网名“鱼树”的学员聂龙浩，

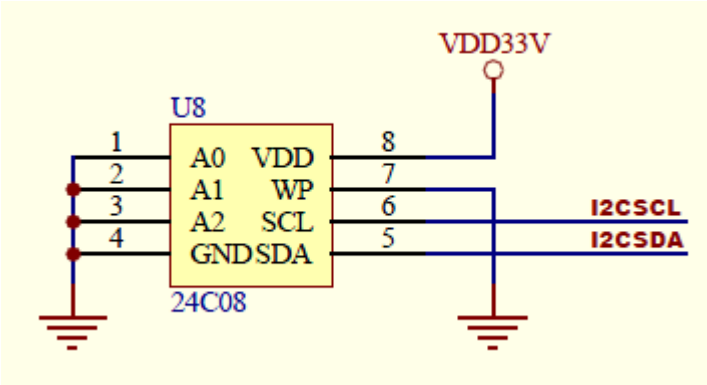
学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

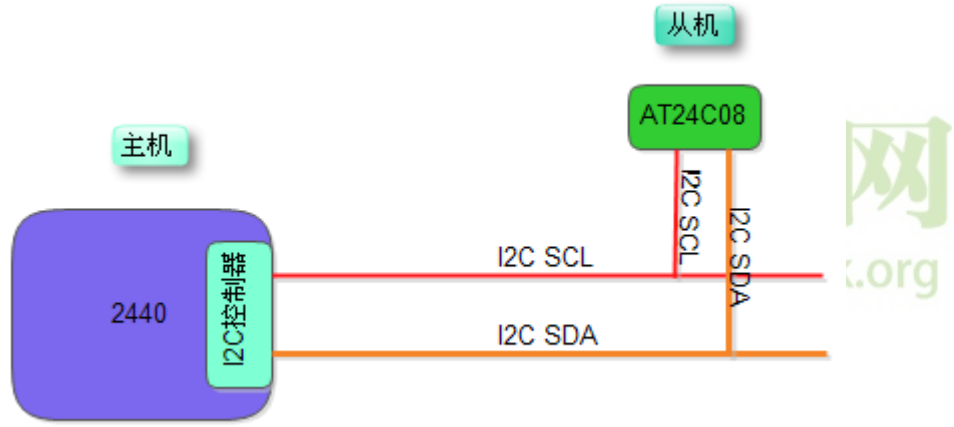
也许有错漏，请自行分辨。

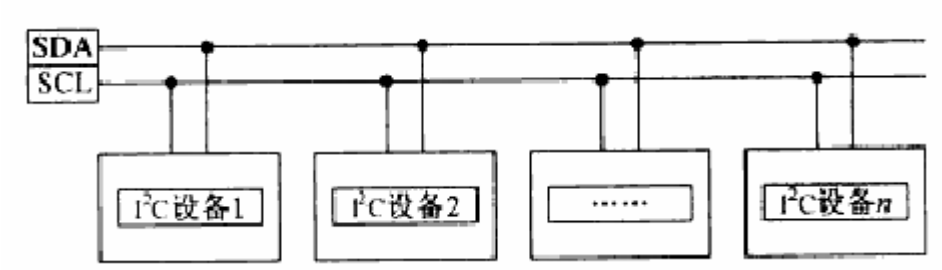
目录

I2C 总线介绍

电路图

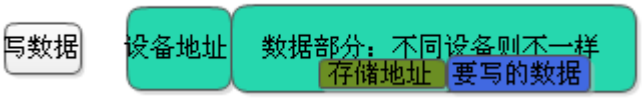






I2C 是个主从结构，即所有的传输都是从主机发起。从机不可能主动引起数据的传输。对于 I2C 协议来说，它只能规定到发出的第一个数据是“地址”，后面发出的内容是什么，每个 I2C 可能不同。

下面是一种情况：



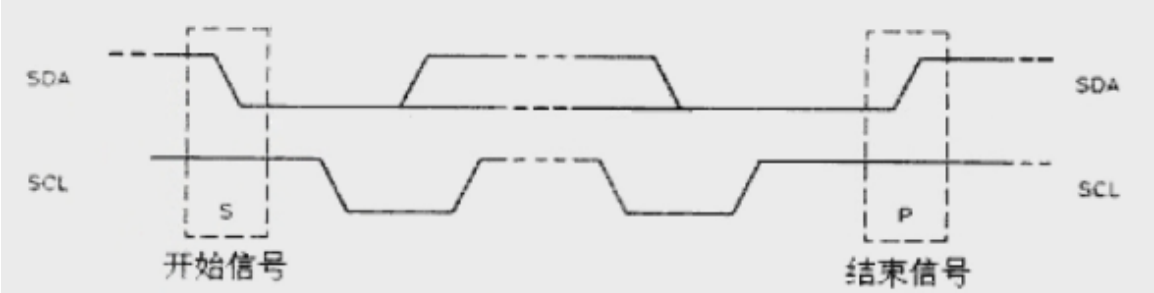
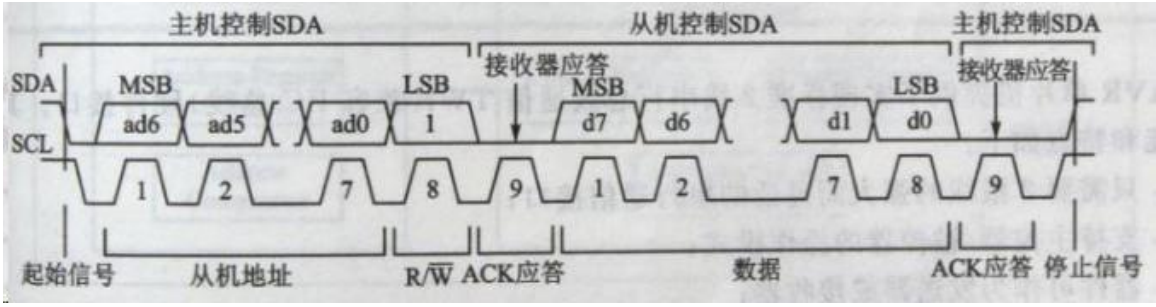


图 12.2　开始信号（S）和结束信号（P）



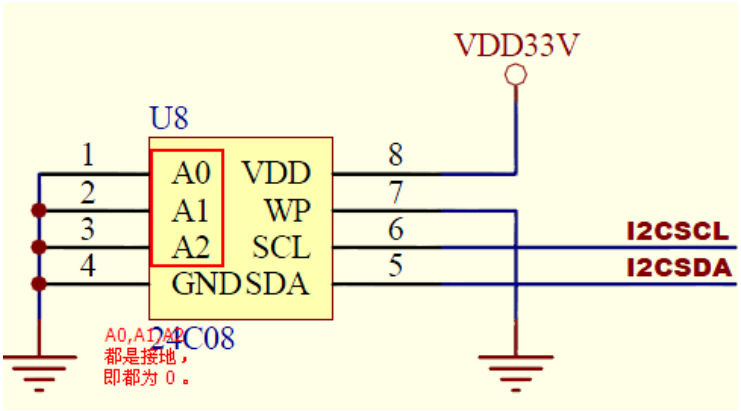
1，平时 SDA 和 SCL 都是高电平。
2，开始信号：当 SCL 为高电平的同时，SDA 有个脉冲。
3，数据在 SCL 低电平时变化 ，在 SCL 脉冲为高电平间稳定。
4，发出地址，每个 I2C 芯片里面都有一个地址，是固化在芯片里面。

一，设备地址：寻址过程 与 写数据

1，设备地址总共是 7bit，“1010”固定，后面“A2,A1,A0”可以从硬件上敲定。

看开发板实际的接线。

2K	1	0	1	0	A_2	A_1	A_0	R/W
	MSD							LSB
4K	1	0	1	0	A_2	A_1	P0	R/W
8K	1	0	1	0	A_2	P1	P0	R/W



开发板上，A0,A1,A2 都是接地，即此三位都是 0.

2，则此 AT24C08 的地址：

发出的前面 7 位就是设备地址。此时后面接的 I2C 设备，接收到一个 start 信号时，就知道下面紧接着的就是
7bit 的设备地址。若发现此地址与固化在设备内的地址相同时，就知道是访问自己。

3，访问时，是“读”还是“写”：由 7bit 后的第 8 位决定。

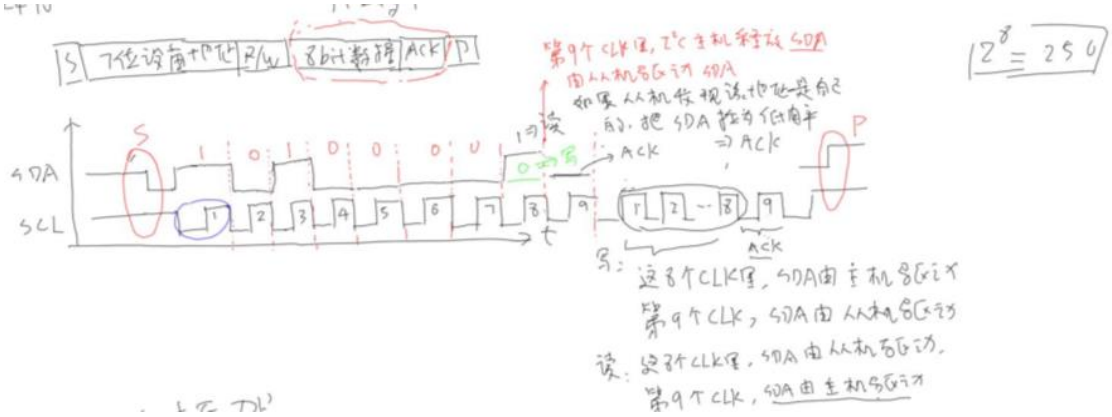
启动一个传输时，主机先发出 S 信号，然后发出 8 位 数据。这 8 位数据的前 7bit 为从机的地址，第 8bit 表示传输的方向（0 表示写操作，1 表示读操作）。

4，ACK 回应信号：

在第 9 个 CLK 里，I2C 主机释放 SDA，由从机驱动驱动 SDA.若从机发现该“设备地址”是自己的，是把 SDA 拉
为低电平。这时主机就能知道此“设备地址”的设备是存在的，后面就能再发数据了。

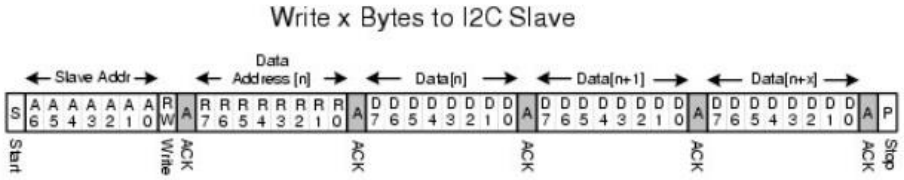
5.后面接着再 8 个 CLK 时钟，是具体的数据，是与设备有关的。

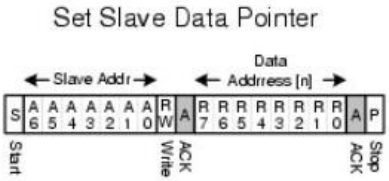
后面还有一个第 9bit，是从机把 SDA 拉低确认 ACK。

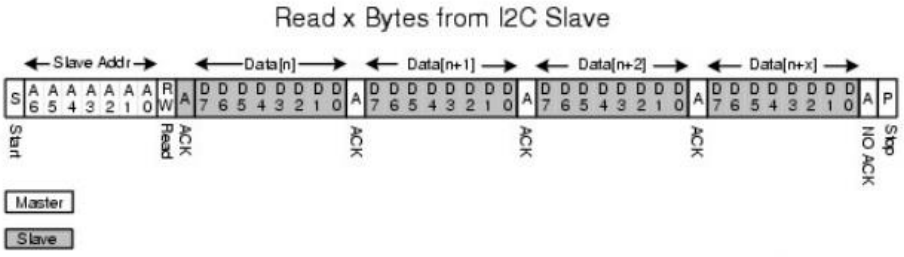


若为主机到从机的读：

寻址后的 8 个时钟是由 从机响应，从机会把设备驱动到 SDA 中，最后的第 9 个时钟也是 ACK，但是由主机响应。就是说主机已经接收到数据了，就到 ACK 拉为低电平。







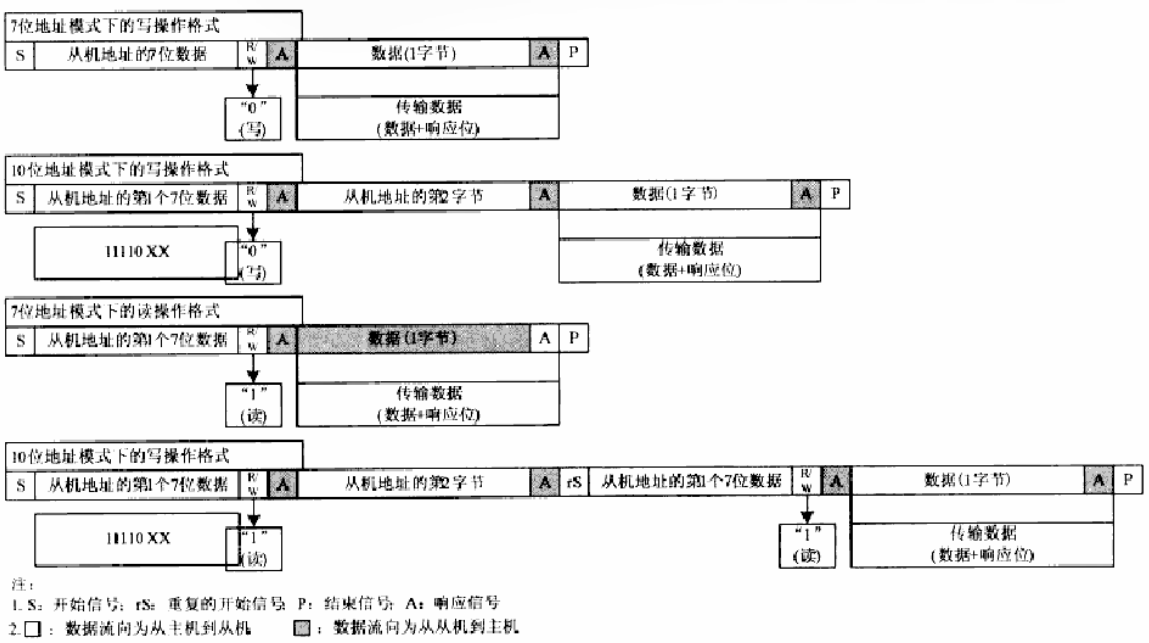


图 12.5 I^2C 总线上数据传输的格式

结束传输：

SCL 时钟信号在高电平期间，SDA 由低电平变成高电平时结束传输。

I2C 裸机程序

看一个裸板程序从“入口函数”开始看：head.S

一，开发板上电初始化工作：

假设 NAND 启动，一上电，程序的代码前 4K 就会复制到 2440 片内内存。

1，第一步：跳转到 reset 处：

```
_start:
@*************************************************************************
@ 中断向量，本程序中，除Reset和HandleIRQ外，其它异常都没有使用
@*************************************************************************
    b   Reset                                   @跳转到reset处。
```


2，reset 处代码工作：

①，设置栈、关看门狗、设置时钟、设置内存控制器、初始化 NANDFLASH。

```
Reset:
    ldr sp, =4096               @ 设置栈指针，以下都是C函数，调用前需要设好栈
    bl  disable_watch_dog       @ 关闭WATCHDOG，否则CPU会不断重启
    bl  clock_init              @ 设置MPLL，改变FCLK、HCLK、PCLK
    bl  memsetup                @ 设置存储控制器以使用SDRAM
    bl  nand_init               @ 初始化NAND Flash
```

这些函数现在是在 0 地址（片内内存）处运行。

②，把代码复制到 0x30000000 地址处。

```
                            @ 复制代码到SDRAM中
ldr r0, =0x30000000         @ 1. 目标地址 = 0x30000000，这是SDRAM的起始地址
mov r1, #0                  @ 2. 源地址   = 0
ldr r2, =__bss_start
sub r2, r2, r0              @ 3. 复制长度
bl  CopyCode2SDRAM          @ 调用C函数CopyCode2SDRAM
```

复制到 0x30000000 处，是因为“链接地址”规定。

```
SECTIONS {
    . = 0x00000000;
    .init : AT(0){ head.o init.o nand.o}
    . = 0x30000000;
    .text : AT(4096) { *(.text) }
    .rodata ALIGN(4) : AT((LOADADDR(.text)+SIZEOF(.text)+3)&~(0x03)) {*(.rodata*)}
    .data ALIGN(4)   : AT((LOADADDR(.rodata)+SIZEOF(.rodata)+3)&~(0x03)) { *(.data) }
    __bss_start = .;
    .bss ALIGN(4)  : { *(.bss)  *(COMMON) }
    __bss_end = .;
}
```

之前是把程序分成了两段，实际上可以分成一段。修改如下：

a,链接地址：0x3000000

```
. = 0x30000000; 链接地址都是从这里开始
```

b,代码段：

```
.text : { head.o(.text)
          init.o(.text)
          nand.o(.text)
          *(.text)
        }
```

```
Head.o 的代码段          : head.o(.text)
Init.o 的代码段          : init.o(.text)
Nand.o 的代码段          : nand.o(.text)
*      其他文件的代码段 :  *(.text)
```

c，只读数据段：所有文件的只读数据段。

```
.rodata ALIGN(4) : {*(.rodata*)}
```

d，所有文件的数据段：

```
.data   ALIGN(4) : { *(.data) }
```

e，bss 段：

```
__bss_start = .;
.bss ALIGN(4)  : { *(.bss)  *(COMMON) }
__bss_end = .;
```

所有文件的的 bss 和 COMMON 段。

3，链接地址：

```
ldr sp, =4096
bl    disable_watch_dog
bl    clock_init
```

```
bl  memsetup
bl  nand_init
```

这些函数现在是在 0 地址（片内内存）处运行。它们的链接地址是“0x30000000”。链接地址并不等于它们当前所在的位置。所以它们就需要以“位置无关码”来写。就是说写这些函数时，要用“位置无关码”来写，不能用全局变量等。

①，关看门狗：

```
/*
 * 关闭WATCHDOG，否则CPU会不断重启
 */
void disable_watch_dog(void)
{
    WTCON = 0;  // 关闭WATCHDOG很简单，往这个寄存器写0即可
}
```

②，初始化时钟：

```
#define FCLK        200000000
#define HCLK        100000000
#define PCLK        50000000
#define S3C2410_MPLL_200MHZ     ((0x5c<<12)|(0x04<<4)|(0x00))
#define S3C2440_MPLL_200MHZ     ((0x5c<<12)|(0x01<<4)|(0x02))
/*
 * 对于MPLLCON寄存器，[19:12]为MDIV，[9:4]为PDIV，[1:0]为SDIV
 * 有如下计算公式：
 *  S3C2410：MPLL(FCLK) = (m * Fin)/(p * 2^s)
 *  S3C2410：MPLL(FCLK) = (2 * m * Fin)/(p * 2^s)
 *  其中：m = MDIV + 8, p = PDIV + 2, s = SDIV
 * 对于本开发板，Fin = 12MHz
 * 设置CLKDIVN，令分频比为：FCLK:HCLK:PCLK=1:2:4，
 * FCLK=200MHz,HCLK=100MHz,PCLK=50MHz
 */
void clock_init(void)
{
    // LOCKTIME = 0x00ffffff;   // 使用默认值即可
    CLKDIVN  = 0x03;            // FCLK:HCLK:PCLK=1:2:4, HDIVN=1,PDIVN=1

    /* 如果HDIVN非0，CPU的总线模式应该从“fast bus mode”变为“asynchronous bus mode” */
__asm__(
    "mrc    p15, 0, r1, c1, c0, 0\n"        /* 读出控制寄存器 */
    "orr    r1, r1, #0xc0000000\n"          /* 设置为“asynchronous bus mode” */
    "mcr    p15, 0, r1, c1, c0, 0\n"        /* 写入控制寄存器 */
    );

    /* 判断是S3C2410还是S3C2440 */
    if ((GSTATUS1 == 0x32410000) || (GSTATUS1 == 0x32410002))
    {
        MPLLCON = S3C2410_MPLL_200MHZ;  /* 现在，FCLK=200MHz,HCLK=100MHz,PCLK=50MHz */
    }
    else
    {
        MPLLCON = S3C2440_MPLL_200MHZ;  /* 现在，FCLK=200MHz,HCLK=100MHz,PCLK=50MHz */
    }
}
```

③，内存设置：

```
/*
 * 设置存储控制器以使用SDRAM
 */
void memsetup(void)
{
    volatile unsigned long *p = (volatile unsigned long *)MEM_CTL_BASE;

    /* 这个函数之所以这样赋值，而不是像前面的实验(比如mmu实验)那样将配置值
     * 写在数组中，是因为要生成"位置无关的代码"，使得这个函数可以在被复制到
     * SDRAM之前就可以在steppingstone中运行
     */
    /* 存储控制器13个寄存器的值 */
    p[0] = 0x22011110;     //BWSCON
    p[1] = 0x00000700;     //BANKCON0
    p[2] = 0x00000700;     //BANKCON1
    p[3] = 0x00000700;     //BANKCON2
    p[4] = 0x00000700;     //BANKCON3
    p[5] = 0x00000700;     //BANKCON4
    p[6] = 0x00000700;     //BANKCON5
    p[7] = 0x00018005;     //BANKCON6
    p[8] = 0x00018005;     //BANKCON7

                                    /* REFRESH,
                                     * HCLK=12MHz:  0x008C07A3,
                                     * HCLK=100MHz: 0x008C04F4
                                     */
    p[9]  = 0x008C04F4;
    p[10] = 0x000000B1;     //BANKSIZE
    p[11] = 0x00000030;     //MRSRB6
    p[12] = 0x00000030;     //MRSRB7
}
```

④，初始化 NAND：

```
/* 初始化NAND Flash */
void nand_init(void)
{
    S3C2410_NAND * s3c2410nand = (S3C2410_NAND *)0x4e000000;
    S3C2440_NAND * s3c2440nand = (S3C2440_NAND *)0x4e000000;

#define TACLS   0
#define TWRPH0  3
#define TWRPH1  0

    /* 判断是S3C2410还是S3C2440 */
    if ((GSTATUS1 == 0x32410000) || (GSTATUS1 == 0x32410002))
    {
        /* 使能NAND Flash控制器，初始化ECC，禁止片选，设置时序 */
        s3c2410nand->NFCONF = (1<<15)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0);

        /* 复位NAND Flash */
        s3c2410_nand_reset();
    }
    else
    {
        /* 设置时序 */
        s3c2440nand->NFCONF = (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);
        /* 使能NAND Flash控制器，初始化ECC，禁止片选 */
        s3c2440nand->NFCONT = (1<<4)|(1<<1)|(1<<0);

        /* 复位NAND Flash */
        s3c2440_nand_reset();
    }
}
```

位置无关码不能用全局变量，这里的 "S3C2410_NAND”是局部变量。所以这个“nand.c”中的代码都要修改。看修改过的“nand.c”把全局变量全换成局部变量，即把全局变量放到局部中（函数里）。

4，head.S 重定位：

把 NANDFLASH 0 地址从 0 地址拷贝。拷贝的长度是"__bss_start = ."减去“.=0x30000000”.这样便有足够长了。

```
    . = 0x30000000; 链接地址都是从这里开始
    .text : { head.o(.text)
              init.o(.text)
              nand.o(.text)
              *(.text)
            }
    .rodata ALIGN(4) : {*(.rodata*)}
    .data   ALIGN(4) : { *(.data) }
      bss_start = .;
```

以前为了简单代码，直接把长度写了 16K：

```
mov r2, #16*1024          @ 3. 复制长度 = 16K，对于本实验，这是足够了
```

修改为如下：就是不包括 bss 段的长度。

```
ldr r2, =__bss_start
sub r2, r2, r0          @ 3. 复制长度
```

bss 段是放那些初始值为 0 的全局变量的。假设有 1W 个 0 时，没有必要把所有的 0 全放在最后的二进制文件里面，要是带上这 1W 个 0 就会很浪费空间。那么二进制文件里面就不去含有 bss 段。bss 段部分需要我们自己来清零。

4，head.S 接着调用：CopyCode2SDRAM 函数，将代码拷贝到内存中去。

```
bl  CopyCode2SDRAM      @ 调用C函数CopyCode2SDRAM
```

```
int CopyCode2SDRAM(unsigned char *buf, unsigned long start_addr, int size)
{
    extern void nand_read(unsigned char *buf, unsigned long start_addr, int size);
    nand_read(buf, start_addr, size);
    return 0;
}
```

用"nand_read"函数来拷贝。

5，head.S 中将代码拷贝到内存中后，接着清 bss 段：

就是将“ __bss_start”到“ __bss_end;”间全部清为 0.

```
bl  clean_bss           @ 清除bss段，未初始化或初值为0的全局/静态变量保存在bss段
```

```
void clean_bss(void)
{
    extern int __bss_start, __bss_end;
    int *p = &__bss_start;

    for (; p < &__bss_end; p++)
        *p = 0;
}
```



```
    __bss_start = .;
    .bss ALIGN(4)   : { *(.bss)  *(COMMON) }
    __bss_end = .;
```

这里在链接时会分配链接地址。就把这段清为 0.那么以前访问这些全局变量时就都是 0.

6，head.S 中其他就是中断相关的设置，最后是 main 函数。

二，主函数：main.c

1,初始化串口和 I2C 设置：

a,初始化串口：

```
uart0_init();   // 波特率115200，8N1(8个数据位，无校验位，1个停止位)
```

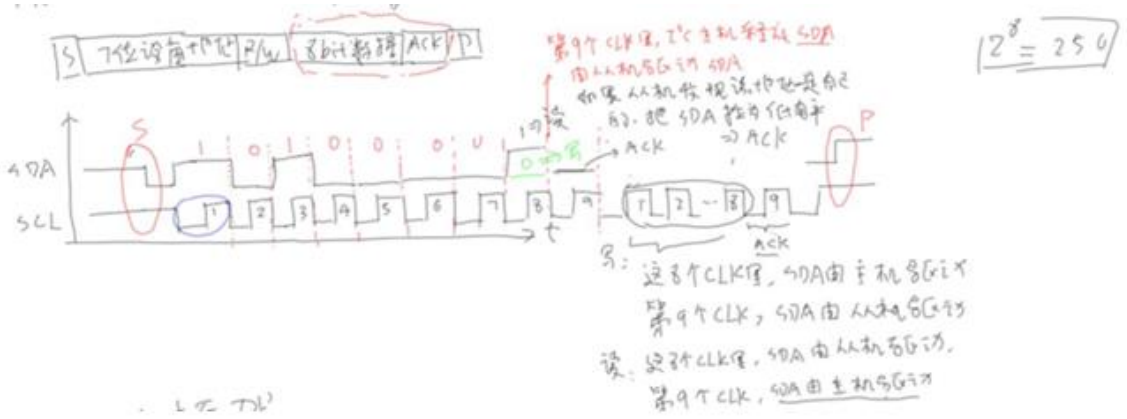
```c
/*
 * 初始化UART0
 * 115200,8N1,无流控
 */
void uart0_init(void)
{
    GPHCON  |= 0xa0;    // GPH2,GPH3用作TXD0,RXD0
    GPHUP   = 0x0c;     // GPH2,GPH3内部上拉

    ULCON0  = 0x03;     // 8N1(8个数据位，无较验，1个停止位)
    UCON0   = 0x05;     // 查询方式，UART时钟源为PCLK
    UFCON0  = 0x00;     // 不使用FIFO
    UMCON0  = 0x00;     // 不使用流控
    UBRDIV0 = UART_BRD; // 波特率为115200
}
```

b,初始化 I2C：

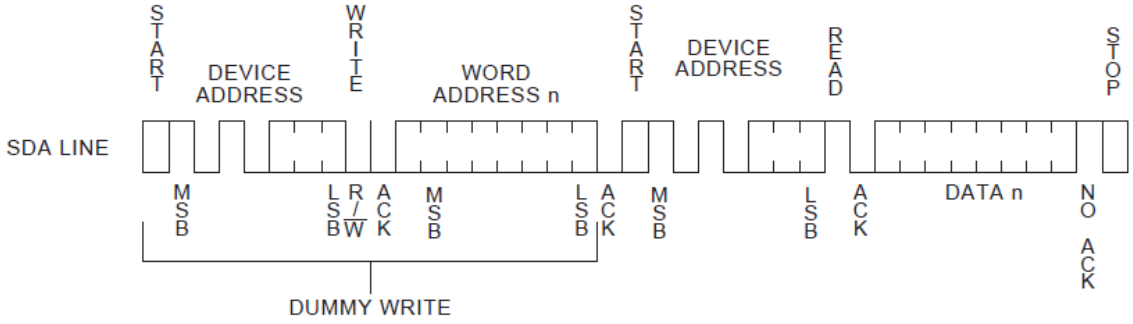
```c
i2c_init();
```

```c
/*
 * I2C初始化
 */
void i2c_init(void)
{
    GPEUP  |= 0xc000;       // 禁止内部上拉
    GPECON |= 0xa0000000;   // 选择引脚功能：GPE15:IICSDA, GPE14:IICSCL

    INTMSK &= ~(BIT_IIC);

    /* bit[7] = 1, 使能ACK
     * bit[6] = 0, IICCLK = PCLK/16
     * bit[5] = 1, 使能中断
     * bit[3:0] = 0xf, Tx clock = IICCLK/16
     * PCLK = 50MHz, IICCLK = 3.125MHz, Tx Clock = 0.195MHz
     */
    IICCON = (1<<7) | (0<<6) | (1<<5) | (0xf);  // 0xaf

    IICADD  = 0x10;     // S3C24xx slave address = [7:1]
    IICSTAT = 0x10;     // I2C串行输出使能(Rx/Tx)
}
```

3，两个读写函数：



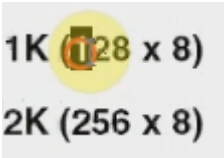
Random Read **随机读的时序：**



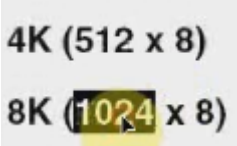
前 7bit 发出设备地址找到从机，第 8bit 从机回应。再接着发出存储地址写给从机。接着就开始一个“START”信号，再发出设备地址，这时候表示读。最后读到此数据（WORD ADDRESS n 处）。

AT24108,是 8K（1024*8）即要 10 位地址才能表示 1024。“WORD ADDRESSn”是 8 位根本无法寻址 1K 的空间。2 的 8 次方是最多访问到 256 个地址。但这里 E2PROM 的容量有 1K、2K 时，显然 8 位的地址是不够的。

Device Addressing **设备寻址：**



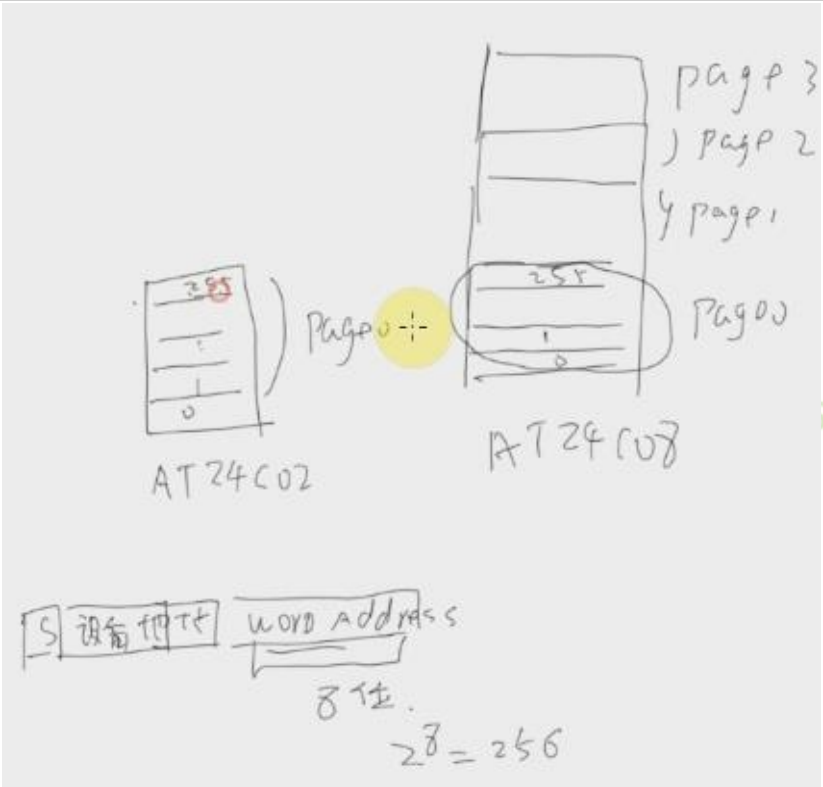
里面的“128”、“256”可以用 8 位来寻址到。



而 512，1024 则有 2 页或 4 页数据。

2K	1	0	1	0	A_2	A_1	A_0	R/W
	MSD							LSB
4K	1	0	1	0	A_2	A_1	P0	R/W
8K	1	0	1	0	A_2	P1	P0	R/W

2K是256B字节时，它所有的存储地址就可以用 8位来寻址。
4K即512字节时，P0等于0时表示访问第一页。P1等于1时表示访问第1页。
8K即1024字节时，P1,P0总共就可以表示 4 页。
16K即2048字节时，P2,P1,P0共表示8页。



对于小容量的 AT24C02 为 256 字节，就可以用“WORD ADDRESS”8 位地址来寻址。但对于上面 AT24C08 这个 1024 字节的大容量 E2PROM 设备，则"WORD ADDRESS"只能寻址它的“PAGE0”空间。对于其他页的空间，以前面发出“START”信号后“设备地址”：

8K	1	0	1	0	A_2	P1	P0	R/W

A2 是硬件引脚，P1,P0 可以变。只要发出“1010+A2”就能访问到设备（A2 硬件上接低电平时，则 A2 为 0；硬件上接的是高电平，则 A2 为 1），P1,P0 就是用来表示访问哪一页。AT20C08 有 4 页（每页 256 字节）。

最后编译测试：

```
book@book-desktop:/work/drivers_and_test/18th_i2c/at24cxx$ ls -l i2c.bin
-rwxr-xr-x 1 book book 10856 2011-12-26 22:01 i2c.bin
book@book-desktop:/work/drivers_and_test/18th_i2c/at24cxx$
```

得到了 bin 有 10K。

```
SECTIONS {
    . = 0x30000000;
    .text : { head.o(.text)
              init.o(.text)
              nand.o(.text)
              *(.text)
            }
    .rodata ALIGN(4) : {*(.rodata*)}
    .data   ALIGN(4) : { *(.data) }
    __bss_start = .;
    .bss ALIGN(4)   : { *(.bss)  *(COMMON) }
    __bss_end = .;
}
```

链接脚本要保证“head.o”"init.o"“nand.o”三个文件位于前面 4K，因为这个三个文件在重定位之前，都得位于片内内存里面。可以看反汇编文件：4K=4096=0x1000,即是 0x30001000 处：

```
30000ff8:       e5d0c000        ldrb    ip, [r0]
30000ffc:       e1a03004        mov     r3, r4
30001000:       e15c0003        cmp     ip, r3
```

链接脚本中最后一个 nand.o 的最后一个函数是：

```
/* 读函数 */
void nand_read(unsigned char *buf, unsigned long start_addr, int size)
```

在反汇编文件中搜索这个“nand_read”的链接地址：

```
300003e4 <nand_read>:
300003e4:       e1a03a81        mov     r3, r1, lsl #21
300003e8:       e1a03aa3        mov     r3, r3, lsr #21
```

链接地址是“3e4”没有超过“4K”。