第1章　I2C—读写 EEPROM

本章参考资料：《STM32 中文参考手册》I2C 章节及《I2C 总线协议》。

若对 I2C 通讯协议不了解，可先阅读《I2C 总线协议》文档的内容学习。

关于 EEPROM 存储器，请参考“常用存储器介绍”章节，实验中的 EEPROM，请参考其规格书《AT24C02》来了解。

1.1 I2C 协议简介

I2C 通讯协议(Inter－Integrated Circuit)是由 Phiilps 公司开发的，由于它引脚少，硬件实现简单，可扩展性强，不需要 USART、CAN 等通讯协议的外部收发设备，现在被广泛地使用在系统内多个集成电路(IC)间的通讯。

在计算机科学里，大部分复杂的问题都可以通过分层来简化。如芯片被分为内核层和片上外设；STM32 标准库则是在寄存器与用户代码之间的软件层。对于通讯协议，我们也以分层的方式来理解，最基本的是把它分为物理层和协议层。物理层规定通讯系统中具有机械、电子功能部分的特性，确保原始数据在物理媒体的传输。协议层主要规定通讯逻辑，统一收发双方的数据打包、解包标准。简单来说物理层规定我们用嘴巴还是用肢体来交流，协议层则规定我们用中文还是英文来交流。

下面我们分别对 I2C 协议的物理层及协议层进行讲解。

1.1.1 I2C 物理层

I2C 通讯设备之间的常用连接方式见图 1-1。

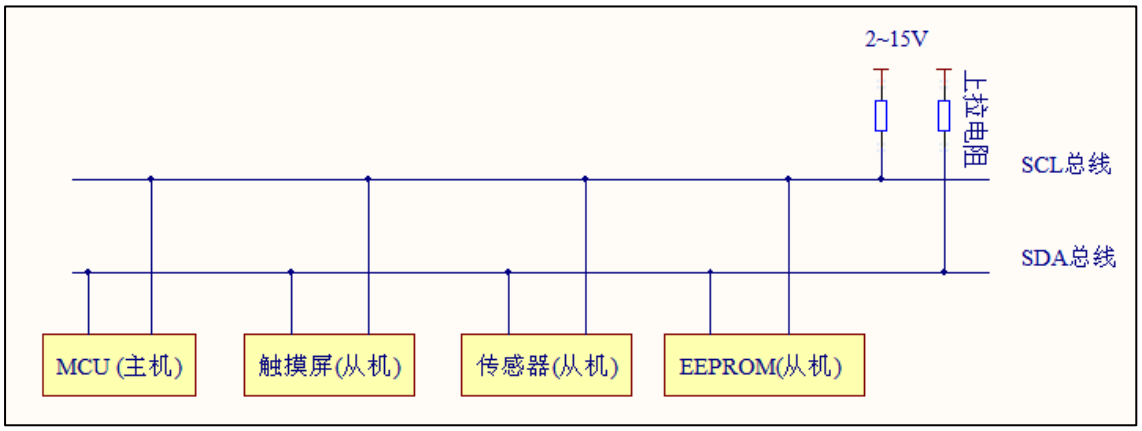


图 1-1 常见的 I2C 通讯系统

它的物理层有如下特点：

(1) 它是一个支持设备的总线。“总线”指多个设备共用的信号线。在一个 I2C 通讯总线中，可连接多个 I2C 通讯设备，支持多个通讯主机及多个通讯从机。

(2) 一个 I2C 总线只使用两条总线线路，一条双向串行数据线(SDA)，一条串行时钟线(SCL)。数据线即用来表示数据，时钟线用于数据收发同步。

(3) 每个连接到总线的设备都有一个独立的地址，主机可以利用这个地址进行不同设备之间的访问。

(4) 总线通过上拉电阻接到电源。当 I2C 设备空闲时，会输出高阻态，而当所有设备都空闲，都输出高阻态时，由上拉电阻把总线拉成高电平。

(5) 多个主机同时使用总线时，为了防止数据冲突，会利用仲裁方式决定由哪个设备占用总线。

(6) 具有三种传输模式：标准模式传输速率为 100kbit/s ，快速模式为 400kbit/s ，高速模式下可达 3.4Mbit/s，但目前大多 I^2C 设备尚不支持高速模式。

(7) 连接到相同总线的 IC 数量受到总线的最大电容 400pF 限制 。

1.1.2 协议层

I2C 的协议定义了通讯的起始和停止信号、数据有效性、响应、仲裁、时钟同步和地址广播等环节。

1. I2C 基本读写过程

先看看 I2C 通讯过程的基本结构，它的通讯过程见图 1-2、图 1-3 及图 1-4。

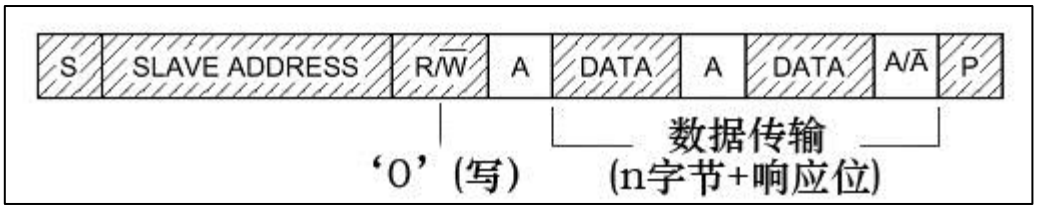


图 1-2 主机写数据到从机

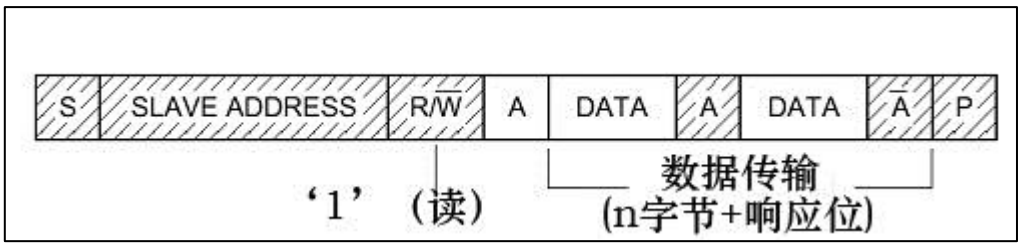


图 1-3 主机由从机中读数据

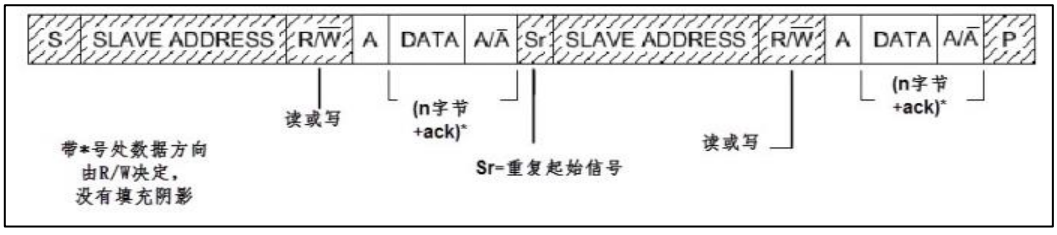


图 1-4 I2C 通讯复合格式

图例： 数据由主机传输至从机　S： 传输开始信号

SLAVE_ADDRESS: 从机地址

□ 数据由从机传输至主机　R/$\overline{W}$： 传输方向选择位，1 为读，0 为写

A/$\overline{A}$ ： 应答(ACK)或非应答(NACK)信号

P ： 停止传输信号

这些图表示的是主机和从机通讯时，SDA 线的数据包序列。

其中 S 表示由主机的 I2C 接口产生的传输起始信号(S)，这时连接到 I2C 总线上的所有从机都会接收到这个信号。

起始信号产生后，所有从机就开始等待主机紧接下来广播 的从机地址信号(SLAVE_ADDRESS)。在 I2C 总线上，每个设备的地址都是唯一的，当主机广播的地址与某个设备地址相同时，这个设备就被选中了，没被选中的设备将会忽略之后的数据信号。根据 I2C 协议，这个从机地址可以是 7 位或 10 位。

在地址位之后，是传输方向的选择位，该位为 0 时，表示后面的数据传输方向是由主机传输至从机，即主机向从机写数据。该位为 1 时，则相反，即主机由从机读数据。

从机接收到匹配的地址后，主机或从机会返回一个应答(ACK)或非应答(NACK)信号，只有接收到应答信号后，主机才能继续发送或接收数据。

写数据

若配置的方向传输位为“写数据”方向，即第一幅图的情况，广播完地址，接收到应答信号后，主机开始正式向从机传输数据(DATA)，数据包的大小为 8 位，主机每发送完一个字节数据，都要等待从机的应答信号(ACK)，重复这个过程，可以向从机传输 N 个数据，这个 N 没有大小限制。当数据传输结束时，主机向从机发送一个停止传输信号(P)，表示不再传输数据。

读数据

若配置的方向传输位为“读数据”方向，即第二幅图的情况，广播完地址，接收到应答信号后，从机开始向主机返回数据(DATA)，数据包大小也为 8 位，从机每发送完一个数据，都会等待主机的应答信号(ACK)，重复这个过程，可以返回 N 个数据，这个 N 也没有大小限制。当主机希望停止接收数据时，就向从机返回一个非应答信号(NACK)，则从机自动停止数据传输。

读和写数据

除了基本的读写，I2C 通讯更常用的是复合格式，即第三幅图的情况，该传输过程有两次起始信号(S)。一般在第一次传输中，主机通过 SLAVE_ADDRESS 寻找到从设备后，发送一段“数据”，这段数据通常用于表示从设备内部的寄存器或存储器地址(注意区分它与 SLAVE_ADDRESS 的区别)；在第二次的传输中，对该地址的内容进行读或写。也就是说，第一次通讯是告诉从机读写地址，第二次则是读写的实际内容。

以上通讯流程中包含的各个信号分解如下：

1. 通讯的起始和停止信号

前文中提到的起始(S)和停止(P)信号是两种特殊的状态，见图 1-5。当 SCL 线是高电平时 SDA 线从高电平向低电平切换，这个情况表示通讯的起始。当 SCL 是高电平时 SDA 线由低电平向高电平切换，表示通讯的停止。起始和停止信号一般由主机产生。

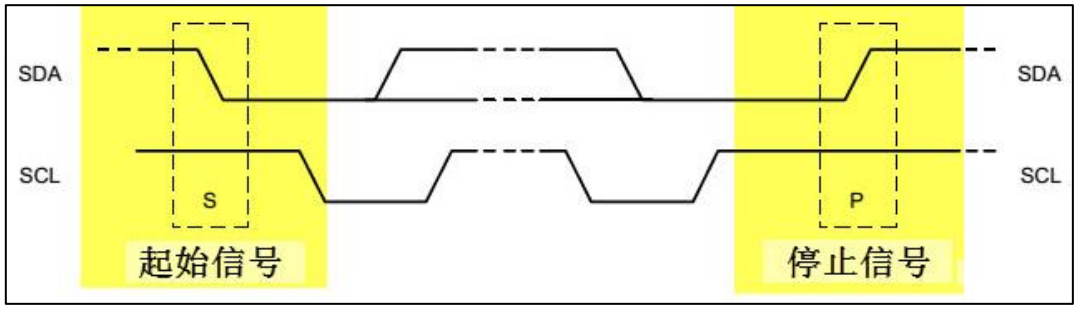


图 1-5 起始和停止信号

2. 数据有效性

I2C 使用 SDA 信号线来传输数据，使用 SCL 信号线进行数据同步。见图 1-6。SDA 数据线在 SCL 的每个时钟周期传输一位数据。传输时，SCL 为高电平的时候 SDA 表示的数据有效，即此时的 SDA 为高电平时表示数据“1”，为低电平时表示数据“0”。当 SCL 为低电平时，SDA 的数据无效，一般在这个时候 SDA 进行电平切换，为下一次表示数据做好准备。

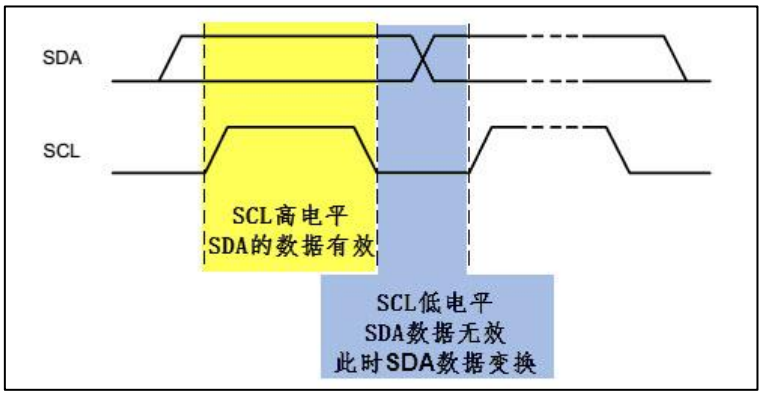


图 1-6 数据有效性

每次数据传输都以字节为单位，每次传输的字节数不受限制。

3. 地址及数据方向

I2C 总线上的每个设备都有自己的独立地址，主机发起通讯时，通过 SDA 信号线发送设备地址(SLAVE_ADDRESS)来查找从机。I2C 协议规定设备地址可以是 7 位或 10 位，实际中 7 位的地址应用比较广泛。紧跟设备地址的一个数据位用来表示数据传输方向，它是

数据方向位(R/$\overline{W}$)，第 8 位或第 11 位。数据方向位为“1”时表示主机由从机读数据，该位为“0”时表示主机向从机写数据。见图 1-7。

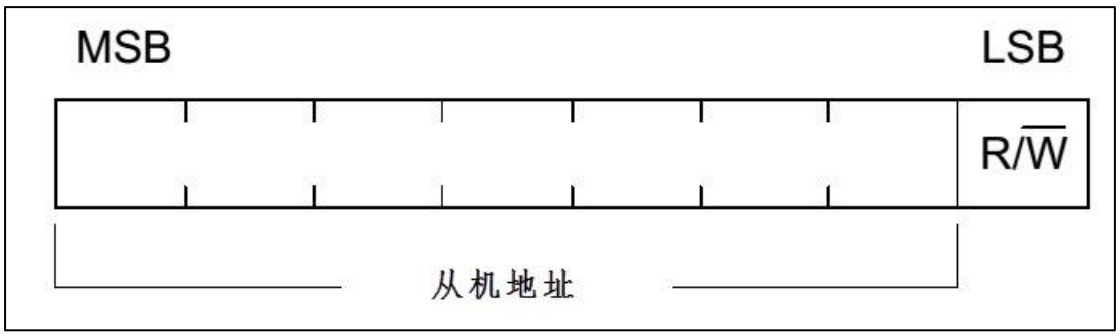


图 1-7 设备地址(7 位)及数据传输方向

读数据方向时，主机会释放对 SDA 信号线的控制，由从机控制 SDA 信号线，主机接收信号，写数据方向时，SDA 由主机控制，从机接收信号。

4. 响应

I2C 的数据和地址传输都带响应。响应包括“应答(ACK)”和“非应答(NACK)”两种信号。作为数据接收端时，当设备(无论主从机)接收到 I2C 传输的一个字节数据或地址后，若希望对方继续发送数据，则需要向对方发送“应答(ACK)”信号，发送方会继续发送下一个数据；若接收端希望结束数据传输，则向对方发送“非应答(NACK)”信号，发送方接收到该信号后会产生一个停止信号，结束信号传输。见图 1-8。

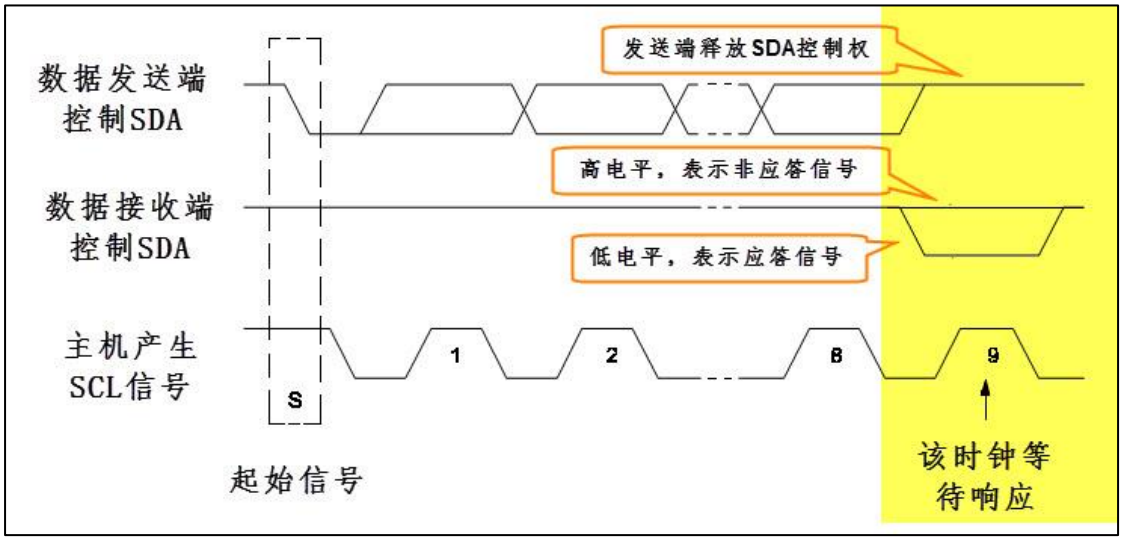


图 1-8 响应与非响应信号

传输时主机产生时钟，在第 9 个时钟时，数据发送端会释放 SDA 的控制权，由数据接收端控制 SDA，若 SDA 为高电平，表示非应答信号(NACK)，低电平表示应答信号(ACK)。

1.2 STM32 的 I2C 特性及架构

如果我们直接控制 STM32 的两个 GPIO 引脚，分别用作 SCL 及 SDA，按照上述信号的时序要求，直接像控制 LED 灯那样控制引脚的输出(若是接收数据时则读取 SDA 电平)，就可以实现 I2C 通讯。同样，假如我们按照 USART 的要求去控制引脚，也能实现 USART 通讯。所以只要遵守协议，就是标准的通讯，不管您如何实现它，不管是 ST 生产的控制器还是 ATMEL 生产的存储器， 都能按通讯标准交互。

由于直接控制 GPIO 引脚电平产生通讯时序时，需要由 CPU 控制每个时刻的引脚状态，所以称之为“软件模拟协议”方式。

相对地，还有“硬件协议”方式，STM32 的 I2C 片上外设专门负责实现 I2C 通讯协议，只要配置好该外设，它就会自动根据协议要求产生通讯信号，收发数据并缓存起来，CPU 只要检测该外设的状态和访问数据寄存器，就能完成数据收发。这种由硬件外设处理 I2C 协议的方式减轻了 CPU 的工作，且使软件设计更加简单。

1.2.1 STM32 的 I^2C 外设简介

STM32 的 I2C 外设可用作通讯的主机及从机，支持 100Kbit/s 和 400Kbit/s 的速率，支持 7 位、10 位设备地址，支持 DMA 数据传输，并具有数据校验功能。它的 I2C 外设还支持 SMBus2.0 协议，SMBus 协议与 I2C 类似，主要应用于笔记本电脑的电池管理中，本教程不展开，感兴趣的读者可参考《SMBus20》文档了解。

1.2.2 STM32 的 I²C 架构剖析

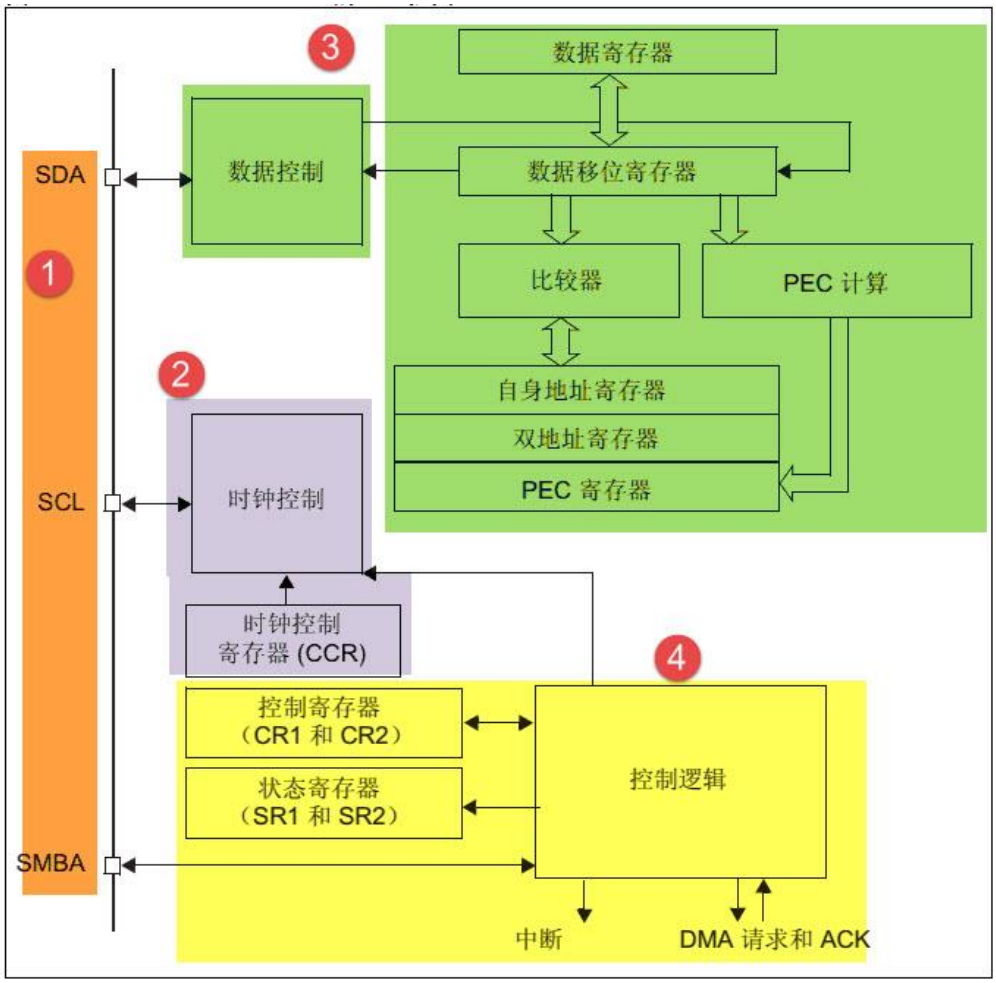


图 1-9 I2C 架构图

1. 通讯引脚

I²C 的所有硬件架构都是根据图中左侧 SCL 线和 SDA 线展开的(其中的 SMBA 线用于 SMBUS 的警告信号，I2C 通讯没有使用)。STM32 芯片有多个 I2C 外设，它们的 I2C 通讯信号引出到不同的 GPIO 引脚上，使用时必须配置到这些指定的引脚，见

表 1-1。关于 GPIO 引脚的复用功能，以规格书为准。

表 1-1 STM32F10x 的 I2C 引脚

引脚	I2C1	I2C2
SCL	PB5 / PB8(重映射)	PB10
SDA	PB6 / PB9(重映射)	PB11

2. 时钟控制逻辑

SCL 线的时钟信号，由 I^2C 接口根据时钟控制寄存器(CCR)控制，控制的参数主要为时钟频率。配置 I2C 的 CCR 寄存器可修改通讯速率相关的参数：

- 可选择 I2C 通讯的“标准/快速”模式，这两个模式分别 I2C 对应 100/400Kbit/s 的通讯速率。

- 在快速模式下可选择 SCL 时钟的占空比，可选 $T_{low}/T_{high}=2$ 或 $T_{low}/T_{high}=16/9$ 模式，我们知道 I2C 协议在 SCL 高电平时对 SDA 信号采样，SCL 低电平时 SDA 准备下一个数据，修改 SCL 的高低电平比会影响数据采样，但其实这两个模式的比例差别并不大，若不是要求非常严格，这里随便选就可以了。

- CCR 寄存器中还有一个 12 位的配置因子 CCR，它与 I2C 外设的输入时钟源共同作用，产生 SCL 时钟，STM32 的 I2C 外设都挂载在 APB1 总线上，使用 APB1 的时钟源 PCLK1，SCL 信号线的输出时钟公式如下：

标准模式：

$T_{high}=CCR*T_{PCKL1}$　　$T_{low} = CCR*T_{PCLK1}$

快速模式中 $T_{low}/T_{high}=2$ 时：

$T_{high} = CCR*T_{PCKL1}$　　$T_{low} = 2*CCR*T_{PCKL1}$

快速模式中 $T_{low}/T_{high}=16/9$ 时：

$T_{high} = 9*CCR*T_{PCKL1}$　　$T_{low} = 16*CCR*T_{PCKL1}$

例如，我们的 PCLK1=45MHz，想要配置 400Kbit/s 的速率，计算方式如下：

PCLK 时钟周期：　　TPCLK1 = 1/45000000

目标 SCL 时钟周期：　　TSCL = 1/400000

SCL 时钟周期内的高电平时间：THIGH = TSCL/3

SCL 时钟周期内的低电平时间：TLOW = 2*TSCL/3

计算 CCR 的值：　　CCR = THIGH/TPCLK1 = 37.5

计算结果为小数，而 CCR 寄存器是无法配置小数参数的，所以我们只能把 CCR 取值为 38，这样 I2C 的 SCL 实际频率无法达到 400KHz (约为 394736Hz)。要想它实际频率达到 400KHz，需要修改 STM32 的系统时钟，把 PCLK1 时钟频率改成 10 的倍数才可以，但修改 PCKL 时钟影响很多外设，所以一般我们不会修改它。SCL 的实际频率不达到 400KHz，除了通讯稍慢一点以外，不会对 I2C 的标准通讯造成其它影响

3. 数据控制逻辑

I2C 的 SDA 信号主要连接到数据移位寄存器上，数据移位寄存器的数据来源及目标是数据寄存器(DR)、地址寄存器(OAR)、PEC 寄存器以及 SDA 数据线。当向外发送数据的时候，数据移位寄存器以“数据寄存器”为数据源，把数据一位一位地通过 SDA 信号线发送出去；当从外部接收数据的时候，数据移位寄存器把 SDA 信号线采样到的数据一位一位地存储到“数据寄存器”中。若使能了数据校验，接收到的数据会经过 PCE 计算器运算，运算结果存储在“PEC 寄存器”中。当 STM32 的 I2C 工作在从机模式的时候，接收到设备地址信号时，数据移位寄存器会把接收到的地址与 STM32 的自身的“I2C 地址寄存器”的值作比较，以便响应主机的寻址。STM32 的自身 I2C 地址可通过修改“自身地址寄存器”修改，支持同时使用两个 I2C 设备地址，两个地址分别存储在 OAR1 和 OAR2 中。

4. 整体控制逻辑

整体控制逻辑负责协调整个 I2C 外设，控制逻辑的工作模式根据我们配置的“控制寄存器(CR1/CR2)”的参数而改变。在外设工作时，控制逻辑会根据外设的工作状态修改“状态寄存器(SR1 和 SR2)”，我们只要读取这些寄存器相关的寄存器位，就可以了解 I2C 的工作状态。除此之外，控制逻辑还根据要求，负责控制产生 I2C 中断信号、DMA 请求及各种 I2C 的通讯信号(起始、停止、响应信号等)。

1.2.3 通讯过程

使用 I2C 外设通讯时，在通讯的不同阶段它会对“状态寄存器(SR1 及 SR2)”的不同数据位写入参数，我们通过读取这些寄存器标志来了解通讯状态。

1. 主发送器

见图 1-10。图中的是“主发送器”流程，即作为 I2C 通讯的主机端时，向外发送数据时的过程。

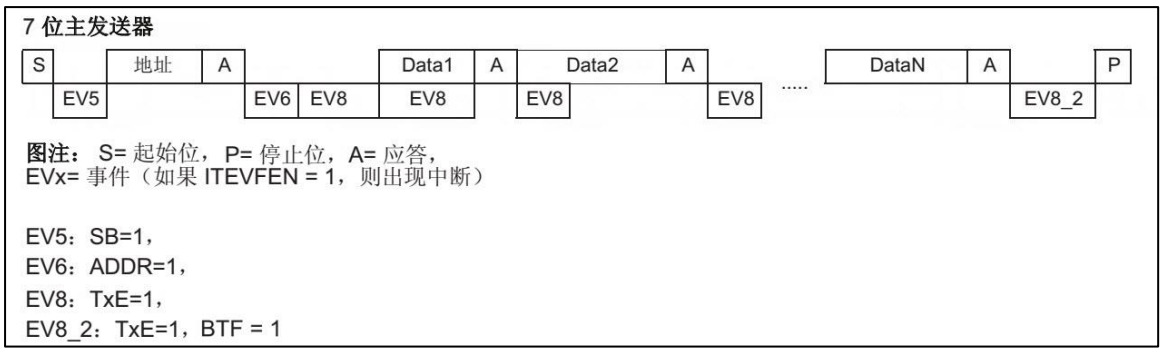


图 1-10 主发送器通讯过程

主发送器发送流程及事件说明如下：

(1) 控制产生起始信号(S)，当发生起始信号后，它产生事件“EV5”，并会对 SR1 寄存器的“SB”位置 1，表示起始信号已经发送；

(2) 紧接着发送设备地址并等待应答信号，若有从机应答，则产生事件“EV6”及“EV8”，这时 SR1 寄存器的“ADDR”位及“TXE”位被置 1，ADDR 为 1 表示地址已经发送，TXE 为 1 表示数据寄存器为空；

(3) 以上步骤正常执行并对 ADDR 位清零后，我们往 I2C 的“数据寄存器 DR”写入要发送的数据，这时 TXE 位会被重置 0，表示数据寄存器非空，I2C 外设通过 SDA 信号线一位位把数据发送出去后，又会产生“EV8”事件，即 TXE 位被置 1，重复这个过程，就可以发送多个字节数据了；

(4) 当我们发送数据完成后，控制 I2C 设备产生一个停止信号(P)，这个时候会产生 EV8_2 事件，SR1 的 TXE 位及 BTF 位都被置 1，表示通讯结束。

假如我们使能了 I2C 中断，以上所有事件产生时，都会产生 I2C 中断信号，进入同一个中断服务函数，到 I2C 中断服务程序后，再通过检查寄存器位来判断是哪一个事件。

2. 主接收器

再来分析主接收器过程，即作为 I2C 通讯的主机端时，从外部接收数据的过程，见图 1-11。

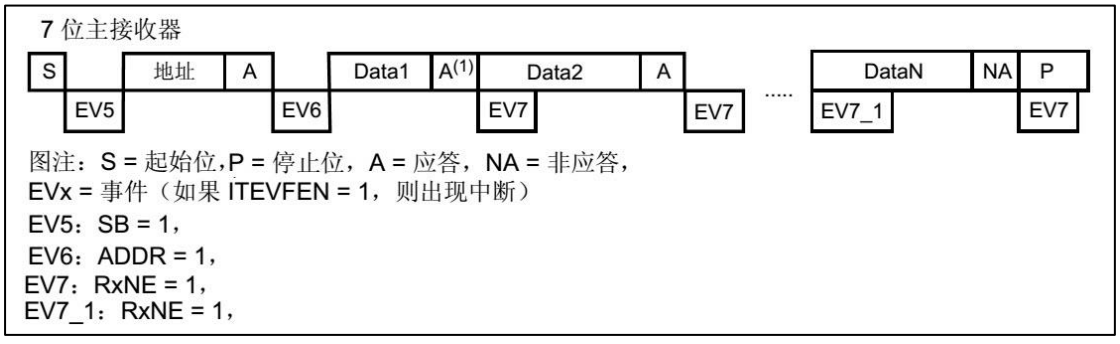


图 1-11 主接收器过程

主接收器接收流程及事件说明如下：

(1) 同主发送流程，起始信号(S)是由主机端产生的，控制发生起始信号后，它产生事件“EV5”，并会对 SR1 寄存器的“SB”位置 1，表示起始信号已经发送；

(2) 紧接着发送设备地址并等待应答信号，若有从机应答，则产生事件“EV6”这时 SR1 寄存器的“ADDR”位被置 1，表示地址已经发送。

(3) 从机端接收到地址后，开始向主机端发送数据。当主机接收到这些数据后，会产生“EV7”事件，SR1 寄存器的 RXNE 被置 1，表示接收数据寄存器非空，我们读取该寄存器后，可对数据寄存器清空，以便接收下一次数据。此时我们可以控制 I2C 发送应答信号(ACK)或非应答信号(NACK)，若应答，则重复以上步骤接收数据，若非应答，则停止传输；

(4) 发送非应答信号后，产生停止信号(P)，结束传输。

在发送和接收过程中，有的事件不只是标志了我们上面提到的状态位，还可能同时标志主机状态之类的状态位，而且读了之后还需要清除标志位，比较复杂。我们可使用 STM32 标准库函数来直接检测这些事件的复合标志，降低编程难度。

1.3 I2C 初始化结构体详解

跟其它外设一样，STM32 标准库提供了 I2C 初始化结构体及初始化函数来配置 I2C 外设。初始化结构体及函数定义在库文件“stm32f10x_i2c.h”及“stm32f10x_i2c.c”中，编程时我们可以结合这两个文件内的注释使用或参考库帮助文档。了解初始化结构体后我们就能对 I2C 外设运用自如了，见代码清单 1-1。

代码清单 1-1 I2C 初始化结构体

```
typedef struct {
    uint32_t I2C_ClockSpeed; /*!< 设置 SCL 时钟频率，此值要低于 400000*/
    uint16_t I2C_Mode;       /*!< 指定工作模式，可选 I2C 模式及 SMBUS 模式 */
    uint16_t I2C_DutyCycle; /*指定时钟占空比，可选 low/high = 2:1 及 16:9 模式*/
    uint16_t I2C_OwnAddress1;     /*!< 指定自身的 I2C 设备地址 */
    uint16_t I2C_Ack;                 /*!< 使能或关闭响应(一般都要使能) */
    uint16_t I2C_AcknowledgedAddress; /*!< 指定地址的长度，可为 7 位及 10 位 */
} I2C_InitTypeDef;
```

这些结构体成员说明如下，其中括号内的文字是对应参数在 STM32 标准库中定义的宏：

(1) I2C_ClockSpeed

本成员设置的是 I2C 的传输速率，在调用初始化函数时，函数会根据我们输入的数值经过运算后把时钟因子写入到 I2C 的时钟控制寄存器 CCR。而我们写入的这个参数值不得高于 400KHz。实际上由于 CCR 寄存器不能写入小数类型的时钟因子，影响到 SCL 的实际频率可能会低于本成员设置的参数值，这时除了通讯稍慢一点以外，不会对 I2C 的标准通讯造成其它影响。

(2) I2C_Mode

本成员是选择 I^2C 的使用方式，有 I^2C 模式(I2C_Mode_I2C)和 SMBus 主、从模式(I2C_Mode_SMBusHost、 I2C_Mode_SMBusDevice) 。I2C 不需要在此处区分主从模式，直接设置 I2C_Mode_I2C 即可。

(3) I2C_DutyCycle

本成员设置的是 I^2C 的 SCL 线时钟的占空比。该配置有两个选择，分别为低电平时间比高电平时间为 2：1 (I2C_DutyCycle_2)和 16：9 (I2C_DutyCycle_16_9)。其实这两个模式的比例差别并不大，一般要求都不会如此严格，这里随便选就可以。

(4) I2C_OwnAddress1

本成员配置的是 STM32 的 I2C 设备自己的地址，每个连接到 I2C 总线上的设备都要有一个自己的地址，作为主机也不例外。地址可设置为 7 位或 10 位(受下面 I2C_AcknowledgeAddress 成员决定)，只要该地址是 I2C 总线上唯一的即可。

STM32 的 I2C 外设可同时使用两个地址，即同时对两个地址作出响应，这个结构成员 I2C_OwnAddress1 配置的是默认的、OAR1 寄存器存储的地址，若需要设置第二个地址寄存器 OAR2，可使用 I2C_OwnAddress2Config 函数来配置，OAR2 不支持 10 位地址，只有 7 位。

(5) I2C_Ack_Enable

本成员是关于 I^2C 应答设置，设置为使能则可以发送响应信号。本实验配置为允许应答(I2C_Ack_Enable)，这是绝大多数遵循 I^2C 标准的设备的通讯要求，改为禁止应答(I2C_Ack_Disable)往往会导致通讯错误。

(6) I2C_AcknowledgeAddress

本成员选择 I2C 的寻址模式是 7 位还是 10 位地址。这需要根据实际连接到 I2C 总线上设备的地址进行选择，这个成员的配置也影响到 I2C_OwnAddress1 成员，只有这里设置成 10 位模式时，I2C_OwnAddress1 才支持 10 位地址。

配置完这些结构体成员值，调用库函数 I2C_Init 即可把结构体的配置写入到寄存器中。

1.4 I2C—读写 EEPROM 实验

EEPROM 是一种掉电后数据不丢失的存储器，常用来存储一些配置信息，以便系统重新上电的时候加载之。EEPOM 芯片最常用的通讯方式就是 I^2C 协议，本小节以 EEPROM 的读写实验为大家讲解 STM32 的 I^2C 使用方法。实验中 STM32 的 I2C 外设采用主模式，分别用作主发送器和主接收器，通过查询事件的方式来确保正常通讯。

1.4.1 硬件设计

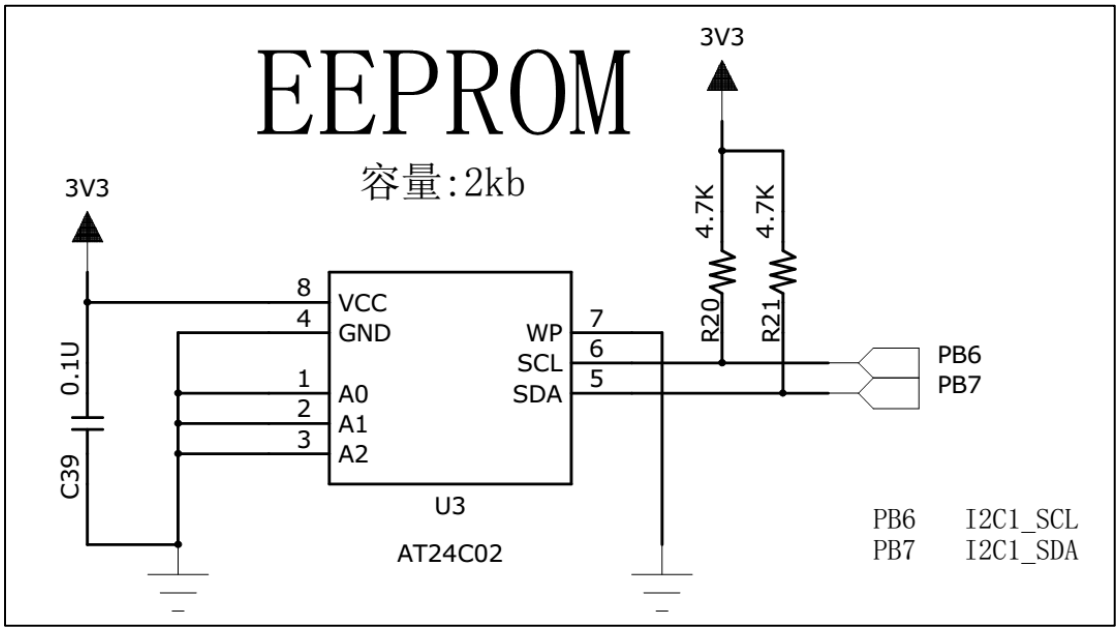


图 1-12 EEPROM 硬件连接图

本实验板中的 EEPROM 芯片(型号：AT24C02)的 SCL 及 SDA 引脚连接到了 STM32 对应的 I2C 引脚中，结合上拉电阻，构成了 I2C 通讯总线，它们通过 I2C 总线交互。EEPROM 芯片的设备地址一共有 7 位，其中高 4 位固定为：1010 b，低 3 位则由 A0/A1/A2 信号线的电平决定，见图 1-13，图中的 R/W 是读写方向位，与地址无关。

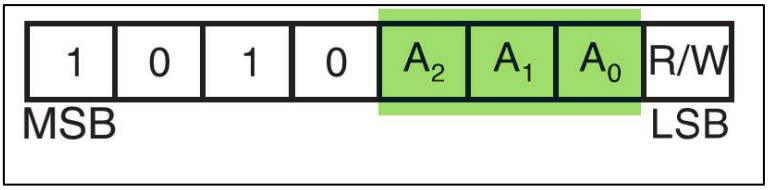


图 1-13 EEPROM 设备地址(摘自《AT24C02》规格书)

按照我们此处的连接，A0/A1/A2 均为 0，所以 EEPROM 的 7 位设备地址是：1010000b，即 0x50。由于 I2C 通讯时常常是地址跟读写方向连在一起构成一个 8 位数，且当 R/W 位为 0 时，表示写方向，所以加上 7 位地址，其值为“0xA0”，常称该值为 I2C 设备的“写地址”；当 R/W 位为 1 时，表示读方向，加上 7 位地址，其值为“0xA1”，常称该值为“读地址”。

EEPROM 芯片中还有一个 WP 引脚，具有写保护功能，当该引脚电平为高时，禁止写入数据，当引脚为低电平时，可写入数据，我们直接接地，不使用写保护功能。

关于 EEPROM 的更多信息，可参考其数据手册《AT24C02》来了解。若您使用的实验板 EEPROM 的型号、设备地址或控制引脚不一样，只需根据我们的工程修改即可，程序的控制原理相同。

1.4.2 软件设计

为了使工程更加有条理，我们把读写 EEPROM 相关的代码独立分开存储，方便以后移植。在“工程模板”之上新建“bsp_i2c_ee.c”及“bsp_i2c_ee.h”文件，这些文件也可根据您的喜好命名，它们不属于 STM32 标准库的内容，是由我们自己根据应用需要编写的。

1. 编程要点

(1) 配置通讯使用的目标引脚为开漏模式；
(2) 使能 I2C 外设的时钟；
(3) 配置 I2C 外设的模式、地址、速率等参数并使能 I2C 外设；
(4) 编写基本 I2C 按字节收发的函数；
(5) 编写读写 EEPROM 存储内容的函数；
(6) 编写测试程序，对读写数据进行校验。

2. 代码分析

I2C 硬件相关宏定义

我们把 I2C 硬件相关的配置都以宏的形式定义到“bsp_i2c_ee.h”文件中，见代码清单 1-2。

代码清单 1-2 I2C 硬件配置相关的宏

```
1  // STM32 I2C 快速模式
2  #define I2C_Speed              400000
3
4  // 这个地址只要与 STM32 外挂的 I2C 器件地址不一样即可
5  #define I2Cx_OWN_ADDRESS7      0X0A
6
7  #define EEPROM_ADDRESS 0xA0
8  // AT24C01/02 每页有 8 个字节
9  #define I2C_PageSize           8
10
11 /*********************I2C 参数定义，I2C1 或 I2C2**************/
12 #define    I2Cx                        I2C1
13 #define    I2C_APBxClock_FUN           RCC_APB1PeriphClockCmd
14 #define    I2C_CLK                     RCC_APB1Periph_I2C1
15
16 #define    I2C_GPIO_APBxClock_FUN      RCC_APB2PeriphClockCmd
17 #define    I2C_GPIO_CLK                RCC_APB2Periph_GPIOB
18 #define    I2C_SCL_PORT                GPIOB
19 #define    I2C_SCL_PIN                 GPIO_Pin_6
20 #define    I2C_SDA_PORT                GPIOB
21 #define    I2C_SDA_PIN                 GPIO_Pin_7
```

以上代码根据硬件连接，把与 EEPROM 通讯使用的 I2C 号 、引脚号都以宏封装起来，并且定义了自身的 I2C 地址及通讯速率，以便配置模式的时候使用。

初始化 I2C 的 GPIO

利用上面的宏，编写 I2C GPIO 引脚的初始化函数，见**错误!未找到引用源。**。

代码清单 1-3 I2C GPIO 初始化函数

```
1  static void I2C_GPIO_Config(void)
2  {
```

```
    GPIO_InitTypeDef  GPIO_InitStructure;

    /* 开启 I2C 的时钟 */
    I2C_APBxClock_FUN ( I2C_CLK, ENABLE );
    /* 开启 I2C GPIO 端口时钟 */
    I2C_GPIO_APBxClock_FUN ( I2C_GPIO_CLK, ENABLE );

    /* 引脚设置*/
    GPIO_InitStructure.GPIO_Pin = I2C_SCL_PIN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    /* 复用开漏输出 */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
    GPIO_Init(I2C_SCL_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = I2C_SDA_PIN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    /* 复用开漏输出 */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
    GPIO_Init(I2C_SDA_PORT, &GPIO_InitStructure);
}
```

开启相关的时钟并初始化 GPIO 引脚，函数执行流程如下：

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量，以便下面用于存储 GPIO 配置；

(2) 调用库函数 RCC_APB1PeriphClockCmd 使能 I2C 外设时钟，调用 RCC_APB2PeriphClockCmd 来使能 I2C 引脚使用的 GPIO 端口时钟，调用时我们使用“|”操作同时配置两个引脚。

(3) 向 GPIO 初始化结构体赋值，把引脚初始化成复用开漏模式，要注意 I2C 的引脚必须使用这种模式。

(4) 使用以上初始化结构体的配置，调用 GPIO_Init 函数向寄存器写入参数，完成 GPIO 的初始化。

配置 I2C 的模式

以上只是配置了 I2C 使用的引脚，还不算对 I2C 模式的配置，见代码清单 1-4。

代码清单 1-4 配置 I2C 模式

```
/**
  * @brief  I2C 工作模式配置
  * @param  无
  * @retval 无
  */
static void I2C_Mode_Configu(void)
{
    I2C_InitTypeDef  I2C_InitStructure;

    /* I2C 模式 */
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
    /* 高电平数据稳定，低电平数据变化 SCL 时钟线的占空比 */
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
    /* I2C 自身地址 */
    I2C_InitStructure.I2C_OwnAddress1 =I2Cx_OWN_ADDRESS7;
    /* 使能响应 */
```

```
    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable ;
    /* I2C 的寻址模式 */
    I2C_InitStructure.I2C_AcknowledgedAddress =
    I2C_AcknowledgedAddress_7bit;
    /* 通信速率 */
    I2C_InitStructure.I2C_ClockSpeed = I2C_Speed;
    /* I2C 初始化 */
    I2C_Init(I2Cx, &I2C_InitStructure);
    /* 使能 I2C */
    I2C_Cmd(I2Cx, ENABLE);
}

/**
  * @brief  I2C 外设(EEPROM)初始化
  * @param  无
  * @retval 无
  */
void I2C_EE_Init(void)
{
    I2C_GPIO_Config();
    I2C_Mode_Configu();
}
```

熟悉 STM32 I2C 结构的话，这段初始化程序就十分好理解，它把 I2C 外设通讯时钟 SCL 的低/高电平比设置为 2，使能响应功能，使用 7 位地址 I2C_OWN_ADDRESS7 以及速率配置为 I2C_Speed(前面在 bsp_i2c_ee.h 定义的宏)。最后调用库函数 I2C_Init 把这些配置写入寄存器，并调用 I2C_Cmd 函数使能外设。

为方便调用，我们把 I2C 的 GPIO 及模式配置都用 I2C_EE_Init 函数封装起来。

向 EEPROM 写入一个字节的数据

初始化好 I2C 外设后，就可以使用 I2C 通讯，我们看看如何向 EEPROM 写入一个字节的数据，见代码清单 1-5。

代码清单 1-5 向 EEPROM 写入一个字节的数据

```
/*
 * 通讯时常见的几种标志位，具体参考手册中的 I2C 主发送器传送序列图(24.3.3 章节)
 * EV5 = 1:起始信号发送成功
 * EV6 = 1:地址发送成功
 * EV7 = 1:数据寄存器非空
 * EV8 = 1:数据发送成功
 */
void I2C_EE_ByteWrite(u8* pBuffer, u8 WriteAddr)
{
    /* 发送起始信号 */
    I2C_GenerateSTART(I2Cx, ENABLE);

    /* 检测 EV5 事件并清除标志*/
    while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

    /* 发送 EEPROM 设备地址 + 写方向 */
    I2C_Send7bitAddress(I2Cx, EEPROM_ADDRESS, I2C_Direction_Transmitter);

    /* 检测 EV6 事件并清除标志*/
    while (!I2C_CheckEvent(I2Cx,
                           I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

    /* 发送要写入的 EEPROM 内部地址(即 EEPROM 内部存储器的地址) */
    I2C_SendData(I2Cx, WriteAddr);

    /* 检测 EV8 事件并清除标志 */
    while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
```

```
28
29      /* 发送一字节要写入的数据 */
30      I2C_SendData(I2Cx, *pBuffer);
31
32      /* 检测 EV8 事件并清除标志*/
33      while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
34
35      /* 发送停止信号 */
36      I2C_GenerateSTOP(I2Cx, ENABLE);
37  }
```

I2C_EE_ByteWrite 函数，这个函数实现了前面讲的 I2C 主发送器通讯流程：

(1) 使用库函数 I2C_GenerateSTART 产生 I2C 起始信号，其中的 I2Cx 宏是前面硬件定义相关的 I2C 编号；然后调用库函数 I2C_CheckEvent()来检测 EV5 事件：起始信号是否发送成功，这里检测的是 I2C_CR1 的 SB 位。

(2) 调用库函数 I2C_Send7bitAddress 发送 EEPROM 的设备地址，并把数据传输方向设置为 I2C_Direction_Transmitter(即发送方向)，这个数据传输方向就是通过设置 I2C 通讯中紧跟地址后面的 R/W 位实现的。发送地址后以同样的方式检测 EV6 事件：地址是否发送成功，这里检测的是 I2C_SR1 的 ADDR 位；

(3) 调用库函数 I2C_SendData 向 EEPROM 发送要写入的内部地址，该地址是 I2C_EE_ByteWrite 函数的输入参数，发送完毕后等待 EV8 事件：检测的是 I2C_SR1 的 TxE 位。要注意这个内部地址跟上面的 EEPROM 地址不一样，上面的是指 I2C 总线设备的独立地址，而此处的内部地址是指 EEPROM 内数据组织的地址，也可理解为 EEPROM 内存的地址或 I2C 设备的寄存器地址；

(4) 调用库函数 I2C_SendData 向 EEPROM 发送要写入的数据，该数据是 I2C_EE_ByteWrite 函数的输入参数，发送完毕后等待 EV8 事件；

(5) 一个 I2C 通讯过程完毕，调用 I2C_GenerateSTOP 发送停止信号。

在这个通讯过程中，STM32 实际上通过 I2C 向 EEPROM 发送了两个数据，但为何第一个数据被解释为 EEPROM 的内存地址？这是由 EEPROM 的自己定义的单字节写入时序，见图 1-14。

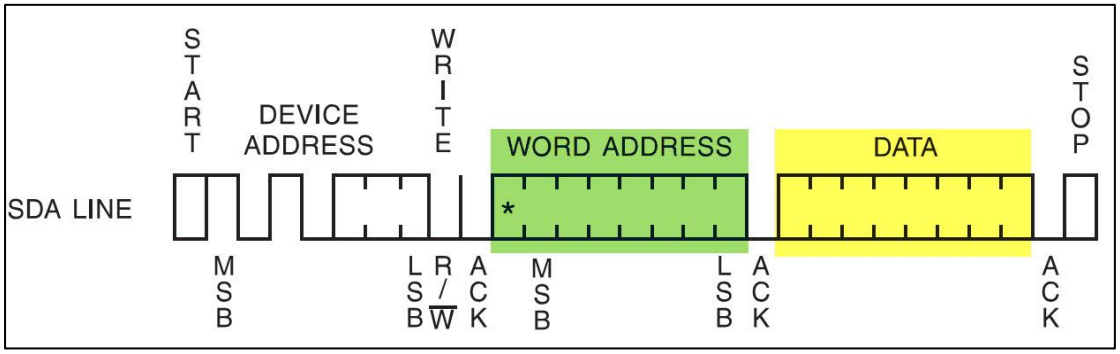


图 1-14 EEPROM 单字节写入时序(摘自《AT24C02》规格书)

EEPROM 的单字节时序规定，向它写入数据的时候，第一个字节为内存地址，第二个字节是要写入的数据内容。所以我们需要理解：命令、地址的本质都是数据，对数据的解释不同，它就有了不同的功能。

多字节写入及状态等待

单字节写入通讯结束后，EEPROM 芯片会根据这个通讯结果擦写该内存地址的内容，这需要一段时间，所以我们在多次写入数据时，要先等待 EEPROM 内部擦写完毕。多个数据写入过程见代码清单 1-6。

代码清单 1-6 多字节写入

```
/**
  * @brief   将缓冲区中的数据写到 I2C EEPROM 中，采用单字节写入的方式，
  *          速度比页写入慢
  * @param  pBuffer:缓冲区指针
  * @param  WriteAddr:写地址
  * @param  NumByteToWrite:写的字节数
  * @retval 无
  */
uint8_t I2C_EE_ByetsWrite(uint8_t* pBuffer,uint8_t WriteAddr,
                          uint16_t NumByteToWrite)
{
    uint16_t i;
    uint8_t res;

    /*每写一个字节调用一次 I2C_EE_ByteWrite 函数*/
    for (i=0; i<NumByteToWrite; i++)
    {
        /*等待 EEPROM 准备完毕*/
        I2C_EE_WaitEepromStandbyState();
        /*按字节写入数据*/
        res = I2C_EE_ByteWrite(pBuffer++,WriteAddr++);
    }
    return res;
}
```

这段代码比较简单，直接使用 for 循环调用前面定义的 I2C_EE_ByteWrite 函数一个字节一个字节地向 EEPROM 发送要写入的数据。在每次数据写入通讯前调用了 I2C_EE_WaitEepromStandbyState 函数等待 EEPROM 内部擦写完毕，该函数的定义见代码清单 1-7。

代码清单 1-7 等待 EEPROM 处于准备状态

```
/**
  * @brief  等待 EEPROM 到准备状态
  * @param  无
  * @retval 无
  */
void I2C_EE_WaitEepromStandbyState(void)
{
    vu16 SR1_Tmp = 0;

    do {
        /* 发送起始信号 */
        I2C_GenerateSTART(I2Cx, ENABLE);
```

```
13
14          /* 读 I2C1 SR1 寄存器 */
15          SR1_Tmp = I2C_ReadRegister(I2Cx, I2C_Register_SR1);
16
17          /* 发送 EEPROM 地址 + 写方向 */
18          I2C_Send7bitAddress(I2Cx, EEPROM_ADDRESS,
19                              I2C_Direction_Transmitter);
20      }
21      // SR1 位 1 ADDR：1 表示地址发送成功，0 表示地址发送没有结束
22      // 等待地址发送成功
23      while (!(I2C_ReadRegister(I2Cx, I2C_Register_SR1) & 0x0002));
24
25      /* 清除 AF 位 */
26      I2C_ClearFlag(I2Cx, I2C_FLAG_AF);
27      /* 发送停止信号 */
28      I2C_GenerateSTOP(I2Cx, ENABLE);
29  }
```

这个函数主要实现是向 EEPROM 发送它设备地址，检测 EEPROM 的响应，若 EEPROM 接收到地址后返回应答信号，则表示 EEPROM 已经准备好，可以开始下一次通讯。函数中检测响应是通过读取 STM32 的 SR1 寄存器的 ADDR 位及 AF 位来实现的，当 I2C 设备响应了地址的时候，ADDR 会置 1，若应答失败，AF 位会置 1。

EEPROM 的页写入

在以上的数据通讯中，每写入一个数据都需要向 EEPROM 发送写入的地址，我们希望向连续地址写入多个数据的时候，只要告诉 EEPROM 第一个内存地址 address1，后面的数据按次序写入到 address2、address3… 这样可以节省通讯的时间，加快速度。为应对这种需求，EEPROM 定义了一种页写入时序，见图 1-15。

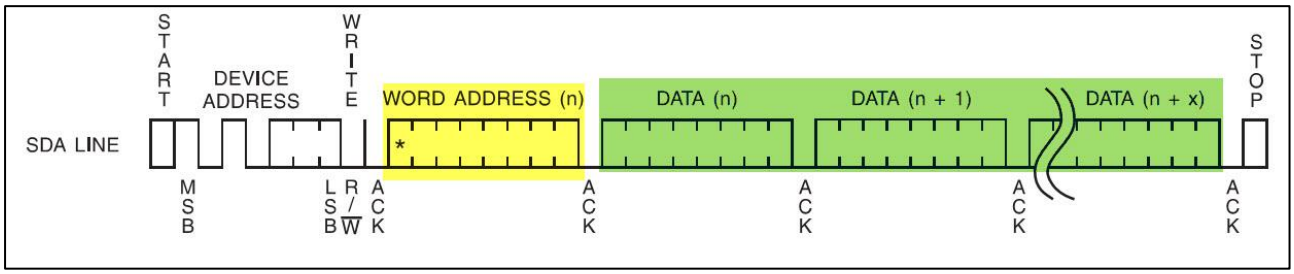


图 1-15 EEPROM 页写入时序(摘自《AT24C02》规格书)

根据页写入时序，第一个数据被解释为要写入的内存地址 address1，后续可连续发送 n 个数据，这些数据会依次写入到内存中。其中 AT24C02 型号的芯片页写入时序最多可以一次发送 8 个数据(即 n = 8)，该值也称为页大小，某些型号的芯片每个页写入时序最多可传输 16 个数据。EEPROM 的页写入代码实现见代码清单 1-8。

代码清单 1-8 EEPROM 的页写入

```
1  void I2C_EE_PageWrite(u8* pBuffer, u8 WriteAddr, u8 NumByteToWrite)
2  {
3      /* 等待总线空闲 */
4      while (I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));
5
6      /* 发送起始信号 */
7      I2C_GenerateSTART(I2Cx, ENABLE);
8
9      /* 检测 EV5 事件并清除标志 */
10     while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));
```

```
11
12      /* 发送 EEPROM 设备地址+写方向 */
13      I2C_Send7bitAddress(I2Cx, EEPROM_ADDRESS, I2C_Direction_Transmitter);
14
15      /* 检测 EV6 事件并清除标志*/
16      while (!I2C_CheckEvent(I2Cx,
17                             I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
18
19      /* 发送要写入的 EEPROM 内部地址(即 EEPROM 内部存储器的地址) */
20      I2C_SendData(I2Cx, WriteAddr);
21
22      /* 检测 EV8 事件并清除标志*/
23      while (! I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
24
25      /* 循环发送 NumByteToWrite 个数据 */
26      while (NumByteToWrite--) {
27          /* 发送缓冲区中的数据 */
28          I2C_SendData(I2Cx, *pBuffer);
29
30          /* 指向缓冲区中的下一个数据 */
31          pBuffer++;
32
33          /* 检测 EV8 事件并清除标志*/
34          while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
35      }
36
37      /* 发送停止信号 */
38      I2C_GenerateSTOP(I2Cx, ENABLE);
39  }
```

这段页写入函数主体跟单字节写入函数是一样的，只是它在发送数据的时候，使用 for 循环控制发送多个数据，发送完多个数据后才产生 I2C 停止信号，只要每次传输的数据小于等于 EEPROM 时序规定的页大小，就能正常传输。

快速写入多字节

利用 EEPROM 的页写入方式，可以改进前面的“多字节写入”函数，加快传输速度，见代码清单 1-9。

代码清单 1-9 快速写入多字节函数

```
 1 // AT24C01/02 每页有 8 个字节
 2 #define I2C_PageSize        8
 3
 4 /**
 5   * @brief   将缓冲区中的数据写到 I2C EEPROM 中
 6   * @param
 7   *   @arg pBuffer:缓冲区指针
 8   *   @arg WriteAddr:写地址
 9   *     @arg NumByteToWrite:写的字节数
10   * @retval  无
11   */
12 void I2C_EE_BufferWrite(u8* pBuffer, u8 WriteAddr,
13                          u16 NumByteToWrite)
14 {
15     u8 NumOfPage=0,NumOfSingle=0,Addr =0,count=0,temp =0;
16
17     /*mod 运算求余，若 writeAddr 是 I2C_PageSize 整数倍，
18      运算结果 Addr 值为 0*/
19     Addr = WriteAddr % I2C_PageSize;
20
21     /*差 count 个数据值，刚好可以对齐到页地址*/
22     count = I2C_PageSize - Addr;
23
```

```
24      /*计算出要写多少整数页*/
25      NumOfPage =  NumByteToWrite / I2C_PageSize;
26
27      /*mod 运算求余，计算出剩余不满一页的字节数*/
28      NumOfSingle = NumByteToWrite % I2C_PageSize;
29
30      // Addr=0,则 WriteAddr 刚好按页对齐 aligned
31      // 这样就很简单了，直接写就可以，写完整页后
32      // 把剩下的不满一页的写完即可
33      if (Addr == 0) {
34          /* 如果 NumByteToWrite < I2C_PageSize */
35          if (NumOfPage == 0) {
36              I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
37              I2C_EE_WaitEepromStandbyState();
38          }
39          /* 如果 NumByteToWrite > I2C_PageSize */
40          else {
41              /*先把整数页都写了*/
42              while (NumOfPage--) {
43                  I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
44                  I2C_EE_WaitEepromStandbyState();
45                  WriteAddr +=  I2C_PageSize;
46                  pBuffer += I2C_PageSize;
47              }
48              /*若有多余的不满一页的数据，把它写完*/
49              if (NumOfSingle!=0) {
50                  I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
51                  I2C_EE_WaitEepromStandbyState();
52              }
53          }
54      }
55      // 如果 WriteAddr 不是按 I2C_PageSize 对齐
56      // 那就算出对齐到页地址还需要多少个数据，然后
57      // 先把这几个数据写完，剩下开始的地址就已经对齐
58      // 到页地址了，代码重复上面的即可
59      else {
60          /* 如果 NumByteToWrite < I2C_PageSize */
61          if (NumOfPage== 0) {
62              /*若 NumOfSingle>count，当前面写不完，要写到下一页*/
63              if (NumOfSingle > count) {
64                  // temp 的数据要写到写一页
65                  temp = NumOfSingle - count;
66
67                  I2C_EE_PageWrite(pBuffer, WriteAddr, count);
68                  I2C_EE_WaitEepromStandbyState();
69                  WriteAddr +=  count;
70                  pBuffer += count;
71
72                  I2C_EE_PageWrite(pBuffer, WriteAddr, temp);
73                  I2C_EE_WaitEepromStandbyState();
74              } else { /*若 count 比 NumOfSingle 大*/
75                  I2C_EE_PageWrite(pBuffer, WriteAddr, NumByteToWrite);
76                  I2C_EE_WaitEepromStandbyState();
77              }
78          }
79          /* 如果 NumByteToWrite > I2C_PageSize */
80          else {
81              /*地址不对齐多出的 count 分开处理，不加入这个运算*/
82              NumByteToWrite -= count;
83              NumOfPage =  NumByteToWrite / I2C_PageSize;
84              NumOfSingle = NumByteToWrite % I2C_PageSize;
85
86              /*先把 WriteAddr 所在页的剩余字节写了*/
87              if (count != 0) {
88                  I2C_EE_PageWrite(pBuffer, WriteAddr, count);
```

```
89                  I2C_EE_WaitEepromStandbyState();
90
91                  /*WriteAddr 加上 count 后，地址就对齐到页了*/
92                  WriteAddr += count;
93                  pBuffer += count;
94              }
95              /*把整数页都写了*/
96              while (NumOfPage--) {
97                  I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
98                  I2C_EE_WaitEepromStandbyState();
99                  WriteAddr +=  I2C_PageSize;
100                  pBuffer += I2C_PageSize;
101             }
102             /*若有多余的不满一页的数据，把它写完*/
103             if (NumOfSingle != 0) {
104                 I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
105                 I2C_EE_WaitEepromStandbyState();
106             }
107         }
108     }
109 }
```

很多读者觉得这段代码的运算很复杂，看不懂，其实主要抓住最主要的两点就很好理解了，Buffer 写的时候我们分两种情况来讨论 。

地址对齐

当写数据的开始地址是 8 的整数倍的时候叫地址对齐，其余的都不对齐。刚好在对齐地址开始写的时候，可以连续写 8 个字节把该页写满，这样的速度非常快。地址对齐的时候，代码比较好理解。通过“整除”计算要写入的数据 NumByteToWrite 能写满多少“完整的页”，计算得的值存储在 NumOfPage 中，但有时数据不是刚好能写满完整页的，会多一点出来，通过“求余”计算得出“不满一页的数据个数”就存储在 NumOfSingle 中。然后调用 I2C_EE_PageWrite()函数把数据写到 EERPOM 中。

地址不对齐

当写地址不是刚好对齐到页的首地址，会需要一个 count 值，用于存储从该首地址开始写满该地址所在的页，还能写多少个数据。实际传输时，先把这部分 count 个数据先写入，填满该页，然后把剩余的数据(NumByteToWrite-count)，再重复上述求出 NumOPage 及 NumOfSingle 的过程，按页传输到 EEPROM。

最后，强调一下，EEPROM 支持的页写入只是一种加速的 I2C 的传输时序，实际上并不要求每次都以页为单位进行读写，EEPROM 是支持随机访问的(直接读写任意一个地址)，如前面的单个字节写入。在某些存储器，如 NAND FLASH，它是必须按照 Block 写入的，例如每个 Block 为 512 或 4096 字节，数据写入的最小单位是 Block，写入前都需要擦除整个 Block；NOR FLASH 则是写入前必须以 Sector/Block 为单位擦除，然后才可以按字节写入。而我们的 EEPROM 数据写入和擦除的最小单位是“字节”而不是“页”，数据写入前不需要擦除整页。

从 EEPROM 读取数据

从 EEPROM 读取数据是一个复合的 I2C 时序，它实际上包含一个写过程和一个读过程，见图 1-16。

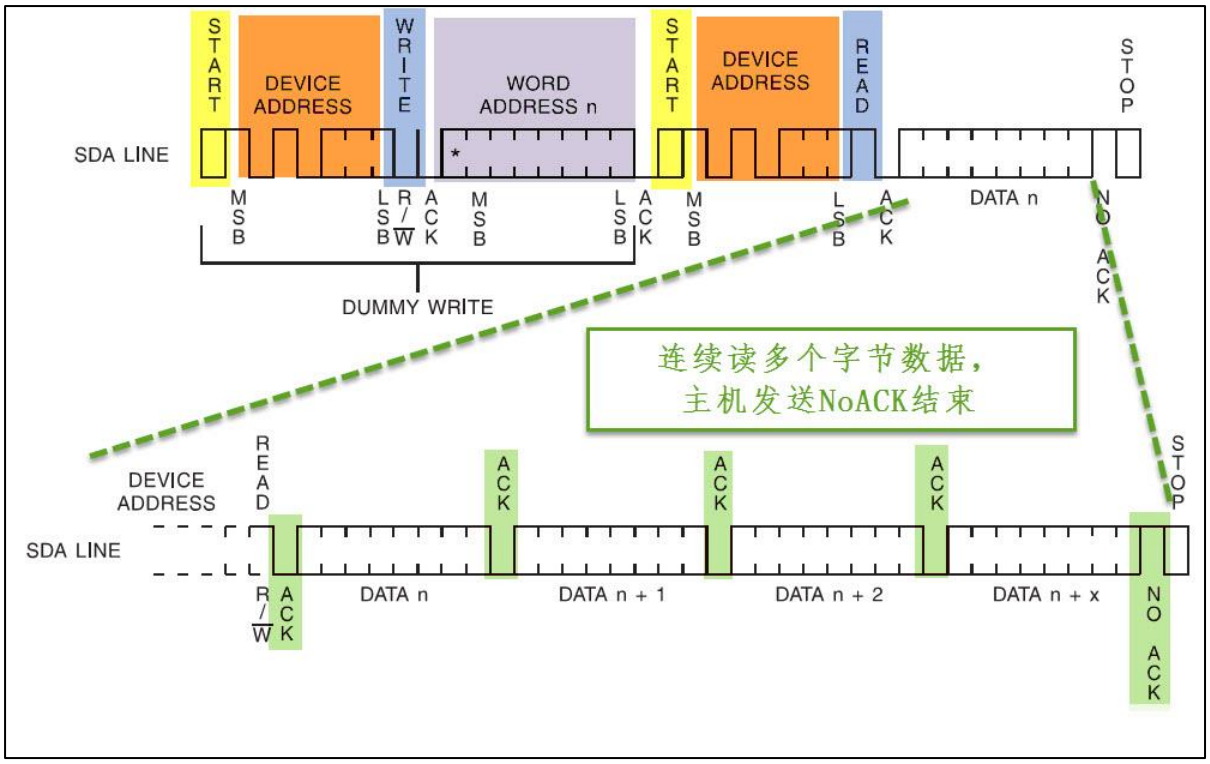


图 1-16 EEPROM 数据读取时序

读时序的第一个通讯过程中，使用 I2C 发送设备地址寻址(写方向)，接着发送要读取的“内存地址”；第二个通讯过程中，再次使用 I2C 发送设备地址寻址，但这个时候的数据方向是读方向；在这个过程之后，EEPROM 会向主机返回从“内存地址”开始的数据，一个字节一个字节地传输，只要主机的响应为“应答信号”，它就会一直传输下去，主机想结束传输时，就发送“非应答信号”，并以“停止信号”结束通讯，作为从机的 EEPROM 也会停止传输。实现代码见代码清单 1-10。

代码清单 1-10 从 EEPROM 读取数据

```
/**
  * @brief   从 EEPROM 里面读取一块数据
  * @param
  *   @arg pBuffer:存放从 EEPROM 读取的数据的缓冲区指针
  *   @arg WriteAddr:接收数据的 EEPROM 的地址
  *     @arg NumByteToWrite:要从 EEPROM 读取的字节数
  * @retval  无
  */
void I2C_EE_BufferRead(u8* pBuffer, u8 ReadAddr,
                       u16 NumByteToRead)
{
    // 等待总线空闲
    while (I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));

    /* 发送起始信号 */
    I2C_GenerateSTART(I2Cx, ENABLE);

    /* 检测 EV5 事件并清除标志*/
    while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

    /* 发送 EEPROM 设备地址+写方向 */
    I2C_Send7bitAddress(I2Cx, EEPROM_ADDRESS, I2C_Direction_Transmitter);

    /* 检测 EV6 事件并清除标志*/
```

```
25      while (!I2C_CheckEvent(I2Cx,
26                             I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
27
28      /*通过重新设置 PE 位清除 EV6 事件 */
29      I2C_Cmd(I2Cx, ENABLE);
30
31      /* 发送要读取的 EEPROM 内部地址(即 EEPROM 内部存储器的地址) */
32      I2C_SendData(I2Cx, ReadAddr);
33
34      /* 检测 EV8 事件并清除标志*/
35      while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
36
37      /* 产生第二次 I2C 起始信号 */
38      I2C_GenerateSTART(I2Cx, ENABLE);
39
40      /* 检测 EV5 事件并清除标志*/
41      while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));
42
43      /* 发送 EEPROM 设备地址 + 读方向*/
44      I2C_Send7bitAddress(I2Cx, EEPROM_ADDRESS, I2C_Direction_Receiver);
45
46      /* 检测 EV6 事件并清除标志*/
47      while (!I2C_CheckEvent(I2Cx,
48                             I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
49
50      /* 读取 NumByteToRead 个数据*/
51      while (NumByteToRead) {
52          /*若 NumByteToRead=1，表示已经接收到最后一个数据了，
53          发送非应答信号，结束传输*/
54          if (NumByteToRead == 1) {
55              /* 发送非应答信号 */
56              I2C_AcknowledgeConfig(I2Cx, DISABLE);
57
58              /* 发送停止信号 */
59              I2C_GenerateSTOP(I2Cx, ENABLE);
60          }
61
62          /* 检测 EV7 事件并清除标志*/
63          if (I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_RECEIVED)) {
64              /*通过 I2C，从设备中读取一个字节的数据 */
65              *pBuffer = I2C_ReceiveData(I2Cx);
66
67              /* 存储数据的指针指向下一个地址 */
68              pBuffer++;
69
70              /* 接收数据计数自减 */
71              NumByteToRead--;
72          }
73      }
74      /* 使能应答，方便下一次 I2C 传输 */
75      I2C_AcknowledgeConfig(I2Cx, ENABLE);
76  }
```

这段中的写过程跟前面的写字节函数类似，而读过程中接收数据时，需要使用库函数 I2C_ReceiveData 来读取。响应信号则通过库函数 I2C_AcknowledgeConfig 来发送，DISABLE 时为非响应信号，ENABLE 为响应信号。

3. main 文件

EEPROM 读写测试函数

完成基本的读写函数后，接下来我们编写一个读写测试函数来检验驱动程序，见代码清单 1-11。

代码清单 1-11 EEPROM 读写测试函数

```
1  void I2C_Test(void)
2  {
3      u16 i;
4
5      EEPROM_INFO("写入的数据");
6
7      for ( i=0; i<=255; i++ ) { //填充缓冲
8          I2c_Buf_Write[i] = i;
9
10         printf("0x%02X ", I2c_Buf_Write[i]);
11         if (i%16 == 15)
12             printf("\n\r");
13     }
14
15     //将 I2c_Buf_Write 中顺序递增的数据写入 EERPOM 中
16     I2C_EE_BufferWrite( I2c_Buf_Write, EEP_Firstpage,256);
17     EEPROM_INFO("写结束");
18
19     EEPROM_INFO("读出的数据");
20     //将 EEPROM 读出数据顺序保持到 I2c_Buf_Read 中
21     I2C_EE_BufferRead(I2c_Buf_Read, EEP_Firstpage, 256);
22
23     //将 I2c_Buf_Read 中的数据通过串口打印
24     for (i=0; i<256; i++) {
25         if (I2c_Buf_Read[i] != I2c_Buf_Write[i]) {
26             printf("0x%02X ", I2c_Buf_Read[i]);
27             EEPROM_ERROR("错误:I2C EEPROM 写入与读出的数据不一致");
28             return;
29         }
30         printf("0x%02X ", I2c_Buf_Read[i]);
31         if (i%16 == 15)
32             printf("\n\r");
33
34     }
35     EEPROM_INFO("I2C(AT24C02)读写测试成功");
36 }
```

代码中先填充一个数组，数组的内容为 1,2,3 至 N，接着把这个数组的内容写入到 EEPROM 中，写入时可以采用单字节写入的方式或页写入的方式。写入完毕后再从 EEPROM 的地址中读取数据，把读取得到的与写入的数据进行校验，若一致说明读写正常，否则读写过程有问题或者 EEPROM 芯片不正常。其中代码用到的 EEPROM_INFO 跟 EEPROM_ERROR 宏类似，都是对 printf 函数的封装，使用和阅读代码时把它直接当成 printf 函数就好。具体的宏定义在“bsp_i2c_ee.h 文件中”，在以后的代码我们常常会用类似的宏来输出调试信息。

main 函数

最后编写 main 函数，函数中初始化串口、I2C 外设，然后调用上面的 I2C_Test 函数进行读写测试，见代码清单 1-12。

代码清单 1-12 main 函数

```
int main(void)
{
    /* 串口初始化 */
    USART_Config();
    printf("\r\n 欢迎使用秉火  STM32 F103 开发板 \r\n");

    /* I2C 外设初始化 */
    I2C_EE_Init();
    printf("\r\n 这是一个 I2C 外设(AT24C02)读写测试例程 \r\n");

    I2C_Test();

    while (1) {
    }
}
```

1.4.3 下载验证

用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到 EEPROM 测试的调试信息。

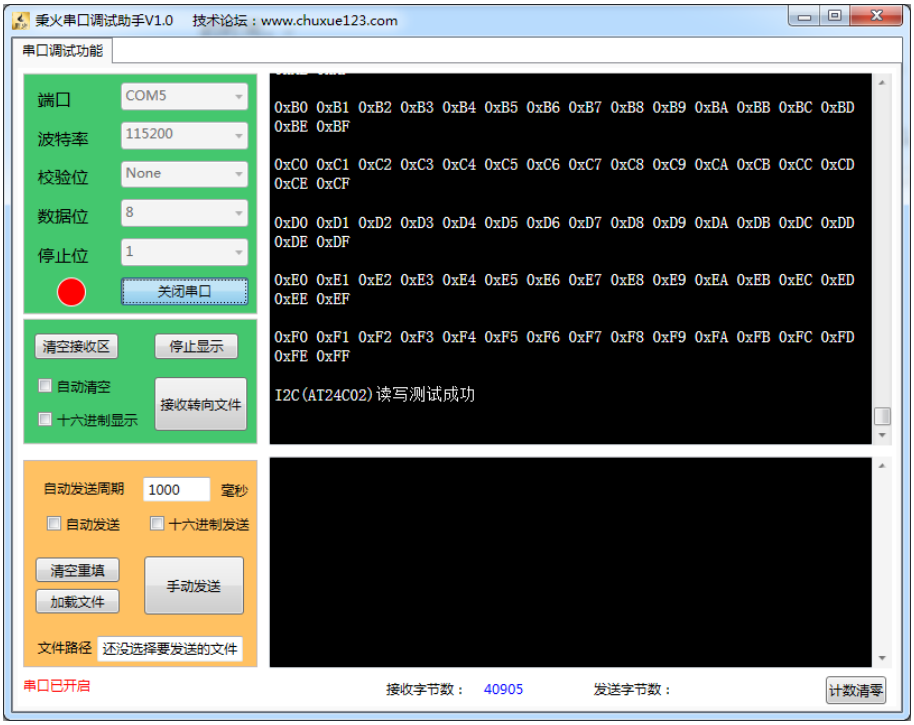


图 1-17 EEPROM 测试成功

1.5 每课一问

1. 在 EEPROM 测试程序中，分别使用单字节写入及页写入函数写入数据，对比它们消耗的时间。
2. 尝试使用 EEPROM 存储 int 整型变量，float 型浮点变量，编写程序写入数据，并读出校验。
3. 尝试把 I2C 通讯引脚的模式改成非开漏模式，测试是否还能正常通讯，为什么？
4. 查看“bsp_i2c_ee.h”文件中 EEPROM_ERROR、EEPROM_INFO、EEPROM_DEBUG 宏，解释为何要使用这样的宏输出调试信息，而不直接使用 printf 函数。