Bootloader 问题

1. 中断向量表

这张表就是**中断向量表。**

可以把它想象成下面这样：

编号	发生的事件	CPU要跳转的地址
0	上电复位	Reset_Handler
1	严重错误	HardFault_Handler
2	定时器中断	Timer_Handler
3	串口接收中断	UART_Handler
4	ADC完成中断	ADC_Handler

真实的向量表里，保存的主要不是函数名称，而是这些函数在存储器中的**地址。**

例如：

```
定时器中断发生
      ↓
CPU知道它是第20号中断
      ↓
去向量表找第20项
      ↓
读出定时器中断函数的地址
      ↓
跳到这个地址执行
```

中断向量表默认放在 flash 的开头，且向量表第一项存储的是初始栈指针，

所以单片机刚上电时，CPU通常会：

1. 读取向量表第0项，设置栈指针SP
2. 读取向量表第1项，跳转到Reset_Handler
3. Reset_Handler进行初始化
4. 最后进入main函数

因此，中断向量表不只负责普通中断，还负责程序最开始的启动。

对于 bootloader 来说，他是和 app 分离开的如下：（放在同一个 flash 中，但是需要分区，所以他们的栈顶指针一样，但是他们的复位地址不同，需要自己分配）



所以刚上电时候，用的是 bootloader 的向量表，而后才由 bootloader 决定跳转到 APP

所以 Bootloader 跳转 APP 时，通常要完成几件事：

1. 关闭或清理 Bootloader 使用的中断
2. 找到 APP 向量表的位置
3. 读取 APP 的初始栈指针
4. 设置 CPU 使用 APP 的向量表
5. 跳转到 APP 的复位处理函数



中断向量表的首地址是什么，怎么来的：

初始栈顶地址

= RAM 首地址

+ .bss 大小

+ .data 大小

+ 其他区域大小

+ 栈大小

但是一般都是 RAM 首地址+RAM 大小

因为 RAM 大小里就包含了.data 与.bss

假如初始栈地址为 0x20010000 存入数据需要往下减的



压入 4 字节后



2. 复位函数地址

单片机上电或复位之后，第一段程序应该从哪里开始执行。

CPU上电后做什么

假设向量表开头是：

```
10 04 00 20 | 49 01 00 08
```

按照小端格式解析：

```
第0项：0x20000410  初始栈顶地址
第1项：0x08000149  Reset_Handler地址
```

CPU复位后会自动进行：

```
SP = 0x20000410
PC = 0x08000149
```

其中：

- `SP` 是栈指针；
- `PC` 是程序计数器，决定CPU下一条指令去哪里执行。

所以把 `0x08000149` 放入 `PC` 后，CPU就会跳到对应位置执行 `Reset_Handler`。

Reset_Handler

    │

    ├── 初始化栈

    ├── 初始化系统时钟

    ├── 把.data 初始值从 Flash 复制到 RAM

    ├── 把.bss 区域清零

    ├── 初始化 C 语言运行环境

    └── 调用 main()

为什么不能上电直接执行 main

如

```
uint32_t count = 100;
uint32_t error_count;
```

count 属于 .data，它的初始值 100 先保存在 Flash 中，启动时要复制进 RAM；

error_count 属于 .bss，启动时需要清零。

这些事情都需要在进入 main() 前完成。

否则执行 main() 时，变量的初始值可能不正确。

3. 进制与字节

字节 Byte　　位 bit　　一个字节（Byte）= 8 个位（bit）　即一字节= 0000 0000

一个十六进制 =　四位二进制 = 0000

于是两个十六进制等于一字节

4. 函数指针（指向函数的指针）

函数指针的价值不是传参，而是“保存一个函数的位置，然后通过地址调用它”。

```
uint32_t app_reset_handle = 0x0800149
typedef void (*pFunc)(void);
pFunc jump_to_app = (pFunc)app_reset_handle;
jump_to_app();
```

第一个为函数指针，按理说 typedef 写法是 typedef A B，以后出现的 B 都代表 A

对于这个函数则为 typedef void（* ）（void）pFunc

变量名 A 为 void（* ）（void），函数名 B 为 pFunc。

所以 pFunc 代表的就是 void（* ）（void）的一个函数指针

对于 pFunc jump_to_app， jump_to_app 是一个指针，现在在内存里为

jump_to_app

↓

？？？

对于 app_reset_handle，仅仅就是一个数 0x0800149。(pFunc)是将 app_reset_handle 强转成一个函数指针**变量** pFunc，此时这个 app_reset_handle 便不是数，而是一个指向一个函数的指针**类型的值**了（注意：他也只是值，并不是函数指针，所以也没法写成 app_reset_handle();）

此时 jump_to_app

↓

0x0800149

jump_to_app();就是执行 0x0800149 的函数。

5. 企业级注意事项

①不能接收到文件直接刷写，要有备份位置，以便后期回滚。

②可以自动检测是否需要升级，需要就自动跳转，不需要就直接进入系统

6. EEPROM 涉及的 I2C 知识

i2c 的读逻辑：（ACK 是主机发给从机）

启动信号->写从设备地址->ACK->寄存器地址->ACK->启动信号->读从设备地址

->ACK->读数据->回复 ACK->读数据->回复 ACK->读完回复 NO ACK->停止信号

（第一次启动是为了告诉系统应该去哪读）

第一次是主设备主要操作 SDA，第二次是从设备

i2c 的写逻辑：（ACK 是从机发给主机）

启动信号->写从设备地址->ACK->从设备寄存器地址->ACK->写数据->ACK->写数据->

ACK……写完回复 NO ACK->停止信号

对于写数据，只能一页一页写。EEPROM 的一页有（16byte 字节），假设从 0x00 开

始写可以写到 0x10，如果从 0x05 开始写，也只能写道 0x10.

7. 更新标识与数据安全性

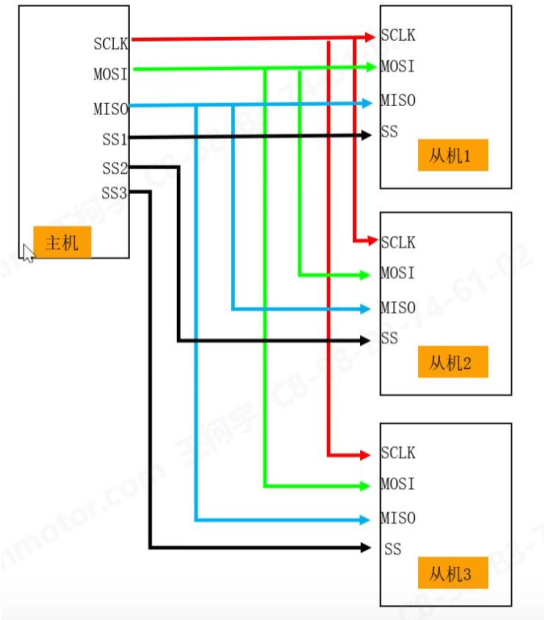
为了让系统自动判断是否需要进行更新，需要在 EEPROM 中的一个地址写上标志位，

来作为是否需要更新的标识符。另外由于安全性考量，需要加一个安全校验，防止误修

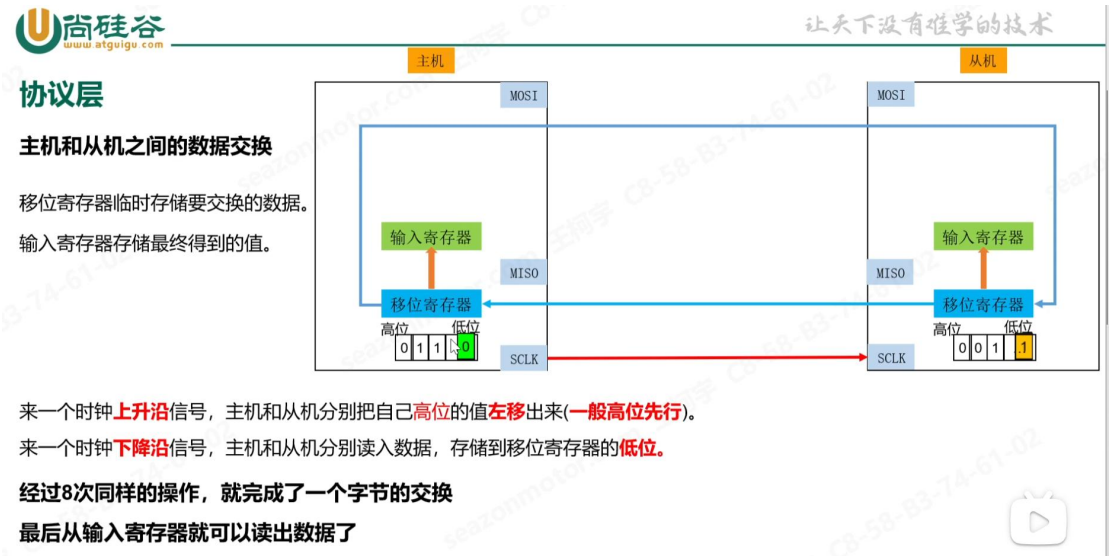
改标志位进而错误的进入更新模式

```h
#ifndef __APP_BOOTLOADER_H
#define __APP_BOOTLOADER_H

#include "Int_w24c02.h"

// 添加校验的秘钥
#define CHECK_KEY_ADDR 0X11
#define CHECK_KEY 0X5A6B
// 存储更新状态的位置
#define CHECK_UPDATE_ADDR 0x10
// 更新状态的值
#define BOOT_UPDATE 0X01
#define BOOT_NO_UPDATE 0X02

/**
 * @brief  判断当前是否需要进行更新
 *
 */
void App_bootloader_check_update(void);

#endif // !__APP_BOOTLOADER_H
```

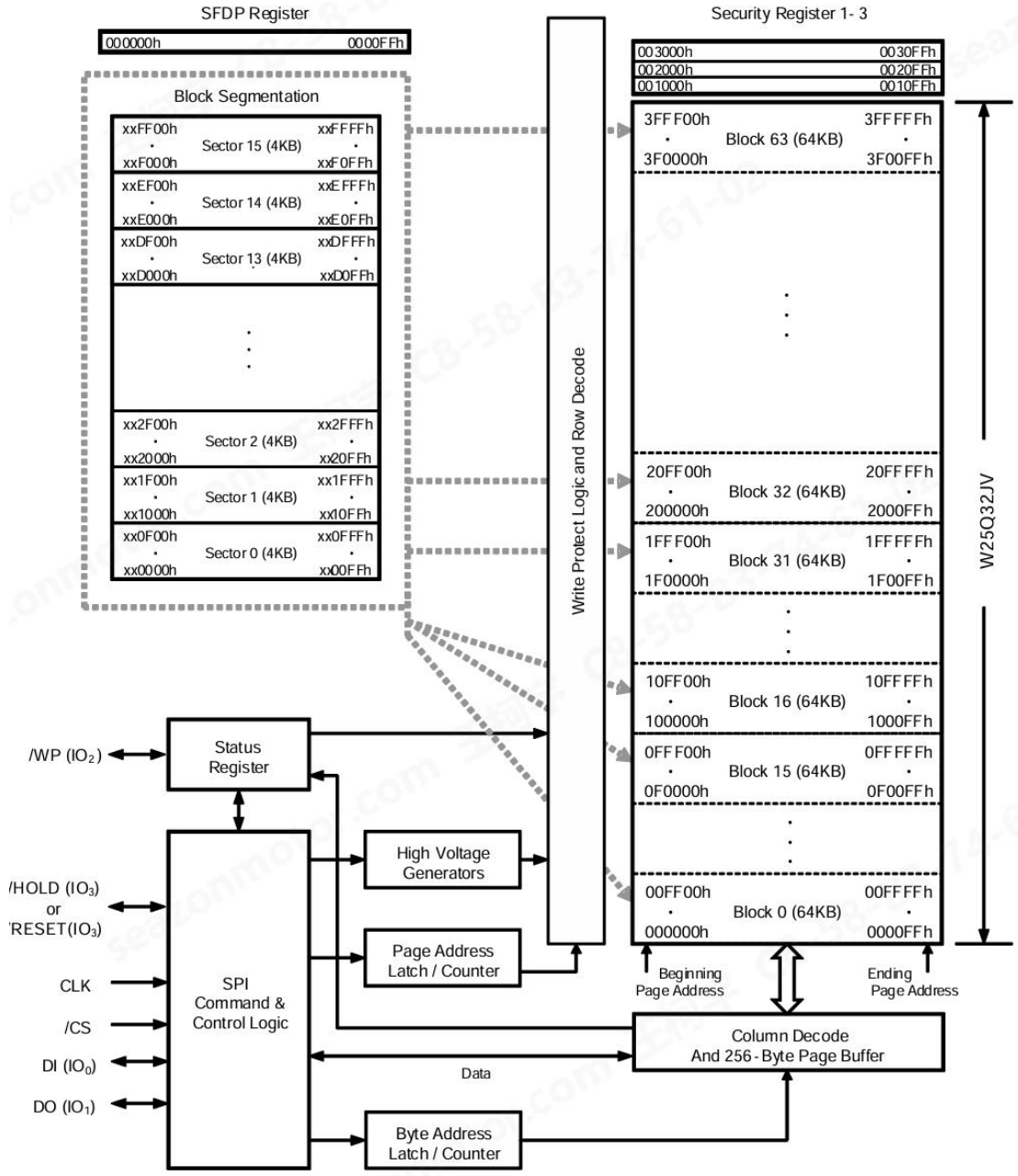
8. SPI







9. W25sq32 外置 flash



整体是 4m 容量->分为 64 个 block 块（64kb）->每个块又分为 16 个 sector 扇区（4kb）

->每个扇区又分为 256 个字节

如 0x023411　为第二个块第三个扇区第四页的第 17 字节

在使用时最小单位为页

Data Input Output	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Number of Clock(1-1-1)	8	8	8	8	8	8	8
Write Enable	06h						
Volatile SR Write Enable	50h						
Write Disable	04h						
Release Power-down / ID	ABh	Dummy	Dummy	Dummy	(ID7-ID0)(2)		
Manufacturer/Device ID	90h	Dummy	Dummy	00h	(MF7-MF0)	(ID7-ID0)	
JEDEC ID	9Fh	(MF7-MF0)	(ID15-ID8)	(ID7-ID0)			
Read Unique ID	4Bh	Dummy	Dummy	Dummy	Dummy	(UID63-0)	
Read Data	03h	A23-A16	A15-A8	A7-A0	(D7-D0)		
Fast Read	0Bh	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0)	
Page Program	02h	A23-A16	A15-A8	A7-A0	D7-D0	D7-D0(3)	
Sector Erase (4KB)	20h	A23-A16	A15-A8	A7-A0			
Block Erase (32KB)	52h	A23-A16	A15-A8	A7-A0			
Block Erase (64KB)	D8h	A23-A16	A15-A8	A7-A0			
Chip Erase	C7h/60h						
Read Status Register-1	05h	(S7-S0)(2)					

如何操作这个 flash：以读数据为理财， byte1 为命令位，通过输入这些来达到某种功

（1）写入操作前，必须先进行写使能。

（2）每个数据位只能由 1 改写为 0，不能由 0 改写为 1。

（3）写入数据前必须先擦除，擦除后，所有数据位变为 1。擦除必须按最小擦除单元进行。

（4）连续写入多字节时，最多写入一页的数据，超过页尾位置的数据，会回到页首覆盖写入。

能。byte234 位地址为地址，byte5 为返回数据

10.出现问题恢复出厂设置



11.flash 驱动

flash 是非易失性存储

擦除是 0-1　编程是 1-0

比如 1111 可以通过编程改为 1010 但是如果再想要改为 1110，则需要把整个 flash 擦除为 1111，然后再通过编程写入为 1110

flash 划分：P-flash：放程序

D-flash:主要放需要掉电保存的数据

flash 擦除是按扇区擦除的 这个 s32k144 是 4kb 擦除的，就是 4096bit 即 0x1000

所以最好要内存对齐，要不然很乱

flash 写入跟擦除也不一样，这个是 8byte 一写入。

```
status_t FLASH_DRV_Init(const flash_user_config_t * const pUserConf,
                        flash_ssd_config_t * const pSSDConfig)
{
    DEV_ASSERT(pUserConf != NULL);
    DEV_ASSERT(pSSDConfig != NULL);
    status_t ret = STATUS_SUCCESS;
#if FEATURE_FLS_HAS_FLEX_NVM
    uint8_t DEPartitionCode;    /* store D/E-Flash Partition Code */
#endif

    pSSDConfig->PFlashBase = pUserConf->PFlashBase;
    pSSDConfig->PFlashSize = pUserConf->PFlashSize;
    pSSDConfig->DFlashBase = pUserConf->DFlashBase;
    pSSDConfig->EERAMBase = pUserConf->EERAMBase;
    pSSDConfig->CallBack = pUserConf->CallBack;
```

常量指针与指针常量：

这个函数就能证明为啥前面的参数地址与数值都不能改

后面的地址不能改但是值个可以改

（DEV_ASSERT 是验证数值是否为空）

12.Bootloader 中的函数指针

定义函数指针：typedef void (*boot_app_entry_t)(void);

基本结构 void (*boot_app_entry_t)(void)

boot_app_entry_t 本身就代表了 void (*)(void)

作用就是创建一个指针，指向一个函数

后期我可以写 boot_app_entry_t app_entry；表示 app_entry 是

boot_app_entry_t 类型的函数指针

我如何跳入这个函数呢?

如果 address = 0x00011235

我需要 app_entry = （boot_app_entry_t）address，然后执行 app_entry ()；

其实写（（boot_app_entry_t）address）（）；也行

函数指针指向一个函数，为啥我不直接执行这个函数，还非得用地址来执行?

因为这个是分了 BOOT 与 APP，从 BOOT 中不知道 APP 中的函数名，肯定要通过指

针执行。

13.0x00010000 与 0x00010004

```
APP 起始地址 = 0x00010000
```

这里不是直接放 APP 的第一条普通业务代码，而是 APP 的**向量表**。

例如可能是：

```
Flash 地址          Flash 里面存的值

0x00010000          0x20007000
0x00010004          0x00011235
0x00010008          0x00011301
0x0001000C          0x00011345
...
```

PS：详细解释一下地址

0x01 与 0x00 相差 1 个 byte

0x1000 与 0x2000 相差 1000（但是是以 16 进制表达的）

所以为 16 进制的 1000 转化为 10 进制为 4096byte = 4kb

一个地址能存 1byte 也就是 8bit，如 0x01

比如要存 0x12345678,这是 8 个 16 进制，是 32bit，也就是 4 个 byte，需要 4 位地址

于是 0x01 存 12

0x02 存 34

0x03 存 56

0x04 存 78

所以上述地才 4 位 4 位增加，右面正好就是 32bit 内容，正好这芯片是 32 位的，正好

也可以把右侧当成地址来看。

补充一下，这个函数指针与代码入口以及回调函数之类的，他们传入的地址最低为要为1 是为了符合 Thumb 要求。

14.APP 跳转函数

读取 app 信息→关闭中断→关闭时钟→切换中断向量表→把 reset_handler 转换为指针→msp 切换到 app 栈顶→跳转到 reset_handler 地址→while（）保护

15.flash 与 ram

flash 不可以直接对某个数据进行更改，永久保存

ram 可以直接对某个数据进行更改，掉电丢失

所以比如一个 update 的信息表（crc 版本号 更新状态）内的内容需要频繁更改，所以在 flash 里直接操作不太好，而且 flash 还有使用寿命，可以在第一步就把相关结构体从 flash 放进 ram，然后随着程序运行进行修改，在完毕后写回 flash 中。

16.CRC

模二算法（不进位算法） 加减乘除同理，然后加减计算结果都相同

$$
\begin{array}{r}
1101 \\
111 \\
\oplus \quad 101 \\
\hline
1111
\end{array}
$$

这里重点关注模二除法，因为它与CRC算法密切相关，它有三个性质：

1. 当最后余数的位数小于除数位数时，除法停止。
2. 当被除数的位数小于除数位数时，则商数为0，被除数就是余数。
3. 只要被除数或部分余数的位数与除数一样多，且最高位为1，不管其他位是什么数，皆可商1。

$$
\begin{array}{rl}
 & 101\ (\text{商数}) \\
(\text{除数})\ 101\ \big) & 10011\ (\text{被除数}) \\
 & -101 \\
\hline
 & 111\ (\text{部分余数}) \\
 & -101 \\
\hline
 & 10\ (\text{最后余数})
\end{array}
$$

二进制系数多项式

对任意的二进制数都构造与其对应的一个二进制系数多项式。例如：10011B，其对应的二进制系数多项式为$P(x) = x^4 + x + 1$

CRC算法中，对于二进制数都是以二进制系数多项式去描述的。

CRC算法

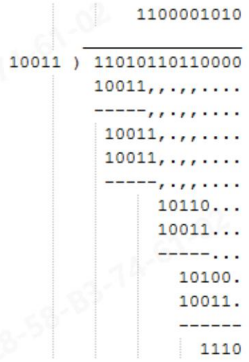
基于上述铺垫，我们现在可以对CRC算法进行具体说明：CRC 算法的基本思想是将传输的数据当做一个位数很长的数。将这个数模二除以另一个数。得到的余数作为校验数据附加到原数据后面。



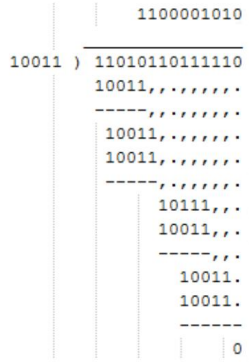
实际应用时，发送方和接收方按以下方式通信：

1. 发送方和接收方在通信前，约定好一个预设整数作为除数。
2. 发送方在发送前根据原始数据和约定好的除数进行模二除法运算生成余数（即CRC码），然后将其附加到原始数据后面一起发送给接收方。
3. 接收方收到后将其模二除以约定好的除数，当且仅当余数为0时接收方认为没有差错。

假设要传输的原始数据为1101011011B，发送方和接收方在通信前约定好的除数为10011B。由于除数10011B是五位数（5bit），那么假设余数（即CRC码）为四位数（4bit）。因为现在余数未知，所以在进行模二除法运算前先将余数设为0000B，即待发送的数据为11010110110000B。下面开始进行模二除法运算来确定余数（即CRC码）：



可见余数（即CRC码）为1110B，因此发送方实际发送的是11010110111110B。接收方在接收后需要将其模二除以10011B来进行CRC校验：



可见余数为0，因此本次通信没有差错。

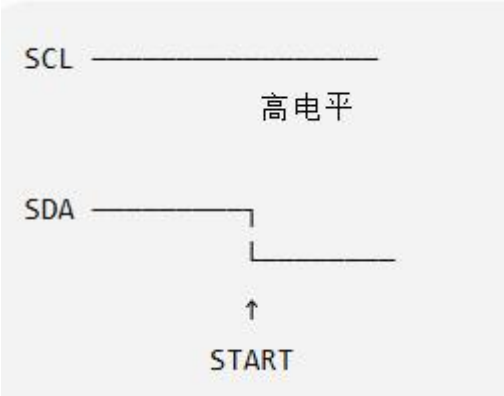
本程序里是 上位机通过 CRC32 验证一个值 传入 B 区 b 区算完与传入值核对，复制到 A 区后再进行核对一次

I2C：

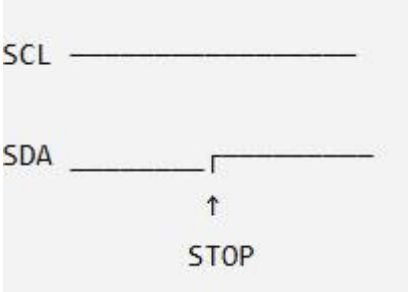
本身是开漏输出（默认高电平 1 拉低低电平 0，高电平由上拉电阻产生）

通信协议：

开始：SCL 高电平时拉低 SDA



停止：SCL 高电平时拉高 SDA



数据传输规则：

SCL 高电平时 SDA 需保持稳定　　所以数据变化要发生在 SCL 低电平期间

传输大小：

每次传输 8Bit + 1Bit ACK　且高位先发　　0x11110000　1->1->1->1->0->0->0->0

ACK:　0 是接收到了　1 是没接收到

I2C 地址的发送：

7bit 的从机地址（固定 7 位）+ 1 位 R/W　（0 写 1 读）

I2C 写数据:与读数据：上面有

谁控制 SDA:

写：主机控制 SDA,回复时 ACK 时从机控制 SDA

读：从机控制 SDA,回复时 ACK 时主机控制 SDA

只有一个能控制 SDA，为半双工

对于仲裁：

由于是开漏输出 0>1 所以设备 A 输出 1 设备 B 输出 0，结果就是 0

Master A：1 0 1 1 ...

Master B：1 0 1 0 ... B 胜出 A 释放 谁地址小谁优先级高

为什么不会短路或者数据冲突：

若用普通推挽，A 拉高 B 拉低 会直接导通，产生大电流

但是用开漏就 A 输出 1（开漏输出默认 1，此时视为断开）B 输出 0（拉低），结果为 0

补充：

- 函数指针、指针函数、数组指针、指针数组

1. 函数指针上文提了

2. 指针函数：

int * p(void) {return &A}

3. 数组指针：用 int* Fun = P( );

有即组数*据的 int/[&A] [8]

有组数据，有几行，每行8个int

最用来调用此函数来返回一些地址

int (*p) [8] = A

于是 P[8][7] = 1. 即把8行7列处改为1

P+1 代表跳8个 int

用来表示多维定长度的数据

4. 指针数组： ....]

int *p [10] = [&A,

用来管理多个地址（外设）

如 P(0) 即为 &A