嵌入式基础知识概述 - 小智课程 - HQ

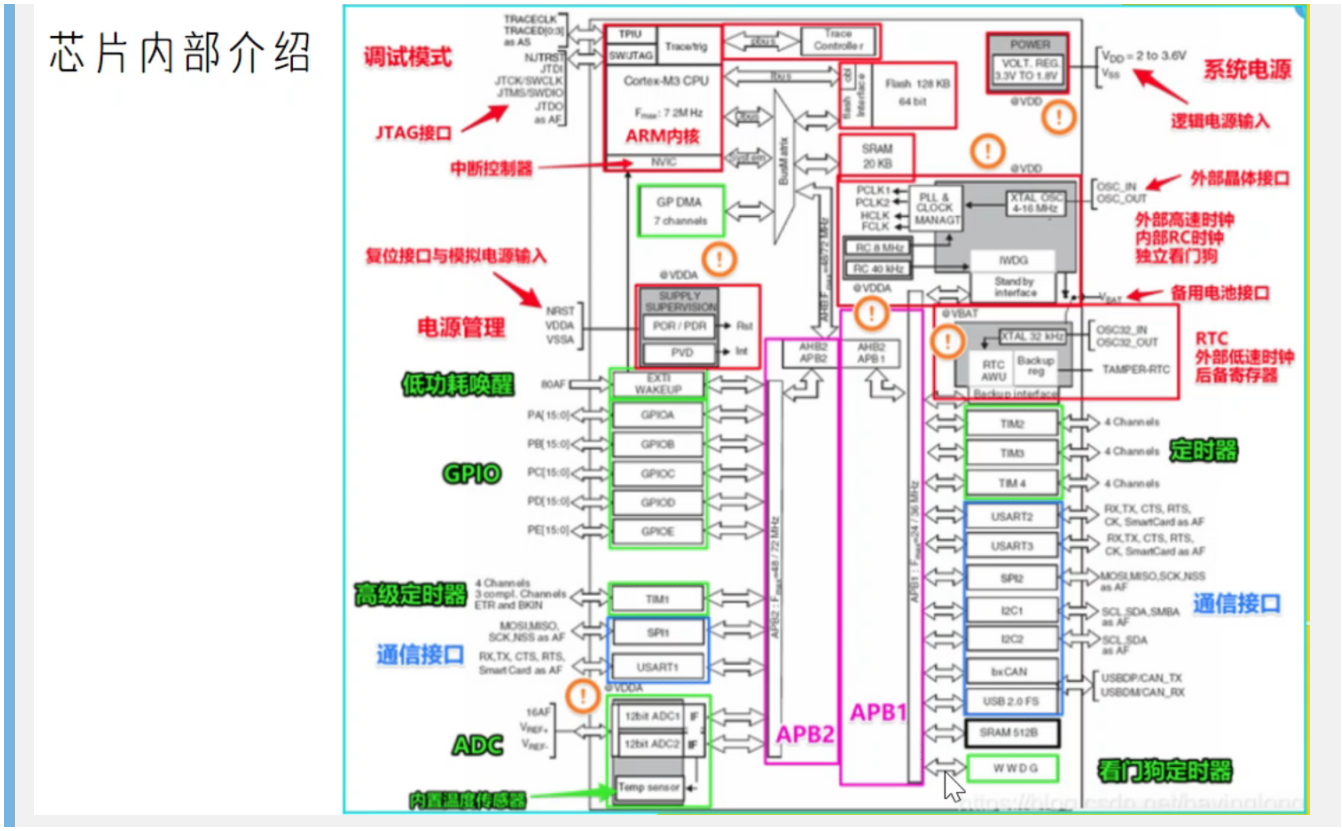
[TOC]

注意

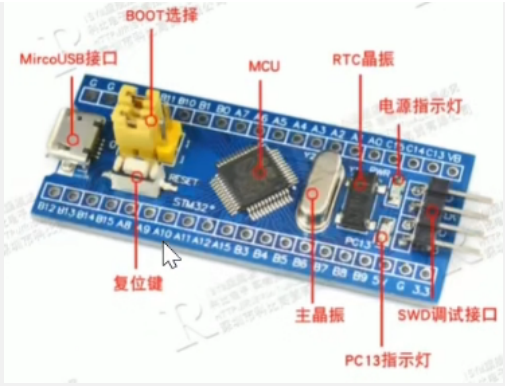
-

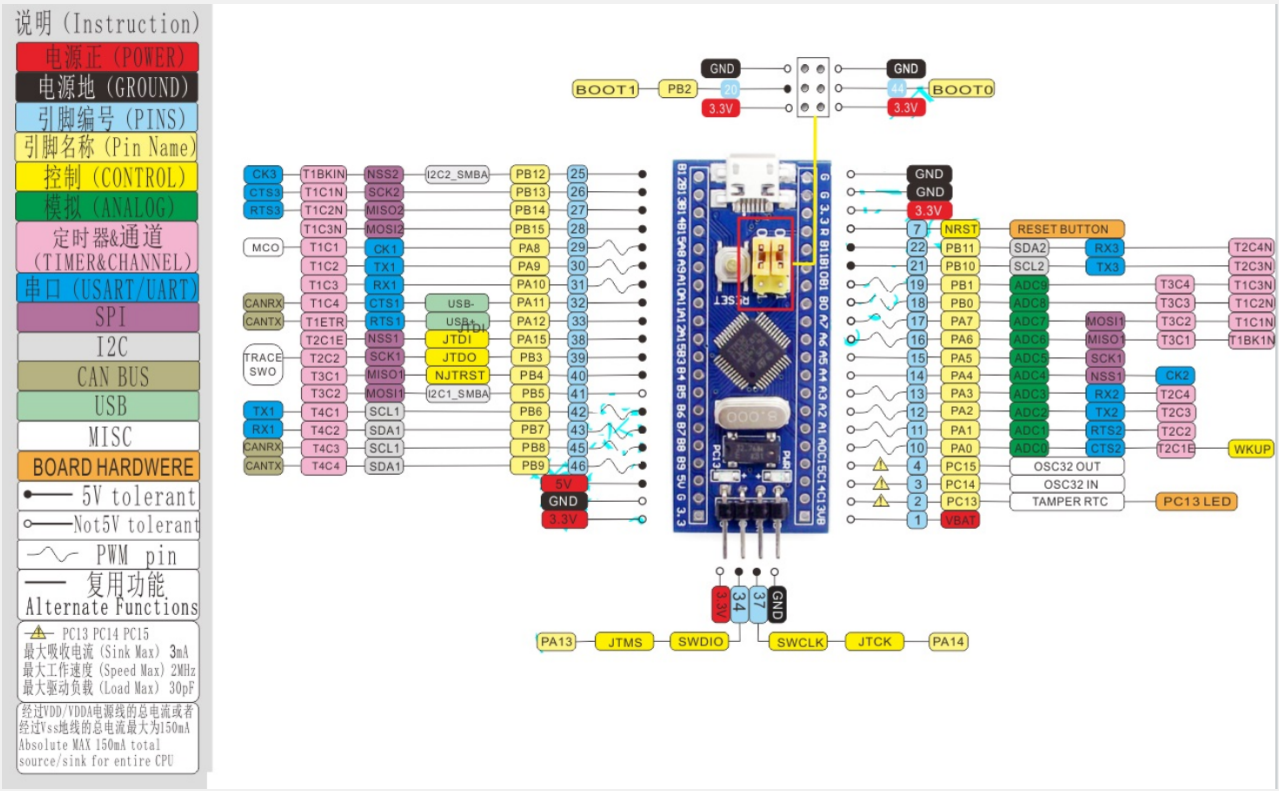
STM32学习 - b站嵌入式小智课程

芯片内部介绍

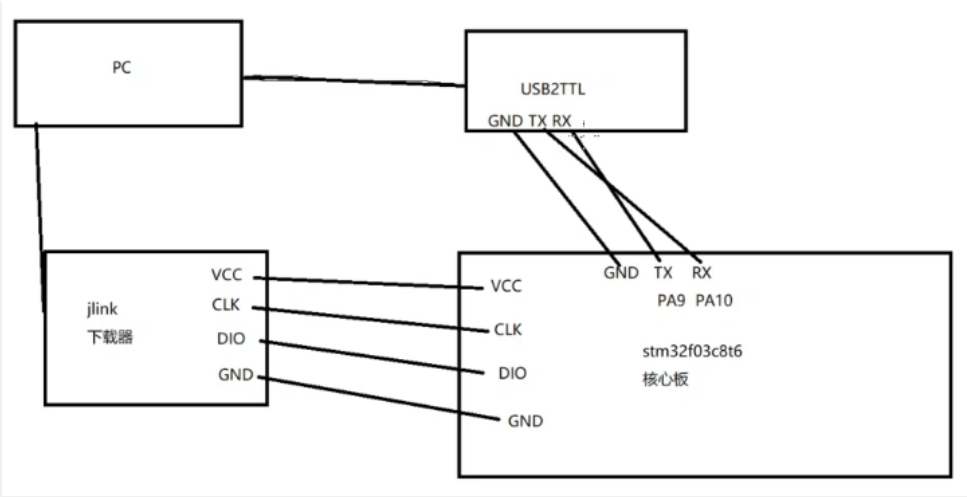


STM32F103C8T6芯片引脚图





STM32与ST-Link v2，串口转TTL模块接线图

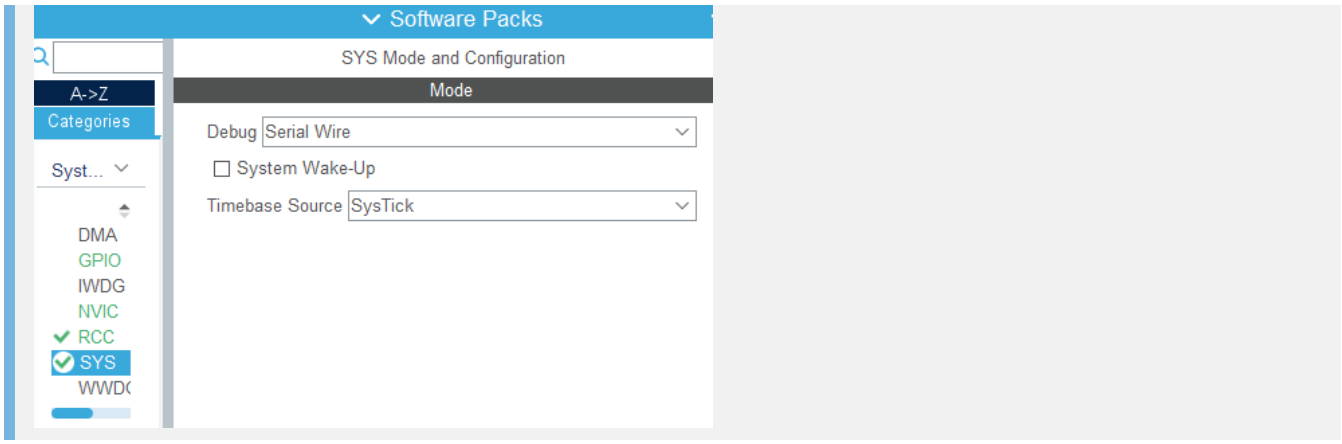


串口打印实验

1.STM32CubeMX配置USART1

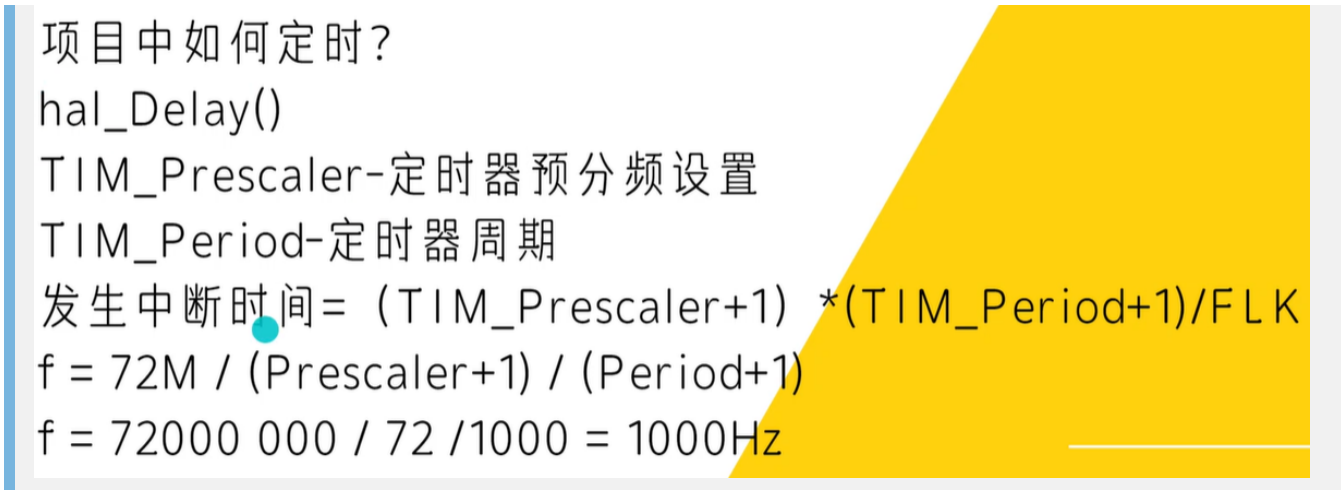
Asynchronous同步模式，Synchronous异步模式(需要连同步时钟信号)

特别，sys要设置一下

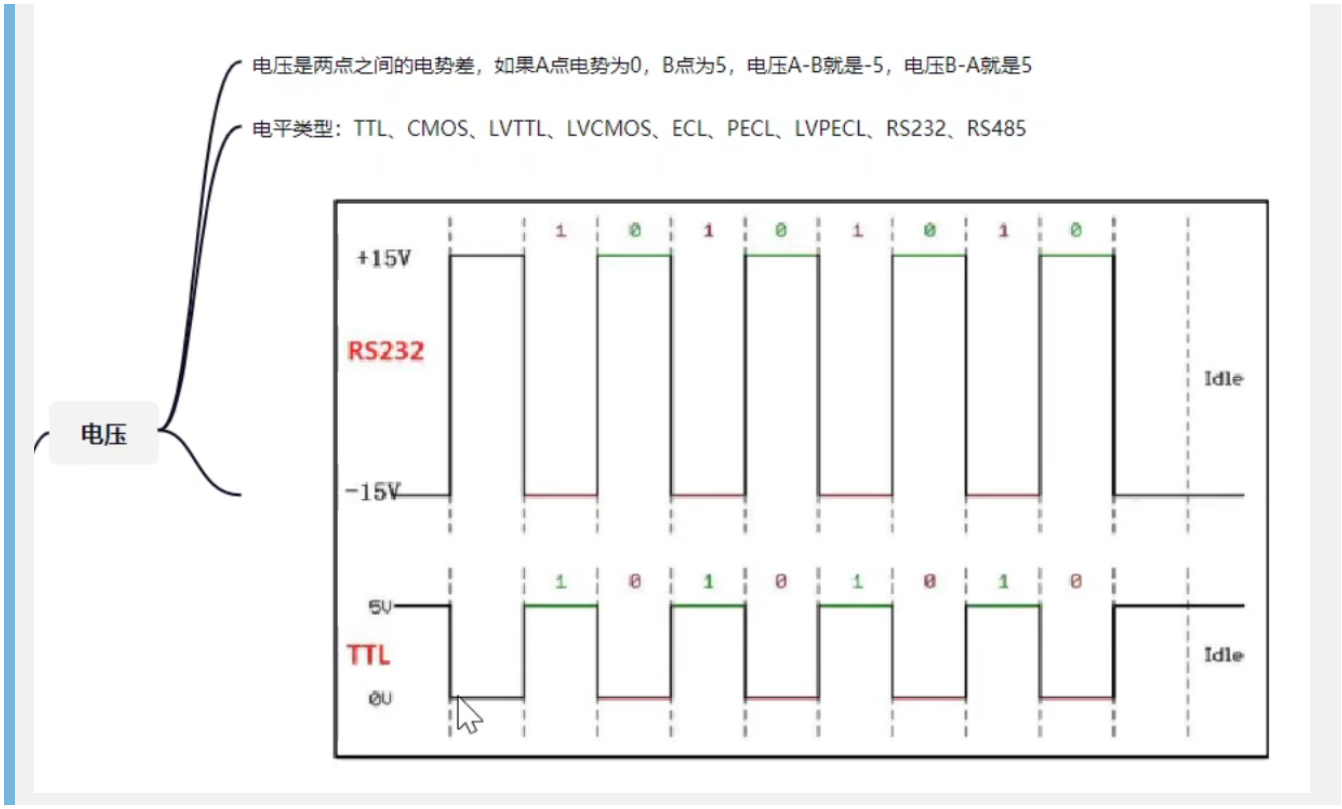


不知道怎么写程序，怎么函数复用，卒

定时器



电压

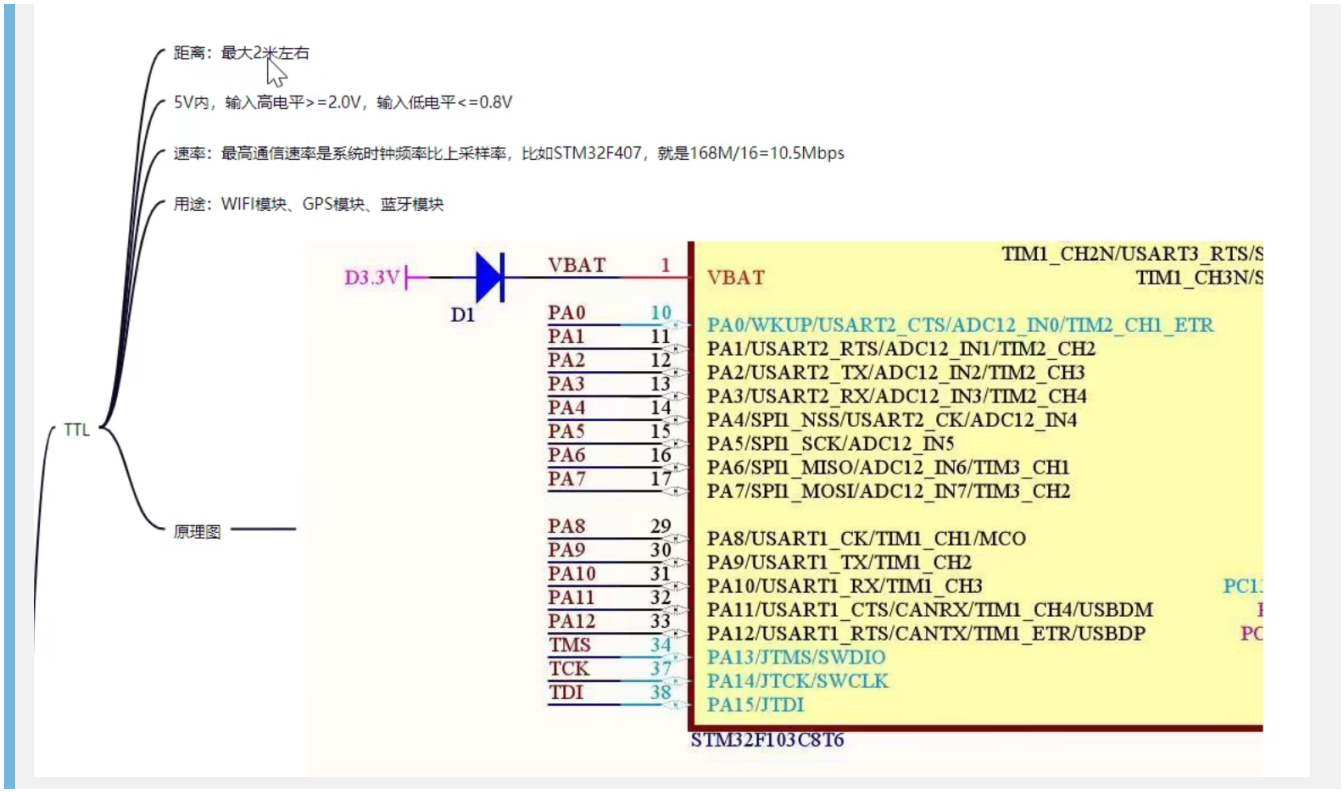

电压
电压是两点之间的电势差，如果A点电势为0，B点为5，电压A-B就是-5，电压B-A就是5
电平类型：TTL、CMOS、LVTTL、LVCMOS、ECL、PECL、LVPECL、RS232、RS485
+15V
RS232
-15V
Idle
1 0 1 0 1 0 1 0
5V
TTL
0V
Idle
1 0 1 0 1 0 1 0

IO

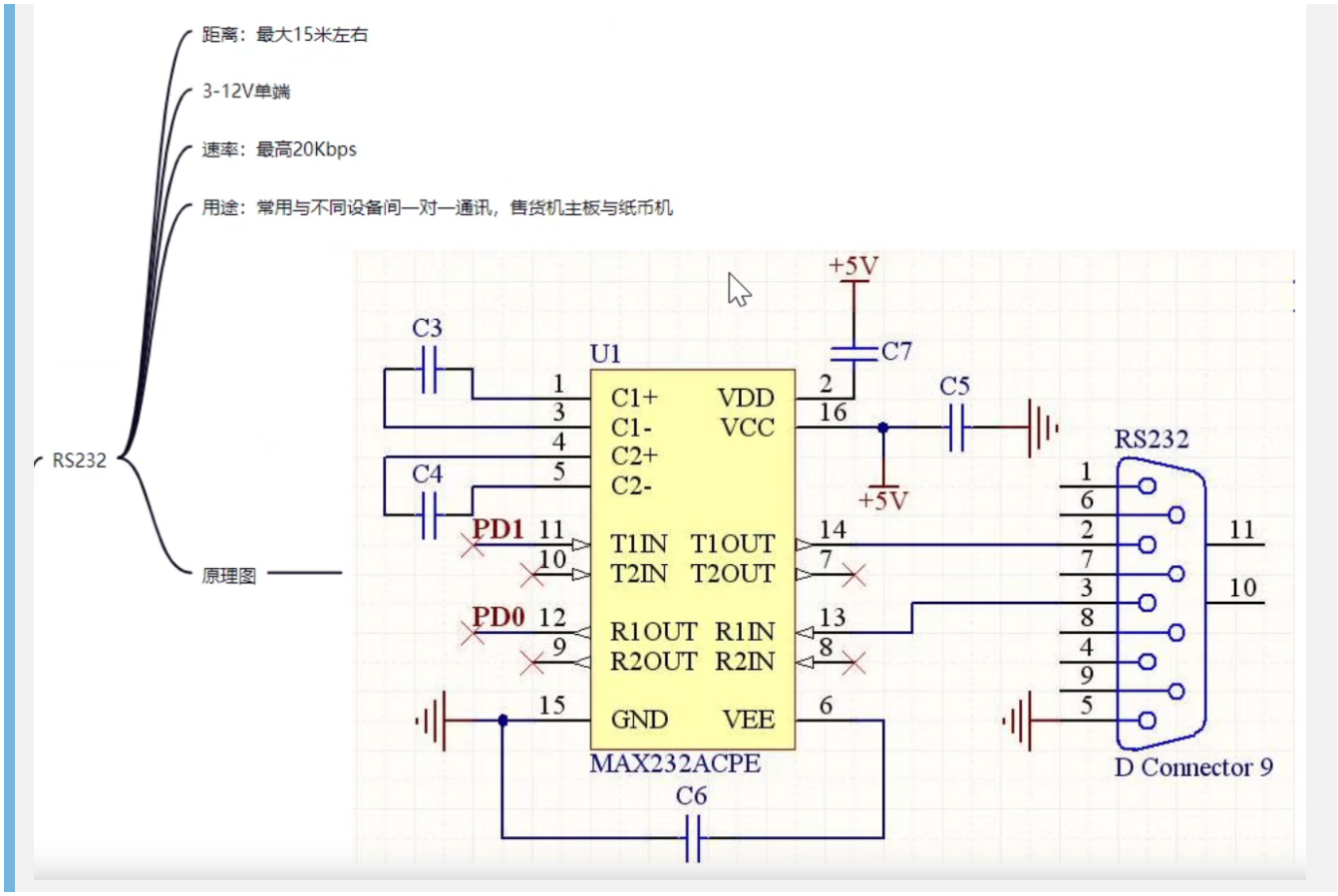

IO
为何需要IO —— 开关
为何需要不同模式 —— 让IO可以适配更多应用场景 —— ADC KEY 双向IO 内置外设
IO输入模式
1、模拟输入： 顾名思义，即为输入模拟信号 ADC
2、浮空输入：I/O口的状态是不确定的，完全由外部输入决定 外部KEY有上下拉
3、下拉输入：在没有信号输入的时候默认输入低电平 外部KEY接VCC
4、上拉输入：在没有信号输入的时候默认输入高电平 外部KEY接GND
IO输出模式
5、开漏输出：IO输出0接GND，IO输出1，悬空，需要外接上拉电阻，才能实现输出高电平。当输出为1时，IO口的状态由上拉电阻拉高电平，但由于是开漏输出模式，这样IO口也就可以由外部电路改变为低电平或不变。可以读IO输入电平变化，实现IO双向功能，模拟I2C
6、推挽输出：IO输出0-接GND， IO输出1 -接VCC，读输入值是未知的
7、复用功能的推挽输出_AF_PP——片内外设功能（I2C的SCL，SDA）
8、复用功能的开漏输出_AF_OD——片内外设功能（MOSI，MISO.SCK.SS）
如何使用
作为普通GPIO输入输出：根据需要配置引脚，同时不要使能该引脚的多有的复用功能模块
作为普通模拟输入：配置该IO口为模拟输入模式，同时不要使能该引脚对应的所有的复用功能模块
作为内置外设的输入的设置：根据需要配置该引脚为 浮空输入 带弱上拉输入或者带弱下拉输入，同时使能该引脚对应的某个复用功能模块；
作为内置外设输出：根据需要配置该引脚为复用推挽输出或则复用开漏输出，同时使能该引脚对用德尔复用功能模块
用途
按键信号 —— 复位、音量按键
继电器 —— 电灯、直流电机开关（定速）
IO输入量传感器 —— 售货机光栅

串口协议

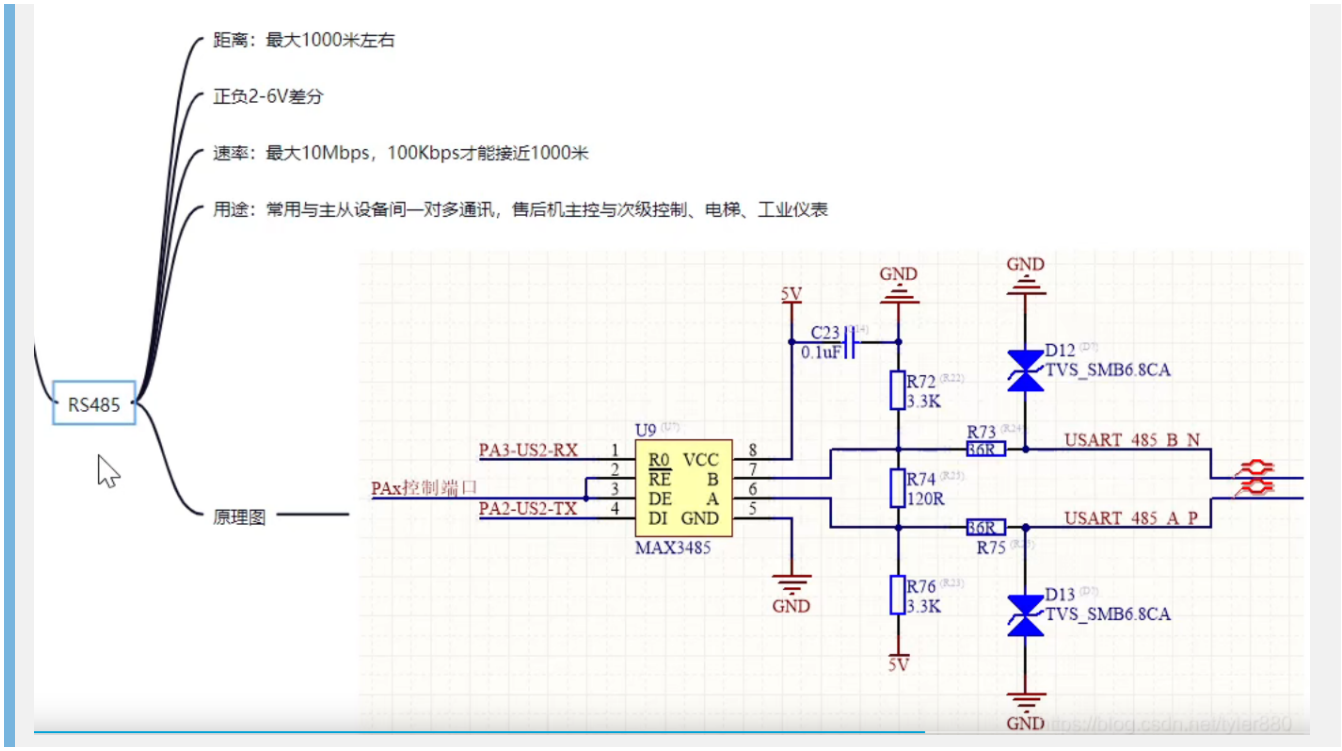
TTL


距离：最大2米左右
5V内，输入高电平>=2.0V，输入低电平<=0.8V
速率：最高通信速率是系统时钟频率比上采样率，比如STM32F407，就是168M/16=10.5Mbps
用途：WIFI模块、GPS模块、蓝牙模块
TTL
原理图
D3.3V
D1
VBAT 1
PA0 10
PA1 11
PA2 12
PA3 13
PA4 14
PA5 15
PA6 16
PA7 17
PA8 29
PA9 30
PA10 31
PA11 32
PA12 33
TMS 34
TCK 37
TDI 38
VBAT
PA0/WKUP/USART2_CTS/ADC12_IN0/TIM2_CH1_ETR
PA1/USART2_RTS/ADC12_IN1/TIM2_CH2
PA2/USART2_TX/ADC12_IN2/TIM2_CH3
PA3/USART2_RX/ADC12_IN3/TIM2_CH4
PA4/SPI1_NSS/USART2_CK/ADC12_IN4
PA5/SPI1_SCK/ADC12_IN5
PA6/SPI1_MISO/ADC12_IN6/TIM3_CH1
PA7/SPI1_MOSI/ADC12_IN7/TIM3_CH2
PA8/USART1_CK/TIM1_CH1/MCO
PA9/USART1_TX/TIM1_CH2
PA10/USART1_RX/TIM1_CH3
PA11/USART1_CTS/CANRX/TIM1_CH4/USBDM
PA12/USART1_RTS/CANTX/TIM1_ETR/USBDP
PA13/JTMS/SWDIO
PA14/JTCK/SWCLK
PA15/JTDI
STM32F103C8T6

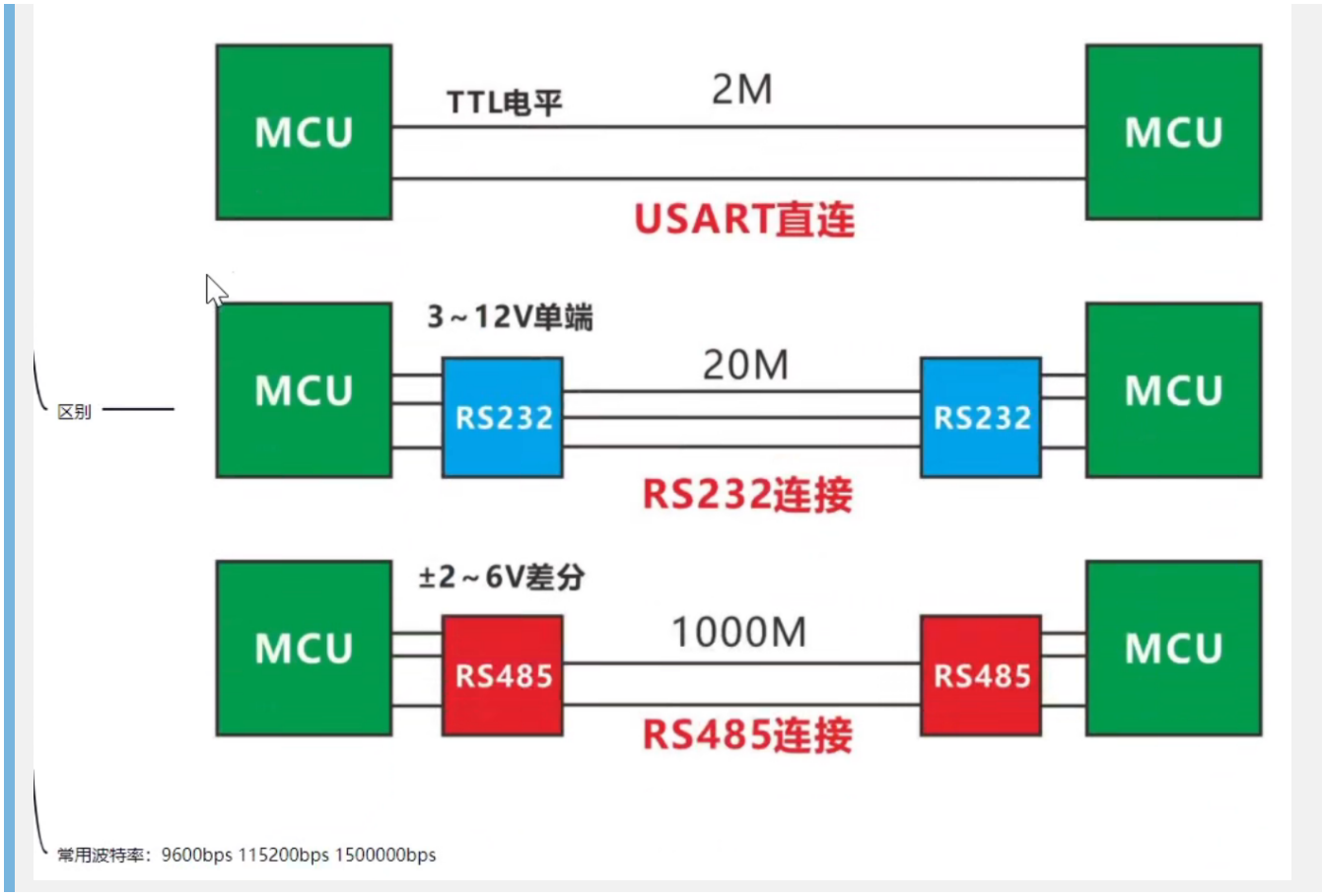
232


距离：最大15米左右
3-12V单端
速率：最高20Kbps
用途：常用与不同设备间一对一通讯，售货机主板与纸币机
RS232
原理图
+5V
C3
C4
C7
C5
U1
C1+
C1-
C2+
C2-
VDD
VCC
T1IN
T2IN
T1OUT
T2OUT
R1OUT
R2OUT
R1IN
R2IN
GND
VEE
PD1
PD0
+5V
MAX232ACPE
C6
RS232
D Connector 9

485

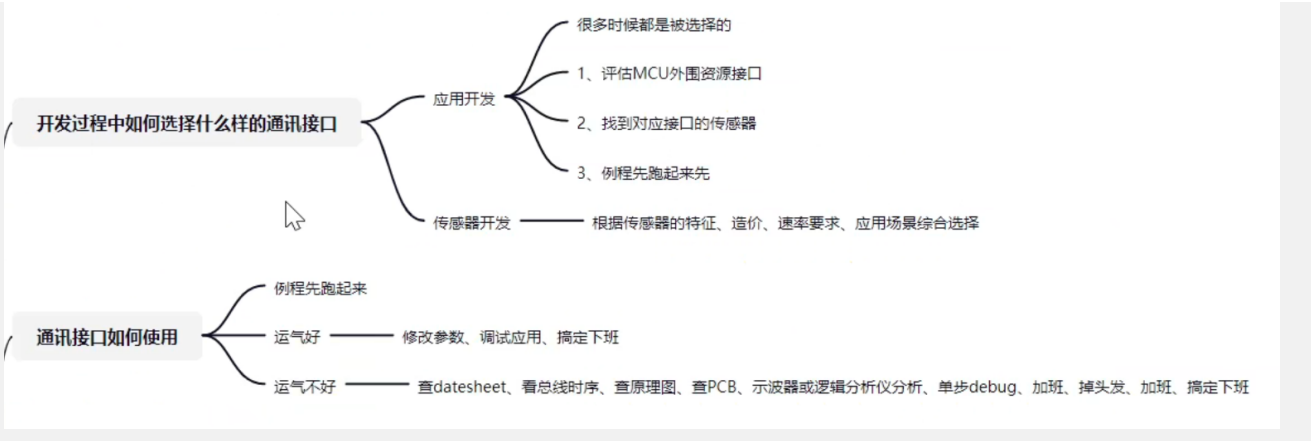


区别

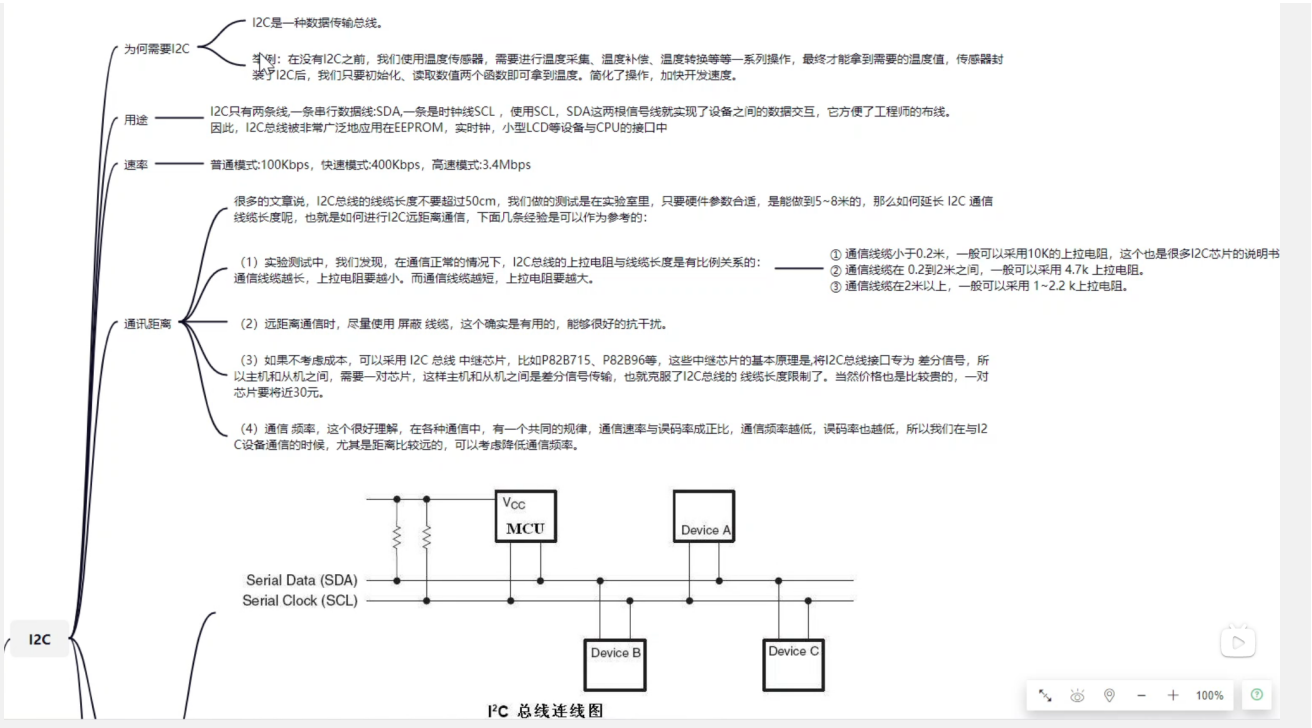


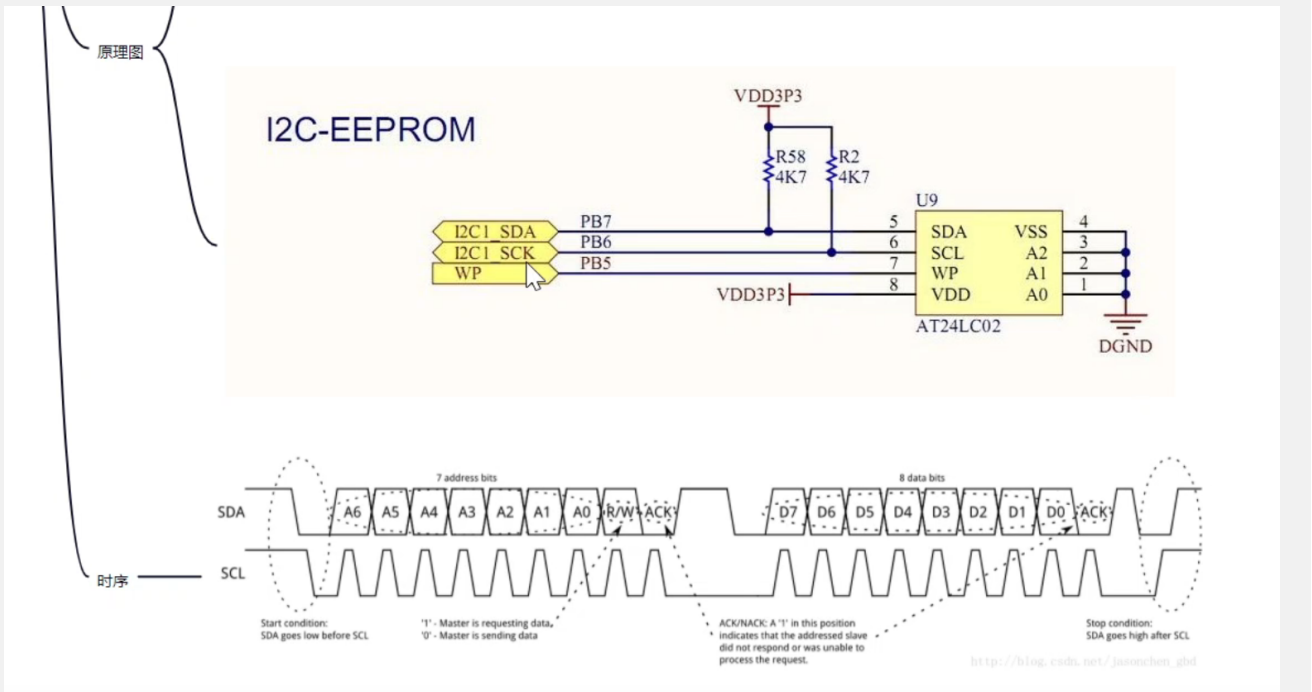
主控与模块通讯

开发过程中如何选择什么样的通讯接口 & 通讯接口如何使用


开发过程中如何选择什么样的通讯接口
应用开发
很多时候都是被选择的
1、评估MCU外围资源接口
2、找到对应接口的传感器
3、例程先跑起来先
传感器开发
根据传感器的特征、造价、速率要求、应用场景综合选择
通讯接口如何使用
例程先跑起来
运气好
修改参数、调试应用、搞定下班
运气不好
查datesheet、看总线时序、查原理图、查PCB、示波器或逻辑分析仪分析、单步debug、加班、掉头发、加班、搞定下班

IIC


I2C
为何需要I2C
I2C是一种数据传输总线。
举例：在没有I2C之前，我们使用温度传感器，需要进行温度采集、温度补偿、温度转换等等一系列操作，最终才能拿到需要的温度值，传感器封装了I2C后，我们只要初始化、读取数值两个函数即可拿到温度。简化了操作，加快开发速度。
用途
I2C只有两条线,一条串行数据线:SDA,一条是时钟线SCL，使用SCL，SDA这两根信号线就实现了设备之间的数据交互，它方便了工程师的布线。
因此，I2C总线被非常广泛地应用在EEPROM，实时钟，小型LCD等设备与CPU的接口中
速率
普通模式:100Kbps，快速模式:400Kbps，高速模式:3.4Mbps
通讯距离
很多的文章说，I2C总线的线缆长度不要超过50cm，我们做的测试是在实验室里，只要硬件参数合适，是能做到5~8米的，那么如何延长 I2C 通信线缆长度呢，也就是如何进行I2C远距离通信，下面几条经验是可以作为参考的：
（1）实验测试中，我们发现，在通信正常的情况下，I2C总线的上拉电阻与线缆长度是有比例关系的：通信线缆越长，上拉电阻要越小。而通信线缆越短，上拉电阻要越大。
① 通信线缆小于0.2米，一般可以采用10K的上拉电阻，这个也是很多I2C芯片的说明书
② 通信线缆在 0.2到2米之间，一般可以采用 4.7k 上拉电阻。
③ 通信线缆在2米以上，一般可以采用 1~2.2 k上拉电阻。
（2）远距离通信时，尽量使用 屏蔽 线缆，这个确实是有用的，能够很好的抗干扰。
（3）如果不考虑成本，可以采用 I2C 总线 中继芯片，比如P82B715、P82B96等，这些中继芯片的基本原理是,将I2C总线接口专为 差分信号，所以主机和从机之间，需要一对芯片，这样主机和从机之间是差分信号传输，也就克服了I2C总线的 线缆长度限制了。当然价格也是比较贵的，一对芯片要将近30元。
（4）通信 频率，这个很好理解，在各种通信中，有一个共同的规律，通信速率与误码率成正比，通信频率越低，误码率也越低，所以我们在与I2C设备通信的时候，尤其是距离比较远的，可以考虑降低通信频率。
Vcc
MCU
Device A
Serial Data (SDA)
Serial Clock (SCL)
Device B
Device C
I²C 总线连线图


原理图
I2C-EEPROM
VDD3P3
R58
4K7
R2
4K7
U9
I2C1_SDA
I2C1_SCK
WP
PB7
PB6
PB5
SDA
SCL
WP
VDD
VSS
A2
A1
A0
AT24LC02
DGND
时序
SDA
SCL
7 address bits
8 data bits
A6 A5 A4 A3 A2 A1 A0 R/W ACK
D7 D6 D5 D4 D3 D2 D1 D0 ACK
Start condition:
SDA goes low before SCL
'1' - Master is requesting data,
'0' - Master is sending data
ACK/NACK: A '1' in this position indicates that the addressed slave did not respond or was unable to process the request.
Stop condition:
SDA goes high after SCL

SPI

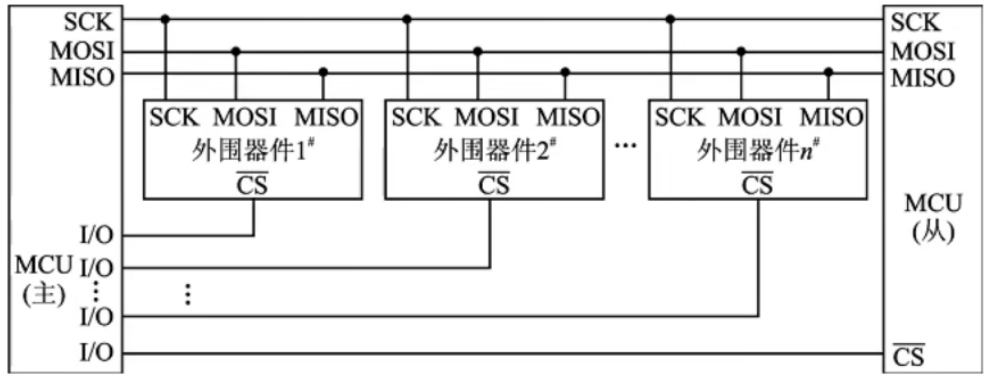
为何需要SPI —— 有了I2C总线后，如果我需要速度更快的总线呢？那SPI是一种选择

应用 —— LCD、NRF24L01、VS1053

特点 —— 是一种高速的，全双工，同步的通信总线

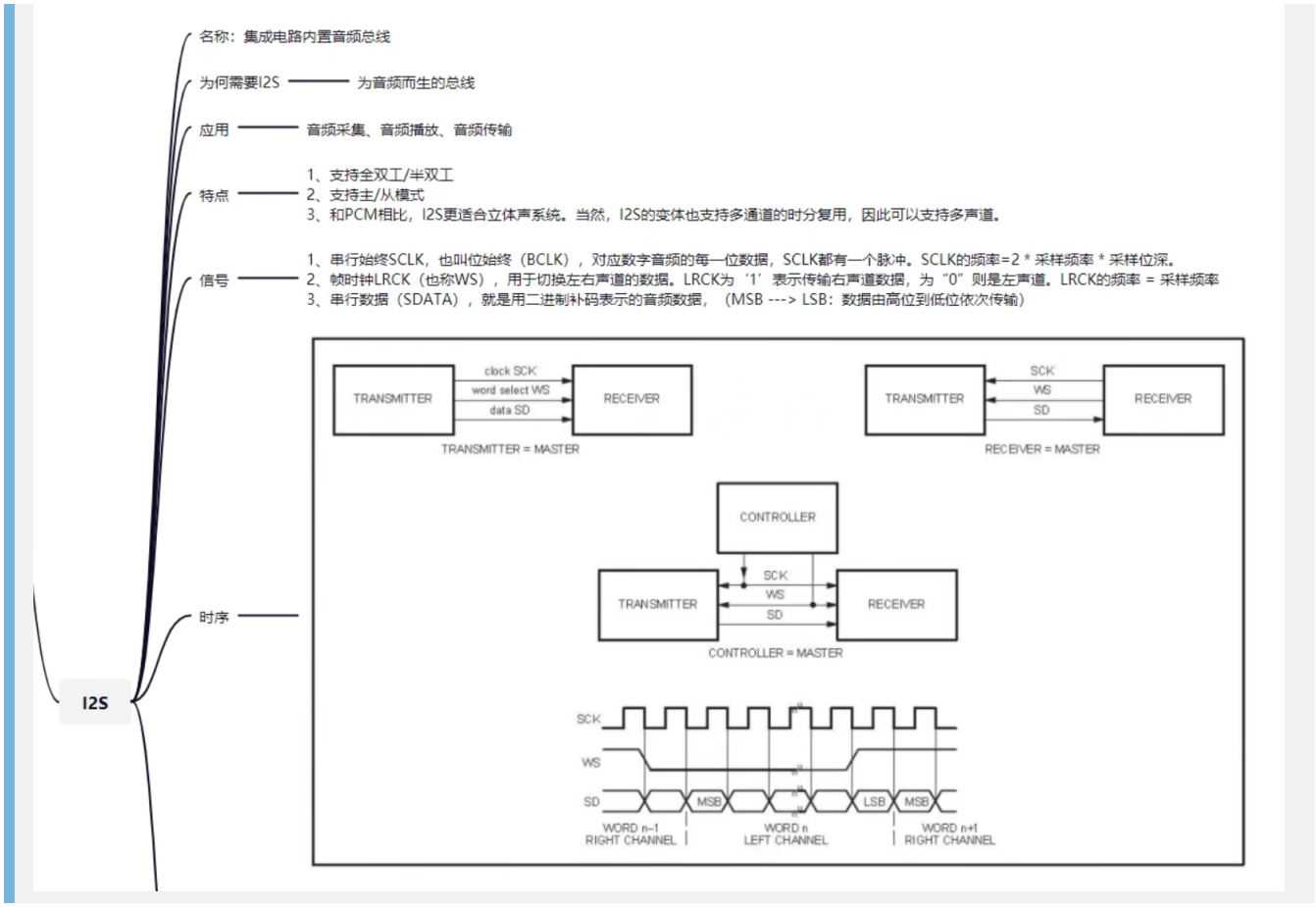
速率 —— 已知的有的器件SPI已达到50Mbps。具体到产品中SPI的速率主要看主从器件SPI控制器的性能限制。
SPI最大传输速率受以下几个条件影响：
SPI的最大时钟频率
CPU处理SPI数据的能力
输出端驱动能力（PCB所允许的最大信号传输速率）

距离 —— 一般20cm
最高扩展到2m

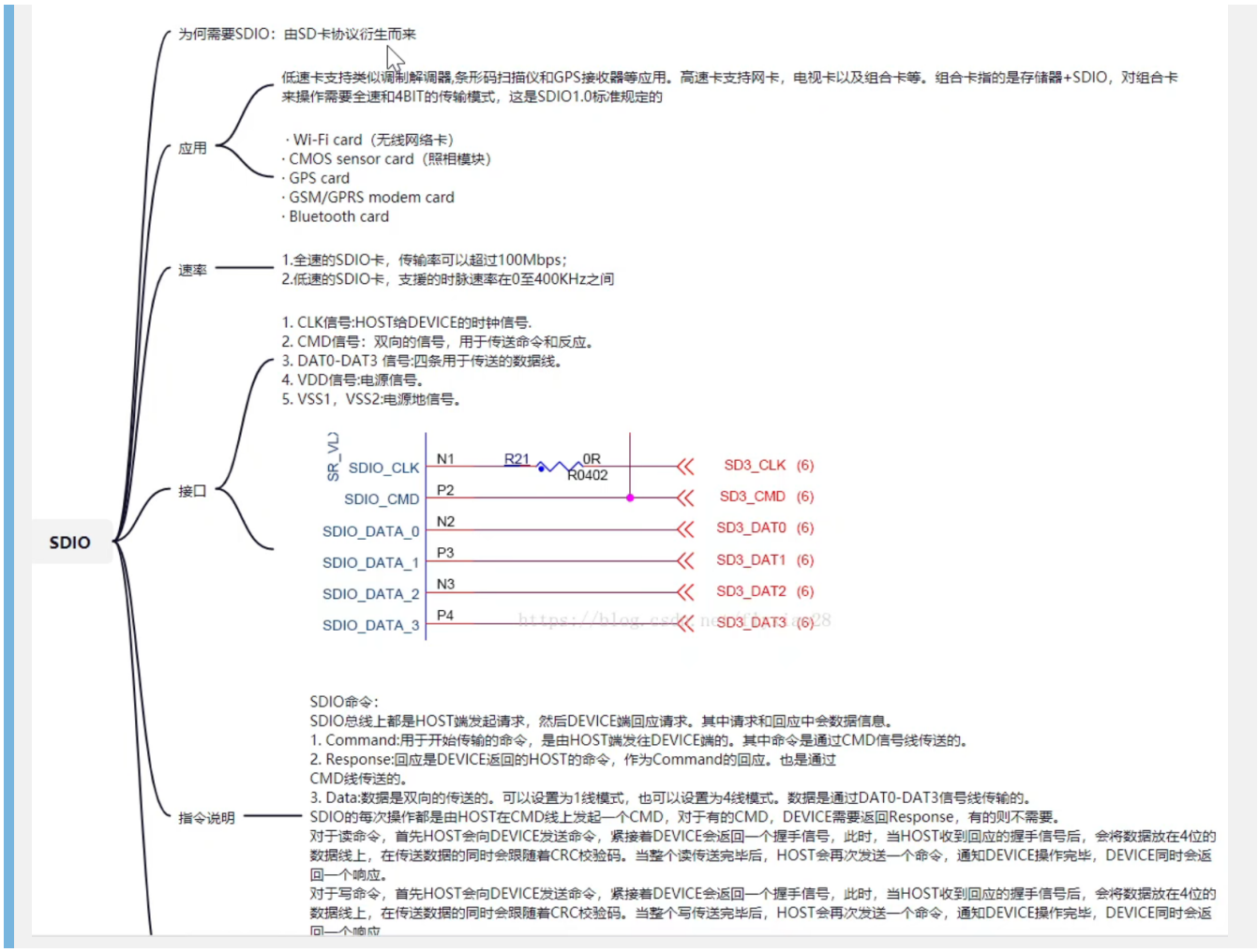
原理图 —— 

时钟极性CPOL是用来配置SCLK的电平出于哪种状态时是空闲态或者有效态，时钟相位CPHA是用来配置数据采样是在第几个边沿：
CPOL=0，表示当SCLK=0时处于空闲态，所以有效状态就是SCLK处于高电平时
CPOL=1，表示当SCLK=1时处于空闲态，所以有效状态就是SCLK处于低电平时
CPHA=0，表示数据采样是在第1个边沿，数据发送在第2个边沿
CPHA=1，表示数据采样是在第2个边沿，数据发送在第1个边沿

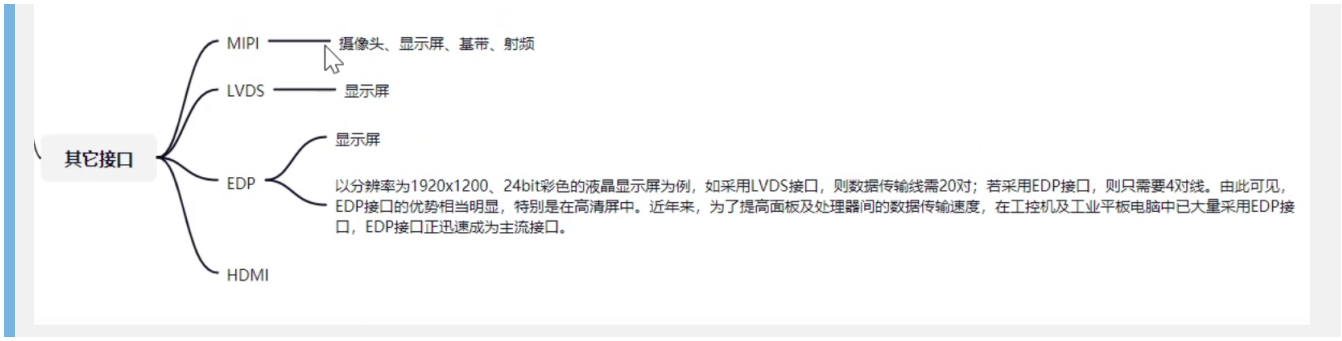
IIS


I2S
名称：集成电路内置音频总线
为何需要I2S
为音频而生的总线
应用
音频采集、音频播放、音频传输
特点
1、支持全双工/半双工
2、支持主/从模式
3、和PCM相比，I2S更适合立体声系统。当然，I2S的变体也支持多通道的时分复用，因此可以支持多声道。
信号
1、串行始终SCLK，也叫位始终（BCLK），对应数字音频的每一位数据，SCLK都有一个脉冲。SCLK的频率=2 * 采样频率 * 采样位深。
2、帧时钟LRCK（也称WS），用于切换左右声道的数据。LRCK为‘1’表示传输右声道数据，为“0”则是左声道。LRCK的频率 = 采样频率
3、串行数据（SDATA），就是用二进制补码表示的音频数据，（MSB ---> LSB：数据由高位到低位依次传输）
时序
TRANSMITTER
clock SCK
word select WS
data SD
RECEIVER
TRANSMITTER = MASTER
SCK
WS
SD
RECEIVER = MASTER
CONTROLLER
CONTROLLER = MASTER
MSB
LSB
WORD n-1
RIGHT CHANNEL
WORD n
LEFT CHANNEL
WORD n+1
RIGHT CHANNEL

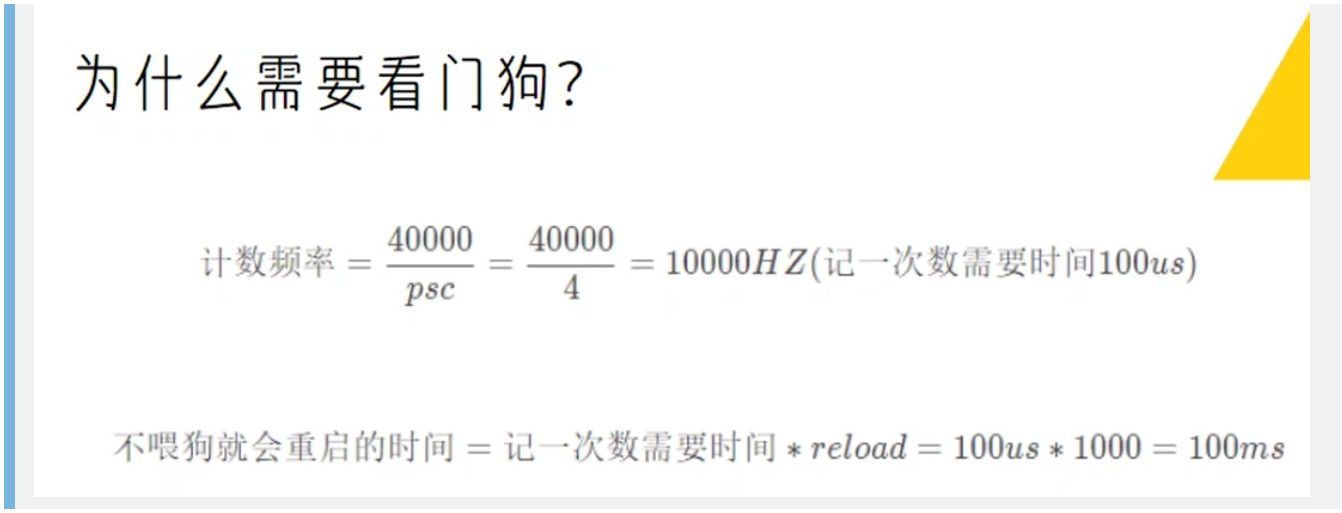
SDIO


SDIO
为何需要SDIO：由SD卡协议衍生而来
应用
低速卡支持类似调制解调器,条形码扫描仪和GPS接收器等应用。高速卡支持网卡，电视卡以及组合卡等。组合卡指的是存储器+SDIO，对组合卡来操作需要全速和4BIT的传输模式，这是SDIO1.0标准规定的
· Wi-Fi card（无线网络卡）
· CMOS sensor card（照相模块）
· GPS card
· GSM/GPRS modem card
· Bluetooth card
速率
1.全速的SDIO卡，传输率可以超过100Mbps；
2.低速的SDIO卡，支援的时脉速率在0至400KHz之间
接口
1. CLK信号:HOST给DEVICE的时钟信号.
2. CMD信号：双向的信号，用于传送命令和反应。
3. DAT0-DAT3 信号:四条用于传送的数据线。
4. VDD信号:电源信号。
5. VSS1，VSS2:电源地信号。
SR_VL)
SDIO_CLK
SDIO_CMD
SDIO_DATA_0
SDIO_DATA_1
SDIO_DATA_2
SDIO_DATA_3
N1
R21
0R
R0402
P2
N2
P3
N3
P4
SD3_CLK (6)
SD3_CMD (6)
SD3_DAT0 (6)
SD3_DAT1 (6)
SD3_DAT2 (6)
SD3_DAT3 (6)
指令说明
SDIO命令：
SDIO总线上都是HOST端发起请求，然后DEVICE端回应请求。其中请求和回应中会数据信息。
1. Command:用于开始传输的命令，是由HOST端发往DEVICE端的。其中命令是通过CMD信号线传送的。
2. Response:回应是DEVICE返回的HOST的命令，作为Command的回应。也是通过
CMD线传送的。
3. Data:数据是双向的传送的。可以设置为1线模式，也可以设置为4线模式。数据是通过DAT0-DAT3信号线传输的。
SDIO的每次操作都是由HOST在CMD线上发起一个CMD，对于有的CMD，DEVICE需要返回Response，有的则不需要。
对于读命令，首先HOST会向DEVICE发送命令，紧接着DEVICE会返回一个握手信号，此时，当HOST收到回应的握手信号后，会将数据放在4位的数据线上，在传送数据的同时会跟随着CRC校验码。当整个读传送完毕后，HOST会再次发送一个命令，通知DEVICE操作完毕，DEVICE同时会返回一个响应。
对于写命令，首先HOST会向DEVICE发送命令，紧接着DEVICE会返回一个握手信号，此时，当HOST收到回应的握手信号后，会将数据放在4位的数据线上，在传送数据的同时会跟随着CRC校验码。当整个写传送完毕后，HOST会再次发送一个命令，通知DEVICE操作完毕，DEVICE同时会返回一个响应

其他接口



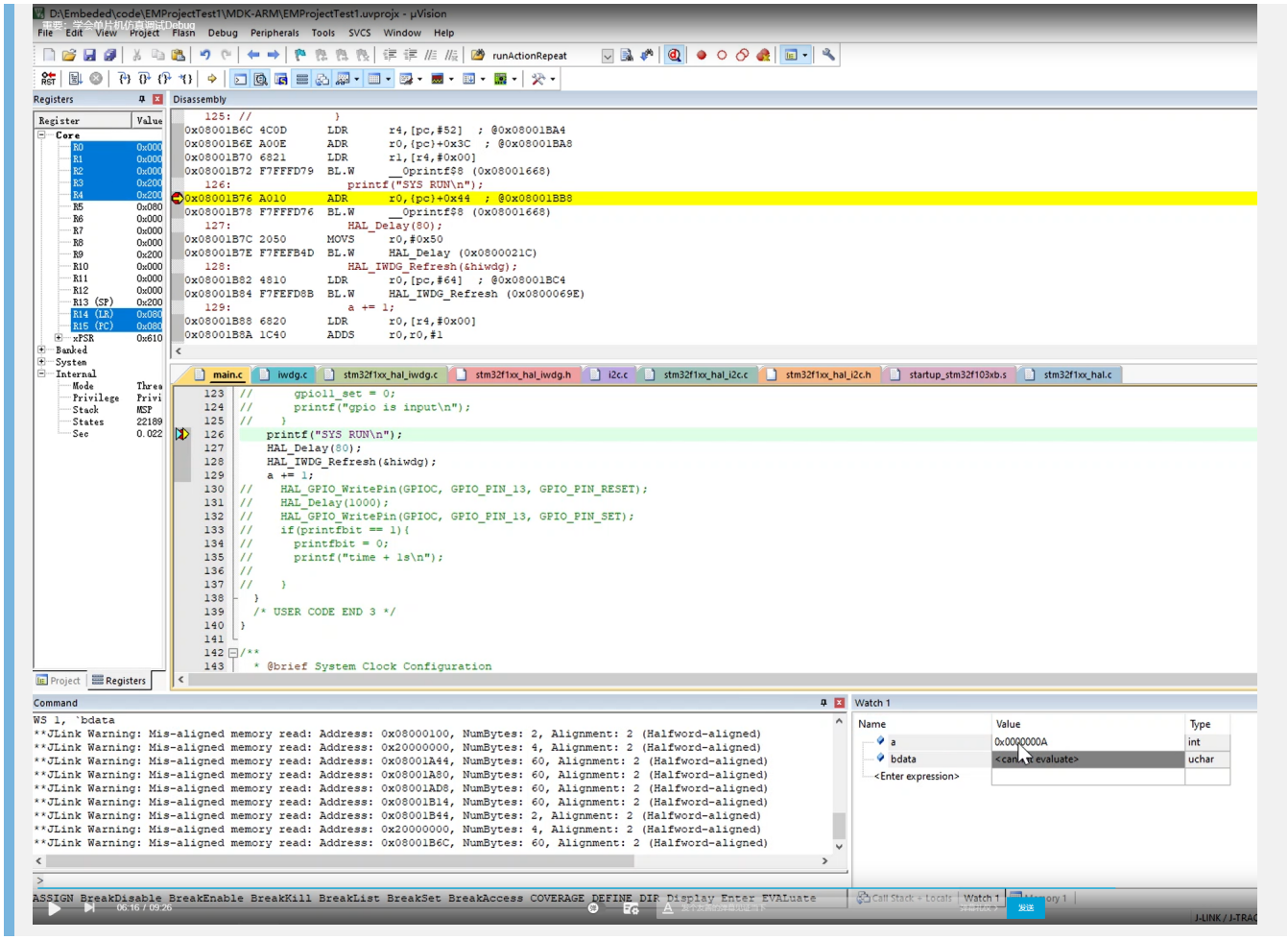
看门狗



调试Debug

可以将变量a的值，加入watch，查看变量的值得变化

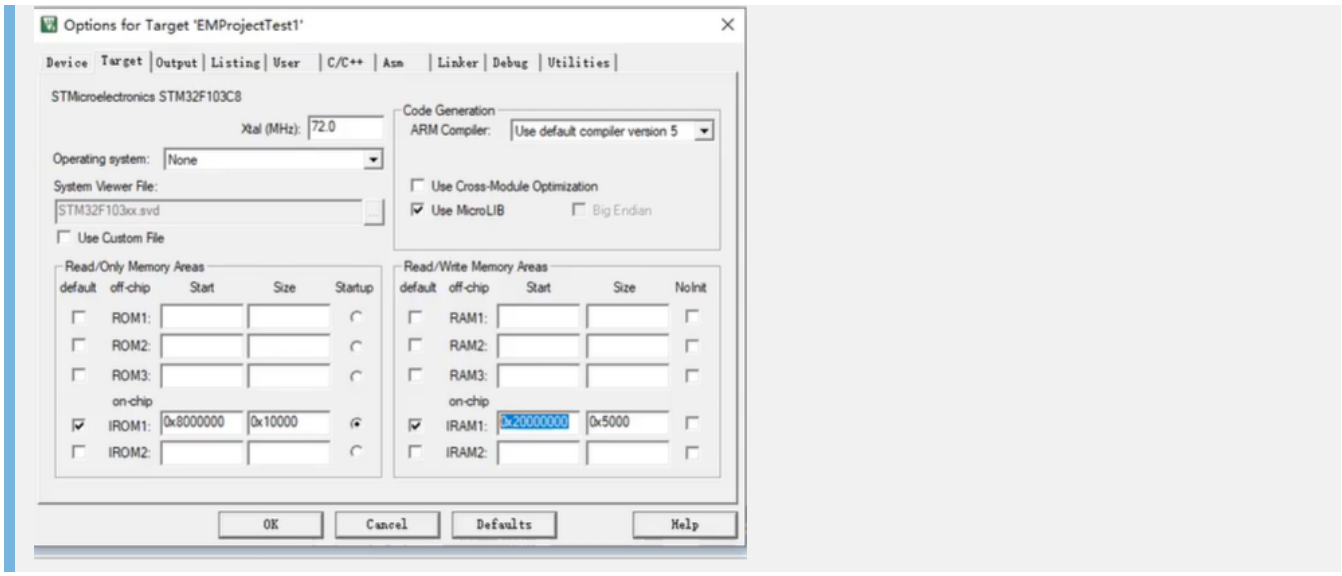
可以设置单步调试等



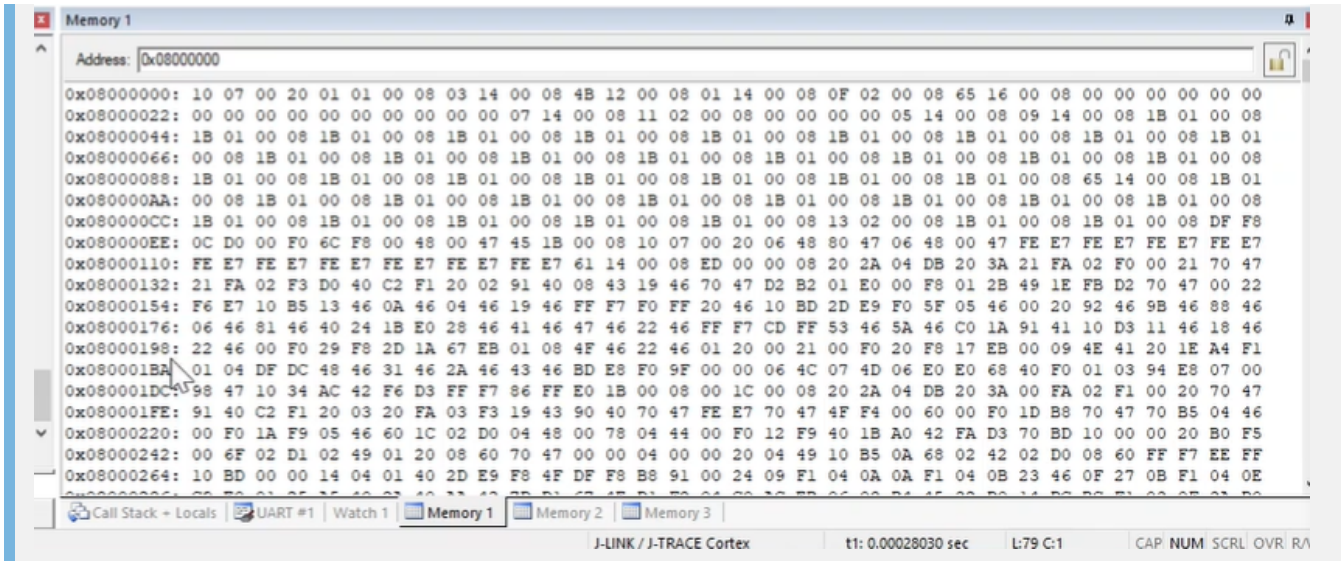
IAP & OTA远程升级

设备（物联网设备）远程升级、更新功能

单片机flash写入地址设置



调试中心，可查看内存内容



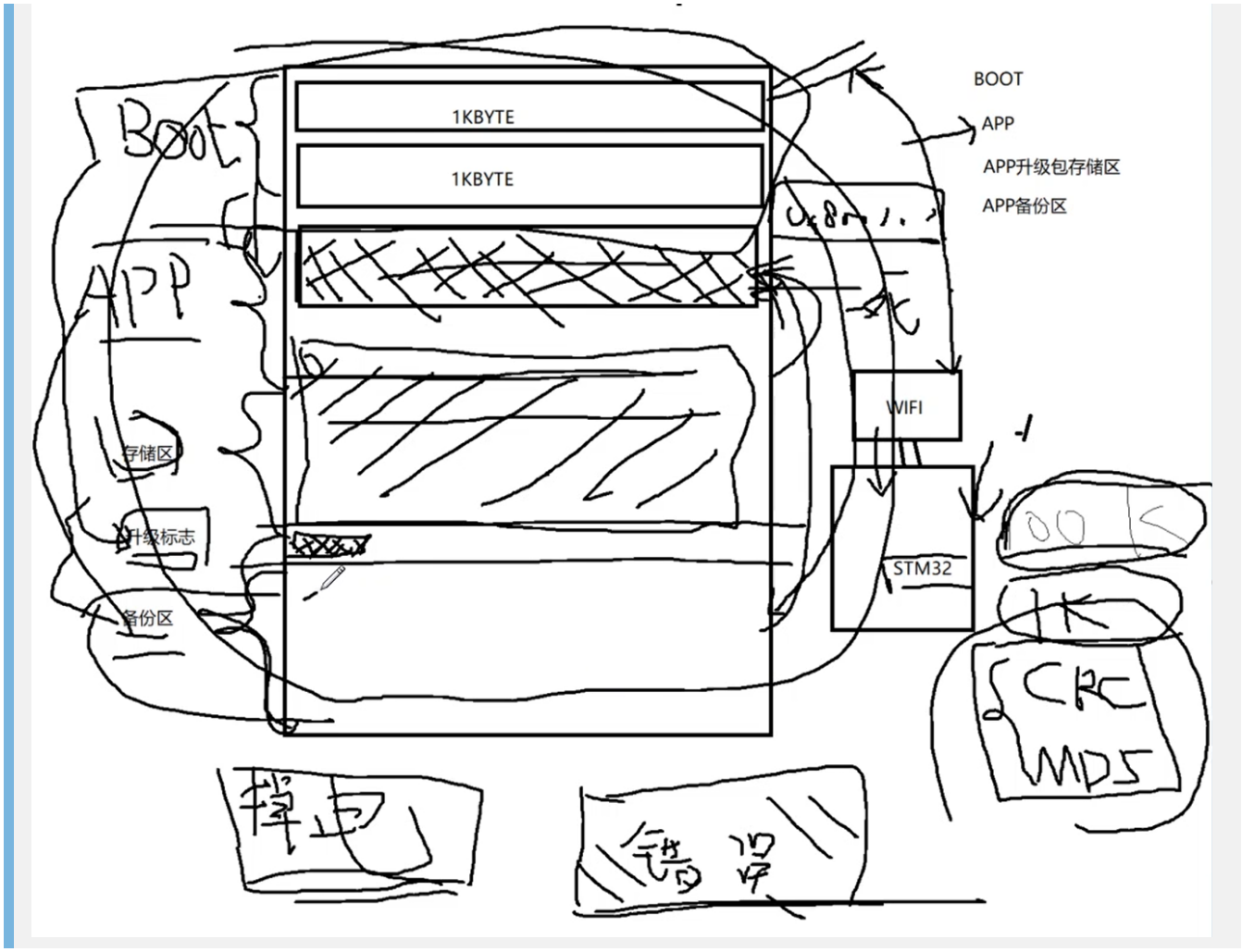
分区设置

BOOT区 -- 引导区，启动

APP区 --- 业务代码

APP升级包存储区 --- 存储接收到的升级代码（加密码，校验码等)

APP备份区 --- 存储备份，保证异常恢复



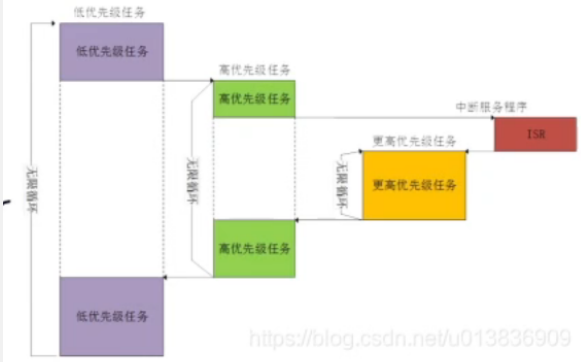
嵌入式RTOS

提高CPU使用率，更好的处理各个进程的运行

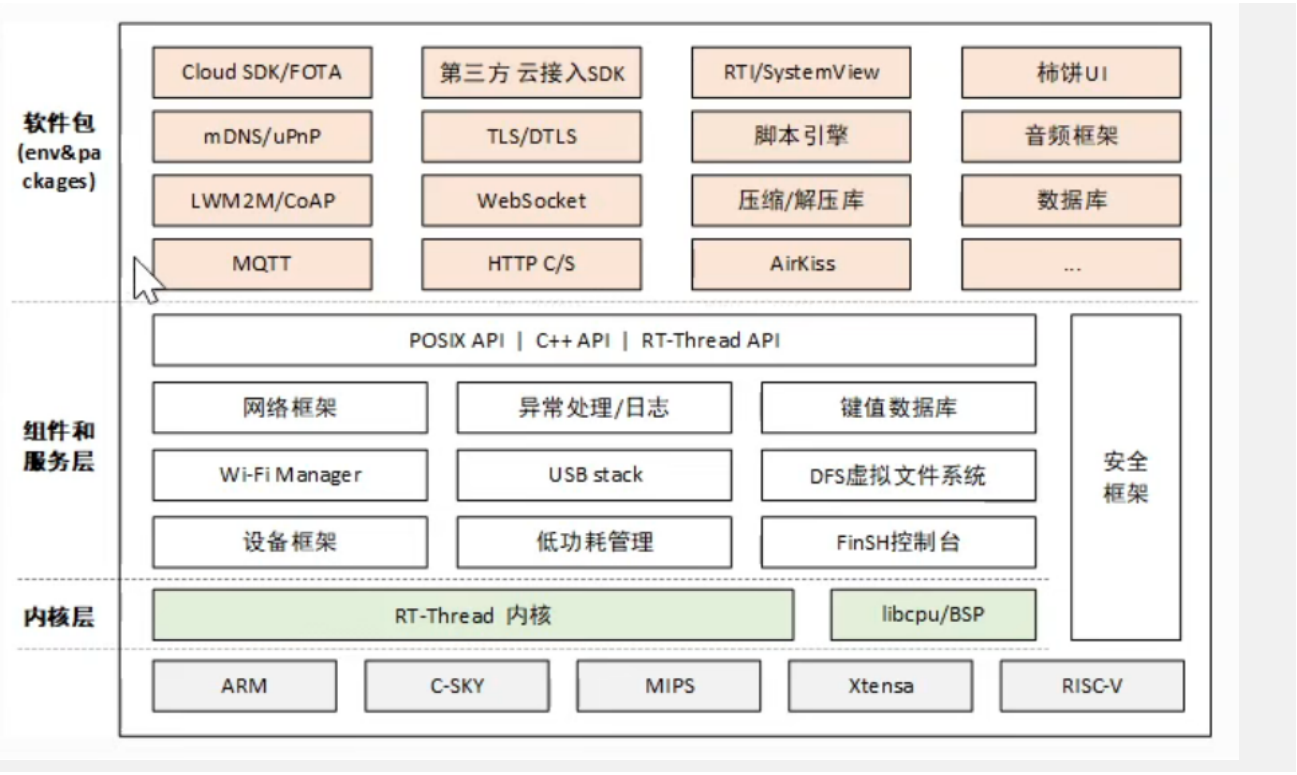
- 实时操作系统

  RTOS提供的事件驱动型设计方式，使得RTOS只是在处理实际任务时才会运行，这能够更合理的利用CPU。在实际项目中，如果程序等待一个超时事件，传统的无RTOS情况下，要么在原地一直等待而不能执行其它任务，要么使用复杂（相对RTOS提供的任务机制而言）的状态机机制。如果使用RTOS，则可以很方便的将当前任务阻塞在该事件下，然后自动去执行别的任务，这显然更方便，并且可以高效的利用CPU。

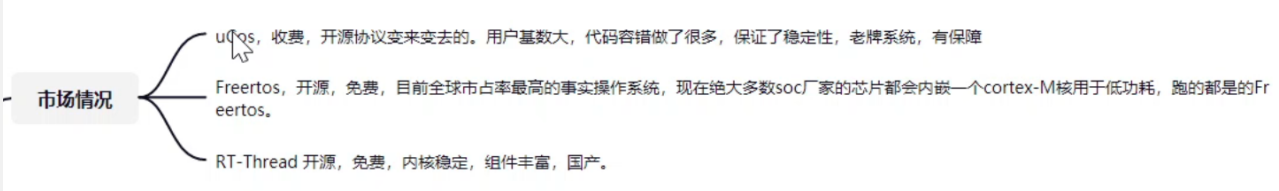
- 哪个场景需要RTOS?



架构图



市场情况



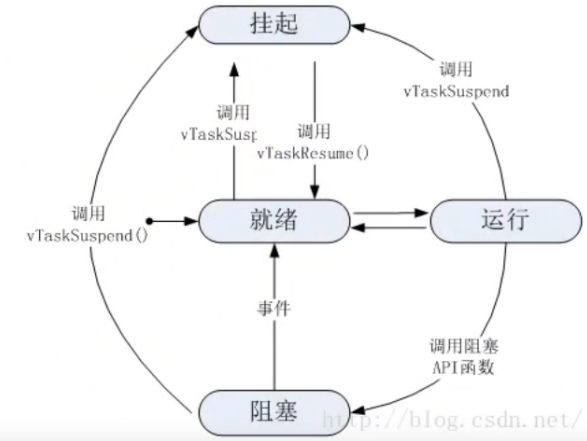
任务管理

为什么需要任务管理?

简介 —— 使用RTOS的实时应用程序可认为是一系列独立任务的集合。每个任务在自己的环境中运行，不依赖于系统中的其它任务或者RTOS调度器。在任何时刻，只有一个任务得到运行，RTOS调度器决定运行哪个任务。调度器会不断的启动、停止每一个任务，宏观看上去就像整个应用程序都在执行。作为任务，不需要对调度器的活动有所了解，在任务切入切出时保存上下文环境（寄存器值、堆栈内容）是调度器主要的职责。为了实现这点，每个任务都需要有自己的堆栈。当任务切出时，它的执行环境会被保存在该任务的堆栈中，这样当再次运行时，就能从堆栈中正确的恢复上次的运行环境。

特点 ——
支持完全抢占
支持优先级
每个任务都有自己的堆栈，消耗RAM较多

状态 ——



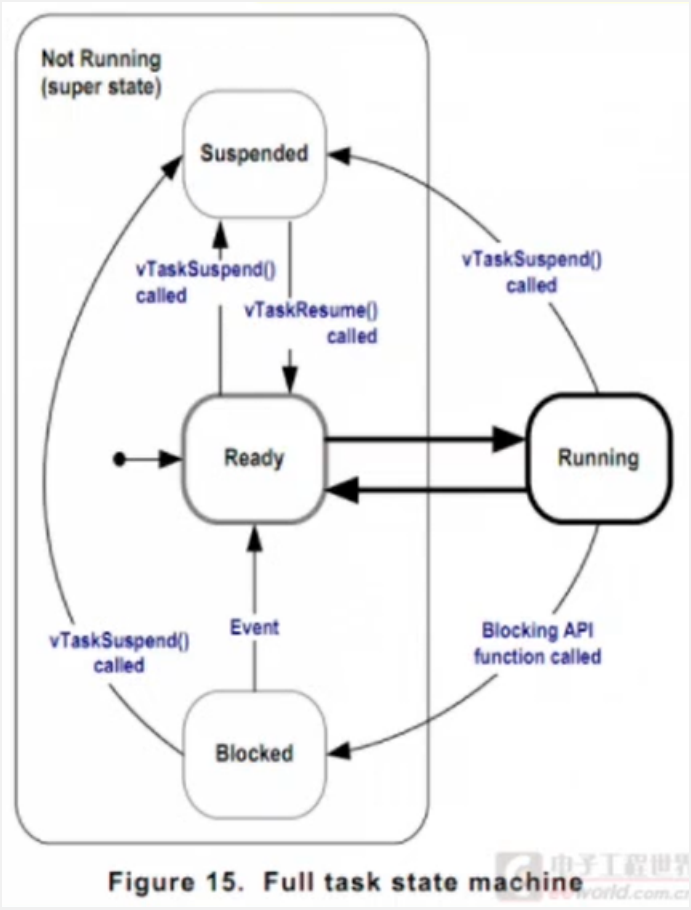


Figure 15. Full task state machine

常用函数

1. 任务创建 `xTaskCreat`
2. 任务删除 `vTaskDelete`
3. 任务挂起 `vTaskHandle`
4. 任务恢复 `vTaskResume`

任务延迟

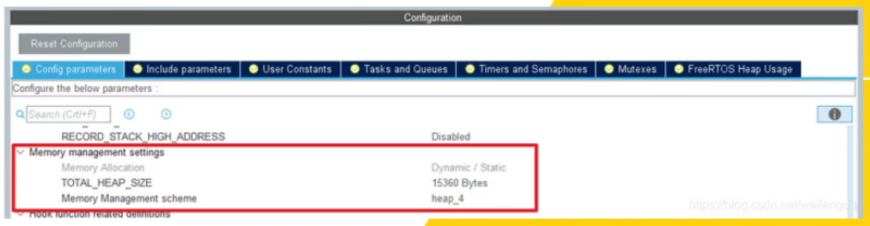
在FreeRTOS里面，提供了相对延迟函数和绝对延迟函数

相对延迟函数，并非严格的延迟，也包括一些中断的耗时

绝对延时函数，时间比较准确，用在如心率测试计时情景

> Systick 系统节拍定时器，Systick是ARM做的，所有相同内核通用。
> 普通定时器是各个厂家自己做的。
> HAL_Delay
>
> 相对延时函数vTaskDelay()和绝对延时函数vTaskDelayUntil()
> 相对延时是指每次延时都是从任务执行函数vTaskDelay()开始，延时指定的时间结束；
> 绝对延时是指每隔指定的时间，执行一次调用vTaskDelayUntil()函数的任务。
> 换句话说：任务以固定的频率执行。

内存管理

> 内存管理与程序好坏？对于安全要求高的应用，尽可能不使用动态内存分配。
>
> malloc()和free()
> 当RTOS内核需要RAM时，调用pvPortMalloc()函数来代替malloc()函数。当RAM要被释放时，调用vPortFree()函数来代替free()函数。
> freeRTOS层使用c语言申请内存的方式申请一个很大的数组（此处configTOTAL_HEAP_SIZE 的数值是15360），极端情况下，我们把可以申请到的所有空间全部申请下来。然后用户层调用的pvPortMalloc和vPortFree函数则是在此数组中操作，操作的函数原型自然是freeRTOS函数层中定义的。
>
> 
>

FreeRTOS提供的五种内存管理方案

FreeRTOS提供的五种内存分配方案
heap_1.c
适合一些不会删除任务，队列以及信号量的任务，特别是一些比较简单的系统和一些对安全性要求比较高的系统。事实上，很多系统中只要创建了任务之后就会一直执行，不会删除任务。所以这个文件适用环境还是比较多的。
heap_2.c
能分配，能回收，但是回收时不考虑内存碎片的场所。适合一些申请和释放操作不是特别频繁，而且不会申请大数组的场所。
heap_3.c
这种方法实际使用的还是c语言的malloc和free函数。与平台没有太大关系，线程保护。
heap_4.c
内存分配和释放操作比较频繁的系统。例如要重复创建删除任务，队列，信号量等。
heap_5.c
系统需要管理不连续的内存空间，例如接入外部ram的场所。

FreeRTOS内核规定的内存管理函数

FreeRTOS内核规定的几个内存管理函数原型为：
void *pvPortMalloc(size_t xSize)：内存申请函数
void vPortFree(void *pv)：内存释放函数
void vPortInitialiseBlocks(void)：初始化内存堆函数
size_t xPortGetFreeHeapSize(void)：获取当前未分配的内存堆大小
size_t xPortGetMinimumEverFreeHeapSize(void)：获取未分配的内存堆历史最小值

信号量 & 互斥量

信号量&互斥量:创建、获取、释放、删除
信号量用于同步，任务间或者任务和中断间同步；
互斥量用于互锁，用于保护同时只能有一个任务访问的资源，为资源上一把锁。

二值信号量：同步
计数信号量：资源使用统计
互斥量：互斥信号量其实就是一个拥有优先级继承的二值信号量
递归互斥：互斥中嵌套互斥

优先级反转，由于高优先级任务在等低优先级任务释放信号量的过程中，低优先级被中优先级任务抢夺CPU，最终导致中优先级任务比高优先级任务先运行，也就是优先级翻转
而互斥信号量就可以将这个影响降到最低，当高优先级任务发现需要信号量被低级信号量占用需要等待时，就将低优先级任务变成和自己同级优先级，就不至于在等待的过程中被中优先级抢夺CPU。

消息队列

比如，传结构体：

消息队列，传递的是结构体的指针，

邮箱队列，传递的是整个结构体（开辟一块内存，将结构体复制）。

> 消息队列
> 1、任务级队列处理函数
> 2、中断级队列处理函数（带中断保护）已经在CMSIS接口中封装
> 但写入生产速度比消费速度快的时候，容易出现数据被覆盖
>
> 邮箱队列
> 创建、发送、接收、查询、删除

嵌入式LINUX

快捷键

- 打开新终端 `ctrl + alt + t`
- 终端多开 `ctrl + shift + t`
- 终端切换 `alt + num`
- 查看提示 `ls --help`
- 打印当前路径 `pwd`
- 创建新文件并打开 `gedit test.txt` && `vim test.txt`
- 删除文件夹 `rm test -r`
- 搜索文件 `find -name test.txt`
- 查看文件里的内容 `grep -r "hello"` && `cat test.txt`
- 增加使用权限 `chmod 777 test.txt`
- 进入连接到linux的开发板中 `adb devices adb push`
- 把开发板中的文件下载到linux中 `adb pull`
- tar解压命令是：

```
1、tar -xvf file.tar //解压 tar包

2、tar -xzvf file.tar.gz //解压tar.gz

3、tar -xjvf file.tar.bz2 //解压 tar.bz2

4、tar -xZvf file.tar.Z //解压tar.Z

5、unrar e file.rar //解压rar
```

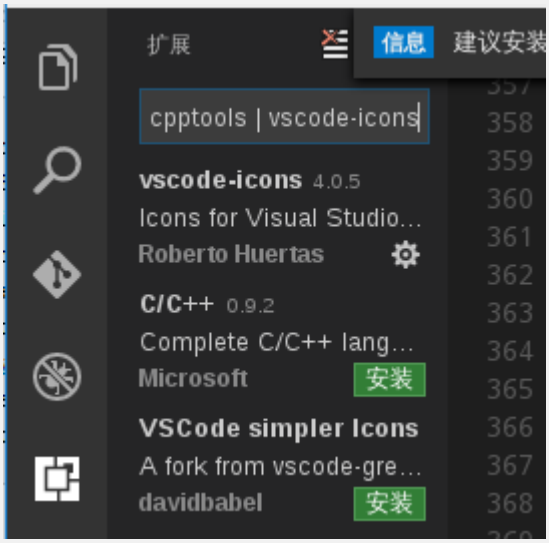
```
6、unzip file.zip //解压zip
```

虚拟机中安装VS Code

> ## 1.从官网下载压缩包(话说下载下来解压就直接可以运行了咧,都不需要make)
>
> 访问Visual Studio Code官网 https://code.visualstudio.com/docs?dv=linux64
>
> 我是64位的:
>
> wget https://az764295.vo.msecnd.net/stable/7ba55c5860b152d999dda59393ca3ebeb1b5c85f/code-stable-code_1.7.2-1479766213_amd64.tar.gz
>
> ## 2.解压 ,如果文件名不对,可能解压不出来的(扩展名:tar.gz)
>
> tar jxcv code-stable-code_1.7.2-1479766213_amd64.tar.gz
>
> ## 3.然后移动到 /usr/local/ 目录
>
> mv VSCode-linux-x64 /usr/local/
>
> ## 4.可能还需要给可执行的权限, 然后就已经可以运行了
>
> chmod +x /usr/local/VSCode-linux-x64/code
>
> ## 5.复制一个VScode图标文件到 /usr/share/icons/ 目录(后面会有用)
>
> cp /usr/local/VSCode-linux-x64/resources/app/resources/linux/code.png /usr/share/icons/
>
> ## 6.创建启动器, 在/usr/share/applications/ 目录, 也可以将它复制到桌面目录
>
> 直接在中断 使用 命令:
>
> vim /usr/share/applications/VSCode.desktop
>
> 然后输入以下文本:
>
> [](javascript:void(0)😉
>
> ```
> [Desktop Entry]
> Name=Visual Studio Code
> Comment=Multi-platform code editor for Linux
> Exec=/usr/local/VSCode-linux-x64/code
> Icon=/usr/share/icons/code.png
> Type=Application
> StartupNotify=true
> Categories=TextEditor;Development;Utility;
> MimeType=text/plain;
> ```

> [ ](javascript:void(0))
>
> 保存后退出, 然后可以复制到桌面:
>
> cp /usr/share/applications/VSCode.desktop ~/桌面/
>
> 之后 就会发现 桌面和 应用程序菜单都有了 VSCode的快捷方式了

8.打开VSCode, 加载插件: cpptools | vscode-icons



Linux工程搭建

1. 最朴素 `gcc main.c -o main`
2. 稍微大一点工程，可手写 Makefile
3. 再大一点工程，推荐用 autotools(本质是自动化生成makefile)
4. 大工程更推荐 cmake(语法简单，高效)
    - 安装 `sudo apt-get install cmake`
    - 查看版本 `cmake -version`
    -

> 1. 安装cmake，并编写 CmakeLists.txt。
> 2. 执行命令“cmake PATH”或者“ccmake PATH”生成 Makefile（PATH 是 CMakeLists.txt 所在的目录）。
> 3. 使用 make 命令进行编译。

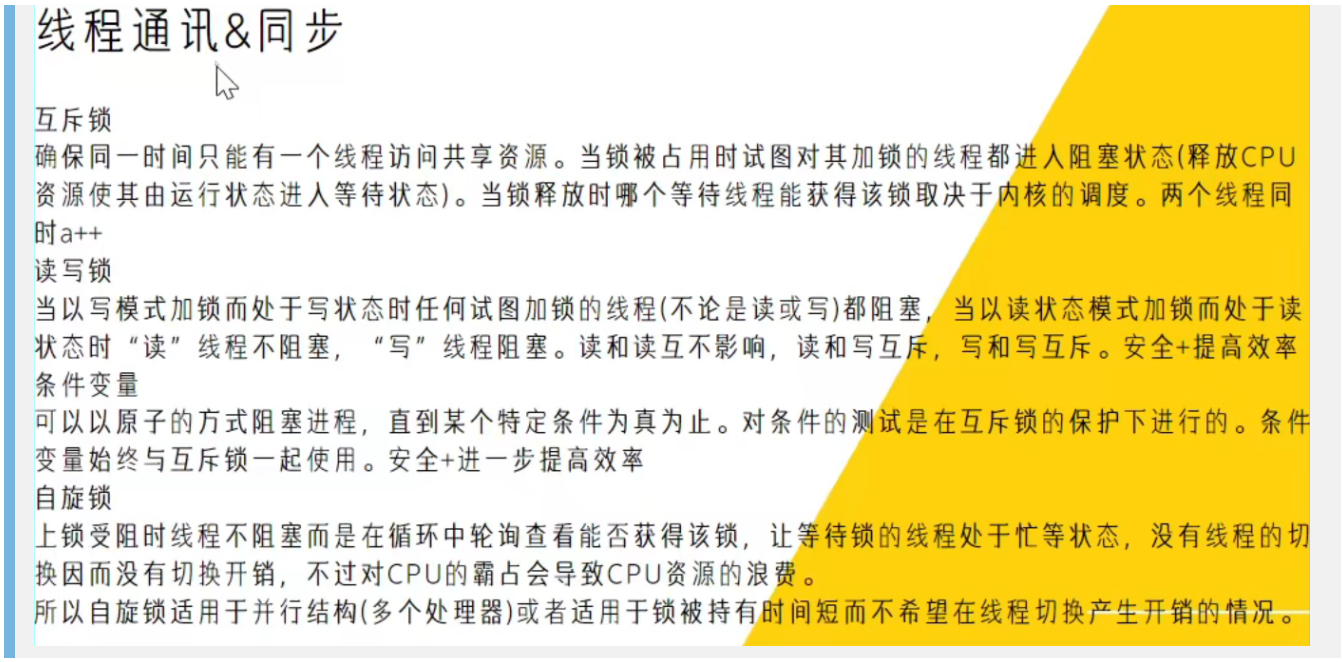
```
cmake工程
#确定cmake最低版本需求
cmake_minimum_required(VERSION 3.0.0)
#打印
MESSAGE(STATUS "This is install dir " ${CMAKE_INSTALL_PREFIX})
#确定工程名
project(XXX)
#添加需要的库
set(CMAKE_PREFIX_PATH ${CMAKE_PREFIX_PATH} "/usr/local/share/OpenCV")
find_package(OpenCV 3.2.0 REQUIRED)
#添加需要的头文件
include_directories(include)
include_directories(${OpenCV_INCLUDE_DIRS})
include_directories(/usr/local/cuda-8.0/include/)
#确定编译语言
#add_definitions(-std=c++11)
set(CMAKE_CXX_STANDARD 11)
#设定变量
ADD_DEFINITIONS( -DGPU -DCUDNN )
#添加源代码
aux_source_directory(${CMAKE_CURRENT_LIST_DIR}/src src)
#编译动态库并链接库文件
link_directories(${PROJECT_SOURCE_DIR})
add_library(plate_recognition SHARED ${SRC})
target_link_libraries(plate_recognition -lianghao  -lpthread -lm -lstdc++)
#生成可执行文件
link_directories(${PROJECT_SOURCE_DIR})
add_executable(Test ${SRC})
target_link_libraries(Test -lianghao  -lpthread -lm -lstdc++)
#make install
```

Linux多线程的使用

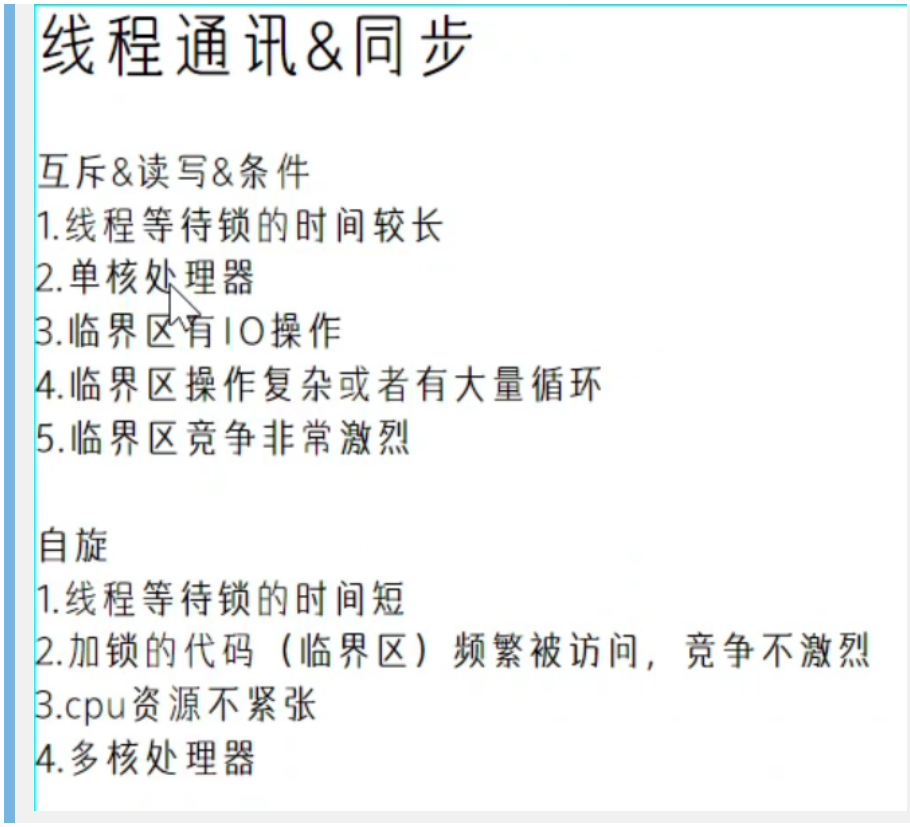
不同系统的多线程比较

事项	WIN32	VxWorks	Linux
线程创建	CreateThread	taskSpawn	pthread_create
线程终止	执行完成后退出；线程自身调用ExitThread函数即终止自己；被其他线程调用函数TerminateThread函数	执行完成后退出；由线程本身调用exit退出；被其他线程调用函数taskDelete终止	执行完成后退出；由线程本身调用pthread_exit 退出；被其他线程调用函数pthread_cance终止
获取线程ID	GetCurrentThreadId	taskIdSelf	pthread_self
创建互斥	CreateMutex	semMCreate	pthread_mutex_init
获取互斥	WaitForSingleObject、WaitForMultipleObjects	semTake	pthread_mutex_lock
释放互斥	ReleaseMutex	semGive	phtread_mutex_unlock
创建信号量	CreateSemaphore	semBCreate、semCCreate	sem_init
等待信号量	WaitForSingleObject	semTake	sem_wait
释放信号量	ReleaseSemaphore	semGive	sem_post

线程通讯 & 同步



自旋锁 与 互斥&读写&条件变量 对比



信号量

> 信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量
> 信号机制(Signal)：类似进程间的信号处理
>
> 信号量&互斥量=同步&资源保护
>
> 检测按键信号、处理按键事件用信号量
> 任务从缓存区读出需要处理的订单，需要用互斥量

Linux多进程的使用

多进程

> linux c多进程
> 什么是进程?
> 什么时候用进程&线程?
> 1、需要频繁创建销毁的优先使用线程；因为对进程来说创建和销毁一个进程代价是很大的。
> 2、线程的切换速度快，所以在需要大量计算，切换频繁时用线程，还有耗时的操作使用线程可提高应用程序的响应。
> 3、多进程可以使用在多机分布式系统，需要扩展到其他机器上，使用多进程，多线程适用于多核处理机。
> 4、需要更稳定安全时，适合选择进程；需要速度时，选择线程更好。

进程通信方式

socket通常用于多机通讯

linux c多进程 进程通信方式

1. 管道（Pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信；
2. 信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）；
3. 队列（消息队列）：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
4. 共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
5. 信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。
6. 套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

消息队列和管道对比

消息队列：用于消息，不是简单的数据信息传递，消息队列还包括消息有优先级、消息到达通知等丰富内容。
管道：低级的通信机制，消息队列比管道高级多了，管道分PIPE和FIFO，PIPE是无名的，所以只能在进程内或父子进程间通信，FIFO可任何两个进程间通信了。不过这两个依然比较低级，完成高级的应用服务器还需要消息队列等。

概念	管道是Unix IPC形式，用于在一个方向上提供信息流。	Message Queue是一个系统VIPC表单，用于存储消息列表。
创建	可以使用 pipe() 函数创建管道，该函数返回两个文件描述符，一个用于读取，另一个用于写入。	使用 msgget() 返回队列标识符的函数创建消息队列。
方向	管道是单向的。	消息队列是双向的。
资料撷取	数据可以先进先出的方式在FIFO中获取。	数据可以以任何顺序获取。
优先事项	管道中没有优先级。	通过在消息类型上附加优先级编号，可以使消息具有优先级。
接收者	为了使管道正常运行，应该存在发送方和接收方进程，以等待在管道中写入和读取消息。	在消息队列中，编写器进程可以编写消息并退出。阅读器进程可以稍后阅读消息。
坚持不懈	如果没有链接的接收器/发送器进程，则将管道从系统中删除。	消息队列在系统中保持活动状态，直到被某个进程显式删除为止。
讯息大小	管道消息的大小最大为4096字节。	消息队列消息大小最多可以为8192字节。

Linux文件编程

日志保存、音视频数据保存、运行数据保存
Linux 一切皆文件 io/iic/hid/
Open close read write

案例代码

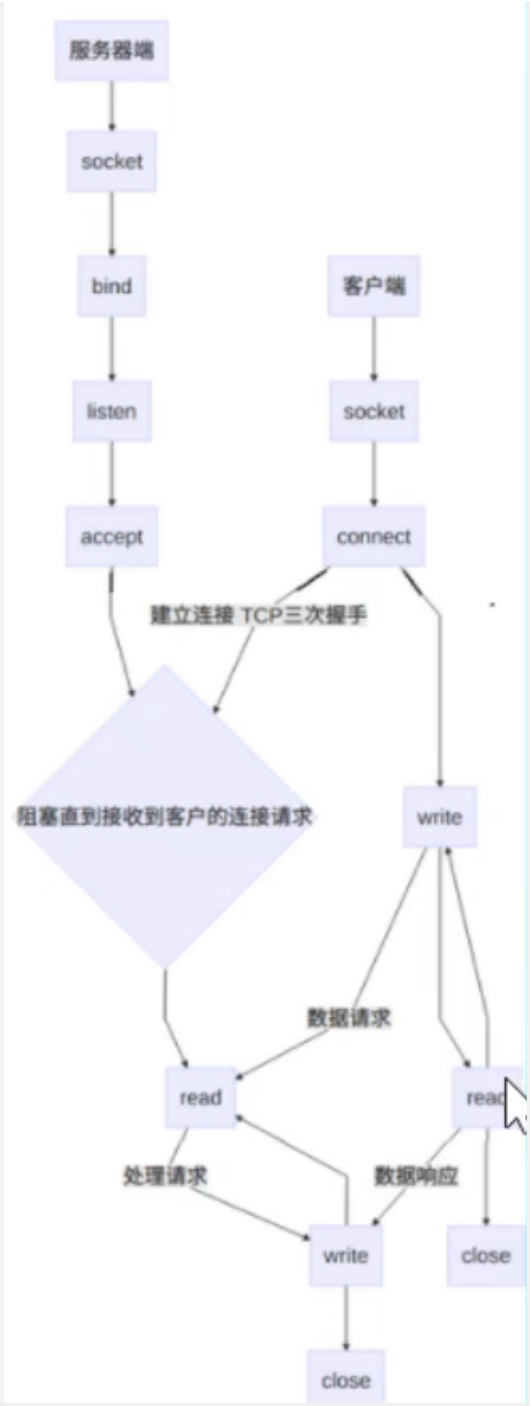
```c
#include <stdio.h>
#include <string.h>
#include <fcntl.h>

int main(void)
{
  int fd,len;
  char *buf = "Hello World!\n",Out[30];
  fd = open("a.txt"O_CREAT | O_TRUNC | O_RDWR, 0600);
  printf("open file:a.txt fd = %d\n", fd);
  len = strlen(buf);
  int size  = write(fd, buf, len);
  close(fd);
  // //Begin to read the file
  fd = open("a.txt", O_RDWR, 0600);
  lseek(fd, 0, SEEK_SET);
  size = read(fd, Out, 12);
  printf("size = %d\nread from file:\n %s\n",size,Out);
  close(fd);
  return 0;
}
```

Linux网络编程

网络7层模型


linux 网络编程
应用层
程序及接口
表示层
对数据进行转换、加密和压缩
会话层
建立、管理和终止会话
传输层
提供可靠的端到端的报文传输和差错控制
网络层
将分组从源端传送到目的端；提供网络互联
数据链路层
将分组数据封装成帧；提供节点到节点方式的传输
物理层
在媒体上传输比特；提供机械的和电气的规约
HTTP 协议对应于应用层，Socket 则是对 TCP/IP 协议的封装和应用
Socket 的出现只是使得程序员更方便地使用 TCP/IP 协议栈而已，是对TCP/IP协议的抽象，从而形成了我们知道 的一些最基本的函数接口，比如 create、listen、connect、accept、send、read和 write 等。
HTTP HTTPS WebSocket MQTT RTMP RTSP

服务器-客户端通讯过程


服务器端
socket
bind
listen
accept
客户端
socket
connect
建立连接 TCP三次握手
阻塞直到接收到客户的连接请求
write
数据请求
read
处理请求
数据响应
write
close
close