STM32 Mini Printer

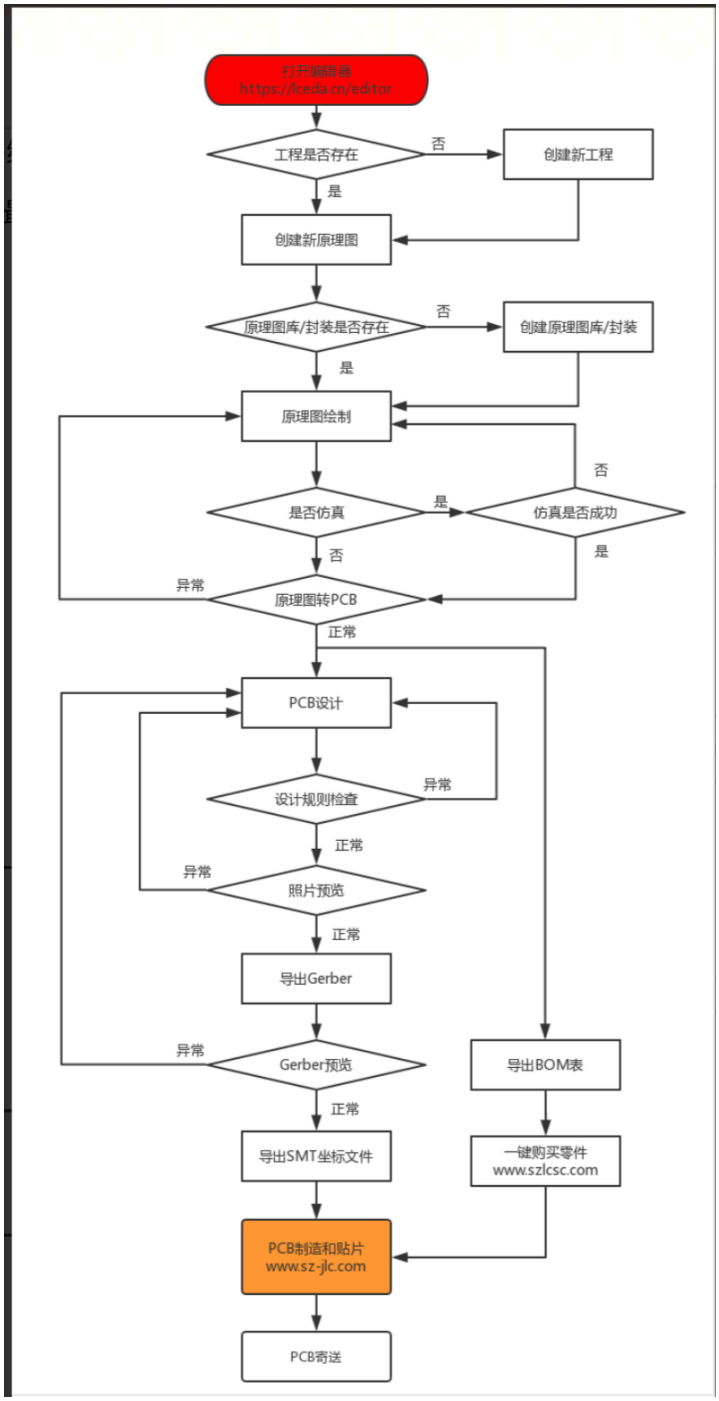
Parts Explanation:

Step down converter: dc converter that reduces in put voltage to a lower output voltage
Arduino: brain of the project, controls robot

Mini打印机Project教程(STM32 knowledge PCB design)

https://x509p6c8to.feishu.cn/docx/JmWhd7qThofnSvx7V87cgvoXnue

Password: HN1b816H



STM 32 Cube MX Design:

确定用哪一个STM32的芯片和确定STM32的对应IO位置

file:///Users/hugoq1/Downloads/stm32f103cb.pdf

以图下红色标的那一个来确认123456等等等然后看default function

打开芯片手册，然后找到pin definitions页面，这里主要查看一些特殊功能的引脚，例如adc、uart、spi用的是哪些io，确定下来后，把其它的通用IO也逐一确定即可，**这些通用的IO后续绘制原理图时，还可以调整。**

Table 5. Medium-density STM32F103xx pin definitions

Pins							Pin name	Type(1)	I / O Level(2)	Main function(3) (after reset)	Alternate functions(4)	
LFBGA100	UFBG100	LQFP48/UFQFPN48	TFBGA64	LQFP64	LQFP100	VFQFPN36					Default	Remap
A3	B2	-	-	-	1	-	PE2	I/O	FT	PE2	TRACECK	
B3	A1	-	-	-	2	-	PE3	I/O	FT	PE3	TRACED0	
C3	B1	-	-	-	3	-	PE4	I/O	FT	PE4	TRACED1	
D3	C2	-	-	-	4	-	PE5	I/O	FT	PE5	TRACED2	
E3	D2	-	-	-	5	-	PE6	I/O	FT	PE6	TRACED3	
B2	E2	1	B2	1	6	-	V_{BAT}	S		V_{BAT}		
A2	C1	2	A2	2	7	-	PC13-TAMPER-RTC(5)	I/O		PC13(6)	TAMPER-RTC	
A1	D1	3	A1	3	8	-	PC14-OSC32_IN(5)	I/O		PC14(6)	OSC32_IN	
B1	E1	4	B1	4	9	-	PC15-OSC32_OUT(5)	I/O		PC15(6)	OSC32_OUT	
C2	F2	-	-	-	10	-	V_{SS_5}	S		V_{SS_5}		
D2	G2	-	-	-	11	-	V_{DD_5}	S		V_{DD_5}		
C1	F1	5	C1	5	12	2	OSC_IN	I		OSC_IN		PD0(7)
D1	G1	6	D1	6	13	3	OSC_OUT	O		OSC_OUT		PD1(7)

ADC: used for analog to digital conversion

:

G2	L2	10	G2	14	23	7	PA0-WKUP	I/O	-	PA0	WKUP/ USART2_CTS(9)/ ADC12_IN0/ TIM2_CH1_ ETR(9)	-
H2	M2	11	H2	15	24	8	PA1	I/O	-	PA1	USART2_RTS(9)/ ADC12_IN1/ TIM2_CH2(9)	-
J2	K3	12	F3	16	25	9	PA2	I/O	-	PA2	USART2_TX(9)/ ADC12_IN2/ TIM2_CH3(9)	-
K2	L3	13	G3	17	26	10	PA3	I/O	-	PA3	USART2_RX(9)/ ADC12_IN3/ TIM2_CH4(9)	-
E4	E3	-	C2	18	27	-	V_{SS_4}	S	-	V_{SS_4}	-	-
F4	H3	-	D2	19	28	-	V_{DD_4}	S	-	V_{DD_4}	-	-
G3	M3	14	H3	20	29	11	PA4	I/O	-	PA4	SPI1_NSS(9)/ USART2_CK(9)/ ADC12_IN4	-
H3	K4	15	F4	21	30	12	PA5	I/O	-	PA5	SPI1_SCK(9)/ ADC12_IN5	-
J3	L4	16	G4	22	31	13	PA6	I/O	-	PA6	SPI1_MISO(9)/ ADC12_IN6/ TIM3_CH1(9)	TIM1_BKIN
K3	M4	17	H4	23	32	14	PA7	I/O	-	PA7	SPI1_MOSI(9)/ ADC12_IN7/ TIM3_CH2(9)	TIM1_CH1N
G4	K5	-	H5	24	33		PC4	I/O	-	PC4	ADC12_IN14	-
H4	L5	-	H6	25	34		PC5	I/O	-	PC5	ADC12_IN15	-
J4	M5	18	F5	26	35	15	PB0	I/O	-	PB0	ADC12_IN8/ TIM3_CH3(9)	TIM1_CH2N
K4	M6	19	G5	27	36	16	PB1	I/O	-	PB1	ADC12_IN9/ TIM3_CH4(9)	TIM1_CH3N

UART: this is used for transmitting and receiving data

Helpful resource: https://www.totalphase.com/blog/2016/06/spi-vs-uart-similarities-differences/

UART (Universal Asynchronous Receiver/Transmitter) is a hardware communication protocol used for serial communication. It typically involves the following signals:

1. **TX (Transmit)**: This is the line used by the UART to send data from one device to another. It's the output from the transmitting device.
2. **RX (Receive)**: This is the line used by the UART to receive data. It's the input to the receiving device.
3. **RTS (Request to Send)**: This is a control signal used in hardware flow control. When the transmitting device is ready to send data, it asserts the RTS line to signal the receiving device to prepare to receive data.
4. **CTS (Clear to Send)**: This is another control signal used in hardware flow control. The receiving device asserts the CTS line to indicate that it's ready to receive data. The transmitting device checks the CTS line before sending data to ensure the receiver is ready.

UART (Universal Asynchronous Receiver/Transmitter)

Characteristics:

- **Asynchronous Communication**: No clock signal is shared between the transmitter and receiver. Instead, both ends must agree on parameters like baud rate, parity, and stop bits.
- **Simple**: Only requires two lines for data transmission: TX (transmit) and RX (receive). Optional control lines include RTS (Request to Send) and CTS (Clear to Send) for flow control.
- **Point-to-Point Communication**: Typically used for communication between two devices.

When to Use UART:

- When you need a simple, low-cost, and easy-to-implement communication method.
- When you're connecting two devices with different clock domains and don't need a synchronous clock.
- For long-distance communication where simplicity and robustness are preferred.

Examples:

- Serial communication with a computer (e.g., using a USB-to-serial adapter).
- Communication between a microcontroller and a GPS module.

D9	D11	29	D7	41	67	20	PA8	I/O	FT	PA8	USART1_CK/ TIM1_CH1(9)/ MCO	-
C9	D10	30	C7	42	68	21	PA9	I/O	FT	PA9	USART1_TX(9)/ TIM1_CH2(9)	-
D10	C12	31	C6	43	69	22	PA10	I/O	FT	PA10	USART1_RX(9)/ TIM1_CH3(9)	-
C10	B12	32	C8	44	70	23	PA11	I/O	FT	PA11	USART1_CTS/ CANRX(9)/ USBDM/ TIM1_CH4(9)	-
B10	A12	33	B8	45	71	24	PA12	I/O	FT	PA12	USART1_RTS/ CANTX(9) /USBDP TIM1_ETR(9)	-

SPI:

Serial Peripheral Interface (SPI) is an interface bus commonly used to send data between microcontrollers and small peripherals such as shift registers, sensors, and SD cards

G3	M3	14	H3	20	29	11	PA4	I/O	-	PA4	SPI1_NSS(9)/ USART2_CK(9)/ ADC12_IN4	-
H3	K4	15	F4	21	30	12	PA5	I/O	-	PA5	SPI1_SCK(9)/ ADC12_IN5	-
J3	L4	16	G4	22	31	13	PA6	I/O	-	PA6	SPI1_MISO(9)/ ADC12_IN6/ TIM3_CH1(9)	TIM1_BKIN
K3	M4	17	H4	23	32	14	PA7	I/O	-	PA7	SPI1_MOSI(9)/ ADC12_IN7/ TIM3_CH2(9)	TIM1_CH1N

外围设置:

Lithium ion battery data sheet https://www.lcsc.com/datasheet/lcsc_datasheet_2401181411_MICRONE-Nanjing-Micro-One-Elec-ME4054BM5G-N_C478383.pdf

Mah is milliampere hour, how much charge a battery can hold for hour, like 2000 mah 2000 ma per hour

Wheres Coulomb is how much this battery discharges, a 2000 mah with a 1C discharge rate means it discharges 2000 milliamps in one hour; if it is 2C, it means it discharges 2000 milliamps in half an hour

Formula:

$$\text{Current (A)} = \text{C-rate} \times \text{Capacity (Ah)}$$

For a 2000 mAh (or 2 Ah) battery:

- At 1C, the current is $1 \times 2\ \text{Ah} = 2\ \text{A}$.
- At 0.5C, the current is $0.5 \times 2\ \text{Ah} = 1\ \text{A}$.
- At 2C, the current is $2 \times 2\ \text{Ah} = 4\ \text{A}$.

而我们需要的printer的current需求要在500mA之内， 2000mah 而动力电池能达到10C左右

datasheet写的是4.2V lithium battery

一、充电管理

需求：希望产品可以不接电源、外出使用

所以需要添加一颗锂电池用于供电，这里的锂电池我们选择动力电池，因为普通锂电池最大输出电流在1C左右，也就是2000mah的电池，最大输出2A电流，而动力电池可以到10C，这是由打印头的电源需求决定的。

需求：一颗2000mah的锂电池，电压3.7-4.2V，充电电流1000ma内

项目中使用ME4054B-N作为充电管理方案，充电电流最大500mA，而且封装较小。

https://item.szlcsc.com/485645.html

采用 Thin-SOT 封装的独立线性锂离子电池充电芯片 ME4054B-N

概述

ME4054B-N 是一款完整的单节锂离子电池用恒定电流/恒定电压线性充电芯片。其中 ThinSOT 封装与较少的外部元器件数目使得 ME4054B-N 成为便携式应用的理想选择。而且 ME4054B-N 是专为在 USB 电源规范内工作而设计的。

由于采用内部 MOSFET 构架，所以不需要外部检测电阻器和隔离二极管。热反馈可对充电电流进行调节以便在大功率操作或高环境温度条件下对芯片温度加以限制。充电电压固定为 4.2V，而充电电流可通过一个电阻器进行外部设置。当充电电流在达到最终浮充电压之后降至设定值的 1/10 时，ME4054B-N 将自动终止充电循环。

当输入电压（交流适配器或 USB 电源）被拿掉时，ME4054B-N 自动进入一个低电流状态，将电池漏电流降至 2μA 以下，可将 ME4054B-N 置于停机模式，从而将供电电流降至 55μA。

ME4054B-N 的其他特点包括充电电流监控器、欠压闭锁、自动再充电和一个用于指示充电结束和输入电压接入的状态引脚。

特点

- 可编程充电电流范围：20-500mA
- 充电终止：3C/10 充电终止
- 充电状态输出引脚，LED 开/关两种状态
- 无需 MOSFET、检测电阻器和隔离二极管
- 用于单节锂离子电池、采用 ThinSOT 封装的完整线性充电器
- 恒定电流/恒定电压操作，并具有可在无过热危险的情况下实现充电速率最大化的热调节功能
- 直接从 USB 端口给锂离子电池充电
- 高精度的 4.2V 预设充电电压
- 用于电池电量检测的充电电流监控器输出
- 自动再充电
- 停机模式下供电电流为 55μA
- 2.9V 涓流充电门限
- 软启动限制了浪涌电流

可以根据实际情况，调整PROG接口的电阻阻值，用以修改充电电流：

2、充电电流的设定

充电电流是采用一个连接在 PROG 引脚与地之间的电阻器来设定的。电流充电电流是PROG 引脚输出电流的 1000 倍。设定电阻器和充电电流采用下列公式来计算：

$$R_{PROG}=1000V/I_{CHG}, \quad I_{CHG}=1000V/R_{PROG}$$

从BAT引脚输出的充电电流可通过监视PROG 引脚电压随时确定，公式如下：

$$I_{BAT}=1000X\ V_{PROG}/R_{PROG}$$

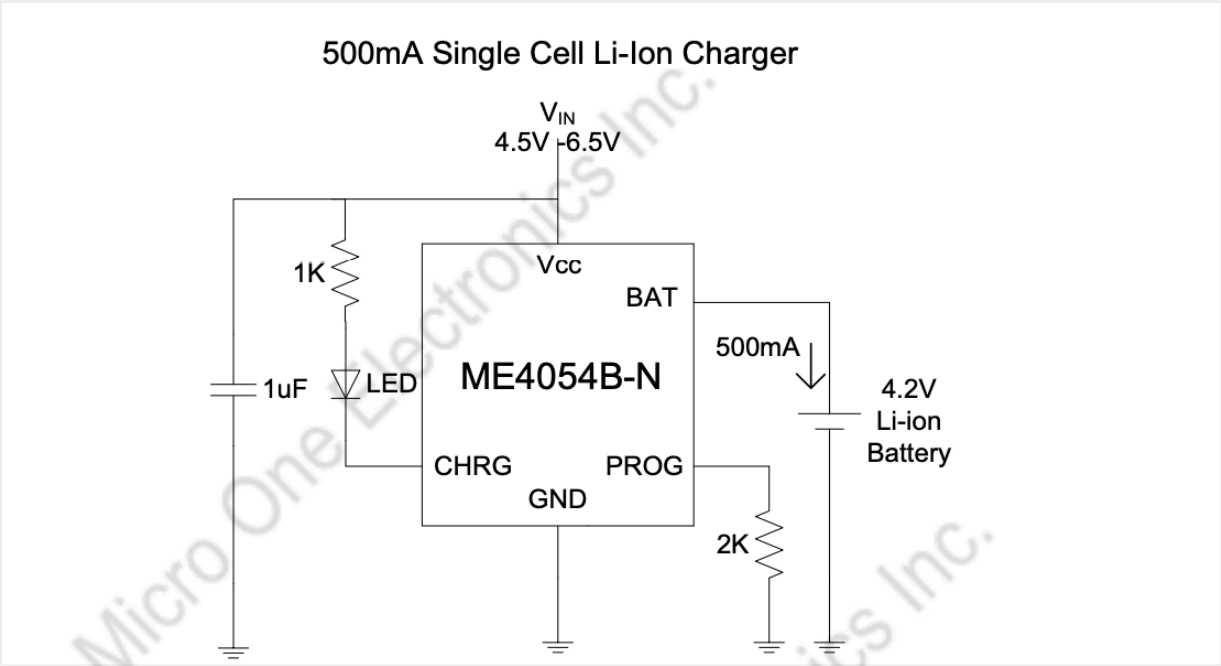
I_{BAT} (mA)	50	100	500
R_{PROG} (KΩ)	20	10	2

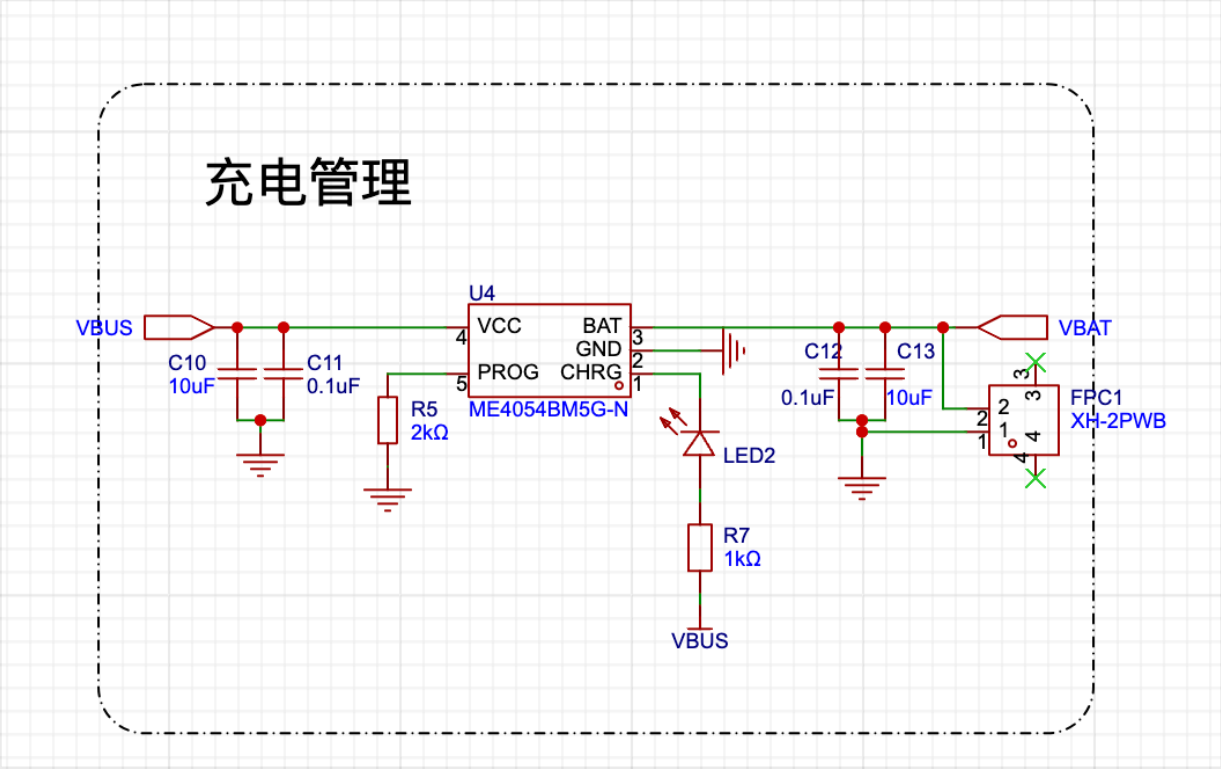
现在可以绘制充电管理的schematics

LED一般在10到30mA左右，LED是diode，所以我们可以选择一个合适的电阻，resistance来防止led破坏

但是datasheet上面写了就不用我们自己找了，1k

这个resistor连接到我们的vbus，用于usb-c接口





下一步是： 电源管理（Battery Management)

USBC输入电源是5V而电脑连接到STM32所需要的是3.3V电压，所以需要绘制转换图，确保不会伤害到芯片 并且是500mA以上的输出电流（current）

选择一个稳压电路，voltage stablizer选择上其实很flexible，只要达到目的就好并不需要独特的

我们选择的是这个： https://item.szlcsc.com/418424.html

芯片&外设需求电源：3.3V

需求：ESP32芯片需要3.3V 500mA以上的电源进行供电。

而单节锂电池提供的电压是3.7-4.2，并不满足电源需求，所以需要添加一个稳压电路，经典的稳压电路像AMS1117。那项目中为什么不使用AMS1117-3.3?

如果使用AMS1117（1.3V@（1A)），电池电压在3.7V时输出的电压将会达不到3V。

XC6210B332MR压差小（100mV@(200mA)），适合电池与低压差供电的场合，输出电流大，额定电流达900mA。电池供电电压在3.7V-4.2V，

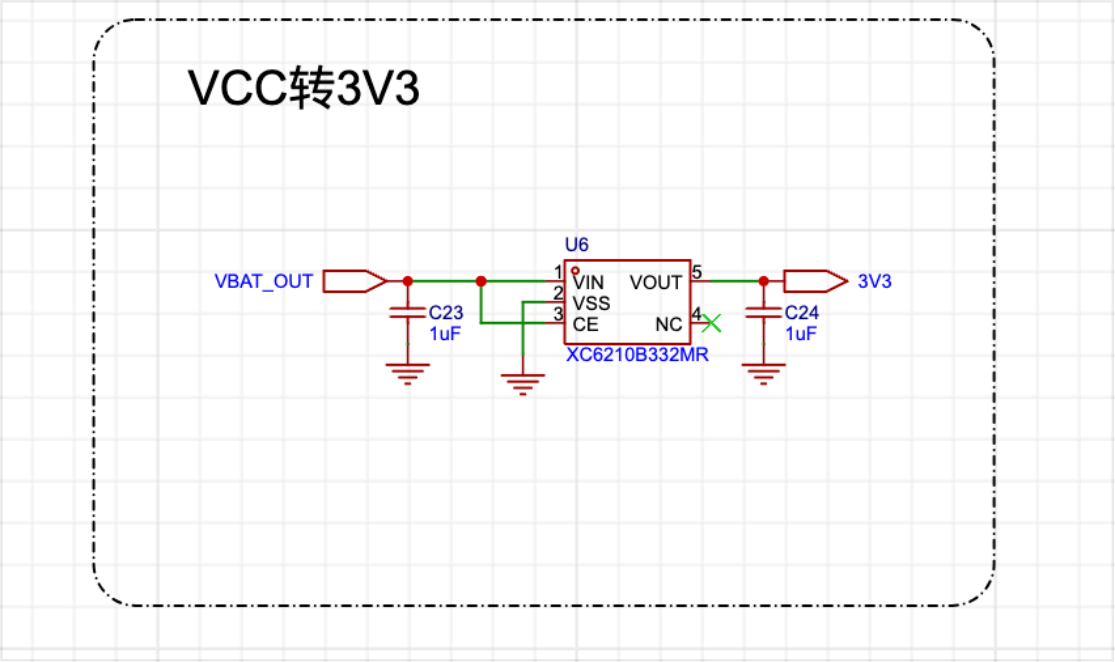
General Description

AMS1117 is a series of low dropout three-terminal regulators with a dropout of 1.3V at 1A load current. AMS1117 features a very low standby current 2mA compared to 5mA of competitor.

■FEATURES

Maximum Output Current	: More than 700mA (800mA limit, TYP.) (1.60V≤V_{OUT}≤5.00V)
Dropout Voltage	50mV @ 100mA 100mV @ 200mA
Operating Voltage Range	: 1.50V ~ 6.00V
Output Voltage Range	: 0.80V ~ 5.00V (0.05V increments)
Highly Accurate	: ±2% (1.55V≦V_{OUT}≦5.00V) ±30mV (0.80V≦V_{OUT}≦1.50V)
Low Power Consumption	: 35μA (TYP.)
High Ripple Rejection	: 60dB @1kHz

我们不用AMS1117是因为他的压差很大，3.7V的时候压差就不够3.3V了，所以我们需要一个压差小一点的Voltage Stableizer



电量检测

三、电量检测

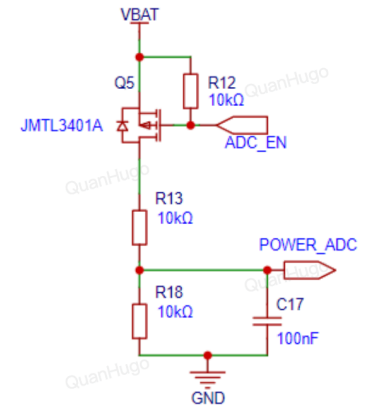
需求：因为是电池供电的产品，我们需要知道电池剩余电量是多少，所以需要添加一个电量检测电路。

电量检测的技术原理是使用电阻分压原理，把电池电压分压后，通过芯片ADC进行读取，和电位器读取原理类似，最终再通过ADC值转换出电压值，电池电压在缺电3.3V，满电4.2V，这样我们就可以按比例估算出剩余的电量。

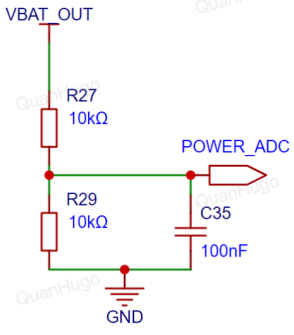
另外，为了节省功耗，可以在电池端添加一个MOS管，控制检测的开关，这样，在不需要读取电量时，就可以关闭MOS管，降低功耗，**(新版本添加了开关后，不需要MOS管开关模块，VBAT经过开关后接R13即可)**。

这里使用PMOS控制开关:当ADC_EN为低电平，此时PMOS导通，可以读取POWER_ADC的值。

https://item.szlcsc.com/372770.html



旧版本



新版本

其实这里使用的是P mosfet的原理， when the battery is 3.7V, which is low battery voltage, the P Mos opens up the gate, allowing voltage to pass through and

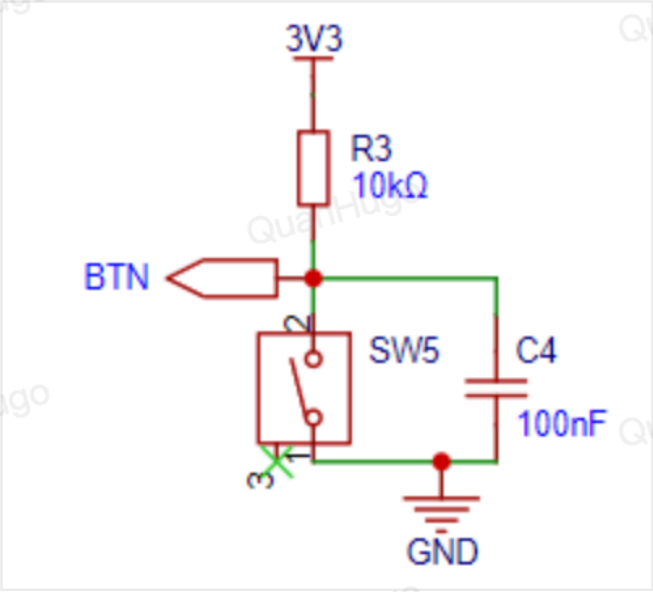
checks the battery through the POWER_ADC and the voltage divider across two same resistors and detect it through power)ADC, which is analog to digital signal, cuss voltage is analog signal, and STM32 uses digital signal

The extra Pmos is used for when battery is full, we don't want it working and wasting power, so we use that to cut off the power, and it just acts as a gate

We don't actually need the extra Pmos because we do have a specific switch for opening and closing; thus, battery goes through the switch then the battery detection

The extra resistor, R12 sets the operating point for gate detection

按键



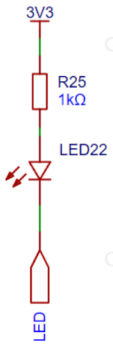
LED:

五、LED指示灯

需求：因为mini printer是没有屏幕的设备，在设备工作过程中，我们想知道设备的运行状态，另外也记录有概率会产生的一些错误。

这时候我们可以添加一颗LED灯，LED有常亮，慢闪，快闪，暗几种状态，于是我们便可以用指示灯来表达设备的工作状态。例如：

状态	灯光
待机	暗
蓝牙连接	常亮
运行异常	快闪
打印中	慢闪



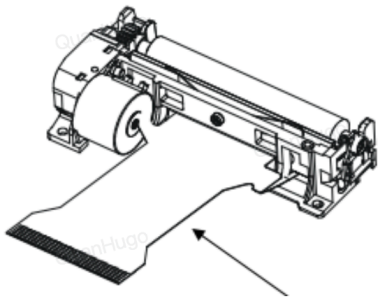
LED is diode, so we just pick the appropriate resistor

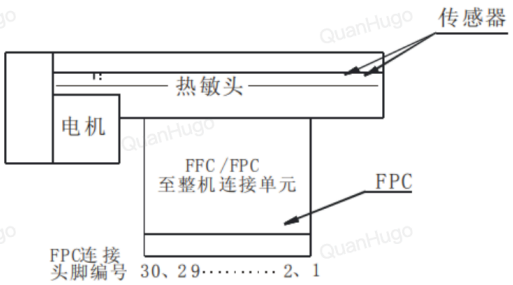
Comes from 3v3 because it is after we stabilize the battery voltage to provide voltage to the LED

打印模组绘制

需要找一个打印头，打印头是有电机，热敏头，传感器,还有fpc

1_JX-2R-01微型热敏打印机芯规格书-V1.5.pdf
490.57KB





打印模组绘制

打印的时候需要热敏电阻驱动，因为打印的时候过热的话会导致出问题，所以我们可以通过Thermistor去做温度检测

通过打印头的datasheet确认IO

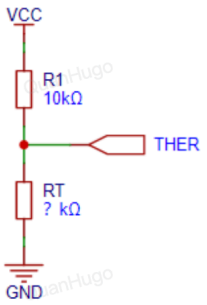
6-1、热敏电阻驱动

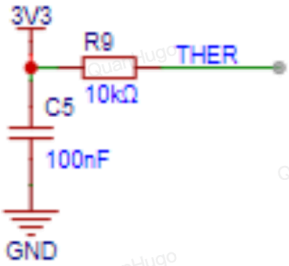
需求：打印模组在工作时，原理是通过加热的方式让热敏纸变色的，如果长时间一直加热，会导致打印头有损坏风险，所以需要通过热敏电阻做温度监测。

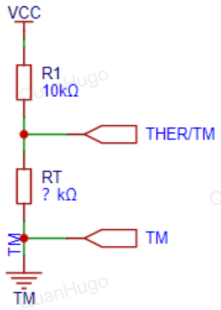
打印头中内置了热敏电阻来监测温度，由于不需要高精度的温度测量，那我们要做的其实就是通过串联一个电阻（普通电阻即可，需要精度高的可以选1%精度的），然后通过分压的公式算出热敏电阻的阻值，最后通过下方计算公式或者RT表转换出实际温度即可。

所以电路特别简单，TM一端接地，另一端接THER口（10k电阻到电源），然后读取THER端的ADC值。

16	TM	热敏电阻端
17	TM	热敏电阻端







中间的是热敏电阻的电路图， 加了一个filter去过滤掉嘈杂的频率，左边图里的RT就是热敏电阻，通过分压来就算出阻值，右边的图就是加上TM的两端，和左边的图一样

THER连到芯片

TM TM from 打印头datasheet 连接到打印头的schematics THER还有GND

缺纸检测

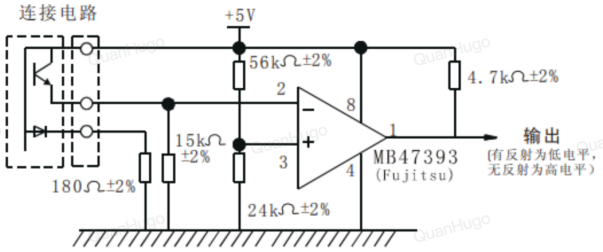
6-2、缺纸侦测

需求：打印模块在工作时，需要放入打印纸，如果没有打印纸，会导致打印异常，所以需要用到缺纸检测模块。

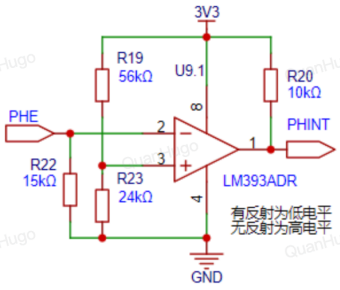
打印机芯内置了一个反射性光电通断侦测传感器，用于缺纸检测。

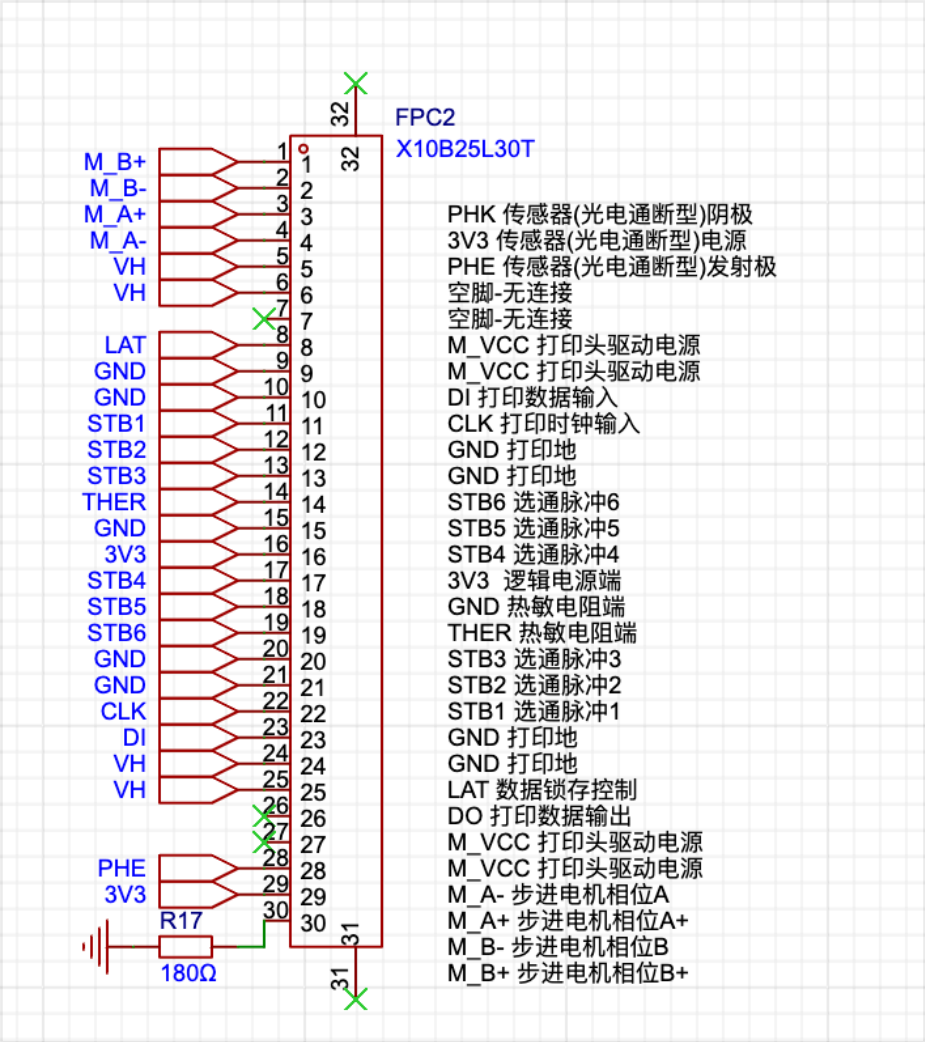
如下图所示，左边是光电传感器、中间是比较电路、右边是输出。

当缺纸时，光电侦测发出的光无法被反射，输出高电平。当纸张正常，光电侦测发出的光被反射，由接收管接收，输出低电平，光电开关的电路驱动，如下图所示，当缺纸时，不要启动打印机加热。



比较器使用的芯片是LM393：C404321





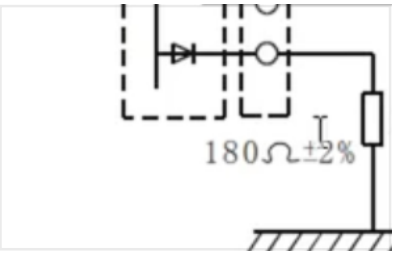
The knowledge here uses a comparator to test the missing of papers,

输出到我们的芯片里面通过芯片来读这个传感器， 然而PHE是连接到我们的打印头那里的

3	PHE	传感器（光电通断型）发射极

还有GND，就是阴极，还有电源都得连接到打印头的schematics那里

1	PHK	传感器（光电通断型）阴极
2	VSEN	传感器（光电通断型）电源



GND需要一个180Ohm的地

电机

需求：打印机的原理是逐行打印，当打印完一行后，需要使用电机拖拽打印纸往前行进一行，以完成多行的打印任务。

打印机芯用的是一个额定电压为3.5-8.5V的两相步机电机，推荐使用L3967、LB1836进行驱动，但是因为这两款芯片太贵了，我们使用TC1508替代。

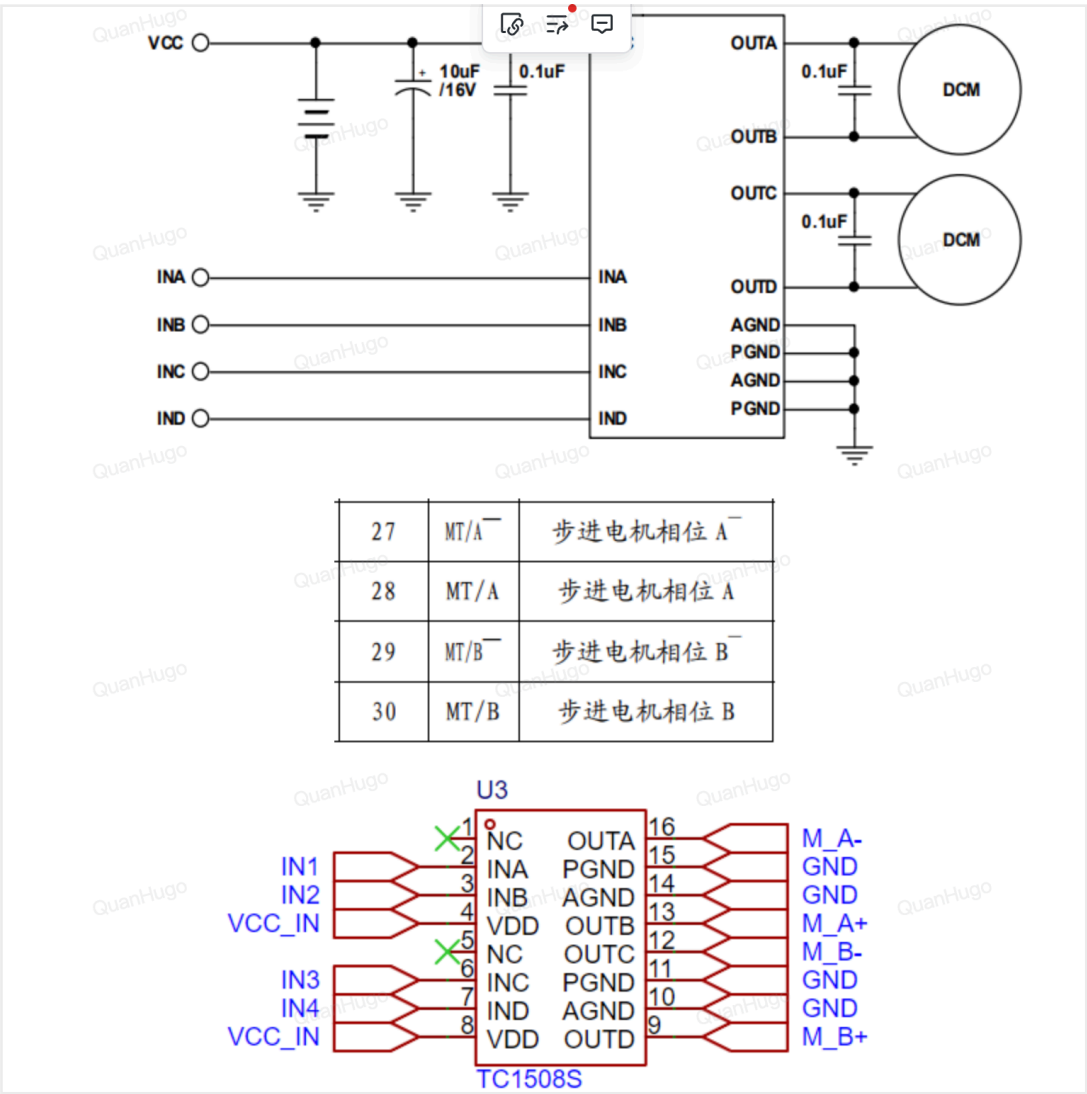
4、步进电机

步进电机每前进一步，纸张前进 0.0625mm.

4.1、步进电机参数

表 10

项 目	规 格	条 件
额定电压	3.5~8.5 DCV	
相 位	2 相	
步距角	9° （1-2 相激励）	
步进距离	0.0625 毫米	
相电阻	10 Ω ± 7%	20℃
相电流	0.357 A	
驱动方式	双极，双相驱动（或 1~2 相）	



27	MT/A‾	步进电机相位 A‾
28	MT/A	步进电机相位 A
29	MT/B‾	步进电机相位 B‾
30	MT/B	步进电机相位 B

M_A+-， M_B+- 对应的是打印级datasheet打印头里面的27,28,29,30

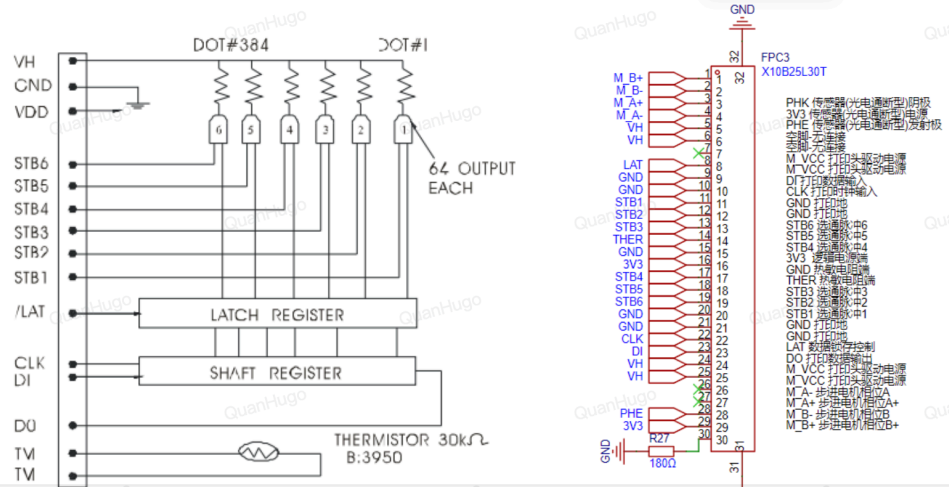
INABCD就是对应IN1234，这些是IO里面的IN1234

剩下的打印头接口就是通过VH，STB的接口来

消耗功率		P。	0.27W/dot	Rav=176Ω, Vdd=5V, 同时打印点数：64 点
供给电压		VH	7.2V	
循环打印		S.L.T	1.25ms/line	
消耗能量	5℃	E。(Ton)	0.16mJ/dot (0.60ms)	64 点同时加热
	25℃		0.13mJ/dot (0.49ms)	
	45℃		0.11mJ/dot (0.41ms)	
消耗电流		I。	2.5A	

STB1-6:通道选择器，接到主控芯片的IO接口。

LAT CLK DI：锁存、时钟、数据线，接到主控的IO接口



The clk sends the data based on certain frequencies, to the shaft register, then they can sent to the latch register, where it will hold it's input until everything is sent and ready to go, then the latch register will send out all the 384 dots for printing

12	STB6	选通脉冲 6	27	MT/$\overline{A}$	步进电机相位 $\overline{A}$
13	STB5	选通脉冲 5	28	MT/A	步进电机相位 A
14	STB4	选通脉冲 4	29	MT/$\overline{B}$	步进电机相位 $\overline{B}$

Vh is 7.2V 得转升压

3.3、推荐参数

表 4

项　目		代 号	电气参数	条　件
消耗功率		P。	0.27W/dot	Rav=176Ω, Vdd=5V, 同时打印点数：64 点
供给电压		VH	7.2V	
循环打印		S.L.T	1.25ms/line	
消耗能量	5℃	E。(Ton)	0.16mJ/dot (0.60ms)	64 点同时加热
	25℃		0.13mJ/dot (0.49ms)	
	45℃		0.11mJ/dot (0.41ms)	
消耗电流		I。	2.5A	

规格书中，打印头一行有384个点，分6个通道控制，每个通道64个点，在同一时间只能加热一个通道，所以同一时间的最大功率是0.27x64W，当供电电压在7.2V，此时电流=(0.27x64)/7.2，接近2.5A，所以需要我们提供一个7.2V，>2.5A的电流的电源。

因为模块使用单个电池供电，电压范围在3.7-4.2V，所以需要选用升压的模块，项目中使用：AP2005

https://item.szlcsc.com/141627.html

输入电压	2.5V~5.5V
输出电压	9V
输出电流	4.5A

输出电压可达到9V可调节，而输出电流则是通过64点每个通道x0.27w每个点除于7.2V最大电压来看看输出最大电流要多少

input is

2、STM32的最小系统绘制

STM32 的供电在2V到3.6V，所以我们可以直接使用电池供电

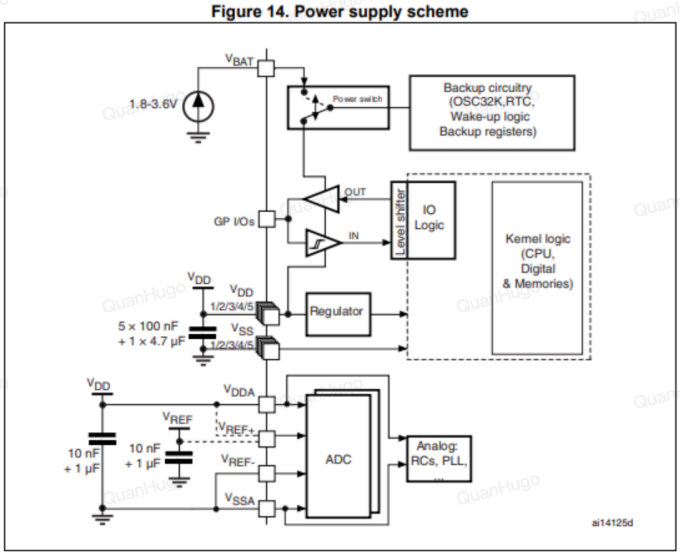
2 Description

The STM32F103xx medium-density performance line family incorporates the high-performance ARM Cortex™-M3 32-bit RISC core operating at a 72 MHz frequency, high-speed embedded memories (Flash memory up to 128 Kbytes and SRAM up to 20 Kbytes), and an extensive range of enhanced I/Os and peripherals connected to two APB buses. All devices offer two 12-bit ADCs, three general purpose 16-bit timers plus one PWM timer, as well as standard and advanced communication interfaces: up to two I^2Cs and SPIs, three USARTs, an USB and a CAN.

The devices operate from a 2.0 to 3.6 V power supply. They are available in both the –40 to +85 °C temperature range and the –40 to +105 °C extended temperature range. A comprehensive set of power-saving mode allows the design of low-power applications.

然后参考芯片手册，如下图，把VBAT VDD VDDA连接到3.3V，VSS连接到GND，为每路电源添加旁路电容，靠近芯片IC放置，确保每路电源干净、纹波小。

Figure 14. Power supply scheme



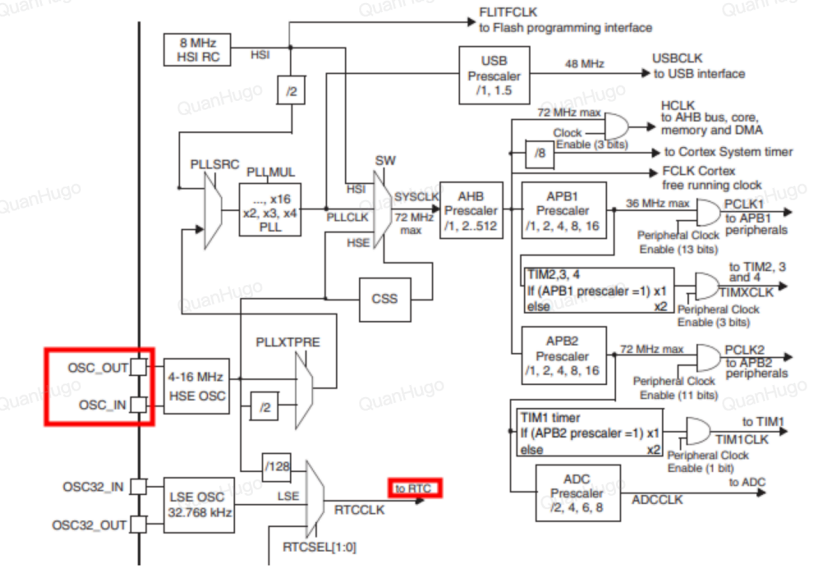
VBAT ， VDD，VDDA，FROM 芯片 到3.3V

VSS 到GND

2-2、然后是外部时钟

这里我们选择常规的8MHz的外部时钟作为高速时钟，由于32.768kHz的时钟只作为RTC使用，而我们项目没有用到RTC功能，所以可以不接。

Figure 2. Clock tree



A real time clock, or RTC, is a digital clock with a primary function to keep accurate track of time even when a power supply is turned off or a device is placed in low power mode.What is a Real Time Clock (RTC)? A real time clock, or RTC, is a digital clock with a primary function to keep accurate track of time even when a power supply is turned off or a device is placed in low power mode.

2-3、然后是启动模式

STM32支持多种启动模式，可以通过IO设置BOOT0和BOOT1的电平，让它从不同地方的代码启动，我们设置为Flash启动即可

2.3.8 Boot modes

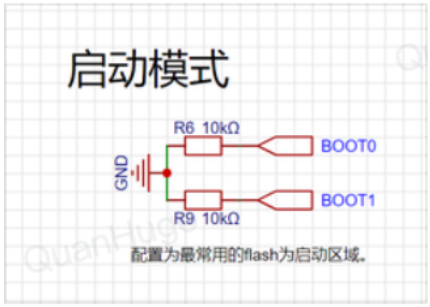
At startup, boot pins are used to select one of three boot options:

- Boot from User Flash

简单来说，大家要想正常跑程序就要把Boot0和Boot1都接到地，就是正常工作模式。

BOOT1	BOOT0	
X	0	用户闪存启动，正常工作模式
0	1	系统存储器启动，厂家设置
1	1	内置SRAM启动，用于调试

Use two 10k resistors to serve as a pull down resistors to keep the pins of boot0 and boot1 at a low logic level, so when they are not used, no other voltage can pass through it



2-4、最后是下载和烧录模式

芯片支持SWD和JTAG进行烧录，SWD接口只需要SWDIO和SWCLK两个IO，非常方便，我们直接选择SWD即可，

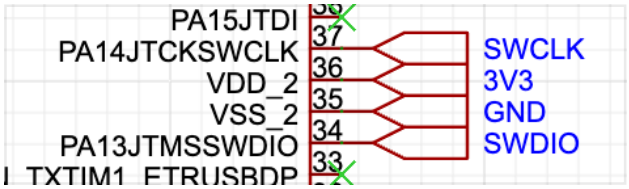
- Debug mode
  - Serial wire debug (SWD) & JTAG interfaces

2.3.24 Serial wire JTAG debug port (SWJ-DP)

The ARM SWJ-DP Interface is embedded. and is a combined JTAG and serial wire debug port that enables either a serial wire debug or a JTAG probe to be connected to the target. The JTAG TMS and TCK pins are shared with SWDIO and SWCLK, respectively, and a specific sequence on the TMS pin is used to switch between JTAG-DP and SW-DP.

烧录就是程序员写好的程序，把程序导入到目标IC上面，实行一个完整的动作。

我们使用SWD的话就可以直接把SWDIO还有SWCLK两个IO连接到芯片上面:



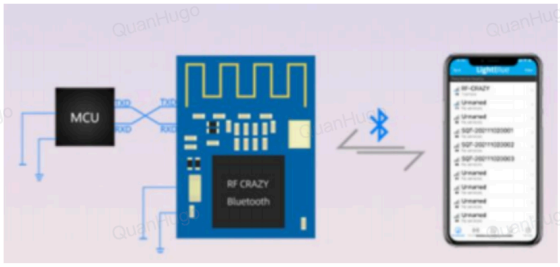
3、BLE透传模组

由于STM32是没有蓝牙功能的，而我们需要和手机进行通讯，这里我们使用了蓝牙透传模组，通过串口与STM32进行数据透传。

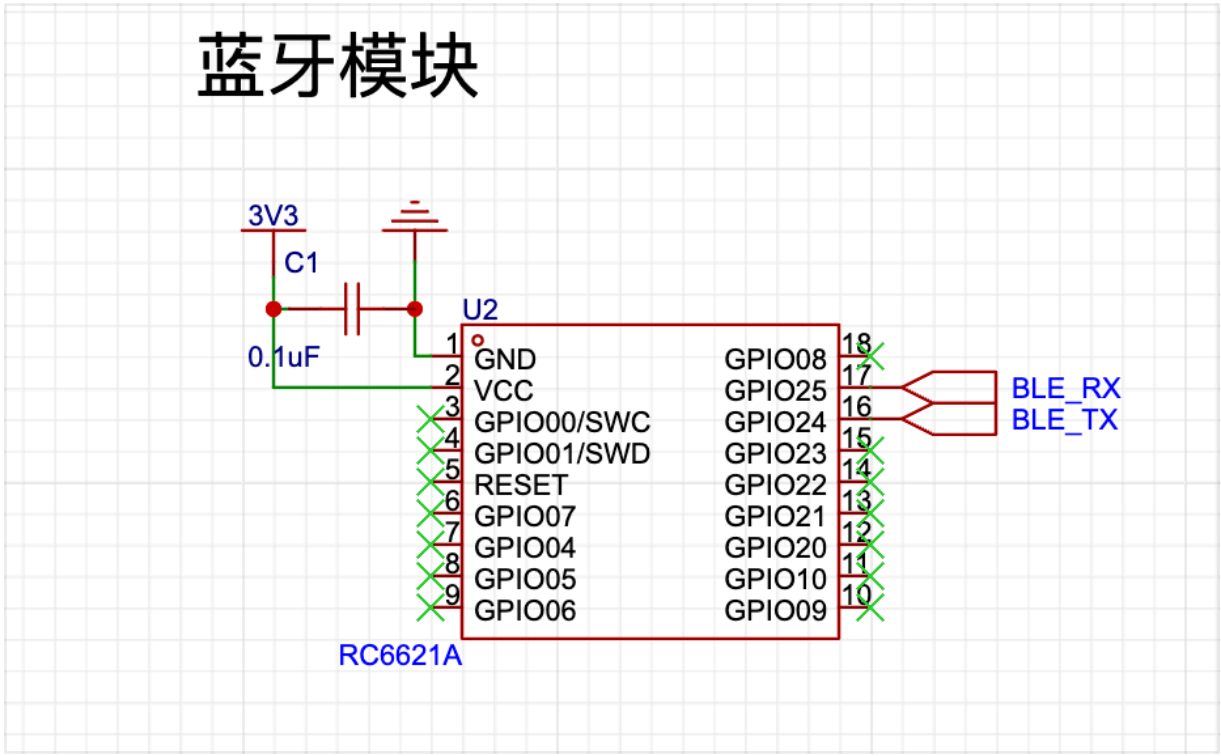
蓝牙部分我们使用串口2进行数据交互，所以需要重写串口2的处理函数

我们使用的蓝牙模组是RF Crazy/智汉的模组，主要是立创上购买方便，大家也可以根据自己需要选择其它模组。

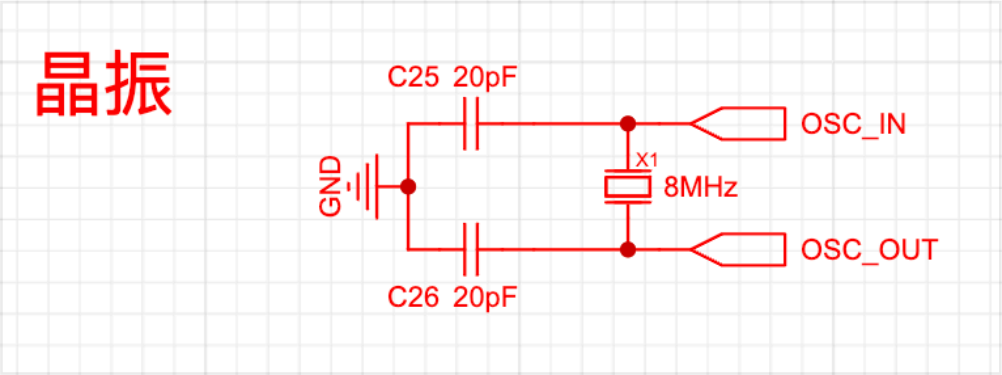
模组的连接非常简单，只需要提供电源和地，然后串口连接STM32的串口即可通讯。



C2886280_蓝牙模块_RC6621A_规格书_RF+CRAZY(智汉)蓝牙模块规...
2.12MB



The capacitor is for bypassing and filtering frequencies that are high before reaching to the bluetooth module



1. **Crystal Operation:**
    - The crystal (X1) in the circuit is a piezoelectric device that mechanically vibrates at a specific frequency (8 MHz in this case) when an electric field is applied.
    - These vibrations generate an oscillating electrical signal at the same frequency.
2. **Capacitors Role:**
    - Capacitors C25 and C26 form a feedback network with the crystal.
    - They create a phase shift and ensure the loop gain is adequate to sustain oscillations.
    - These capacitors help in achieving the necessary conditions for the crystal to start oscillating and maintain stable oscillations.
3. **Oscillation Process:**
    - When power is applied to the circuit, the amplifier (usually integrated into the microcontroller or an external IC, not shown in the schematic) amplifies the noise present in the circuit.
    - This amplified noise is fed back through the crystal and capacitors.
    - The crystal filters this noise, allowing only the 8 MHz signal to pass through and be reinforced.
    - This feedback loop continues, causing the circuit to oscillate at the crystal's resonant frequency (8 MHz).
4. **Signal Output:**
    - The resulting stable 8 MHz oscillating signal is available at the OSC_OUT pin.
    - This signal can then be used to clock digital circuits, such as microcontrollers, providing a

At low frequency, it allows only 8MHZ frequency to pass through and at high frequencies, they are filtered out through the filters from capacitors. And 20pF is used because for a 8MHZ it is usually between 18 to 22 pF

4、把芯片IO和外设连接起来

有了芯片和外设，那芯片如何控制外设或者读取外设状态呢？这里就需要把芯片IO和外设连接起来了。

前面我们已经绘制完最小系统和外设部分，那怎么把把外设接到STM32的IO上了，在第二个章节，我们已经确定了所有的外设分别接到哪些IO上了，所以这里很简单，只需要用标签连接起来即可。

我们可以先list out所有的IO然后通过这个list来连接到我们的芯片上面：

Power and Ground Connections

1. **VBAT (Pin 1):** Connected to a 3V3 supply. This pin powers the RTC and backup registers when VDD is not present.
2. **VSS_A (Pin 8), VSS_1 (Pin 23), VSS_2 (Pin 35), VSS_3 (Pin 47):** Connected to GND. These are ground pins.
3. **VDD_A (Pin 9), VDD_1 (Pin 24), VDD_2 (Pin 34), VDD_3 (Pin 48):** Connected to 3V3. These are the main power supply pins for the microcontroller.

Clock Pins

4. **OSC_IN (Pin 5), OSC_OUT (Pin 6):** Connected to the external crystal oscillator. These are necessary for providing a clock signal to the MCU.
5. **PD0-OSC_IN (Pin 4), PD1-OSC_OUT (Pin 5):** These are for an optional secondary clock, often a 32.768 kHz crystal for RTC.

Reset and Boot Pins

6. **NRST (Pin 7):** Connected to a reset button. This pin resets the MCU when pulled low.
7. **BOOT0 (Pin 44):** Connected to a switch or jumper. This pin determines the boot mode of the MCU.

Peripheral Pins

8. **USART (Pins 12, 13, 14):**
    - **PA9/PA10:** UART1 TX and RX for serial communication.
    - **PA2/PA3:** UART2 TX and RX for serial communication.
9. **SPI (Pins 15, 16, 17, 18, 19, 21):**
    - **PA5, PA6, PA7:** SPI1 SCK, MISO, MOSI for SPI communication.
10. **I2C (Pins 27, 28):**
    - **PB6, PB7:** I2C1 SCL and SDA for I2C communication.
11. **ADC (Various Pins):**
    - **PA0, PA1:** Analog input for ADC.
    - **PB0, PB1:** Additional analog inputs.
12. **PWM/Timer (Various Pins):**
    - **PA8, PA9:** Timer channels for PWM output.
13. **General GPIO:**
    - Various other pins like PB12, PB13, etc., are used as general-purpose

IO pins for connecting LEDs, buttons, etc.

Connectivity Pins

14. **SWD (Pins 36, 38, 37, 32):**
    - **PA13, PA14, PB3:** Used for programming and debugging the MCU.
15. **Other Interfaces:**
    - **CH340_RX, CH340_TX (Pins 30, 31):** These are for UART to USB communication via CH340 IC.

Possible Reassignments

Most IOs on the STM32 can be remapped to different pins using the alternate function remap registers. However, some pins have fixed functions:

- **Power and Ground:** Cannot be changed.
- **OSC and NRST:** Generally should not be changed for stability.
- **BOOT0:** Should remain fixed for reliable booting.

For other peripherals, you can change their connections depending on your application and PCB layout, as long as the chosen pins support the desired function.

Summary

- **Power/GND, Clock, and Reset pins are generally fixed.**
- **Peripheral pins can often be remapped to different IOs using the MCU's remapping capabilities.**
- **Some pins like BOOT0, SWD, and primary oscillator pins should remain as recommended for proper operation.**

Actul Connection:

POWER and GND:

3v3 to VBAT, VDD, VDDA
GND to VSS
VBATOUT, battery output is not connected to the STM32 as

CLOCK-PINS:

OSC_IN and OCS_OUT connect to OSC_IN (Pin 5), OSC_OUT (Pin 6)

Reset and Boot Pins:

NRST (Pin 7): Connected to a reset button. This pin resets the MCU when pulled low.
BOOT0 (Pin 44): Connected to a switch or jumper. This pin determines the boot mode of the MCU.

Peripherals: peripheral pins can usually be remapped depending on the STM32's requirements

IN1234 is from 电机驱动，uses GPIO pins

Timer Pins and Motor Control

1. **Pulse Width Modulation (PWM):**
   - Motor speed and direction are often controlled using PWM signals.
   - Timers in the STM32 microcontroller can generate PWM signals, which are used to control the duty cycle of the motor's power input, thereby adjusting its speed.
2. **Precise Timing Control:**
   - Timers allow for precise control over the timing of signals, which is crucial for motor control applications.
   - Using timers, you can generate consistent and accurate PWM signals necessary for stable motor operation.
3. **Synchronized Control:**
   - When controlling motors, especially in applications like robotics or CNC machines, synchronized signals are essential.
   - Timers can synchronize multiple PWM channels, ensuring coordinated control of motor phases.

MB+- are for printer head connection pins
MA+- are for printer head connection pins
THER, Thermistor can use pin 11 for the adc connection because it needs to convert analog data which is temperature to digital data which can be transferred to uart for connection, so it uses adc communication
POWER_ADC can use pin 10 for adc connection obviously
VCC_IN is not connected to the 32
VH, for printer head's connection, not for stm32
LAT，latch register can use GPIO, pin and most likely uses timer otherwise
STB3456 can use pin 25,26,27,28 as GPIO output
CLK can use pin 15 for spi communication
DI，**DI: data in**

Can use MOSI

LED uses timer signal as it converts digital signals to an analog signals by changing the brightness of the LED, but we can use pin 18 for GPIO because we are not changing brightness, we are purely flashing the LED, that's it
PHE
PHINT can use pin 29 as it can use GPIO pin for communicating interrupts, set it to

external for faster detection
VH_EN from 打印头电源开关 can use pin 32 for GPIO pin output
BTN, button typically uses a **digital signal**. Uses GPIO as an input pin
SWDIO connect to pin SWDIO
SWCLK connect to pin SWCLK
BOOT0 connects to BOOT0
BOOT1 connects to BOOT1
NRST connects to NRST

TX and RX is for UART:

BLERX connect to pin 13 because it is serial communication uart
BLETX pin 12 because it is serial communication so it can use uart

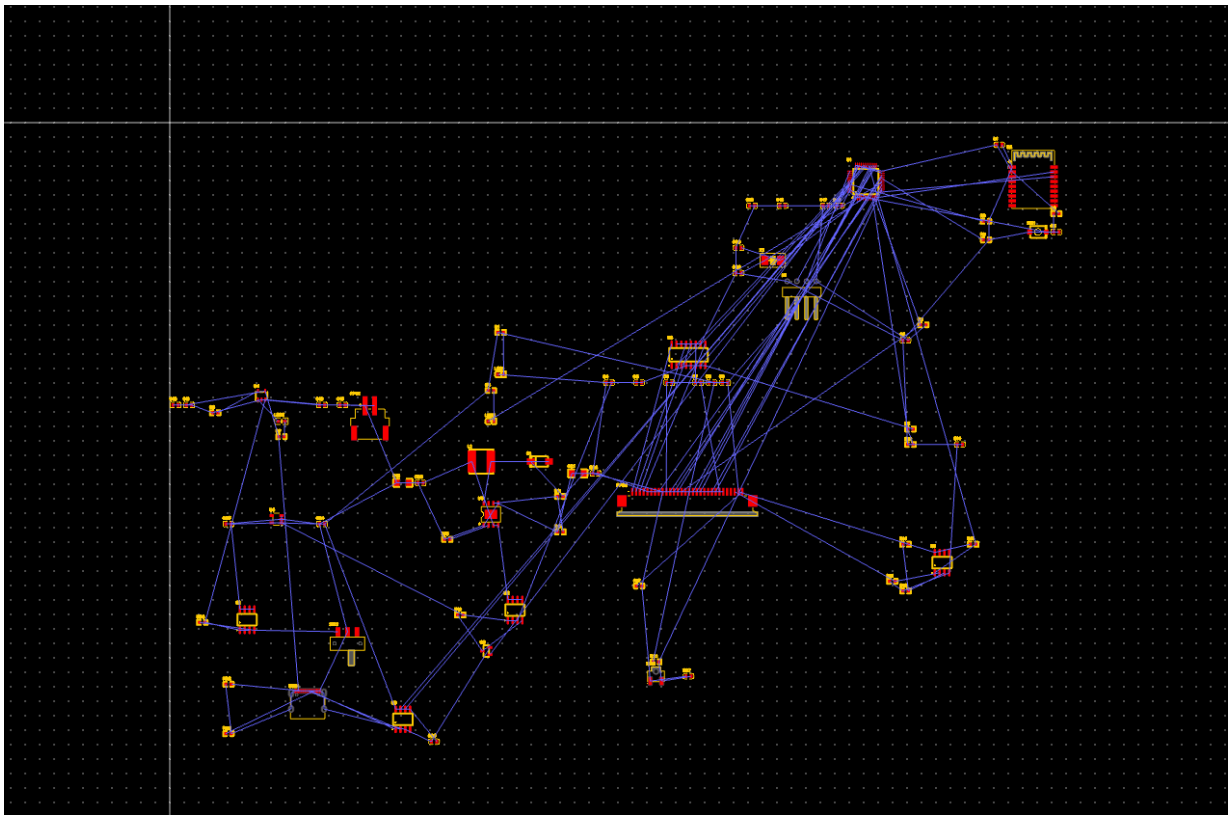
CH340RX, TX is used for UART communication's RX and TX, so pin 30 and 31

PCB绘制

接下来都是pcb绘制了

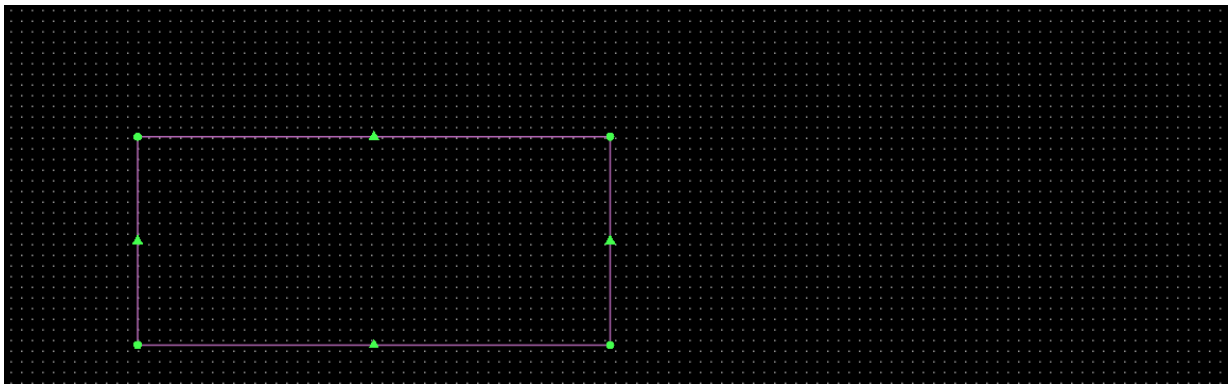
首先我们要检查一下所有的绘制图都是正确的，有没有输出到正确的电路还有芯片上面去

没有问题了之后，我们就可以把他们绘制成pcb了，这里点击一下按钮就可以了：



接下里就是我们要把这个PCB的尺寸设计好，不然的话到时候装机就会出问题

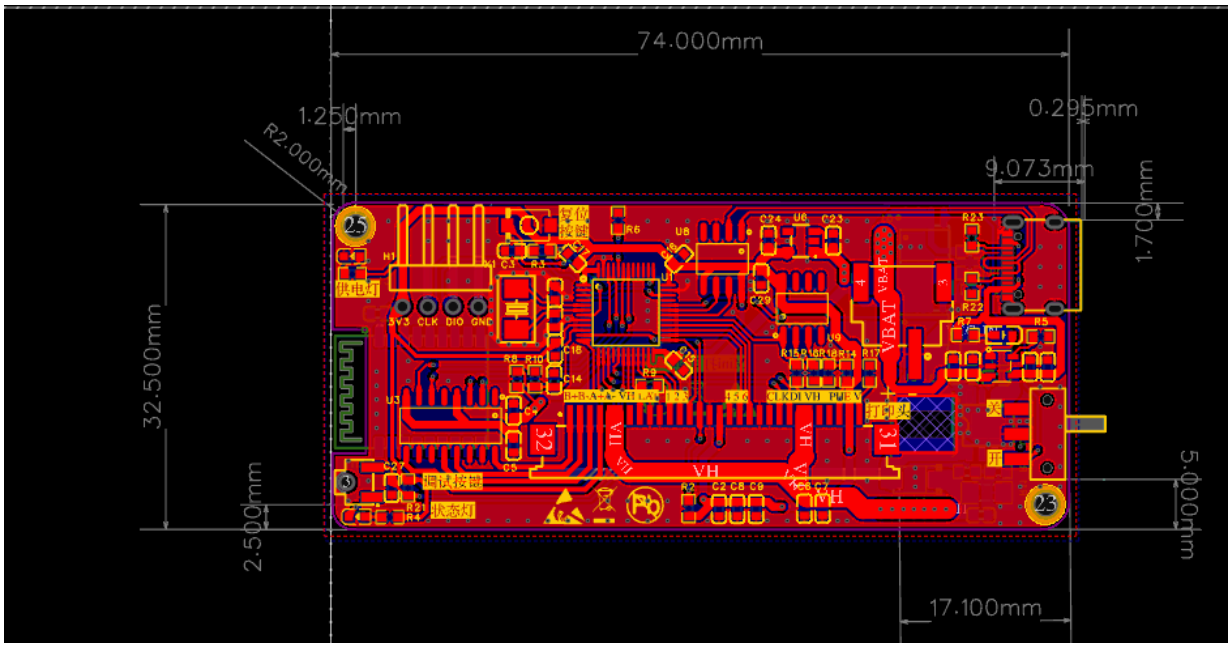
首先绘制一个72x32.5的框：



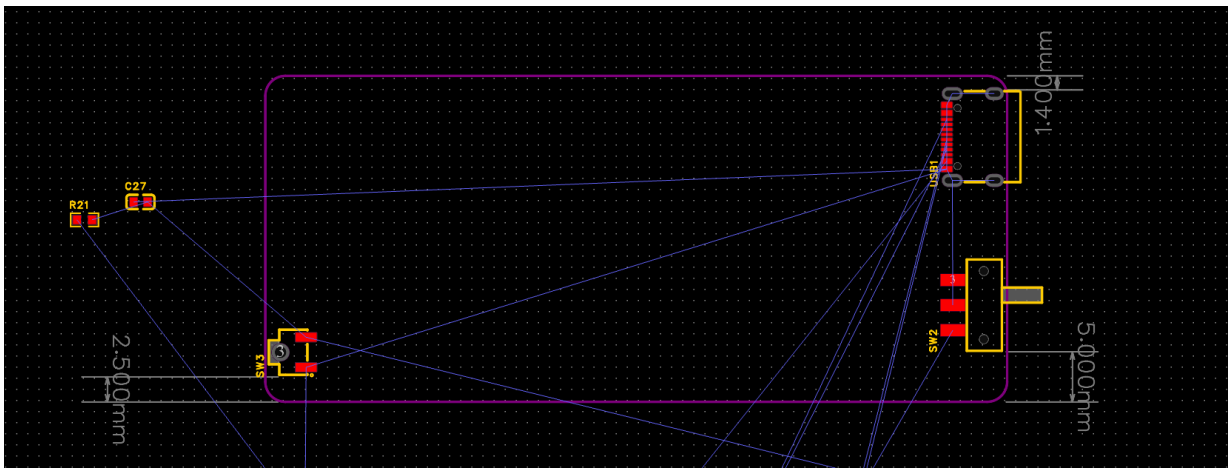
然后添加一个圆角2mm， 点击框框再右键就行

然后根据安装的走向来进行结构的布局，就是

-第一个是项目中的结构已经固定好**TYPEC口位置**，**打印机排线位置**，**开关位置**，**电池端子位置**以及**按键的位置**，所以如果你最终不打算重新设计结构，那你必须按工程中的位置进行摆放。除此之外，其它元器件位置可根据自己需求摆放。



想要更快地找到元器件可以右键然后选择cross prob这样就能在pcb里面快速找到了，像这样

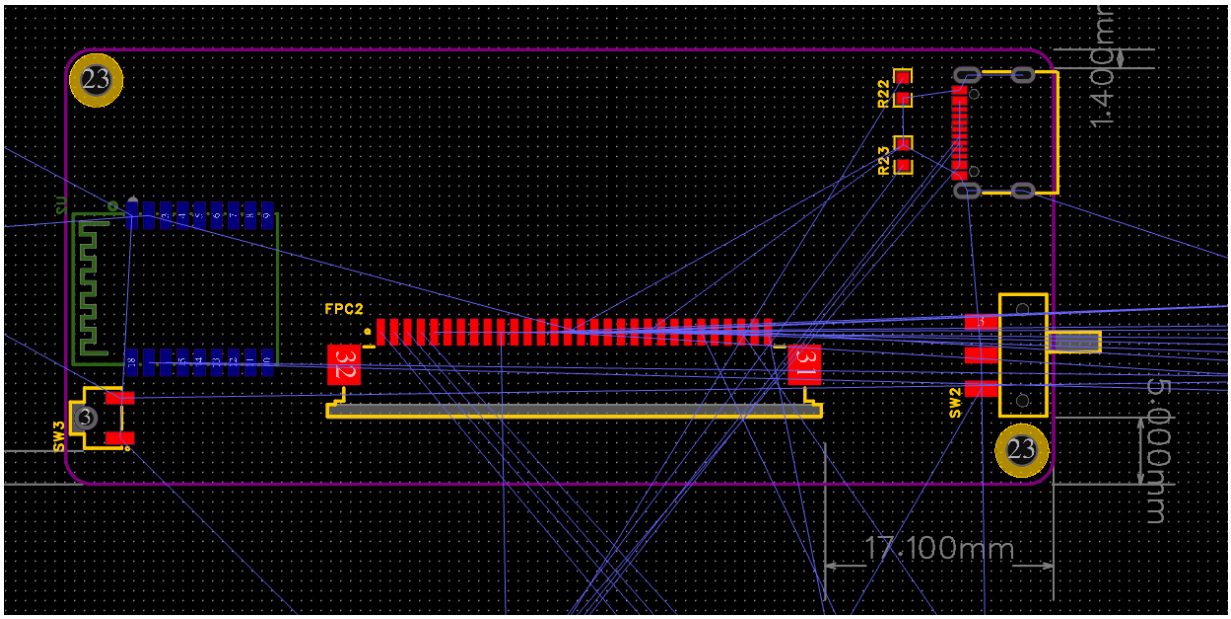


然后，

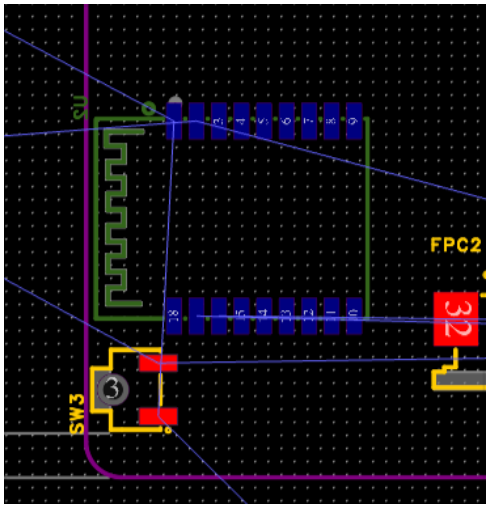
-第二个**PCB的螺丝孔位置**是不能随意更改的，如果你不计划自己设计、修改结构的话。

-第三个是是要注意**模组摆放的位置**，一定要在板卡的边缘，因为模组底部需要掏空，不允许走线，否则会影响信号，所以摆放在PCB边缘，后续好处理。

模组是蓝牙的模组



模组是在底层的



其他的外设就可以进行摆设了：

一起的外设最好是放在一起，这样的话能够很好的组织起来，还有布置的时候最好是把相关的配件把他们放在隔壁，这样排线的时候能够轻松一点：比如说typeC和充电管理

还有一点要注意的话就是：

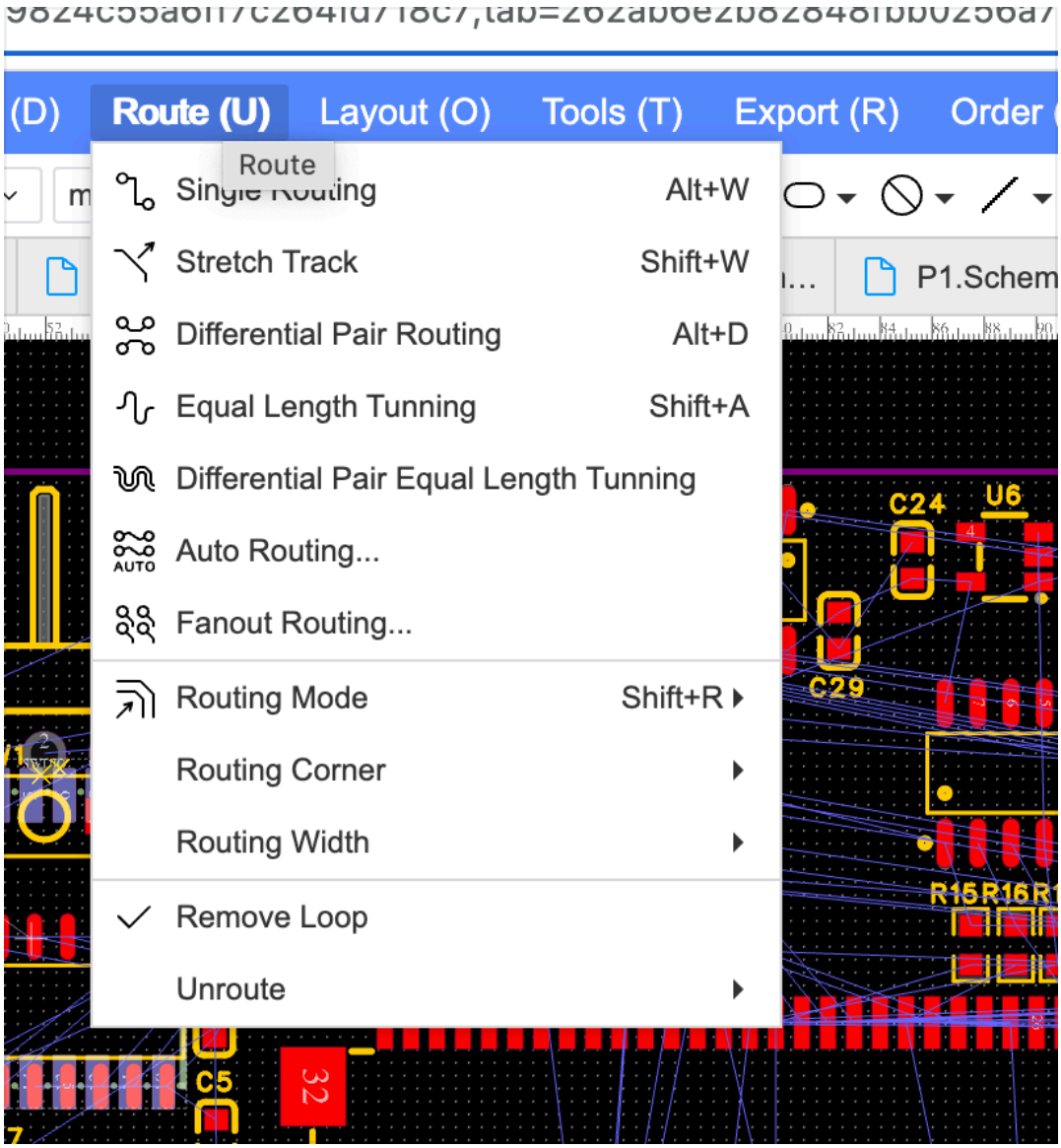
-第四个是**升压模块背面**最好不要走线和摆放元器件，因为这是会产生干扰的地方，所以要预留好这部分的空间。

布线阶段

接下来就是不限阶段了，布线的话就是要通过电容电阻的线路来布线

第一个就是电源的走线，我们选择的是2mm的线路宽度

点击，route，single routing进行布线：



按住tab来改动宽度

其他的就可以改为默认的线宽（0.254）就可以了

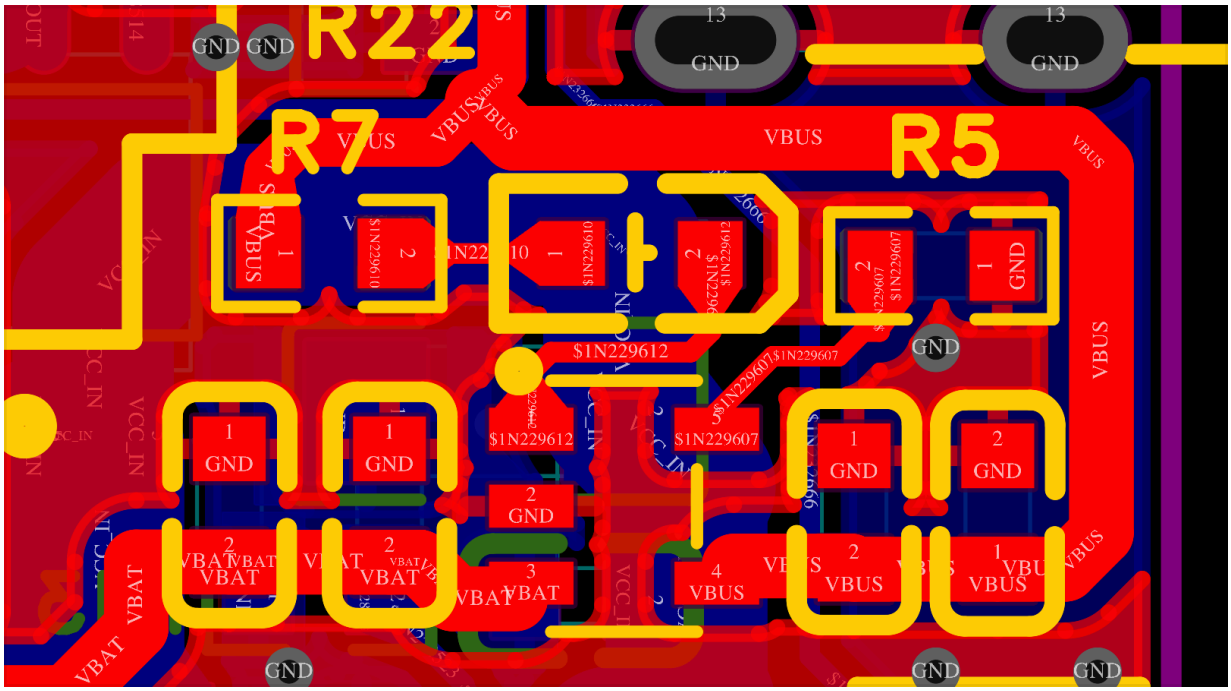
第一个是**电源的走线**，因为打印头工作时最大功率会到20多W，所以电池到打印头部分的走线线宽要粗，满足通过电流。

第二个是升压模块下方**不允许走线**，防止受到干扰影响系统工作。

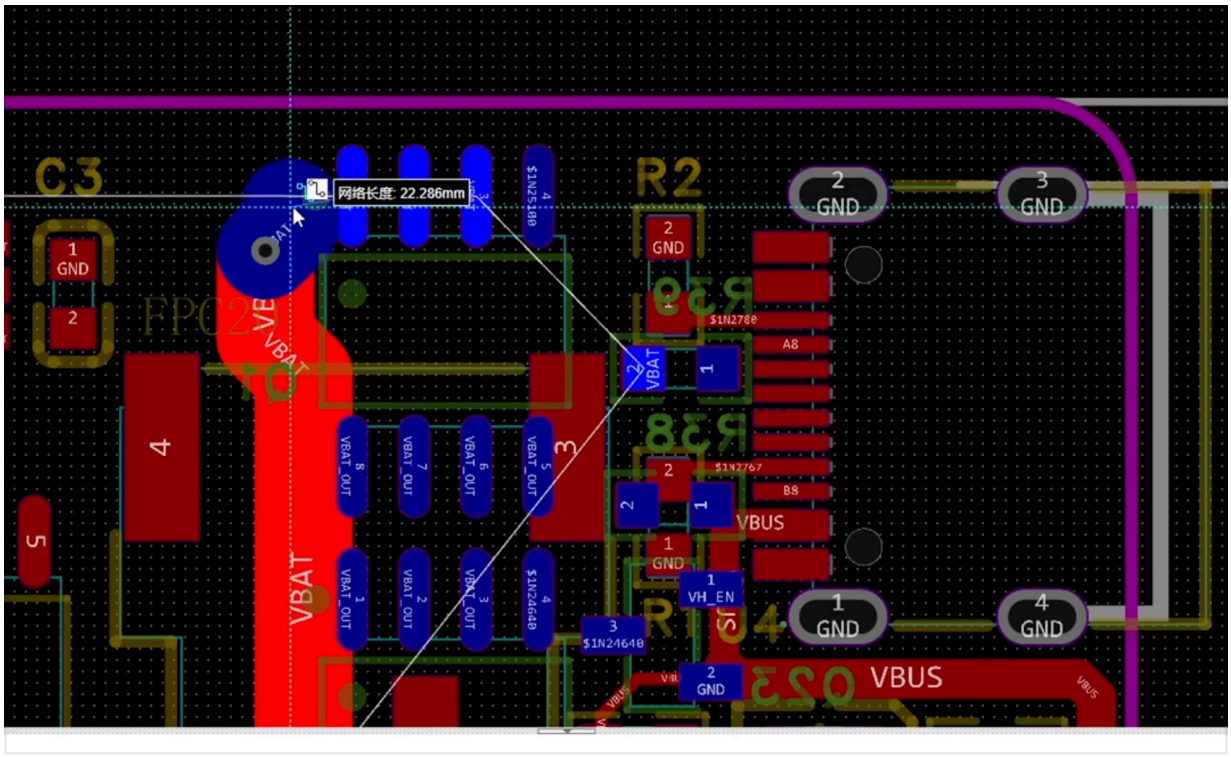
第三个是所有走线都需要**经过旁路电容**再进入模块，避免电容不起作用。

第四个是布线的顺序，一般根据实际情况，**先布置电源或难度高的线，然后再布常规线，地线最后**通过铺铜过孔连接。

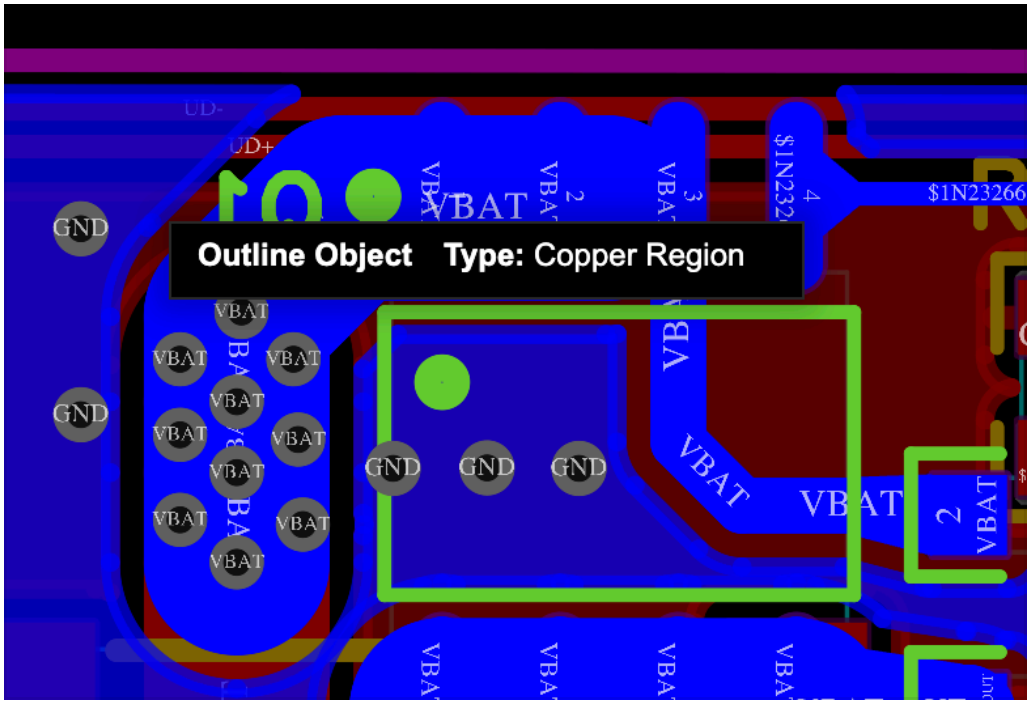
地线可以通过打孔铺铜的方式



如果要连接到底层的话，就得按住alt V，进行过孔连接到底层



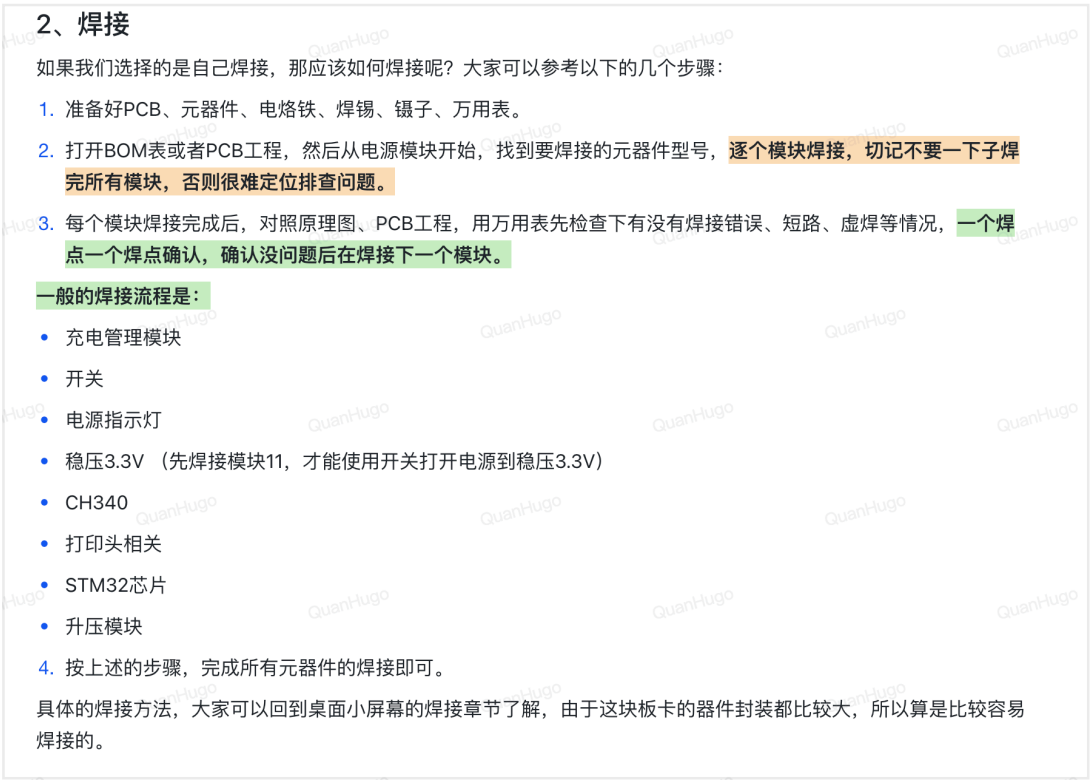
过孔可能一个不够得复制多几个：



下单加焊接阶段

下单的话，我们可以导出制版文件然后到官网进行下单

下单的话就可以全部默认



2、焊接

如果我们选择的是自己焊接，那应该如何焊接呢？大家可以参考以下的几个步骤：

1. 准备好PCB、元器件、电烙铁、焊锡、镊子、万用表。
2. 打开BOM表或者PCB工程，然后从电源模块开始，找到要焊接的元器件型号，**逐个模块焊接，切记不要一下子焊完所有模块，否则很难定位排查问题。**
3. 每个模块焊接完成后，对照原理图、PCB工程，用万用表先检查下有没有焊接错误、短路、虚焊等情况，**一个焊点一个焊点确认，确认没问题后在焊接下一个模块。**

一般的焊接流程是：

- 充电管理模块
- 开关
- 电源指示灯
- 稳压3.3V （先焊接模块11，才能使用开关打开电源到稳压3.3V）
- CH340
- 打印头相关
- STM32芯片
- 升压模块

4. 按上述的步骤，完成所有元器件的焊接即可。

具体的焊接方法，大家可以回到桌面小屏幕的焊接章节了解，由于这块板卡的器件封装都比较大，所以算是比较容易焊接的。

完善STM32CUBEMX的配置

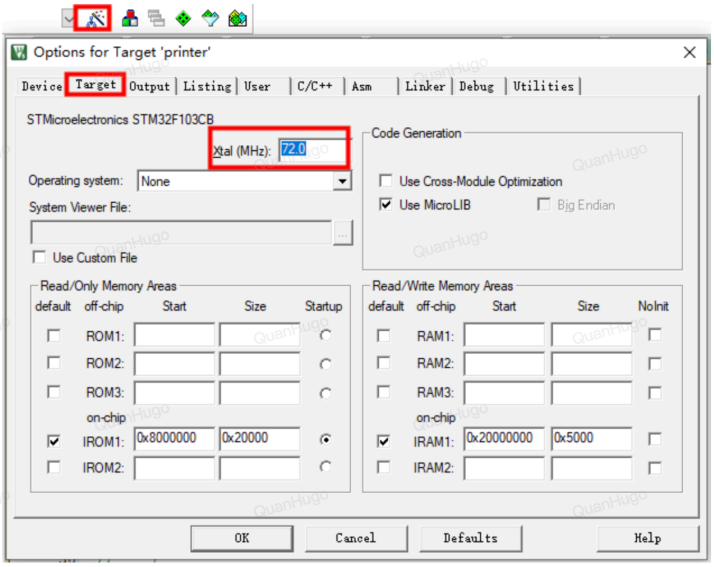
参考：https://x509p6c8to.feishu.cn/docx/RWaHdWTHvoVIYaxNrGUcwCPzn0c

安装KIELMDK然后打开CUBEMX的工程

打开之后进行编译，如果编译成功就会显示0错误

设置烧录仿真配置

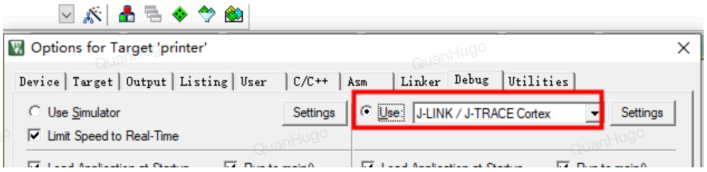
确认下芯片是否是STM32F103CB，设置主频为72MHz（系统时钟这里设置为其它也能跑，但是跑得慢啊），如果需要使用printf打印数据到串口，可以添加右边的“Use MicroLIB”，打上勾



然后设置烧录仿真器，这里我们使用J-LINK的SWD接口下载，大家需要自行购买烧录器

如果是ST-Link，下方选择ST-Link哦，然后把板卡通过烧录器接到电脑，如果Jlink无法识别，可以问卖家要驱动哦。

驱动正常，接线没问题后，我们就可以点击侧面的settings，进入设置页面



接线方式如下：

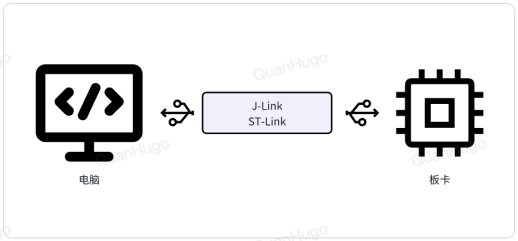
J-Link 板卡

GND 《-------》 GND

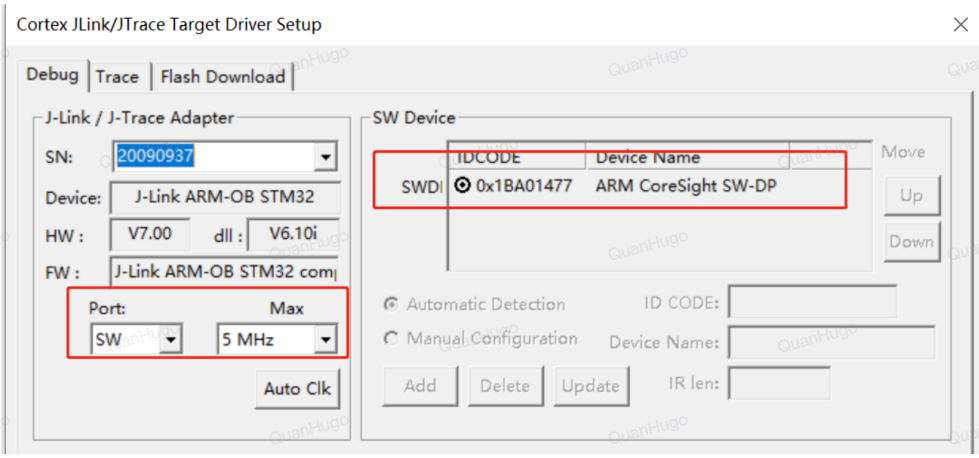
SWDIO《-------》 SWDIO

SWCLK《-------》 SWCLK

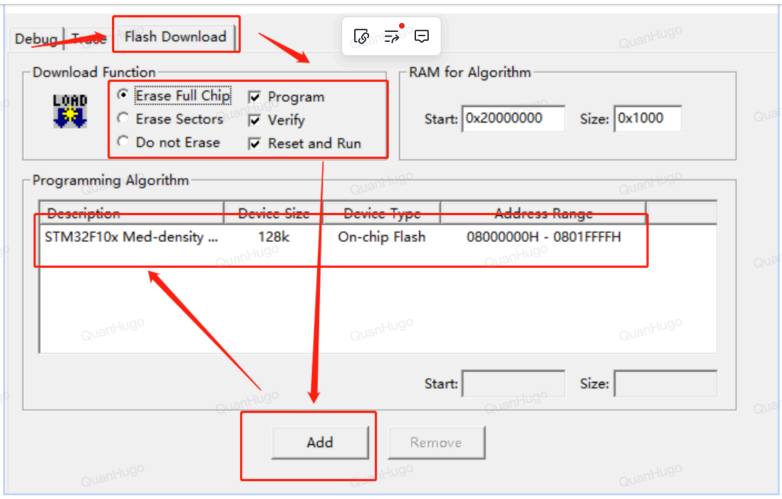
3V3 《-------》 3V3



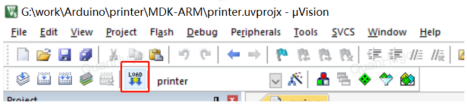
在这个页面中，我们设置模式为SW模式，这时候如果连接正常可以看到右边出现了设备，如果看不到设备，应该是接线和驱动原因哦。



然后就可以设置烧录和下载信息了，设置擦除芯片，烧录完成自动重启，添加烧录算法，使用STM32F10x的即可，都设置好后点击确定关闭设置页面。



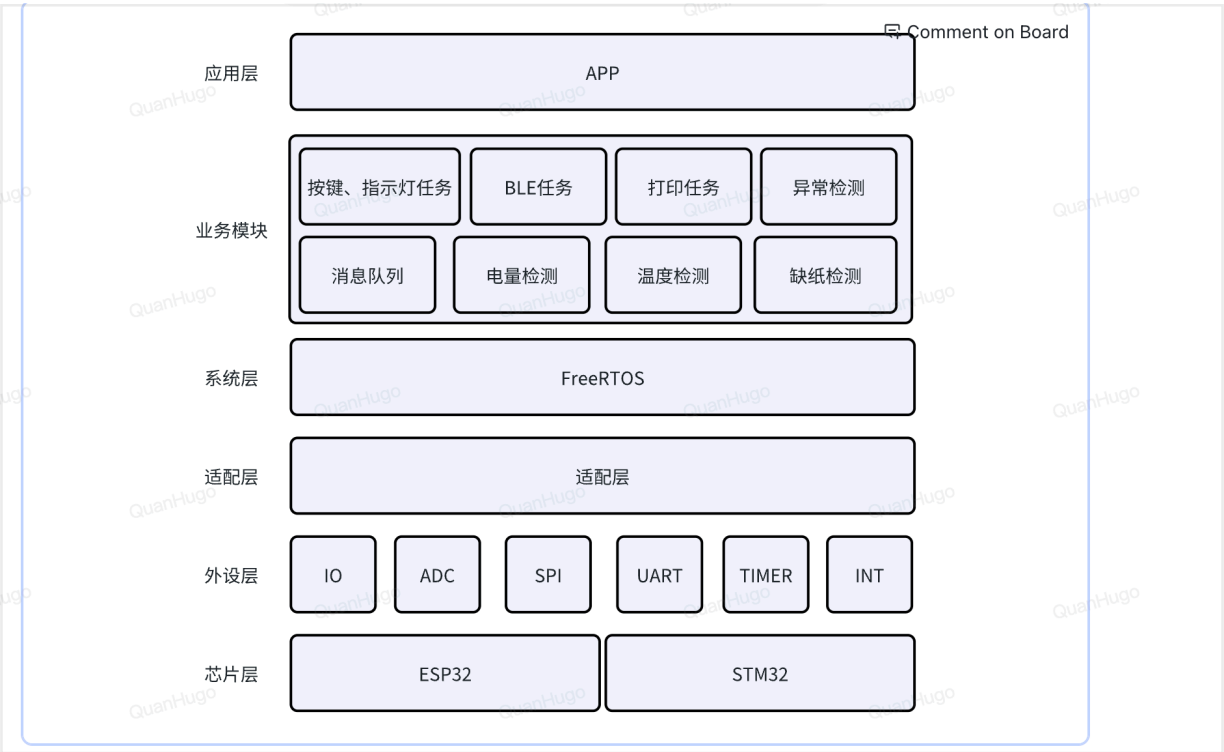
6、烧录下载

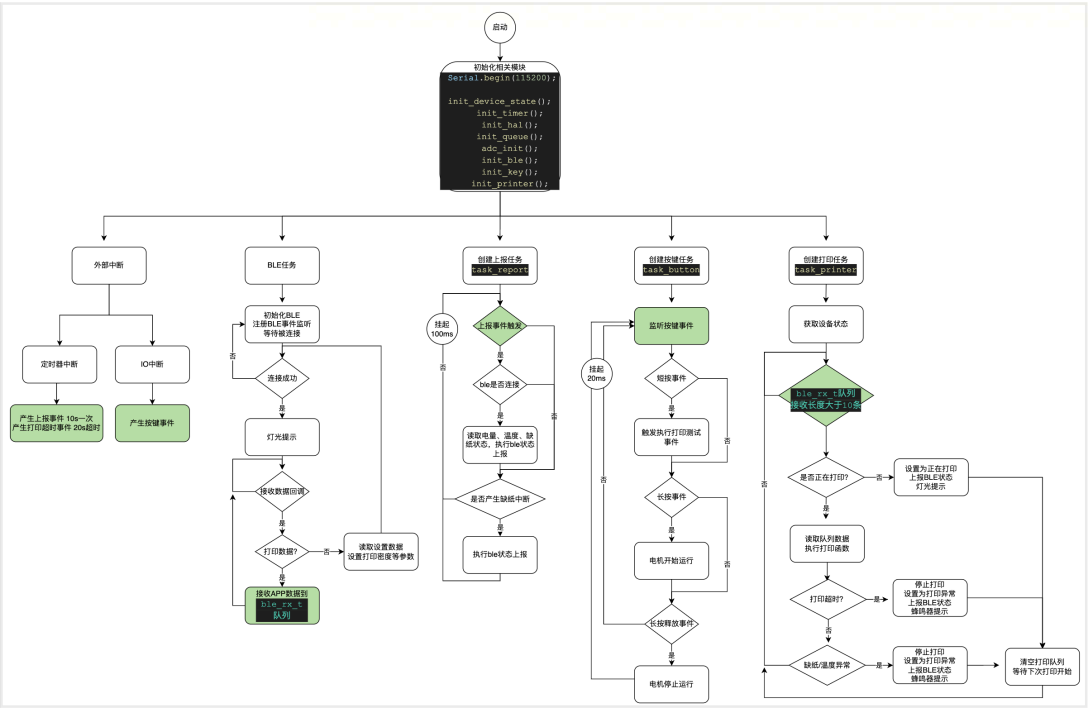


如无意外，你就可以看到烧录完成啦！！！！

功能实现

接下来就是真正的代码时间了


Comment on Board
应用层
APP
业务模块
按键、指示灯任务
BLE任务
打印任务
异常检测
消息队列
电量检测
温度检测
缺纸检测
系统层
FreeRTOS
适配层
适配层
外设层
IO
ADC
SPI
UART
TIMER
INT
芯片层
ESP32
STM32


启动
初始化相关模块
Serial.begin(115200);
init_device_state();
init_timer();
init_hal();
init_queue();
adc_init();
init_ble();
init_key();
init_printer();
外部中断
定时器中断
IO中断
产生上报事件 10s一次
产生打印超时事件 20s超时
产生按键事件
BLE任务
初始化BLE
注册BLE事件监听
等待被连接
连接成功
否
是
灯光提示
接收数据回调
打印数据?
否
是
读取设置数据
设置打印密度等参数
接收APP数据到
ble_rx_t
队列
创建上报任务
task_report
挂起
100ms
上报事件触发
是
否
ble是否连接
读取电量、温度、缺纸状态，执行ble状态上报
是否产生缺纸中断
执行ble状态上报
创建按键任务
task_button
挂起
20ms
监听按键事件
短按事件
触发执行打印测试事件
长按事件
电机开始运行
长按释放事件
电机停止运行
创建打印任务
task_printer
获取设备状态
ble_rx_t队列
接收长度大于10条
是否正在打印?
否
是
设置为正在打印
上报BLE状态
灯光提示
读取队列数据
执行打印函数
打印超时?
是
否
停止打印
设置为打印异常
上报BLE状态
蜂鸣器提示
缺纸/温度异常
是
停止打印
设置为打印异常
上报BLE状态
蜂鸣器提示
清空打印队列
等待下次打印开始

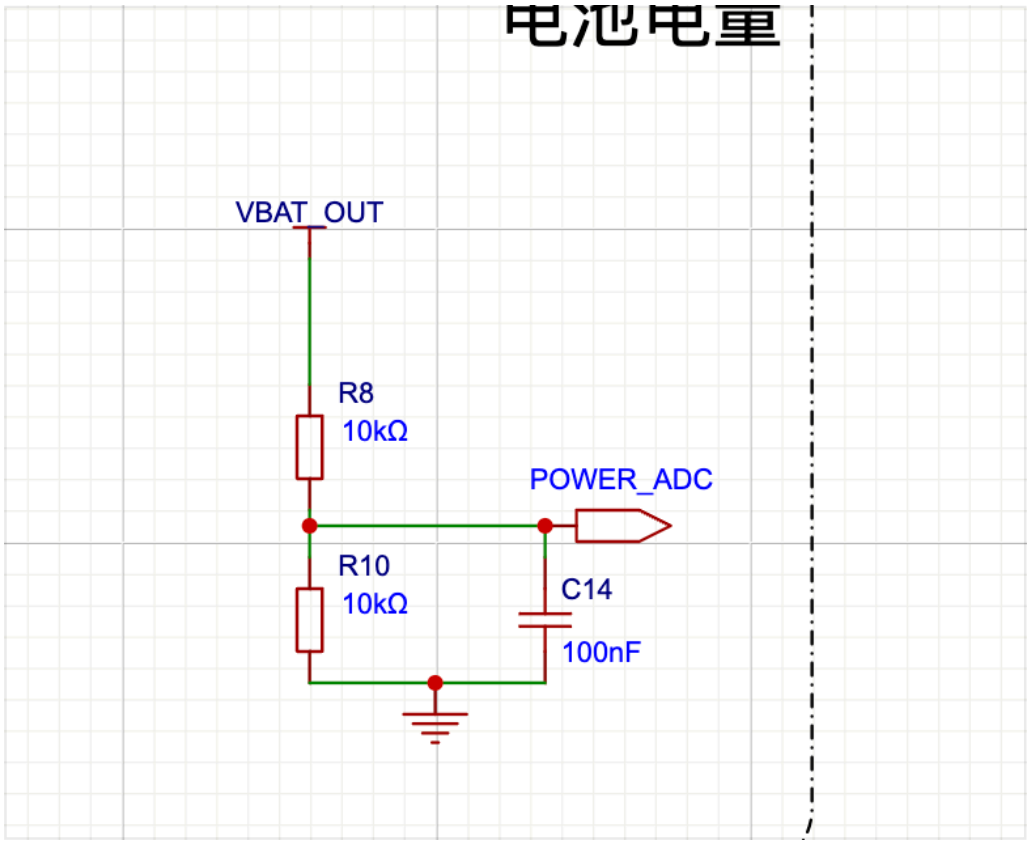
电量检测

7-1、ADC-电量检测

因为电量检测、温度检测的GPIO使用的是ADC1，所以这里我们直接读取两个IO的ADC，进行转换。

ADC模块采样时，做了简单的去抖处理，具体处理方式是：

1. 每个IO采集十次数据，放入数组中
2. 把最高和最低值去掉，然后剩余八组数据求平均值
3. 最终返回ADC平均值转换为实际电压



编程电量检测的话， we can read the ADC signals from two ADC (Analog-to-Digital Converter) channels, process the data, and estimate the battery level based on the measured voltage.

The way we process it is by smoothing the ADC values by limiting the highest and lowest values and just take the average of it.

```
extern ADC_HandleTypeDef hadc1;

#define ADC_READ_TIME 10 //adc channel will be sampled every 10 times
#define EPISON 1e-7

uint32_t ADC_Value[2]; //this array will store the processed ADC values for the two
```

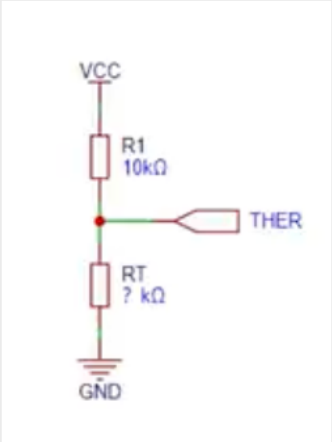
channels

```
void adc_init() {
	HAL_ADC_Start(&hadc1); //Starts the ADC conversion
	HAL_ADC_PollForConversion(&hadc1,1000); //Waits for the adc time out of
1000ms
}

//this function processes the sampled ADC data
uint32_t adc_alg_handle(uint32_t *adc, int size) {
   uint32_t sum = 0; //sum is initially set to 0
   uint32_t min_val = adc[0];//min value is 0
   uint32_t max_val = adc[0]; //max value is 0

   for (int i = 0; i < size; i++) { //a simple for loop for
      if (adc[i] < min_val) {
         min_val = adc[i]; //min value = adc value at I
      }
      else if (adc[i] > max_val) {
         max_val = adc[i]; //same for maximum value
      }
      sum += adc[i]; //add the sum
   }
   sum = sum - (min_val + max_val);
   uint32_t avg_val = sum / (size - 2); //average value obtained
   return avg_val;
}



/**
 * @brief 获取ADC引脚电压值，可根据需要加入滤波算法
 *
 * @return int
 */

//this function reads the voltage value, samples its, then processes the data
int get_adc_volts()
{
  uint32_t data = 0;
  uint32_t adc1[ADC_READ_TIME]; //array
  uint32_t adc2[ADC_READ_TIME]; //array for storing sampled values

//adc read time is 10
  for (int sample_ptr = 0; sample_ptr < ADC_READ_TIME; sample_ptr++)
```

```
 {
        //由于上面设置的ADC是间断模式+扫描模式，所以每采集一次，需要执行一次HAL_ADC_Start
        HAL_ADC_Start(&hadc1); //execute HAL_ADC start
        if (HAL_ADC_PollForConversion(&hadc1, 100) == HAL_OK)//waits the result pool for conversion at a time out of 100ms
        {
            adc1[sample_ptr] = HAL_ADC_GetValue(&hadc1);//stores the digital value from the adc at the current index
        }
        HAL_ADC_Start(&hadc1);
        if (HAL_ADC_PollForConversion(&hadc1, 100) == HAL_OK)
        {
            adc2[sample_ptr] = HAL_ADC_GetValue(&hadc1);
        }
        HAL_ADC_Stop(&hadc1);
 }
 ADC_Value[0] = adc_alg_handle(adc1, ADC_READ_TIME); //uses the function to process the data from these two arrays holding the sampling data
 ADC_Value[1] = adc_alg_handle(adc2, ADC_READ_TIME);

//converts the adc value back to voltage value:

Voltage (V) = (ADC Value) * (Reference Voltage / Resolution)
        = (ADC Value) * (3.3V / 4096)

 printf(" ADC channel0 end value = ->%1.3fV \r\n", ADC_Value[0] * 3.3f / 4096);
 printf(" ADC channel1 end value = ->%1.3fV \r\n", ADC_Value[1] * 3.3f / 4096);

 data = ADC_Value[0] * 3.3f / 4096;

 HAL_ADC_Stop(&hadc1);

 return data;
}

//this function maps a value of x from one range to another. Basically maps voltages to battery percentages.

static long map(long x, long in_min, long in_max, long out_min, long out_max) {
  const long dividend = out_max - out_min;
  const long divisor = in_max - in_min;
  const long delta = x - in_min;
  if(divisor == 0){ //this ensures that if the input range is 0
```

```
        return -1; //AVR returns -1, SAM returns 0
    }
    return (delta * dividend + (divisor / 2)) / divisor + out_min;
}

void read_battery()
{
    //这里我们认为电压在3.3V为0%电量，4.2V为100%电量，进行转换
    get_device_state()->battery = map(get_adc_volts()*2,3300,4200,0,100); //The voltage returned by get_adc_volts() is multiplied by 2. This adjustment compensates for a voltage divider circuit used to measure higher voltages than the ADC can directly read. The factor of 2 indicates that the voltage has been scaled down by half before reaching the ADC, so multiplying by 2 restores the original voltage value. Maps from 3.3V at 0 percentage to 4.2 at 100%

    if(get_device_state()->battery > 100)
     get_device_state()->battery = 100; //if the calculated battery percentage is at 100 clip it at 100.
    printf("battery = %d\n",get_device_state()->battery); //prints the battery
}
```

ADC-温度检测



热敏电阻会随着温度的变化从而变化，so with voltage division, we can find out what is the value of the THER tag is, and thus we can find out the temperature

The formula for sampled data ADC_Value = VR * 4096 / 3.3 , VR is voltage across the resistor divider

This formula can be rearranged to ADC_Value = 3.3 * R / (Rt + R) * 4096 / 3.3 = R /

(Rt + R) * 1024

voltage divider

We can then find Rt from this and

The thermistor's resistance (Rt) is related to temperature using the formula:

Rt = Rp * exp(B * (1/T1 - 1/T2))

Rp is the resistance of the thermistor at a reference temperature (usually 25°C). At 25 degrees it’s usually 30k
B is the thermistor's material constant (B-value).
T1 is the absolute temperature (in Kelvin) at the desired measurement.
T2 is the reference temperature in Kelvin (25°C = 298.15 K). From 25 + 273.15

Thus we can calculate temperature by rearranging:

T = 1 / ((1/T1) + (log(Rt/Rp) / B)) - 273.15

Rp is the fixed resistor value in the voltage divider at T2. It’s about 30k at 25degrees.
Rt is the thermistor resistance at a measured temperature T1
B25/50 = 3950K ± 1% indicates the thermistor's B-value, which is typically provided by the manufacturer.

Rlead:	导线电阻	10Ω

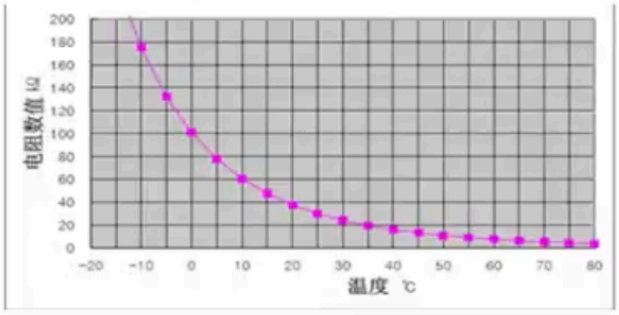
3.5、热敏电阻

计算公式:

$$R_X = R25 \times EXP[B \times (\frac{1}{T_X} - \frac{1}{T25})]$$ (注: T: 绝对温度)

其中:

B	常数	3950K ± 2%
R25	电阻值	30KΩ ± 5%(在 25℃)
Tx	工作温度	-20℃~+80℃
T25	1 冲宽度的工作温度	25℃



热敏电阻的温度曲线图

图 1

表 5 热敏电阻温度表:

温度 (℃)	阻值 KΩ)	温度 (℃)	阻值 (KΩ)	温度 (℃)	阻值 (KΩ)	温度 (℃)	阻值 (KΩ)
-20	269	10	60	40	15.9	70	5.21
-15	208	15	47.1	45	13.1	75	4.4
-10	178	20	37.5	50	10.8		

7-2、ADC-温度检测

上个步骤中，我们已经同时采集了两个IO的ADC值，并转换为实际的IO电压值，所以这个步骤我们只需要把电压转换为实际的温度值即可。

热敏电阻是有RT表（阻值和温度对应表）和公式法两种转换方式的

查表法顾名思义就是有温度和电阻的一个二维数组存储在代码中，读取到ADC值转换为电压，再根据电阻串联分压原理，计算出热敏电阻阻值，最终查表得出实际温度值。

查表法：

ADC采样率8位 串联电阻R1=10K R2=热敏电阻 电压3.3V
已知
Rt = R*(3.3-VR)/VR
VR = 3.3*ADC_Value/1024
得出
ADC_Value = VR*4096/3.3 =3.3*R/(Rt+R)*4096/3.3 = R/(Rt+R)*1024
ADC_Value=3.3/(C5+10)*10/3.3*1023
如果需要用到小数点后面的温度，正确的方法是使用公式RT＝R0*exp(B (1/T-1/T0))在excel中计算得到步进值为0.1°C的温度表
Rt=100*exp(3950*(1/(273.15+T1)-1/(273.15+25)))

```
/**
 * @brief 阻值转换温度
 *
 * @param Rt 热敏电阻阻值
 * @return float 返回摄氏度温度
 */

This function calculates the temperature using Rt

float em_temp_calculate(float Rt)
{
    float Rp = 30000; // 30k
    float T2 = 273.15 + 25;
    float Bx = 3950; // B值
    float Ka = 273.15;
    float temp = 0.0f;
    temp = 1 / (log(Rt / Rp) / Bx + 1 / T2) - Ka + 0.5;
    return temp;
}

/**
 * @brief Get the adc temperatrue object
 *
```

```
 * @return float
 */

//this function gcalculates the Rt and then uses Rt to calculate temperature by first converting adc to voltage and then voltage to

float get_adc_temperatrue(){
  float temp = 0.0f;
  float Rt=0;
  float vol=0;
  //ADC转换为电压 vol=AD/4096*VCC

  vol=(float)ADC_Value[1]*3.3f/4096;
        printf("ADC temperatrue analog value = %f\n",vol);
  //电压转换为阻值 原理图为10k 1%精度  vol/VCC=Rt/(R+Rt)  vol/3.3=Rt/(10000+Rt)
  Rt=(vol*10000)/(3.3-vol);
       printf("ADC temperatrue Rt = %f\n",Rt);
  temp = em_temp_calculate(Rt);
  return temp;
}


This function
void em_adc_test(){
  float Rt=0;
  float vol=3.0f;
  Rt=(vol*10000)/(3.3-vol);
  printf("Rt = %f\n",Rt);

  Rt = 60000; //60k 10°C
  float temp = 0.0f;
  temp = em_temp_calculate(Rt);
  printf("temp = %f\n",temp);

  if(temp >= 1e-7){
     printf("正\n");
  }else{
     printf("负\n");
  }
}
```

UART for printf

In embedded systems, the standard printf function does not have a default output

destination, so it needs to be redefined to send data to a specific communication channel, such as a UART.

We can use UART 1 as the signal for printingF function

7-3、串口重定向

使用过单片机的小伙伴都知道，单片机本身是没有指定printf的打印函数使用的串口的，例如我们项目中，有两个usart，那如何确定串口作为日志口打印日志呢？这里我们需要在代码中实现对printf函数的重定向，把它指向我们想绑定的串口，这里我们使用huart1作为打印串口。

C Copy

```c

#include <stdio.h>

//串口printf重定向
#ifdef __GNUC__
  /* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
     set to 'Yes') calls __io_putchar() */
  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */
/**
  * @brief  Retargets the C library printf function to the USART.
  * @param  None
  * @retval None
  */
PUTCHAR_PROTOTYPE
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the EVAL_COM1 and Loop until the end of transmission */
  HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);

  return ch;
}
```

```
#ifdef __GNUC__
  /* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
     set to 'Yes') calls __io_putchar() */
  #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
  #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */
```

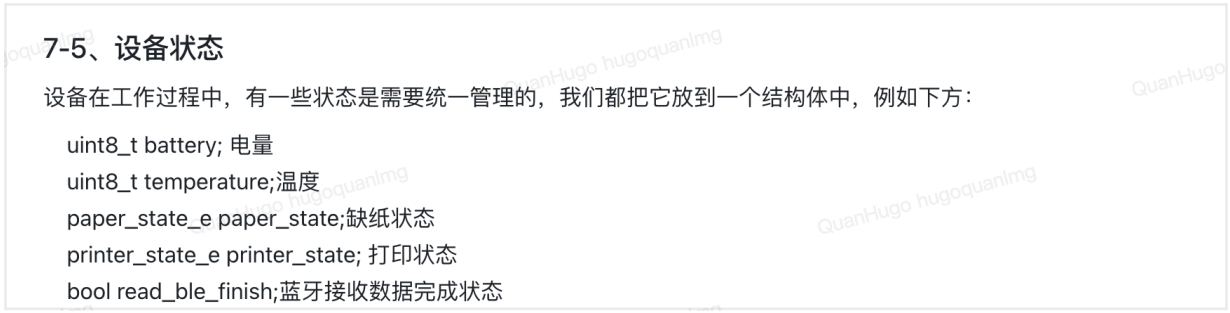
This section defines a macro PUTCHAR_PROTOTYPE based on the compiler being used:

- If GCC (GNU Compiler Collection) is being used, it defines PUTCHAR_PROTOTYPE as int __io_putchar(int ch). This function (__io_putchar) is expected by GCC when using a small printf implementation.

HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);

- HAL_UART_Transmit is a function provided by the HAL (Hardware Abstraction Layer) library for STM32 microcontrollers. This function sends data over the specified UART interface.
- &huart1 is the handle for UART1, which has been configured and initialized elsewhere in the code.
- (uint8_t *)&ch casts the character to an 8-bit unsigned integer pointer, which is the expected data type for transmission.
- 1 specifies the number of bytes to send (only one character here).
- 0xFFFF is the timeout in milliseconds, representing the maximum time the function will wait for the transmission to complete.

设备状态



配置模块

主要是实现一些配置参数

7-5、配置模块

这里主要实现一些通用的配置参数，例如

START_PRINTER_WHEN_FINISH_RAED 定义打开时，需要等待接收到蓝牙的完成传输指令，才会开始打印

PRINT_TIME 打印加热时间，时间越大，打印颜色越深，us单位，过大会损坏打印模组哦

PRINT_END_TIME 打印冷却时间，打印完一个通道数据的等待冷却时间，us单位

MOTOR_WATI_TIME 电机移动一步的时间，us单位

LAT_TIME 打印数据发送后的数据锁存时间，us单位

我们可以看到，上面很多参数的时间单位都是us，而在Freertos中，时间片单位是ms，是没有提供us单位的延时的，所以我们基于定时器TIME1做了us的延时函数。us_delay

```c
//接收完成所有数据才开始打印
#define START_PRINTER_WHEN_FINISH_RAED 1

#define PRINT_TIME 1700         //打印加热时间
#define PRINT_END_TIME 200      //冷却时间
#define MOTOR_WATI_TIME 4000    //电机一步时间
#define LAT_TIME 1              //数据锁存时间


#include "tim.h"
#define DLY_TIM_Handle (&htim1)  // Timer handle
#define hal_delay_us(nus) do { \
    __HAL_TIM_SET_COUNTER(DLY_TIM_Handle, 0); \
    __HAL_TIM_ENABLE(DLY_TIM_Handle); \
    while (__HAL_TIM_GET_COUNTER(DLY_TIM_Handle) < (nus)); \
    __HAL_TIM_DISABLE(DLY_TIM_Handle); \
} while(0)

#define us_delay(us) hal_delay_us(us)

#define LOW              0x0
#define HIGH             0x1
```

Start Printer set to 1

PRINT_TIME: The time needed to heat up the printer.
PRINT_END_TIME: The cooldown time after printing.
MOTOR_WAIT_TIME: The delay for the motor to complete its movement.
LAT_TIME: A small delay for shifting data, possibly between data transmissions.

#include "tim.h"

- This line includes the tim.h header file, which contains functions and definitions for using the timer hardware.

#define DLY_TIM_Handle (&htim1) // Timer handle

- This macro defines DLY_TIM_Handle as a pointer to the timer handle htim1. This handle is associated with a hardware timer (TIM1) configured elsewhere in the project.

#define hal_delay_us(nus) do { \

```
    __HAL_TIM_SET_COUNTER(DLY_TIM_Handle, 0); \
    __HAL_TIM_ENABLE(DLY_TIM_Handle); \
    while (__HAL_TIM_GET_COUNTER(DLY_TIM_Handle) < (nus)); \
    __HAL_TIM_DISABLE(DLY_TIM_Handle); \
} while(0)
```

- This macro defines a function to create a delay in microseconds (nus).
- Here's how it works:
    1. **Reset Timer Counter**: __HAL_TIM_SET_COUNTER(DLY_TIM_Handle, 0); resets the timer counter to 0.
    2. **Enable the Timer**: __HAL_TIM_ENABLE(DLY_TIM_Handle); starts the timer, allowing it to count up.
    3. **Wait for Target Delay**: The while loop waits until the timer's counter reaches the specified delay (nus). It continuously checks if the timer's counter value is less than nus.
    4. **Disable the Timer**: __HAL_TIM_DISABLE(DLY_TIM_Handle); stops the timer after the delay is reached.
-

This delay function is necessary because, in FreeRTOS, the minimum delay unit is milliseconds, but some functions need delays in microseconds.

1. #define LOW 0x0
2. #define HIGH 0x1
    - These are standard definitions for digital signal levels, where LOW (0x0) represents a low or zero signal, and HIGH (0x1) represents a high or one signal. They are often used to set or read the state of GPIO pins.

消息队列

因为设备打印的时候，需要接收来自手机端的数据，如果打印的是较大的图片，那就需要分包进行下发，例如一张图片大小5k，而蓝牙不能一次性传输5k的数据，所以我们把照片的每一行作为一包下发，打印机每行支持384个点，也就是384bit，48Byte，于是每一包就是48Byte。

设备接收到打印数据后，并不是马上开始打印的，因为蓝牙通讯的速度和设备打印速度不一样，如果接收就打开，可能会出现打印完了又没有接收到新的数据，导致一卡一卡的不好体验，所以我们的做法是把接收的数据写入消息队列中，需要打印再进行读取。

操作系统本身是带有消息队列的，这里我们没有使用操作系统的消息队列，而是自己写了个环形缓冲区，实现对打印数据的存储，大家可以参考下。

由于缓冲区最大的空间受STM32芯片的RAM限制，所以STM32版本最多只能打印275行的图片，当然了，大家可以自行优化为动态打印，一边接收一边打印。

```
// 一行最大byte
#define MAX_ONELINE_BYTE 48
// 最大行数
#define MAX_LINE 275

typedef struct
{
    uint8_t buffer[MAX_ONELINE_BYTE];
} ble_rx_buffer_t;

typedef struct
{
    ble_rx_buffer_t printer_buffer[MAX_LINE];
    uint32_t r_index;
    uint32_t w_index;
    uint32_t left_line;
} ble_rx_t;

ble_rx_t g_ble_rx;

SemaphoreHandle_t xHandler = NULL;

void init_queue()
{
    clean_printbuffer();
    xHandler = xSemaphoreCreateMutex();
}

/**
 * @brief 写入一行数据
 *
 * @param pdata
```

```
 * @param length
 */
void write_to_printbuffer(uint8_t *pdata, size_t length)
{
    static BaseType_t xHigherPriorityTaskWoken;
    if (length == 0)
        return;
    if (g_ble_rx.left_line >= MAX_LINE)
        return;
    if (length > MAX_ONELINE_BYTE)
        length = MAX_ONELINE_BYTE;
    // 查看是否可以获得信号量，如果信号量不可用，则用10个时钟滴答来查看信号量是否可用
    //这里可以设置零等待、有限等待、无限等待
    //零等待可能会导致数据丢包，因为正在读取数据时，同时想写入数据，这时候是获取不到信号量的，就不会执行写入循环缓冲区操作。
    //有限等待：阻塞10个时钟滴答是大致评估了数据读出的时长，保证每次写入时都能阻塞等待到获取信号量，代码设计时一般不建议无限阻塞。
    //当然，这个模块可以使用rtos提供的消息队列实现，或者双缓冲区实现会更适合。
    if (xSemaphoreTakeFromISR(xHandler, &xHigherPriorityTaskWoken) == pdPASS)
    {
        memcpy(&g_ble_rx.printer_buffer[g_ble_rx.w_index], pdata, length);
        g_ble_rx.w_index++;
        g_ble_rx.left_line++;
        if (g_ble_rx.w_index >= MAX_LINE)
            g_ble_rx.w_index = 0;
        if (g_ble_rx.left_line >= MAX_LINE)
            g_ble_rx.left_line = MAX_LINE;
        xSemaphoreGiveFromISR(xHandler, &xHigherPriorityTaskWoken);
    }
    if (xHigherPriorityTaskWoken == pdTRUE)
    {
        // 如果有更高优先级的任务需要唤醒，则进行任务切换
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }
}

/**
 * @brief 读取一行数据
 *
 * @return uint8_t*
 */
uint8_t *read_to_printer()
{
```

```
    uint32_t index = 0;
    // 查看是否可以获得信号量，如果信号量不可用，则用10个时钟滴答来查看信号量是否可用
    if (xSemaphoreTake(xHandler, (portTickType)10) == pdPASS)
    {
        if (g_ble_rx.left_line > 0)
        {
            g_ble_rx.left_line--;
            index = g_ble_rx.r_index;
            g_ble_rx.r_index++;
            if (g_ble_rx.r_index >= MAX_LINE)
                g_ble_rx.r_index = 0;
            xSemaphoreGive(xHandler);
            return g_ble_rx.printer_buffer[index].buffer;
        }
        xSemaphoreGive(xHandler);
        return NULL;
    }
    else
        return NULL;
}

/**
 * @brief 清空接收缓存
 *
 */
void clean_printbuffer()
{
    g_ble_rx.w_index = 0;
    g_ble_rx.r_index = 0;
    g_ble_rx.left_line = 0;
}

uint32_t get_ble_rx_leftline()
{
    return g_ble_rx.left_line;
}
```