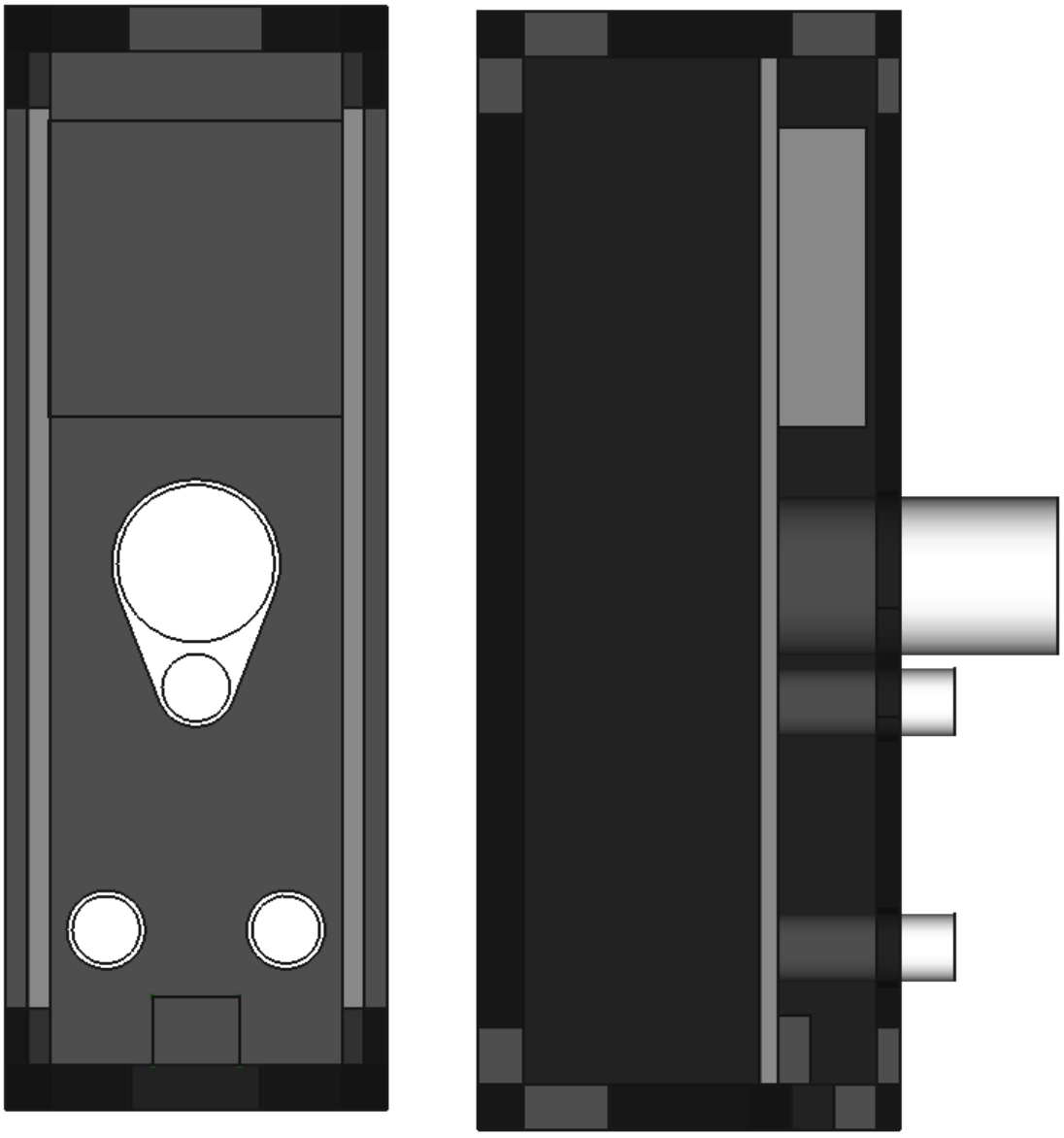
Bottom (4mm Thick)

Note: The top is exactly the same, but without the hole.



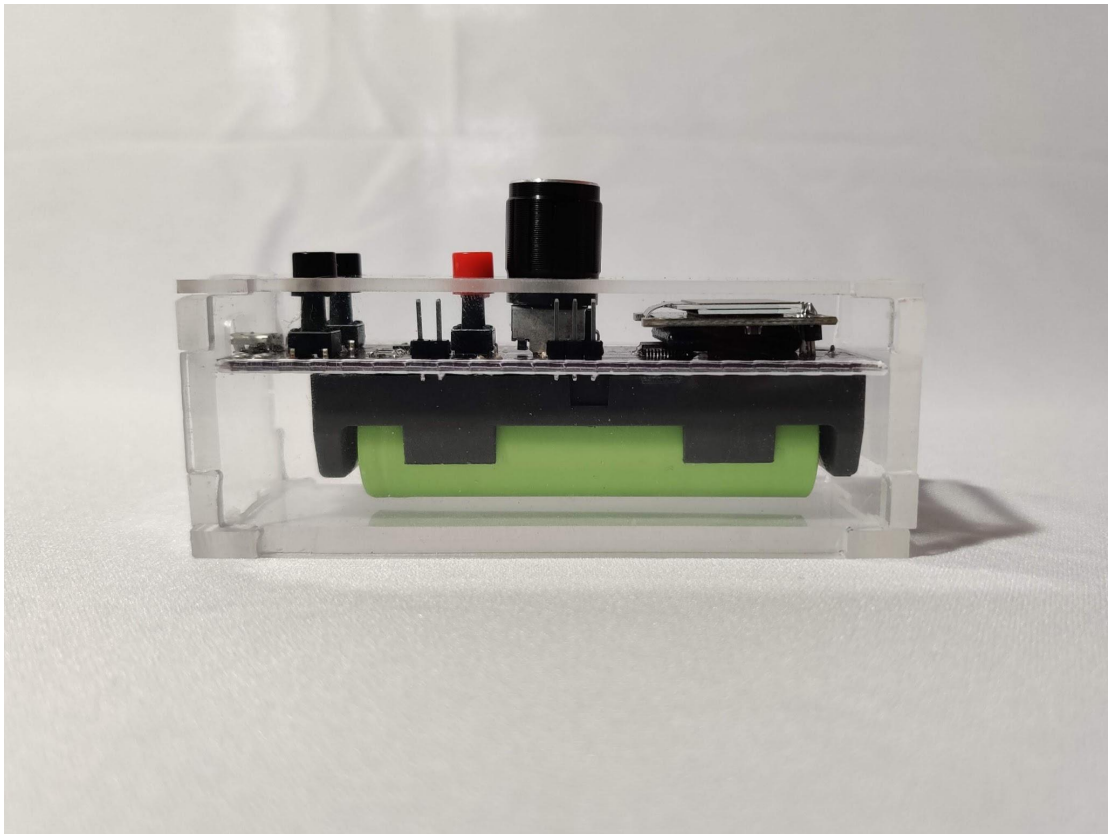
Assembly

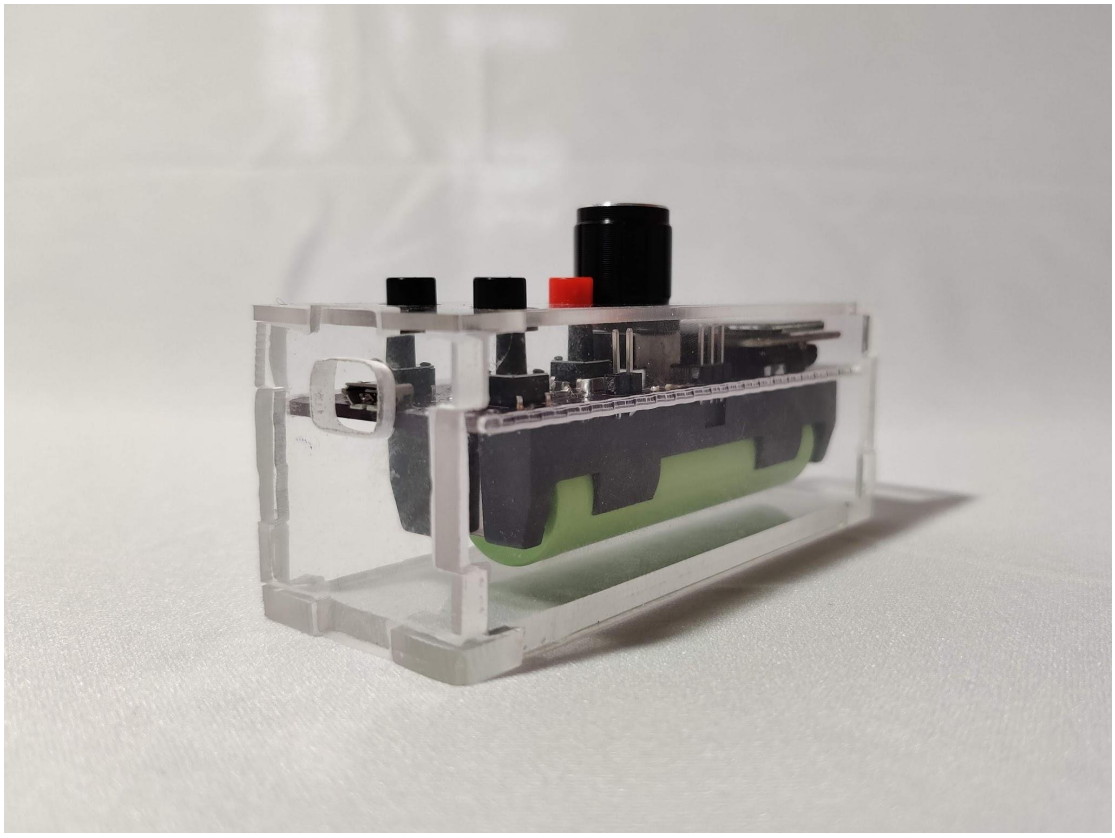


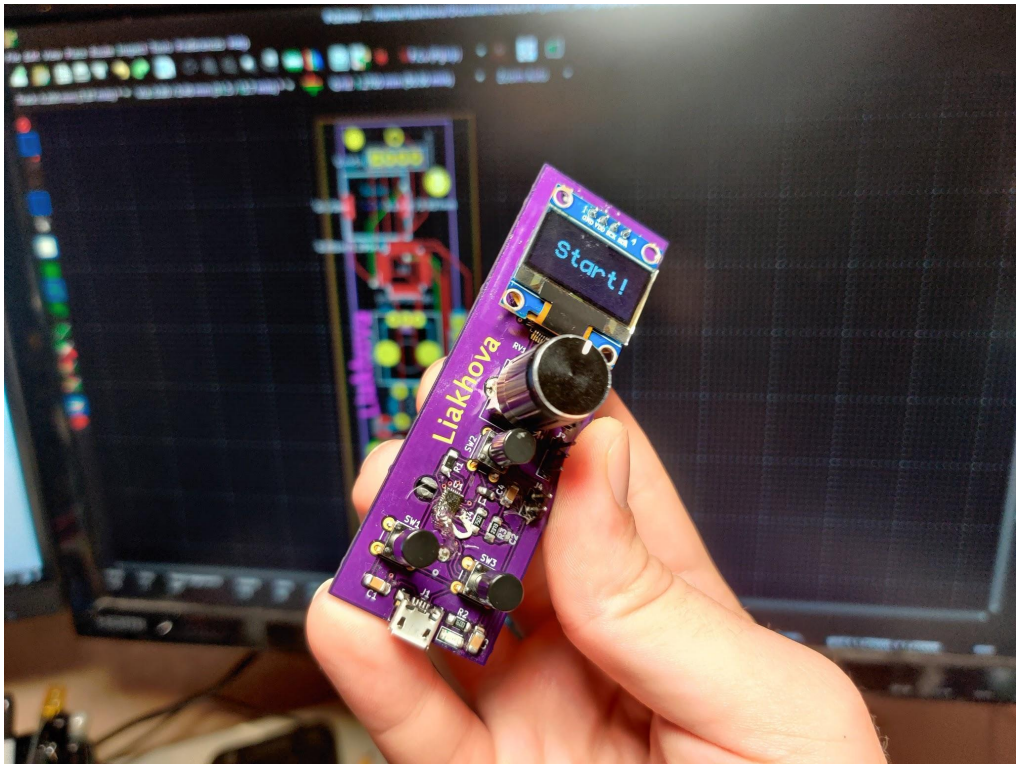


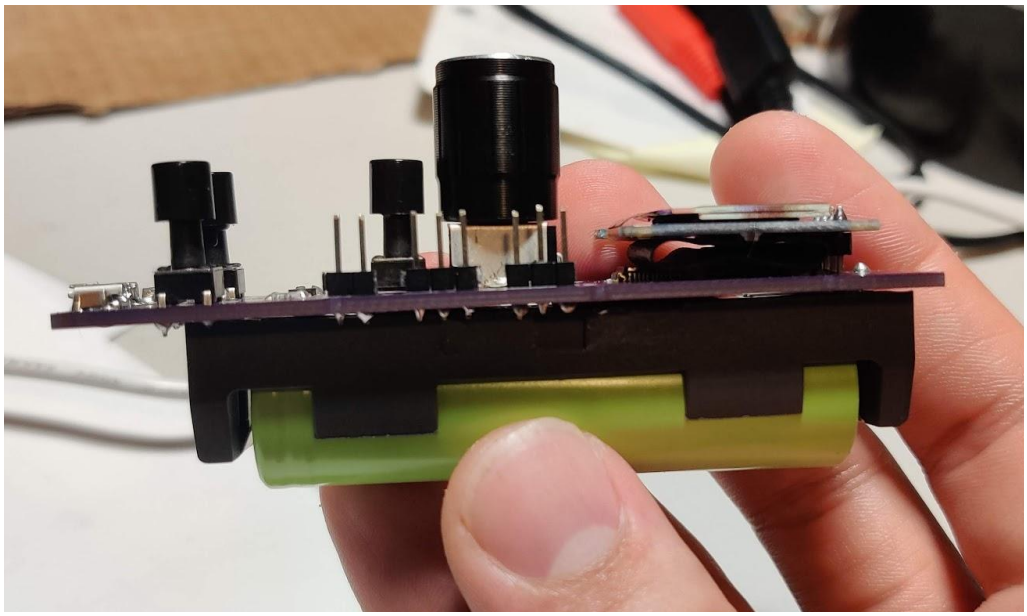
Photographs

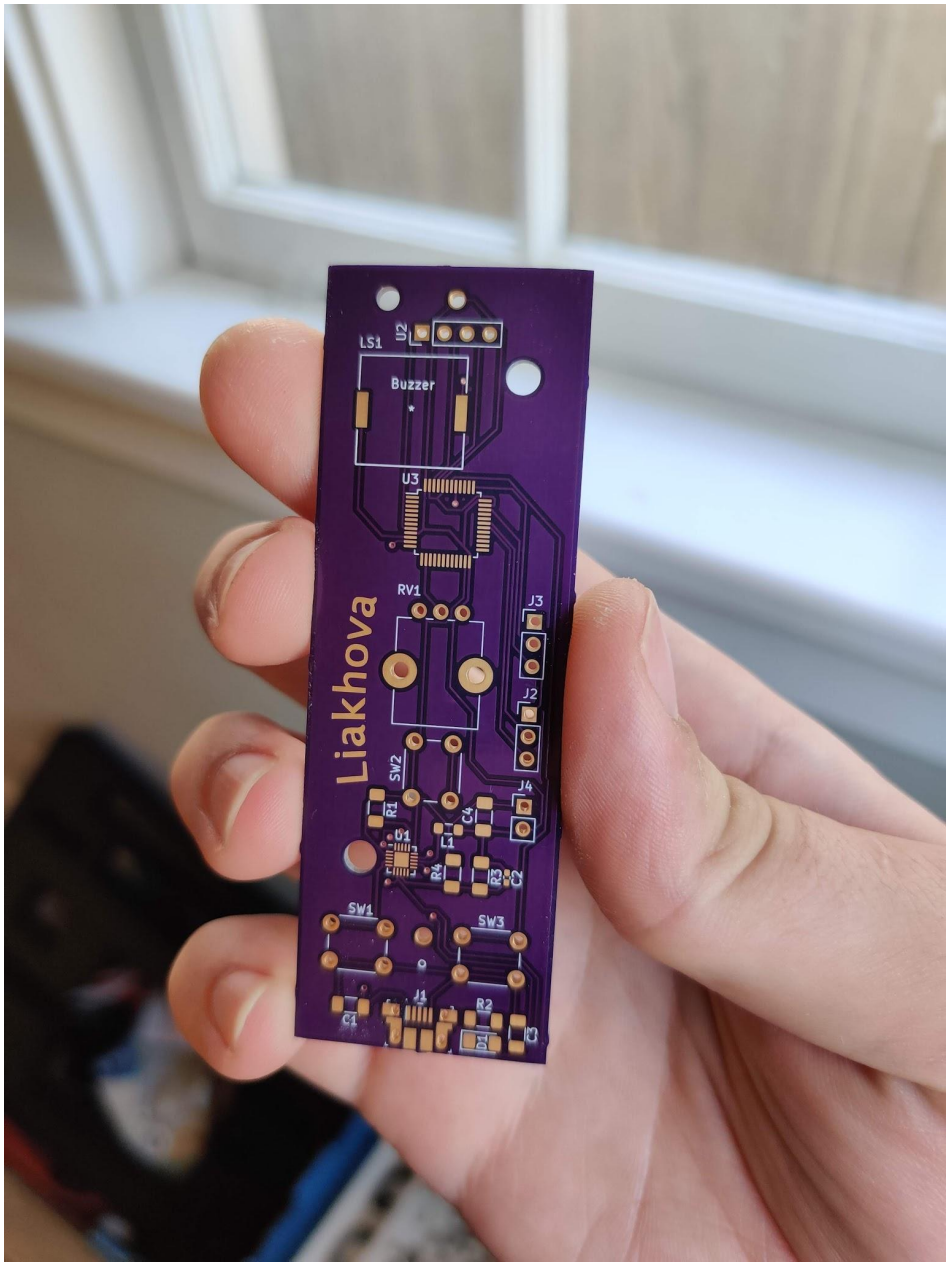

U2
GND VCC SCL SDA
Liakhova
Egg Timer!
RV1
SW2
SW1






Start!
Liakhova




LS1
U2
Buzzer
U3
RV1
J3
J2
J4
SW2
R1
C4
D1
L1
R4
R3
C3
SW1
SW3
J1
R2
C1
D2
C2
Liakhova

Code

https://github.com/liakhovitch/junior_design
Not included:

- Code files containing bitmap constants
- Several other configuration files required to build the project

Cargo.toml

```
[package]
name = "jd_timer"
version = "0.1.0"
authors = ["fountainman <liakhova@oregonstate.edu>"]
edition = "2018"

[dependencies]
cortex-m = "0.7.2"
cortex-m-rt = "0.6.13"
#cortex-m-rtic = { git = "https://github.com/rtic-rs/cortex-m-rtic.git"}
cortex-m-rtic = "0.6.0-alpha.2"
panic-halt = "0.2.0"
usb-device = {version = "0.2.8"}
stm32f1xx-hal = {version="0.7.0", features = ["rt", "stm32f103"]}
embedded-hal = "0.2.5"
ssd1306 = "0.5.2"
embedded-graphics = "0.6.2"
profont = "0.4.0"
heapless = "0.5.6"
dwt-systick-monotonic = "0.1.0-alpha.1"
embedded-time = "0.11.0"

# this lets you use `cargo fix`!
[[bin]]
name = "jd_timer"
test = false
bench = false

[profile.release]
codegen-units = 1 # better optimizations
debug = true # symbols are nice and they don't increase the size on Flash
lto = false # better optimizations
opt-level = 2 # better optimizations apparently

[profile.dev]
codegen-units = 1 # better optimizations
```

```
debug = true # symbols are nice and they don't increase the size on Flash
lto = true # better optimizations
opt-level = "z" # better optimizations apparently
```

Main.rs

```rust
#![no_std]
#![no_main]
#![allow(unused_imports)]

use panic_halt as _;

// Declare modules for the other parts of the project.
// Rust automatically matches these by filename and creates the proper scopes.
mod buttons;
mod pot;
mod ui;
mod rtc;
mod beep;
mod types;
mod config;
mod states;
mod rtc_util;
mod charge;
#[path = "./bitmaps/bigbolt.rs"]
mod bigbolt;
#[path = "./bitmaps/smallbolt.rs"]
mod smallbolt;
#[path = "./bitmaps/logo.rs"]
mod logo;

#[rtic::app(device = stm32f1xx_hal::pac,
peripherals = true, dispatchers = [DMA1_CHANNEL1,DMA1_CHANNEL2,DMA1_CHANNEL3])]
// RTIC application
mod app {

   // Include other parts of the project so the task definitions at the bottom can
access them
   use crate::buttons::*;
   use crate::pot::*;
   use crate::ui::*;
   use crate::types::*;
   use crate::beep::*;
   use crate::rtc::*;
   use crate::states::*;
   use crate::charge::*;
   // My modified version of the stm32f1xx_hal RTC library
   use crate::rtc_util;

   use crate::config::{SLEEP_TIME};
```

```rust
use stm32f1xx_hal::{
    adc::{Adc, SampleTime},
    prelude::*,
    serial,
    gpio::{
        gpiob::{PB8, PB9, PB6, PB5},
        gpioa::{PA0, PA1, PA4, PA9, PA10},
        {Output, PushPull},
        {Input, PullUp},
        {Alternate, OpenDrain},
        Edge::*,
        ExtiPin,
        Analog,
    },
    timer::{Event, Timer, Tim2NoRemap},
    pac::{I2C1, USART1, ADC1, TIM2},
    i2c::{BlockingI2c, DutyCycle, Mode},
    pwm::C1,
    rtc::Rtc,
};

use dwt_systick_monotonic::DwtSystick;

use cortex_m::asm::delay;

//use core::fmt::Write;
use ssd1306::{
    prelude::*,
    Builder,
    I2CDIBuilder,
};

use embedded_graphics::{
    fonts::Text,
    pixelcolor::BinaryColor,
    prelude::*,
    style::TextStyle,
};

use profont::ProFont24Point;

// Import peripheral control methods from general HAL definition
use embedded_hal::digital::v2::{OutputPin, InputPin};

use core::fmt::Write;
use core::future::Future;
use core::ptr::*;
```

```
    use rtic::rtic_monotonic::{Clock, Milliseconds, Nanoseconds};
    use embedded_time::duration::*;
    use core::convert::TryFrom;
    use rtic::rtic_monotonic::embedded_time::fixed_point::FixedPoint;
    use rtic::Monotonic;

    // Declare type for monotonic timer used by RTIC for task scheduling
    #[monotonic(binds = SysTick, default = true)]
    type MyMono = DwtSystick<8_000_000>; // 8 MHz

    // Resources shared by all handlers.
    // All resourced not initialized here by macros are initialized in [init] and
returned to RTIC
    //   in the init::LateResources object.
    #[resources]
    struct Resources {
        display: GraphicsMode<I2CInterface<BlockingI2c<I2C1,
(PB8<Alternate<OpenDrain>>,PB9<Alternate<OpenDrain>>) >>, DisplaySize128x64>,
        button_start: PB5<Input<PullUp>>,
        button_brightness: PB6<Input<PullUp>>,
        chg_pin: PA10<Input<PullUp>>,
        EXTI: stm32f1xx_hal::pac::EXTI,
        clocks: stm32f1xx_hal::rcc::Clocks,
        adc1: Adc<ADC1>,
        pot: PA4<Analog>,
        pot_pos: u16,
        sleep_pin: PA9<Output<PushPull>>,
        buzzer: stm32f1xx_hal::pwm::PwmChannel<TIM2, C1>,
        rtc: stm32f1xx_hal::pac::RTC,
        #[init(0)]
        brightness_state: u8,
        #[init(false)]
        pot_dir: bool,
        #[init(SysState::Setup)]
        sys_state: SysState,
        #[init(0)]
        max_time: u16,
        #[init(0)]
        time_remaining: u16,
        #[init(0)]
        disp_call_cnt: u8,
    }

    // Init function
    #[init ()]
    // CX object contains our PAC (peripheral access crate)
    // Init function initializes resources and returns them to RTIC via the
```

```
LateResources object.
   fn init(cx: init::Context) -> (init::LateResources,init::Monotonics){

// -------------
// General Setup
// -------------

       // Enable cycle counter
       let mut core = cx.core;
       core.DWT.enable_cycle_counter();
       // Take ownership of clock register
       let mut rcc = cx.device.RCC.constrain();
       // Take ownership of flash peripheral
       let mut flash = cx.device.FLASH.constrain();
       // Take ownership of AFIO register
       let mut afio = cx.device.AFIO.constrain(&mut rcc.apb2);

       // Configure clocks and make clock object from clock register
       let clocks = rcc
           .cfgr
           //.use_hse(8.mhz())
           //.sysclk(8.mhz())
           //.pclk1(4.mhz())
           .freeze(&mut flash.acr);

       // Split GPIO ports into smaller pin objects
       let mut gpioa = cx.device.GPIOA.split(&mut rcc.apb2);
       let mut gpiob = cx.device.GPIOB.split(&mut rcc.apb2);

// ---------------------
// Init scheduling timer
// ---------------------
       // This monotonic timer object is returned to the RTIC framework for use in
task scheduling
       let mut mono = DwtSystick::new(&mut core.DCB, core.DWT, core.SYST,
8_000_000);
       unsafe {
           mono.reset();
           mono.enable_timer();
       }

// --------
// Init RTC
// --------
       let mut rtc = cx.device.RTC;

       // We start by modifying some registers that the HAL already has control of,
```

```
        // behind the HAL's back, using unsafe code. We can do this with safe code
before
        // initializing the HAL, but we have no guarantee that HAL initialization
won't
        // overwrite our changes. For reference, this is what that might have looked
like:
        //cx.device.RCC.apb1enr.modify(|_,w| w.pwren().set_bit().bkpen().set_bit());
        //cx.device.RCC.csr.modify(|_,w| w.lsion().set_bit());
        unsafe {
            // Get address of PWR register block
            // Note: PWR access can actually be done safely, but we need the
volatile write to
            // ensure that the compiler doesn't put this after our attempts to write
to RCC_BDCR.
            let pwr_ptr: *mut u32 = stm32f1xx_hal::pac::PWR::ptr() as *mut u32;

            // Get address of RCC register block
            let rcc_ptr: *mut u32 = stm32f1xx_hal::pac::RCC::ptr() as *mut u32;

            // Offset to get apb1enr address
            // Offset is: 0x1C bytes (from datasheet) / 4 = 7 words
            let apb1enr_ptr: *mut u32 = rcc_ptr.offset(7);
            // Get current reg value
            let mut apb1enr_temp: u32 = read_volatile(apb1enr_ptr);
            // Set BKPEN: Enable backup domain access
            apb1enr_temp |= 1<<27;
            // Set PWREN: Enable backup domain access even harder
            apb1enr_temp |= 1<<28;
            // Write changes
            write_volatile(apb1enr_ptr, apb1enr_temp);

            // Offset to get RCC_CSR address
            // Offset is: 0x24 bytes (from datasheet) / 4 = 9 words
            let rcc_csr_ptr: *mut u32 = rcc_ptr.offset(9);
            // Get current reg value
            let mut rcc_csr_temp: u32 = read_volatile(rcc_csr_ptr);
            // Set LSION: Turn on low speed internal oscillator
            rcc_csr_temp |= 1<<0;
            // Write changes
            write_volatile(rcc_csr_ptr, rcc_csr_temp);

            // Offset to get pwr_cr address
            // Offset is: 0x00 bytes (from datasheet) / 4 = 0 words
            let pwr_cr_ptr: *mut u32 = pwr_ptr.offset(0);
            // Get current reg value
            let mut pwr_cr_temp: u32 = read_volatile(pwr_cr_ptr);
            // Set DBP: Disable backup domain write protection
```

```
            pwr_cr_temp |= 1<<8;
            // Write changes
            write_volatile(pwr_cr_ptr, pwr_cr_temp);

            // Offset to get RCC_BDCR address
            // Offset is: 0x20 bytes (from datasheet) / 4 = 8 words
            let rcc_bdcr_ptr: *mut u32 = rcc_ptr.offset(8);
            // Get current reg value
            let mut rcc_bdcr_temp: u32 = read_volatile(rcc_bdcr_ptr);
            // Set RTCEN: Enable RTC
            rcc_bdcr_temp |= 1<<15;
            // Set RTCSEL: Set RTC clock source to LSI (Low Speed Internal) clock
            rcc_bdcr_temp |= 0b10<<8;
            // Write changes
            write_volatile(rcc_bdcr_ptr, rcc_bdcr_temp);
        }

// -------------
// Calibrate RTC
// -------------

        // Procedure:
        //  * Set the RTC prescaler such that one tick takes about 0.25s
        //  * Use the system clock to measure how long that tick takes
        //  * Set the -correct- prescaler based on that data
        // The datasheet recommends a hardware-based method not available on this
particular chip,
        //   so I had to improvise.

        // We will spend 1/CAL_DIV seconds on calibration
        const CAL_DIV:u32 = 4;
        // Conversion factor from time measurement units to seconds
        const CONV_FACTOR:u32 = 1_000_000_000;
        // Initial guess as to the LSI (Low Speed Internal oscillator) frequency
        const LSI_GUESS:u32 = 40_000;

        // Set RTC prescaler. Code partly borrowed from the HAL.
        let prescaler = (LSI_GUESS / CAL_DIV) - 1;
        rtc_util::rtc_write(&mut rtc, |rtc| {
            rtc.prlh.write(|w| unsafe { w.bits(prescaler >> 16) });
            rtc.prll.write(|w| unsafe { w.bits(prescaler as u16 as u32) });
        });

        // Reset the RTC counter
        // This is done before the clock starts to compensate for the RTC read
operation done after
        // the clock starts. Both operations take around three RTC clock cycles,
```

```
which is a lot of
        // system cycles.
        rtc_util::set_time(&mut rtc, 0);

        // Measure initial time
        // Welcome to the wonderful world of embedded Rust!
        let time_start = *(Nanoseconds::<u32>::try_from(
            mono.try_now().unwrap().duration_since_epoch()
        ).unwrap().integer());

        // Blocking wait until RTC ticks
        while rtc_util::current_time(&mut rtc) == 0 {};

        // Measure final time
        let time_stop = *(Nanoseconds::<u32>::try_from(
            mono.try_now().unwrap().duration_since_epoch()
        ).unwrap().integer());

        // This is the amount of time it takes the LSI to tick 10_000 times
        let time_diff = time_stop - time_start;

        // Calculate the correct frequency
        // To do this without FP, we order operations such that numbers get BIG
before they get
        //   small again. Hence, conversion to u64 and then conversion back into
u32.
        let lsi_hz:u32 = (
            (((LSI_GUESS / CAL_DIV) as u64) * (CONV_FACTOR as u64))
            / (time_diff as u64)
        ) as u32;

        // Set RTC prescaler. Code partly borrowed from the HAL.
        let prescaler = (lsi_hz) - 1;
        rtc_util::rtc_write(&mut rtc, |rtc| {
            rtc.prlh.write(|w| unsafe { w.bits(prescaler >> 16) });
            rtc.prll.write(|w| unsafe { w.bits(prescaler as u16 as u32) });
        });

// ---------
// Setup RTC
// ---------

        rtc_util::set_time(&mut rtc, 0);
        rtc_util::unlisten_alarm(&mut rtc);
        rtc_util::listen_seconds(&mut rtc);
        rtc_util::clear_alarm_flag(&mut rtc);
        rtc_util::clear_second_flag(&mut rtc);
```

```
// -----------
// Init buttons
// -----------
        // Start timer button
        let mut button_start = gpiob.pb5.into_pull_up_input(&mut gpiob.crl);
        button_start.make_interrupt_source(&mut afio);
        button_start.trigger_on_edge(&cx.device.EXTI, FALLING);
        button_start.enable_interrupt(&cx.device.EXTI);
        // Brightness button
        let mut button_brightness = gpiob.pb6.into_pull_up_input(&mut gpiob.crl);
        button_brightness.make_interrupt_source(&mut afio);
        button_brightness.trigger_on_edge(&cx.device.EXTI, FALLING);
        button_brightness.enable_interrupt(&cx.device.EXTI);
        // Note: Both buttons trigger the EXTI9_5 interrupt.

// ------------
// Init pot adc
// ------------
        // Setup adc1 with default settings
        let mut adc1 = Adc::adc1(cx.device.ADC1, &mut rcc.apb2, clocks);
        adc1.set_sample_time(SampleTime::T_239);
        // Setup PA4 as analog input
        let mut pot = gpioa.pa4.into_analog(&mut gpioa.crl);
        // Take initial read
        let mut pot_pos = adc1.read(&mut pot).unwrap();
        pot_pos = pot_pos >> 4;

// -----------------
// Init PMIC control
// -----------------
        let mut sleep_pin = gpioa.pa9.into_push_pull_output(&mut gpioa.crh);
        // Tell the PMIC to please not shut us off
        sleep_pin.set_high().unwrap();

// ------------------
// Init PMIC feedback
// ------------------
        let mut chg_pin = gpioa.pa10.into_pull_up_input(&mut gpioa.crh);
        chg_pin.make_interrupt_source(&mut afio);
        chg_pin.trigger_on_edge(&cx.device.EXTI, RISING_FALLING);
        chg_pin.enable_interrupt(&cx.device.EXTI);
        // This will use interrupt EXTI15_10

// ------------
// Init buzzer
// ------------
```

```
        // For now, only using PWM on one pin
        // Polarity flipping for individual timer channels not yet supported by HAL
        let buzz0 = gpioa.pa0.into_alternate_push_pull(&mut gpioa.crl);
        let mut buzz1 = gpioa.pa1.into_push_pull_output(&mut gpioa.crl);
        buzz1.set_low().unwrap();
        let mut buzzer = Timer::tim2(cx.device.TIM2, &clocks, &mut rcc.apb1)
            .pwm::<Tim2NoRemap, _, _, _>(buzz0, &mut afio.mapr, 440.hz()).split();
        buzzer.set_duty(buzzer.get_max_duty() / 2);

// ----------------
// Init I2C display
// ----------------
        // Init IO pins
        let scl = gpiob.pb8.into_alternate_open_drain(&mut gpiob.crh);
        let sda = gpiob.pb9.into_alternate_open_drain(&mut gpiob.crh);

        // Init i2c peripheral
        let i2c = BlockingI2c::i2c1(
            cx.device.I2C1,
            (scl, sda),
            &mut afio.mapr,
            Mode::Fast {
                frequency: 400_000.hz(),
                duty_cycle: DutyCycle::Ratio2to1,
            },
            clocks,
            &mut rcc.apb1,
            1000,
            10,
            1000,
            1000,
        );

        // Init i2c interface
        let interface = I2CDIBuilder::new().init(i2c);
        // Create display in graphics mode (as opposed to terminal mode)
        let mut display: GraphicsMode<_, _> =
Builder::new().connect(interface).into();
        // Init display
        display.init().unwrap();
        display.clear();
        display.flush().unwrap();

// ------------------
// Start bootup tasks
// ------------------
```

```
        // Read initial ADC value
        let _ = handle_adc::spawn(true);
        // Do startup beep
        let _ = beep::spawn(70, 2);
        // Show boot message
        let _ = update_display::spawn(ScreenPage::Boot);

        // Return initialized resources to RTIC so they can be loaned to tasks
        (init::LateResources {
            display,
            button_start, button_brightness,
            chg_pin,
            EXTI: cx.device.EXTI,
            clocks,
            adc1,
            pot,
            pot_pos,
            sleep_pin,
            buzzer,
            rtc,
        },
         // Return timer object so RTIC can use it for task scheduling.
         init::Monotonics(mono))
    }

    // Idle task, run when nothing else is happening. This is where polling happens.
    #[idle(resources = [sys_state])]
    fn idle(cx: idle::Context) -> ! {
        let mut sys_state = cx.resources.sys_state;
        loop {
            // Read off the ADC value
            sys_state.lock(|sys_state| {
                if *sys_state != SysState::Timer {
                    let _ = handle_adc::spawn(false);
                }
            });
        }
    }

    // This is where we declare tasks which are in external files.
    // 'binds' specifies interrupts the task is bound to
    // 'resources' specify global resource permissions
    // 'priority' decides which tasks can preempt each other
    extern "Rust" {
        #[task(binds = EXTI9_5, resources = [&clocks, button_start,
button_brightness, EXTI, display, brightness_state, sys_state], priority=1)]
        fn handle_buttons(cx: handle_buttons::Context);
```

```
        #[task(binds = EXTI15_10, resources = [chg_pin, disp_call_cnt], priority=1)]
        fn handle_charge(cx: handle_charge::Context);
        #[task(resources = [pot, pot_pos, adc1, pot_dir, max_time], priority=1)]
        fn handle_adc(cx: handle_adc::Context, silent:bool);
        #[task(resources = [rtc, display, max_time, brightness_state, disp_call_cnt,
chg_pin], priority=1, capacity=3)]
        fn update_display(cx: update_display::Context, screen_type:ScreenPage);
        #[task(resources = [disp_call_cnt, sys_state], priority=1, capacity=10)]
        fn reset_display(cx: reset_display::Context);
        #[task(resources = [buzzer], priority=2, capacity=1)]
        fn beep(cx: beep::Context, length: u32, count: u8);
        #[task(resources = [buzzer], priority=2, capacity=1)]
        fn unbeep(cx: unbeep::Context, length: u32, count: u8);
        #[task(binds = RTC, resources = [rtc, sys_state, max_time, disp_call_cnt,
chg_pin], priority=2)]
        fn tick(cx: tick::Context);
        #[task(resources = [rtc, sys_state], priority=3, capacity=1)]
        fn kick_dog(cx: kick_dog::Context);
        #[task(resources = [rtc, sys_state, sleep_pin, max_time, disp_call_cnt],
priority=1, capacity=1)]
        fn to_state(cx: to_state::Context, target: SysState);
    }
}
```

beep.rs

```rust
use crate::app;
use crate::app::*;

use rtic::Mutex;

use embedded_hal::PwmPin;

use rtic::time::duration::Milliseconds;

// Play a beep the specified number of times. Each beep is [length] ms long.
pub fn beep(cx: beep::Context, length: u32, count: u8) {
   let mut buzzer = cx.resources.buzzer;
   // Check for end of "recursive algorithm"
   if count == 0 {return};
   // Turn on the buzzer
   buzzer.lock(|buzzer|{
       buzzer.enable();
   });
   // Schedule turning off the buzzer for later, so the program can go back to
other stuff.
   let _ = unbeep::spawn_after(Milliseconds(length), length, count-1);
}

pub fn unbeep(cx: unbeep::Context, length: u32, count: u8) {
   let mut buzzer = cx.resources.buzzer;
   buzzer.lock(|buzzer|{
       buzzer.disable();
   });
   if count != 0 {let _ = beep::spawn_after(Milliseconds(length), length, count);};
}
```

buttons.rs

```rust
#![allow(non_snake_case)]

use crate::app;
use crate::app::*;
use crate::types::{SysState, ScreenPage};

use rtic::Mutex;

use stm32f1xx_hal::{
   prelude::*,
   gpio::{
       gpiob::{PB6, PB5},
       {Input, PullUp},
       ExtiPin,
   },
};

use ssd1306::{
   prelude::*,
   brightness::Brightness,
};

use embedded_graphics::{
   fonts::Text,
   pixelcolor::BinaryColor,
   prelude::*,
   style::TextStyle,
};
use profont::ProFont24Point;

use embedded_hal::digital::v2::InputPin;

// Task triggered by button interrupt.
pub fn handle_buttons(cx: app::handle_buttons::Context){
   // Bring resources into scope
   let (mut button_start, mut button_brightness) =
       (cx.resources.button_start, cx.resources.button_brightness);
   let (mut EXTI, mut display) =
   (cx.resources.EXTI, cx.resources.display);
   let mut brightness_state = cx.resources.brightness_state;
   let mut sys_state = cx.resources.sys_state;
   //let clocks = cx.resources.clocks;

   // Clear interrupt bits and disable interrupts
   EXTI.lock(|EXTI| {
```

```
        button_start.lock(|button_start| {
            button_brightness.lock(|button_brightness| {
                button_start.disable_interrupt(&EXTI);
                button_brightness.disable_interrupt(&EXTI);
                button_start.clear_interrupt_pending_bit();
                button_brightness.clear_interrupt_pending_bit();
            })
        })
    });

    // Kick the dog
    let _ = kick_dog::spawn();

    // Check button state
    let mut button_start_pressed:bool = false;
    let mut button_brightness_pressed:bool = false;
    button_start.lock(|button_start| {
        button_brightness.lock(|button_brightness| {
            button_start_pressed = button_start.is_low().unwrap();
            button_brightness_pressed = button_brightness.is_low().unwrap();
        })
    });

    // Handle button presses
    if button_brightness_pressed == true || button_start_pressed == true {

        if button_start_pressed == true{
            sys_state.lock(|sys_state|{
                match sys_state {
                    SysState::Setup => {
                        let _ = to_state::spawn(SysState::Timer);
                    }
                    SysState::Timer => {
                        let _ = to_state::spawn(SysState::Setup);
                    }
                    SysState::Sleep => {
                        // The earlier dog-kicking code will have already handled
this situation
                    }
                }
            });
            let _ = beep::spawn(100, 1);
        } else if button_brightness_pressed == true {
            let mut brightness:Brightness = Brightness::DIM;
            brightness_state.lock(|brightness_state|{
                *brightness_state = (*brightness_state+1)%3;
                match brightness_state {
```

```
                    0 => {brightness = Brightness::DIM},
                    1 => {brightness = Brightness::NORMAL},
                    _ => {brightness = Brightness::BRIGHTEST},
                }
                });
                display.lock(|display| {
                    display.set_brightness(brightness).unwrap();
                });
                let _ = update_display::spawn(ScreenPage::Brightness);
                let _ = beep::spawn(10, 5);
            }
        }

        // Enable interrupts
        EXTI.lock(|EXTI| {
            button_start.lock(|button_start| {
                button_brightness.lock(|button_brightness| {
                    button_start.enable_interrupt(&EXTI);
                    button_brightness.enable_interrupt(&EXTI);
                })
            })
        });
    }
```

charge.rs

```rust
use crate::app;
use crate::app::*;

use rtic::Mutex;

use stm32f1xx_hal::gpio::ExtiPin;

pub fn handle_charge(cx: handle_charge::Context){
   let (mut chg_pin, mut disp_call_cnt) =
       (cx.resources.chg_pin, cx.resources.disp_call_cnt);

   // Ignore any pending notification messages on screen
   disp_call_cnt.lock(|disp_call_cnt|{
       *disp_call_cnt = 0;
   });

   // Kick the dog
   let _ = kick_dog::spawn();

   // Refresh the display
   let _ = reset_display::spawn();

   // Clear interrupt flag
   chg_pin.lock(|chg_pin| {
       chg_pin.clear_interrupt_pending_bit();
   });

}
```

config.rs

```rust
// General configuration file for defining constants that might need tweaking.

// Maximum time user can set on the timer
pub const MAX_TIME: u16 = 900;
// Time steps to increment time by
pub const TIME_STEPS: u16 = 15;
// Time it takes to cook a soft-boiled egg
pub const SOFT_BOILED: u16 = 240;
// Time it takes to cook a hard-boiled egg
pub const HARD_BOILED: u16 = 600;
// Sleep timeout
pub const SLEEP_TIME: u16 = 30;
```

pot.rs

```
use crate::app;
use crate::app::*;
use crate::types::ScreenPage;
use crate::config::{MAX_TIME, TIME_STEPS};

use rtic::Mutex;

use stm32f1xx_hal::{
   prelude::*,
   gpio::{
       gpioa::{PA4},
       {Input, PullUp},
       {Alternate, OpenDrain},
   },
   timer::{Event, Timer},
   pac::{ADC1},
};

use ssd1306::{
   prelude::*,
   brightness::Brightness,
};

use embedded_graphics::{
   fonts::Text,
   pixelcolor::BinaryColor,
   prelude::*,
   style::TextStyle,
};
use core::fmt::Write;
use heapless::String;
use heapless::consts::*;
use profont::ProFont24Point;
use profont::ProFont14Point;

use embedded_hal::digital::v2::InputPin;



pub fn handle_adc(cx: app::handle_adc::Context, silent:bool){
   // Bring resources into scope
   let (mut pot, mut adc1) =
       (cx.resources.pot, cx.resources.adc1);
   let mut pot_pos = cx.resources.pot_pos;
   let mut pot_dir = cx.resources.pot_dir;
```

```
    let mut max_time = cx.resources.max_time;

    let mut pot_pos_new:u16 = 0;
    let mut sample_sum:u16 = 0;
    let mut middle_sum:u16 = 0;
    let mut outer_sum:u16 = 0;

    // Read ADC. This is a quick and dirty averaging algorithm, will improve if I
have time.
    pot.lock(|pot| {
        adc1.lock(|adc1| {
            for _i in 0..4 {
                middle_sum = 0;
                for _p in 0..4 {
                    sample_sum = 0;
                    for _n in 0..4 {
                        pot_pos_new = adc1.read(pot).unwrap();
                        pot_pos_new = pot_pos_new >> 4;
                        sample_sum += pot_pos_new;
                    }
                    middle_sum += sample_sum >> 2;
                }
                outer_sum += middle_sum >> 2;
            }
            pot_pos_new = outer_sum >> 2;
        })
    });

    // Another awful algorithm to prevent jitter.
    // If pot turn has changed direction, will not update pot_pos until pot has
moved 2 positions.
    pot_pos.lock(|pot_pos| {
        pot_dir.lock(|pot_dir|{
            // Just pretend this code doesn't exist, you will sleep more soundly.
            let mut pot_changed:bool = false;
            if (*pot_dir == true) && (pot_pos_new > *pot_pos || pot_pos_new <
*pot_pos-1){
                pot_changed = true;
            } else if (*pot_dir == false) && (pot_pos_new > *pot_pos+1 ||
pot_pos_new < *pot_pos){
                pot_changed = true;
            }
            // Handle a registered change in pot position
            if pot_changed == true || silent == true {
                if pot_pos_new > *pot_pos {*pot_dir = true}
                else {*pot_dir = false}
                // Convert old pot position into time
```

```
                let max_time_old:u16 =
(((MAX_TIME/TIME_STEPS)*(255_u16-*pot_pos))/255)*TIME_STEPS;
                // Update pot position
                *pot_pos = pot_pos_new;
                // Convert new pot position into time
                let max_time_new:u16 =
(((MAX_TIME/TIME_STEPS)*(255_u16-*pot_pos))/255)*TIME_STEPS;
                // Only update the display if the pot has moved enough to change the
time
                if max_time_old != max_time_new {
                    // Update the time remaining on the clock
                    max_time.lock(|max_time|{
                        *max_time = max_time_new;
                    });
                    // Kick the dog
                    let _ = kick_dog::spawn();
                    // Update the display with the new time
                    if silent == false {
                        let _ = update_display::spawn(ScreenPage::Setup);
                    }
                }

            }
        })
    });

}
```

rtc.rs

```
use crate::app;
use crate::app::*;
use crate::types::{ScreenPage, SysState};
use crate::rtc_util;
use crate::config::SLEEP_TIME;

use rtic::Mutex;

use stm32f1xx_hal::{
   prelude::*,
   rtc::Rtc,
};

use embedded_hal::digital::v2::InputPin;

use cortex_m::asm::delay;

// tick: triggered via interrupt every time the RTC counts down one second
pub fn tick(cx: tick::Context) {
   // Bring resources into scope
   let (mut sys_state, mut max_time) =
       (cx.resources.sys_state, cx.resources.max_time);
   let mut disp_call_cnt = cx.resources.disp_call_cnt;
   let mut rtc = cx.resources.rtc;
   let mut chg_pin = cx.resources.chg_pin;
   // Tiny arbitrary delay to ensure the RTC counter is updated.
   // For some reason the tick interrupt gets triggered just before the counter is
updated.
   // It's in the manual but still odd.
   delay(100);
   sys_state.lock(|sys_state|{
       match sys_state {
           SysState::Setup => {
               // Get time since last user interaction
               let current_time:u16 = rtc.lock(|rtc|{
                   return rtc_util::current_time(rtc) as u16;
               });
               let chg_state:bool = chg_pin.lock(|chg_pin|{
                   return chg_pin.is_low().unwrap();
               });

               // Is it time to fall asleep?
               if current_time >= SLEEP_TIME {
                   // Is the device charging right now?
                   if chg_state == true {
```

```
                        // If charging, show icon fullscreen and reset sleep timer
                        rtc.lock(|rtc|{
                            rtc_util::set_time(rtc,0);
                        });
                        let _ = update_display::spawn(ScreenPage::Charging);

                    } else {
                        // If not charging, got to sleep
                        let _ = to_state::spawn(SysState::Sleep);
                    }
                }
            }
            SysState::Timer => {
                // Get current time
                let current_time:u16 = rtc.lock(|rtc|{
                    return rtc_util::current_time(rtc) as u16;
                });
                // Get maximum time (what the timer was set to)
                let maximum_time:u16 = max_time.lock(|max_time|{return *max_time});
                // Is the timer down to 0?
                if current_time < maximum_time {
                    // If timer is still running, update the display but only if
nothing more
                    //   important is currently showing
                    let cnt:u8 = disp_call_cnt.lock(|disp_call_cnt|{
                        return *disp_call_cnt;
                    });
                    if cnt == 0 {
                        let _ = update_display::spawn(ScreenPage::Timer);
                    }
                } else {
                    // If timer is down to zero, trigger alarm and return to main
menu
                    let _ = update_display::spawn(ScreenPage::Alarm);
                    let _ = beep::spawn(500, 5);
                    let _ = to_state::spawn(SysState::Setup);
                }

            }
            SysState::Sleep => {
                // We dont really care what time it is when we're sleeping
            }
        }
    });
    rtc.lock(|rtc|{rtc_util::clear_second_flag(rtc)});
}
```

```
// kick_dog: spawned by other tasks when user interacts with the main menu.
//   Resets the rtc counter delay auto-sleep.
pub fn kick_dog(cx: kick_dog::Context) {
   let mut sys_state =
       cx.resources.sys_state;
   let mut rtc = cx.resources.rtc;
   sys_state.lock(|sys_state|{
       match sys_state {
           SysState::Setup => {
               rtc.lock(|rtc|{
                   // Put off sleep timer when user interacts with device
                   rtc_util::set_time(rtc,0);
               });
           }
           SysState::Timer => {
               // Nothing to do
           }
           SysState::Sleep => {
               // If the user moves some input while the system is going to sleep,
               //   cancel sleep.
               let _ = to_state::spawn(SysState::Setup);
           }
       }
   });
}
```

rtc_util.rs

```
//--------
//RTC Util
//--------
// IMPORTANT!!!
// Most of this code was copied from the HAL's RTC object at:
// https://github.com/stm32-rs/stm32f1xx-hal/blob/master/src/rtc.rs
// and then modified to work more like a C-style library than an OOP object.

use stm32f1xx_hal::{
   pac::RTC
};

// This implements the RTC config register write algorithm found on page 485 of the
manual.
// The function is a modified version of the one from the stm32f1xx_hal.
pub fn rtc_write(rtc: &mut RTC , func: impl Fn(&mut RTC)) {
   // Wait for the last write operation to be done
   while !(*rtc).crl.read().rtoff().bit() {}
   // Put the clock into config mode
   (*rtc).crl.modify(|_, w| w.cnf().set_bit());

   // Perform the write operation
   func(rtc);

   // Take the device out of config mode
   (*rtc).crl.modify(|_, w| w.cnf().clear_bit());
   // Wait for the write to be done
   while !(*rtc).crl.read().rtoff().bit() {}
}

// Set the current RTC counter value to the specified amount
pub fn set_time(rtc: &mut RTC, counter_value: u32) {
   rtc_write(rtc, |rtc| {
       (*rtc)
           .cnth
           .write(|w| unsafe { w.bits(counter_value >> 16) });
       (*rtc)
           .cntl
           .write(|w| unsafe { w.bits(counter_value as u16 as u32) });
   });
}

/*
 Sets the time at which an alarm will be triggered
 This also clears the alarm flag if it is set
```

```
*/
pub fn set_alarm(rtc: &mut RTC, counter_value: u32) {
   // Set alarm time
   // See section 18.3.5 for explanation
   let alarm_value = counter_value - 1;

       rtc_write(rtc, |rtc| {
       (*rtc)
           .alrh
           .write(|w|  w.alrh().bits((alarm_value >> 16) as u16) );
       (*rtc)
           .alrl
           .write(|w|  w.alrl().bits(alarm_value as u16) );
   });

   clear_alarm_flag(rtc);
}

// Enables the RTC interrupt to trigger when the counter reaches the alarm value.
In addition,
// if the EXTI controller has been set up correctly, this function also enables the
RTCALARM
// interrupt.
/*pub fn listen_alarm(rtc: &mut RTC) {
   // Enable alarm interrupt
   rtc_write(rtc, |rtc| {
       (*rtc).crh.modify(|_, w| w.alrie().set_bit());
   })
}*/

// Stops the RTC alarm from triggering the RTC and RTCALARM interrupts
pub fn unlisten_alarm(rtc: &mut RTC) {
   // Disable alarm interrupt
   rtc_write(rtc, |rtc| {
       (*rtc).crh.modify(|_, w| w.alrie().clear_bit());
   })
}

// Reads the current counter
pub fn current_time(rtc: &mut RTC) -> u32 {
   // Wait for the APB1 interface to be ready
   //rtc_write(rtc, |rtc| (*rtc).crl.modify(|_, w| w.rsf().clear_bit()));
   //while (*rtc).crl.read().rsf().bit() {}

   (*rtc).cnth.read().bits() << 16 | (*rtc).cntl.read().bits()
}
```

```rust
// Enables triggering the RTC interrupt every time the RTC counter is increased
pub fn listen_seconds(rtc: &mut RTC) {
   rtc_write(rtc, |rtc| (*rtc).crh.modify(|_, w| w.secie().set_bit()))
}

// Disables the RTC second interrupt
pub fn unlisten_seconds(rtc: &mut RTC) {
   rtc_write(rtc, |rtc| (*rtc).crh.modify(|_, w| w.secie().clear_bit()))
}

// Clears the RTC second interrupt flag
pub fn clear_second_flag(rtc: &mut RTC) {
   rtc_write(rtc, |rtc| (*rtc).crl.modify(|_, w| w.secf().clear_bit()))
}

// Clears the RTC alarm interrupt flag
pub fn clear_alarm_flag(rtc: &mut RTC) {
   rtc_write(rtc, |rtc| (*rtc).crl.modify(|_, w| w.alrf().clear_bit()))
}
```

states.rs

```
use crate::app;
use crate::app::*;
use crate::types::{ScreenPage, SysState};
use crate::config::{SLEEP_TIME};
use crate::rtc_util;

use rtic::Mutex;

use stm32f1xx_hal::{
   prelude::*,
   gpio::{
       gpioa::{PA9},
       {Output, PushPull},
   },
};

use embedded_hal::digital::v2::OutputPin;

// This task is spawned by other parts of the program. It does some housekeeping to
move the
//   system to a specific state.
pub fn to_state(cx: to_state::Context, target: SysState){
   // Bring resources into scope
   let (mut sys_state, mut sleep_pin) =
       (cx.resources.sys_state, cx.resources.sleep_pin);
   let mut max_time = cx.resources.max_time;
   let mut disp_call_cnt = cx.resources.disp_call_cnt;
   let mut rtc = cx.resources.rtc;
   // Acquire display status message status
   let cnt:u8 = disp_call_cnt.lock(|disp_call_cnt|{
       return *disp_call_cnt;
   });
   // Acquire resource locks
   rtc.lock(|rtc|{
   sys_state.lock(|sys_state|{
   sleep_pin.lock(|sleep_pin|{
       match target {
           SysState::Setup => {
               // Set new system state
               *sys_state = SysState::Setup;
               // Reset RTC
               rtc_util::set_time(rtc, 0);
               // Configure RTC for use as sleep timer
               rtc_util::listen_seconds(rtc);
               rtc_util::clear_second_flag(rtc);
```

```
                // Make sure we aren't shutting off
                sleep_pin.set_high().unwrap();
                // Update the display, unless there's a status message being shown
                if cnt == 0 {
                    let _ = update_display::spawn(ScreenPage::Setup);
                }
            }
            SysState::Timer => {
                // Set new system state
                *sys_state = SysState::Timer;
                // Reset RTC
                rtc_util::set_time(rtc,0);
                max_time.lock(|max_time| {
                    // Set RTC alarm to trigger when timer runs out
                    rtc_util::set_alarm(rtc,*max_time as u32);
                });
                // Configure RTC for use as egg timer
                rtc_util::listen_seconds(rtc);
                rtc_util::clear_second_flag(rtc);
                // Make sure we aren't shutting off
                sleep_pin.set_high().unwrap();
                // Update the display
                let _ = update_display::spawn(ScreenPage::Timer);
            }
            SysState::Sleep => {
                // Update system state
                *sys_state = SysState::Sleep;
                // Shut off RTC alarms
                rtc_util::unlisten_seconds(rtc);
                rtc_util::clear_second_flag(rtc);
                // Tell PMIC to shut us off
                sleep_pin.set_low().unwrap();
                // Show sleep message on display
                if cnt == 0 {
                    let _ = update_display::spawn(ScreenPage::Sleep);
                }
            }
        }
    });});});});
}
```

types.rs

```rust
// Enums used in this project

// State of the overarching state machine
#[derive(PartialEq)]
pub enum SysState {
   Setup, // User is setting time on the timer
   Timer, // Timer is ticking down
   Sleep, // MCU has asked PMIC to shut off power
}

pub enum ScreenPage {
   // Main status pages for the main system states
   Setup,
   Timer,
   Sleep,
   // Temporary notification pages
   Brightness,
   Alarm,
   Boot,
   Charging,
}
```

ui.rs

```rust
use crate::app;
use crate::app::*;
use crate::types::{ScreenPage, SysState};
use crate::config::{HARD_BOILED, SOFT_BOILED};
use crate::logo::LOGO;
use crate::bigbolt::BIGBOLT;
use crate::smallbolt::SMALLBOLT;
use crate::rtc_util;

use rtic::Mutex;

use ssd1306::{
   prelude::*,
   brightness::Brightness,
};

use embedded_graphics::{
   fonts::Text,
   pixelcolor::BinaryColor,
   prelude::*,
   style::TextStyle,
   image::{Image, ImageRaw},
};

use embedded_hal::digital::v2::InputPin;

use core::fmt::Write;
use heapless::String;
use heapless::consts::*;
use profont::ProFont24Point;
use profont::ProFont14Point;
use rtic::time::duration::Seconds;

pub fn update_display(cx: update_display::Context, screen_type:ScreenPage){
   // Bring resources into scope
   let (mut display, mut brightness_state) =
       (cx.resources.display, cx.resources.brightness_state);
   let mut max_time = cx.resources.max_time;
   let mut disp_call_cnt = cx.resources.disp_call_cnt;
   let mut chg_pin = cx.resources.chg_pin;
   let mut rtc = cx.resources.rtc;

   display.lock(|display| {
       // Wipe the slate
       display.clear();
```

```
        let chg_state:bool = chg_pin.lock(|chg_pin|{
            return chg_pin.is_low().unwrap();
        });
        // Are we charging?
        if chg_state == true {
            // If we're charging, show little bolt icon in the corner
            let raw_image: ImageRaw<BinaryColor> = ImageRaw::new(SMALLBOLT, 14, 14);
            Image::new(&raw_image, Point::new(114,0))
                .draw(display)
                .unwrap();
        }
        match screen_type {
            // Display the time set screen
            ScreenPage::Setup => {
                disp_call_cnt.lock(|disp_call_cnt|{*disp_call_cnt = 0});
                max_time.lock(|max_time| {
                    // Format the text
                    let mut data = String::<U16>::from("");
                    let minutes = *max_time/60;
                    let seconds = *max_time%60;
                    let _ = write!(data, "{:>2}:{:>02}", minutes, seconds);

                    // Create the graphics object and draw it on the "buffer"
                    Text::new(&data[..], Point::new(20,16))
                        .into_styled(TextStyle::new(ProFont24Point,
BinaryColor::On))
                        .draw(display)
                        .unwrap();

                    // Determine special status message for special time settings
                    let status_msg: &str = match *max_time {
                        SOFT_BOILED => "Soft-Boiled",
                        HARD_BOILED => "Hard-Boiled",
                        _ => "",
                    };

                    // Render special status message
                    Text::new(status_msg, Point::new(10,48))
                        .into_styled(TextStyle::new(ProFont14Point,
BinaryColor::On))
                        .draw(display)
                        .unwrap();

                    // Render constant status message
                    Text::new("Set Time:", Point::new(10,0))
                        .into_styled(TextStyle::new(ProFont14Point,
BinaryColor::On))
```

```
                    .draw(display)
                    .unwrap();

                // Write buffer to display
                display.flush().unwrap();
            });
        },
        // Display the countdown screen
        ScreenPage::Timer => {
            disp_call_cnt.lock(|disp_call_cnt|{*disp_call_cnt = 0});
            let mut show_start_msg: bool = false;
            let time_remaining: u16 = rtc.lock(|rtc| {
                return max_time.lock(|max_time|{
                    let current_time = rtc_util::current_time(rtc) as u16;
                    if current_time == 0 {show_start_msg = true}
                    if current_time <= *max_time {
                        return *max_time - current_time
                    } else {
                        return 0
                    }
                });
            });

            if show_start_msg == true {
                // Create the graphics object and draw it on the "buffer"
                Text::new("START", Point::new(20,16))
                    .into_styled(TextStyle::new(ProFont24Point,
BinaryColor::On))
                    .draw(display)
                    .unwrap();
            } else {
                // Format the text
                let mut data = String::<U16>::from("");
                let minutes = time_remaining/60;
                let seconds = time_remaining%60;
                let _ = write!(data, "{:>2}:{:>02}", minutes, seconds);

                // Create the graphics object and draw it on the "buffer"
                Text::new(&data[..], Point::new(20,16))
                    .into_styled(TextStyle::new(ProFont24Point,
BinaryColor::On))
                    .draw(display)
                    .unwrap();
            }

            // Write buffer to display
            display.flush().unwrap();
```

```
},
ScreenPage::Brightness => {
    display.clear();
    let mut disp_str: &'static str = "";
    brightness_state.lock(|brightness_state|{
        match brightness_state {
            0 => {disp_str = "Dim"},
            1 => {disp_str = "Med"},
            _ => {disp_str = "Bright"},
        }
    });
    Text::new(disp_str, Point::new(20,16))
        .into_styled(TextStyle::new(ProFont24Point, BinaryColor::On))
        .draw(display)
        .unwrap();
    Text::new("Brightness:", Point::new(10,0))
        .into_styled(TextStyle::new(ProFont14Point, BinaryColor::On))
        .draw(display)
        .unwrap();
    display.flush().unwrap();
    disp_call_cnt.lock(|disp_call_cnt|{*disp_call_cnt += 1});
    let _ = reset_display::spawn_after(Seconds(2_u32));
},
ScreenPage::Alarm => {
    Text::new("Alarm!", Point::new(20,16))
        .into_styled(TextStyle::new(ProFont24Point, BinaryColor::On))
        .draw(display)
        .unwrap();
    display.flush().unwrap();
    // Schedule the screen to go back to what it was previously showing
    disp_call_cnt.lock(|disp_call_cnt|{*disp_call_cnt += 1});
    let _ = reset_display::spawn_after(Seconds(5_u32));
},
ScreenPage::Boot => {
    display.clear();
    let raw_image: ImageRaw<BinaryColor> = ImageRaw::new(LOGO, 128, 64);
    Image::new(&raw_image, Point::zero())
        .draw(display)
        .unwrap();
    Text::new("Egg Timer!", Point::new(20,44))
        .into_styled(TextStyle::new(ProFont14Point, BinaryColor::On))
        .draw(display)
        .unwrap();
    // Write buffer to display
    display.flush().unwrap();
    // Schedule the screen to go back to what it was previously showing
    disp_call_cnt.lock(|disp_call_cnt|{*disp_call_cnt += 1});
```

```
                let _ = reset_display::spawn_after(Seconds(3_u32));
            },
            ScreenPage::Sleep => {
                display.clear();
                let raw_image: ImageRaw<BinaryColor> = ImageRaw::new(LOGO, 128, 64);
                Image::new(&raw_image, Point::zero())
                    .draw(display)
                    .unwrap();
                Text::new("Power Off", Point::new(20,44))
                    .into_styled(TextStyle::new(ProFont14Point, BinaryColor::On))
                    .draw(display)
                    .unwrap();
                // Write buffer to display
                display.flush().unwrap();
            },
            ScreenPage::Charging => {
                display.clear();
                let raw_image: ImageRaw<BinaryColor> = ImageRaw::new(BIGBOLT, 128,
64);
                Image::new(&raw_image, Point::zero())
                    .draw(display)
                    .unwrap();
                /*Text::new("", Point::new(20,44))
                    .into_styled(TextStyle::new(ProFont14Point, BinaryColor::On))
                    .draw(display)
                    .unwrap();*/
                // Write buffer to display
                display.flush().unwrap();
            },
        }
    });
}

// Figure out what the display should be showing for the current system state and
show it
pub fn reset_display(cx: reset_display::Context) {
    let mut disp_call_cnt = cx.resources.disp_call_cnt;
    let mut sys_state = cx.resources.sys_state;
    disp_call_cnt.lock(|disp_call_cnt|{
        // Call count mechanism ensures we don't revert the screen if we're already
showing
        //   another, newer status message.
        if *disp_call_cnt <= 1 {
            *disp_call_cnt = 0;
            sys_state.lock(|sys_state|{
                match *sys_state {
                    SysState::Setup => { let _ =
```

```
update_display::spawn(ScreenPage::Setup); },
                    SysState::Timer => { let _ =
update_display::spawn(ScreenPage::Timer); },
                    SysState::Sleep => { let _ =
update_display::spawn(ScreenPage::Sleep); },
                }
            });
        } else {
            *disp_call_cnt -= 1;
        }
    });
}
```

Resources

- PCB design and schematic capture: KiCAD https://www.kicad.org/
- Mechanical CAD: FreeCAD https://www.freecadweb.org/
- Parts source: DigiKey https://digikey.com
- PCB fab: OSHpark https://oshpark.com
- IDE: Jetbreains Clion
- Language: Rust https://www.rust-lang.org/
- HAL: https://github.com/stm32-rs/stm32f1xx-hal
- Realtime concurrency framework: rtic.rs

PCB Rev 1 Issues

Cosmetic

- Increase LED current limiting resistor to decrease brightness (can be done with no board changes).
- Add pin labels to debug ports.
- Add polarity label outside of battery holder symbol so user knows which way to insert the battery.
- PMIC gets hot. Solution: Add heat sink. Requires no board changes.
- Add board revision number to board.
- Remove redundant ground connections.
- Add extra headers wherever possible to facilitate board hacking.
- Breakout more MCU pins.

Requires case design changes

- Start button is too close to potentiometer. Solutions:
    - Mount start button lower.
    - Use push button potentiometer.

Requires code changes

- Pot is mounted upside-down (notabug wontfix).
- Possible improvement: Connect vout to an ADC pin through a voltage divider to enable battery level monitoring.
- Connect power button to MCU pin to allow for alternate, one-press functions.

Serious

- PMIC buck_en is hardwired to vout, making the state machine always automatically move to the on state. This makes it impossible to turn the device off. Solutions:
    - Temp solution on current board revision: Cut connection from buck_en to vout and jump buck_en to pa9 on the serial connector, repurposing pa9 for MCU power control.
    - Possible solution: Wire buck_en to 3v3 bus, ensuring that buck_en is asserted only when the device is actually on.
    - Possible solution: Wire buck_en to an MCU pin to allow computer control of power state.

Additional Assembly Considerations

When assembling PCB rev 1, some parts need to be installed in nonstandard ways. The following is a list of additional steps that must be taken.

1. When installing the OLED module, cut the pin header as short as possible so as to mount the module flush to the piezoelectric buzzer without the header protruding far from the other side of the board. Add a double layer of gaffer's tape between the buzzer and OLED module to serve as a mechanical buffer and prevent sharp parts on the back of the OLED board from damaging parts on the timer board.
2. The battery holder is meant to be installed flush to the PCB. Through-hole components on the other side prevent this, but the battery holder can still be installed securely if additional steps are followed. Note that mechanical strength is slightly due to the plastic clips on the battery holder not being able to fully lock onto the timer PCB. However, this should not be an issue as the battery holder is fully enclosed by the case. The steps are:
    a. Before installing the potentiometer, cut the mounting tabs to about half of their length so they protrude less from the other side of the PCB. Solder the tabs on from both the top and bottom so the potentiometer remains securely mounted.
    b. After assembling and soldering the top side of the PCB, cut off any sharp tips from joints on the bottom side.
    c. After installing the battery holder, affix a layer of gaffer's tape to the inside of the fixture. This, along with step B, prevents sharp protrusions from puncturing the battery cell through the slot in the battery holder.
3. It might be necessary to apply a small drop of hot glue to the 6mm tactile switches before attaching the 6mm diameter button caps. This is necessary because the caps are designed for switches with a shorter actuator. 6mm tact switches all use the same roughly conical shape for the actuator, where shorter actuators have the shape 'cut' closer to the base. Hence, the top of the switch may be too thin if the cap is made for a more common short switch. Hot glue, while generally not a durable solution, should be sufficient because the caps do not experience any tensile stress under normal operation. The glue is only there to keep them from falling off under their own weight.

Misc Notes

Misc Circuit Design Notes

- ICR18650 battery datasheet says standard charge current is 1.3A for a standard 4h charge and 2.6A for a fast, 2.5h charge. I'm choosing an approximate 2.0A charge current as a compromise between quick charging and longevity. According to the LTC3553 PMIC datasheet, this means I need a 750V/2.0A = 375Ω ≅ 400Ω resistor for a 1.875A charge current.
- For the red charging indicator LED, I chose the IN-S126ATR SMD LED. The LED would be powered from a 3.3V potential, and it has a forward voltage of 1.8V and a maximum current of 20mA. Hence, the current limiting resistor should be (5V-1.8V)/20mA=160Ω.
- The buzzer will be driven directly by two MCU pins with an inverted PWM signal.

Deciding on button and pot length

Part	Req. Clearance from PCB to enclosure (for base)	Max protrusion from surface (arbitrary UX decision)	Requisite total length (panel clearance + min panel thickness + max protrusion)
OLED module	~7mm	N/A	
PTV09A pot	6.8mm	10mm	20mm
Tac switch	Don't care	5mm	15mm
Enclosure top panel	8mm	N/A	

Misc Software Design Notes

- Using Rust as programming language, stm32f1xx_hal as hardware interface library, and RTIC as task scheduling and interrupt management system.
- None of the libraries used are at 1.0, so there will be some growing pains.
- Handle button presses with interrupts, pot input with idle task.
- Use RTC for main timer. I'm not exactly sure how I'll display remaining time, but I might just have an RTC alarm go off every second. The priority is to keep the main clock running no matter what, not even stoping for a few cycles. In the worst case, the CPU can poll the RTC during its idle time. Another potential option is to have a regular timer go off every second, then read the RTC time and apply a correction when setting the next timer.
- Use task scheduling and system clock for UI timers. For example, when changing brightness while the timer is ticking, the brightness notification should show up for some

time and then disappear again in order to show remaining time. Everything that requires a time delay should use scheduling, including beeps. This way, all time that isn't spent actively processing can be spent doing other tasks.

- Make a beep task that can be fired off from anywhere in the program with frequency, length, and repetition parameters. The beep task should come paired with a beep-off task, which turns off the buzzer after the beep is over and reschedules the beep task in case of a multi-beep.