8. STM32的I2C通信

北京科技大学
计算机与通信工程学院

目录

CONTENTS.

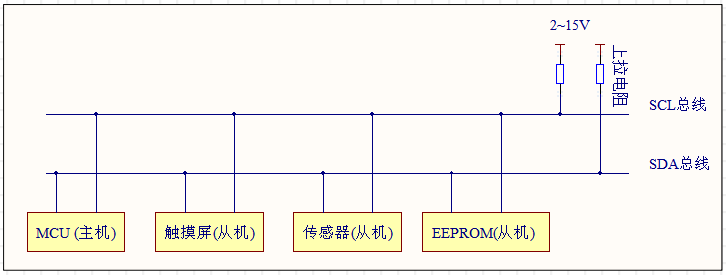
STM32-8

I2C协议简介

I2C协议简介

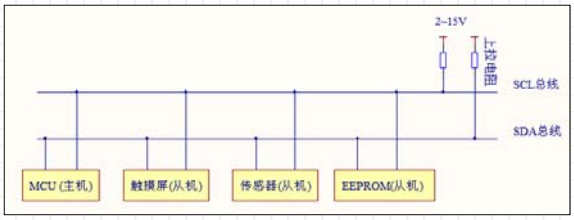
I2C 通讯协议(Inter - Integrated Circuit)是由Phiilps公司开发的，由于它引脚少，硬件实现简单，可扩展性强，不需要USART、CAN等通讯协议的外部收发设备，现在被广泛地使用在系统内多个集成电路(IC)间的通讯。

I2C物理层的特点



I2C协议简介

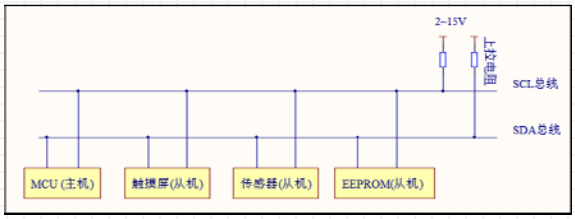
I2C物理层的特点



- 它是一个支持多设备的总线。“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。
- 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。
- 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。

I2C协议简介

I2C物理层的特点



- 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。
- 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。
- 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I²C设备尚不支持高速模式。
- 连接到相同总线的 IC 数量受到总线的最大电容 400pF 限制 。

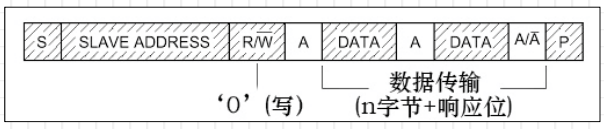
I2C协议简介

I2C的协议层

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

主机写数据到从机：



数据由主机传输至从机

数据由从机传输至主机

S： 传输开始信号

SLAVE_ADDRESS: 从机地址

A/A：应答(ACK)或非应答(NACK)信号

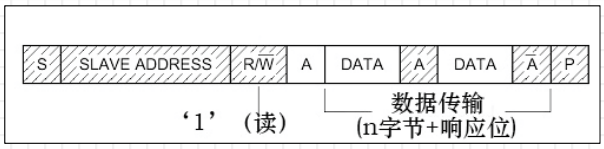
I2C协议简介

I2C的协议层

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

主机由从机中读数据：



数据由主机传输至从机

数据由从机传输至主机

S： 传输开始信号

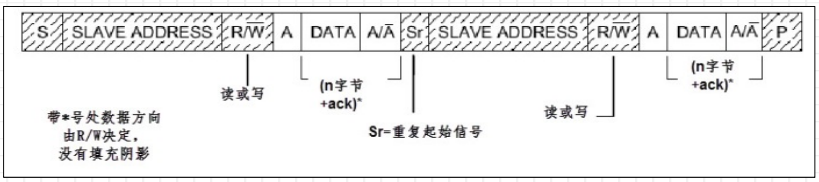
SLAVE_ADDRESS: 从机地址

A/A：应答(ACK)或非应答(NACK)信号

I2C协议简介

1. I2C基本读写过程

通讯复合格式：



数据由主机传输至从机

数据由从机传输至主机

S： 传输开始信号

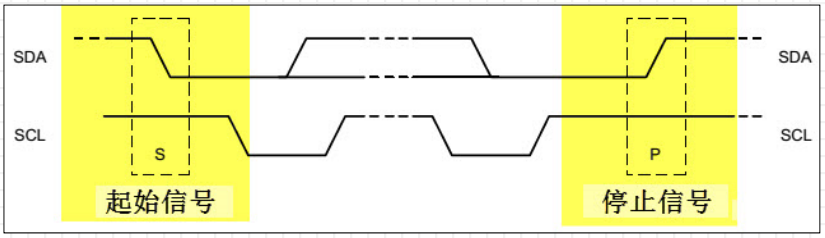
SLAVE_ADDRESS: 从机地址

R/W：传输方向选择位，1为读，0为写

A/A：应答(ACK)或非应答(NACK)信号

I2C协议简介

2.通讯的起始和停止信号

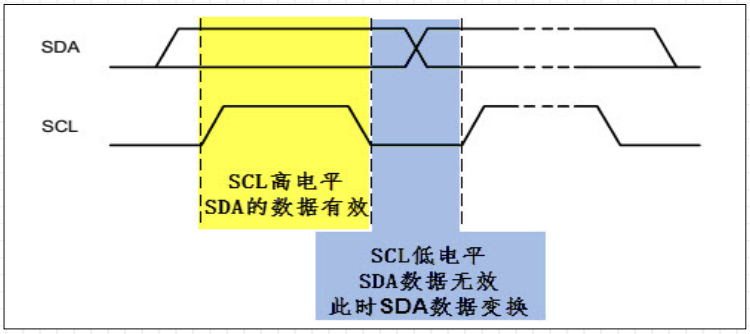


- 当 SCL 线是高电平时 SDA 线从高电平向低电平切换，这个情况表示通讯的起始。
- 当 SCL 是高电平时 SDA 线由低电平向高电平切换，表示通讯的停止。
- 起始和停止信号一般由主机产生。

I2C协议简介

3.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。SDA数据线在SCL的每个时钟周期传输一位数据。



- 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。

I2C协议简介

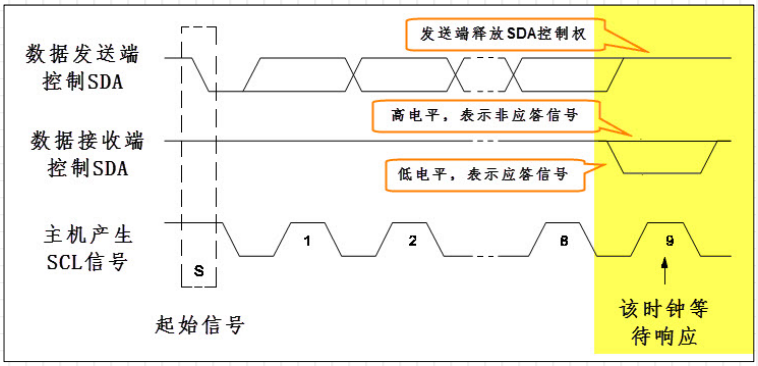
4.地址及数据方向

- 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。设备地址可以是7位或10位。
- 紧跟设备地址的一个数据位R/W用来表示数据传输方向，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。

I2C协议简介

5.响应

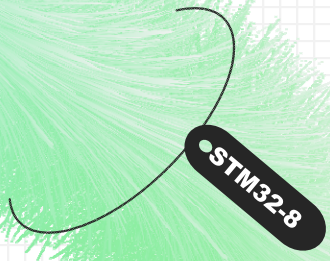
I2C的数据和地址传输都带响应。响应包括“应答(ACK)”和“非应答(NACK)”两种信号。



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)。

目录

CONTENTS.



13

STM32的I2C特性及架构

STM32的I2C特性及架构

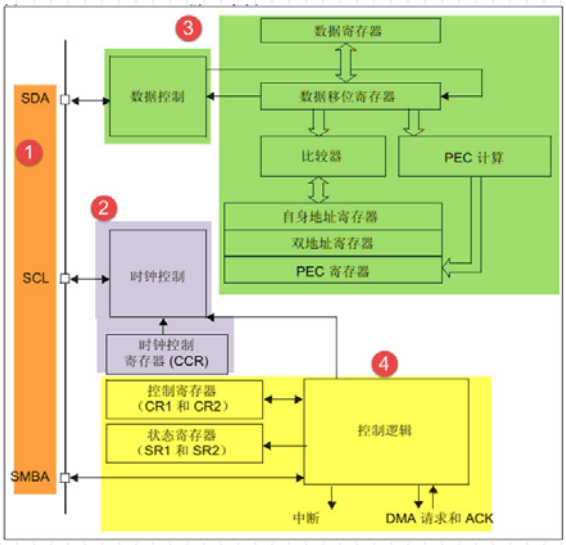
软件模拟协议：使用CPU直接控制通讯引脚的电平，产生出符合通讯协议标准的逻辑。

硬件实现协议：由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。

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。

STM32的I2C特性及架构

STM32的I2C架构剖析



- 通讯引脚
- 时钟控制逻辑
- 数据控制逻辑
- 整体控制逻辑

STM32的I2C特性及架构

1.通讯引脚

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，以《STM32F10x规格书》为准。

引脚	I2C编号	
	I2C1	I2C2
SCL	PB6/PB8(重映射)	PB10
SDA	PB7/PB9(重映射)	PB11

STM32的I2C特性及架构

2.时钟控制逻辑

SCL线的时钟信号，由I²C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。

- 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。
- CCR寄存器中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外设输入时钟源为PCLK1。

STM32的I2C特性及架构

计算时钟频率：

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=36MHz，想要配置400Kbit/s的速率，计算方式如下：

PCLK时钟周期：	TPCLK1 = 1/36000000
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 = 30

计算出来的CCR值写入到寄存器即可。

STM32的I2C特性及架构

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。

STM32的I2C特性及架构

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。

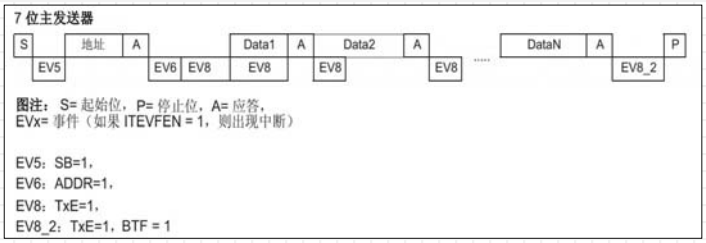
在外设工作时，控制逻辑会根据外设的工作状态修改“状态寄存器(SR1和SR2)”，只要读取这些寄存器相关的寄存器位，就可以了解I2C的工作状态。

STM32的I2C特性及架构

STM32的I2C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，通过读取这些寄存器标志来了解通讯状态。

1.主发送器

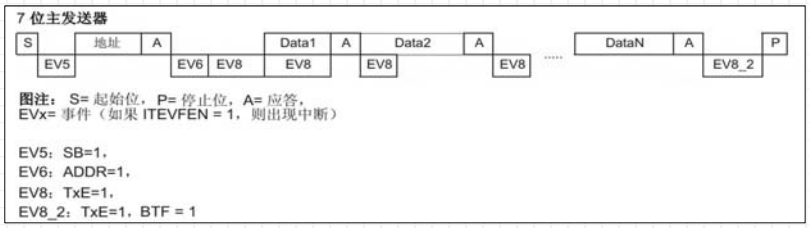


主发送器通讯过程

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。

STM32的I2C特性及架构

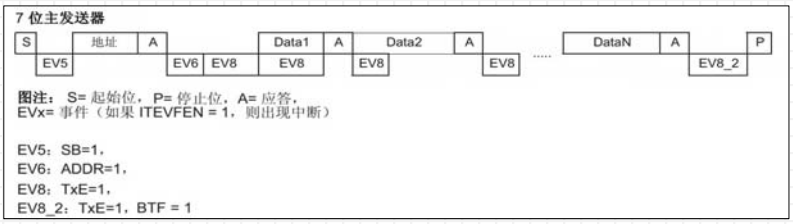
1.主发送器通讯过程



- 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；
- 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；

STM32的I2C特性及架构

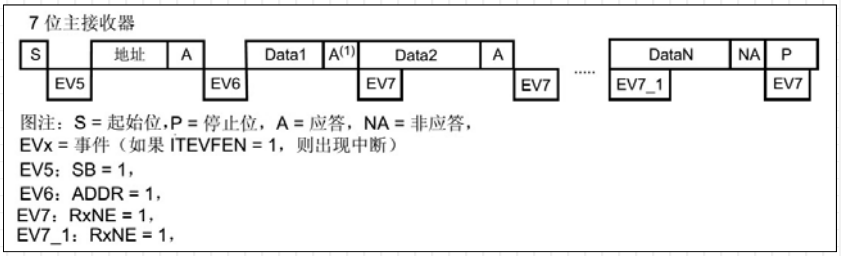
1.主发送器通讯过程



- 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，可以发送多个字节数据；
- 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EV2事件，SR1的TXE位及BTF位都被置1，表示通讯结束。

STM32的I2C特性及架构

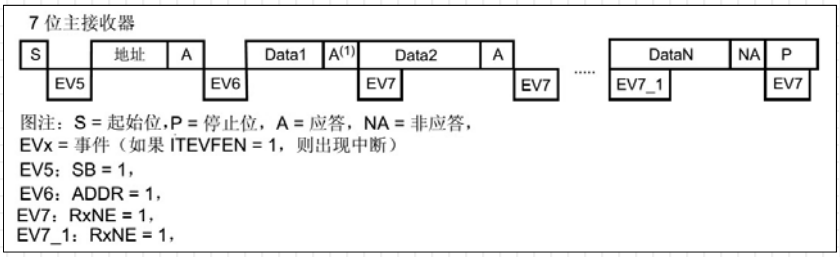
2.主接收器



- 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；
- 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。

STM32的I2C特性及架构

2.主接收器



- 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，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；
- 发送非应答信号后，产生停止信号(P)，结束传输。

目录

CONTENTS.

STM32-8

I2C初始化结构体详解

```
typedef struct {
    uint32_t I2C_ClockSpeed; /*!< 设置SCL时钟频率，此值要低于40 0000*/
    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;
```

- I2C_ClockSpeed

设置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。

I2C初始化结构体详解

```
typedef struct {
    uint32_t I2C_ClockSpeed; /*!< 设置SCL时钟频率，此值要低于40 0000*/
    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;
```

- I2C_Mode

选择I2C的使用方式，有I2C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。

I2C初始化结构体详解

```
typedef struct {
    uint32_t I2C_ClockSpeed; /*!< 设置 SCL 时钟频率，此值要低于 40 0000*/
    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;
```

- I2C_DutyCycle

设置I²C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，这里随便选就可以了。

I2C初始化结构体详解

```
typedef struct {
    uint32_t I2C_ClockSpeed; /*!< 设置 SCL 时钟频率，此值要低于 40 0000*/
    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;
```

- I2C_OwnAddress1

配置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。

I2C初始化结构体详解

```
typedef struct {
    uint32_t I2C_ClockSpeed; /*!< 设置 SCL 时钟频率，此值要低于 40 0000*/
    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;
```

- I2C_Ack_Enable

配置I²C应答是否使能，设置为使能则可以发送响应信号。一般配置为允许应答(I2C_Ack_Enable)，这是绝大多数遵循I²C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。

I2C初始化结构体详解

```
typedef struct {
    uint32_t I2C_ClockSpeed; /*!< 设置 SCL 时钟频率，此值要低于 40 0000*/
    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;
```

- I2C_AcknowledgeAddress

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。

目录

CONTENTS.

STM32-8

33

I2C实例—读写EEPROM

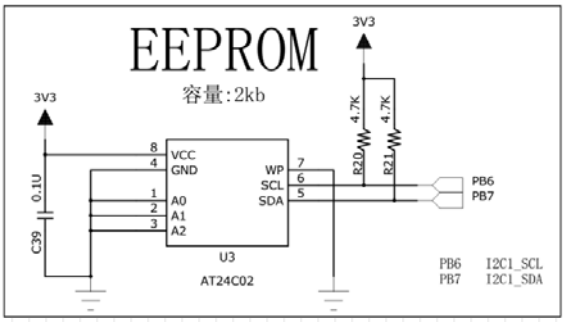
EEPROM 是一种掉电后数据不丢失的存储器，常用来存储一些配置信息，以便系统重新上电的时候加载之。

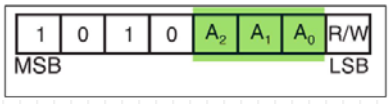
EEPOM 芯片最常用的通讯方式就是I²C 协议，

硬件连接

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。

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，R/W 是读写方向位，与地址无关。





EEPROM设备地址(摘自《AT24C02》规格书)

EEPROM 的7 位设备地址是：101 0000b，即 0x50

34

I2C实例—读写EEPROM

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。

35

(1) 配置通讯使用的目标引脚为开漏模式

I2C 硬件相关宏定义

bsp_i2c_ee.h

```
/**************************I2C 参数定义，I2C1 或 I2C2*********************/
#define       EEPROM_I2Cx                          I2C1
#define       EEPROM_I2C_APBxClock_FUN             RCC_APB1PeriphClockCmd
#define       EEPROM_I2C_CLK                       RCC_APB1Periph_I2C1
#define       EEPROM_I2C_GPIO_APBxClock_FUN        RCC_APB2PeriphClockCmd
#define       EEPROM_I2C_GPIO_CLK                  RCC_APB2Periph_GPIOB
#define             EEPROM_I2C_SCL_PORT                        GPIOB
#define             EEPROM_I2C_SCL_PIN                         GPIO_Pin_6
#define             EEPROM_I2C_SDA_PORT                        GPIOB
#define             EEPROM_I2C_SDA_PIN                         GPIO_Pin_7

/* STM32 I2C 快速模式 */
#define I2C_Speed              400000  //*

/* 这个地址只要与 STM32 外挂的 I2C 器件地址不一样即可 */
#define I2Cx_OWN_ADDRESS7      0X0A

/* AT24C01/02 每页有8 个字节 */
#define I2C_PageSize           8
```

36

(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；

```
1 static void I2C_GPIO_Config(void)
 2 {
 3     GPIO_InitTypeDef  GPIO_InitStructure;
 4 
 5     /* 使能与 I2C 有关的时钟 */
 6     EEPROM_I2C_APBxClock_FUN ( EEPROM_I2C_CLK, ENABLE );             //I2C外设时钟
 7     EEPROM_I2C_GPIO_APBxClock_FUN ( EEPROM_I2C_GPIO_CLK, ENABLE );  //I2C的GPIO时钟
 8 
 9     /* I2C_SCL、I2C_SDA*/
10     GPIO_InitStructure.GPIO_Pin = EEPROM_I2C_SCL_PIN;
11     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
12     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;         // 开漏输出
13     GPIO_Init(EEPROM_I2C_SCL_PORT, &GPIO_InitStructure);
14 
15     GPIO_InitStructure.GPIO_Pin = EEPROM_I2C_SDA_PIN;
16     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
17     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;         // 开漏输出
18     GPIO_Init(EEPROM_I2C_SDA_PORT, &GPIO_InitStructure);
19 }
```

37

(3) 配置 I2C 外设的模式、地址、速率等参数并使能 I2C 外设；

```
 6 static void I2C_Mode_Configu(void)
 7 {
 8     I2C_InitTypeDef  I2C_InitStructure;
 9 
10     /* I2C 配置 */
11     I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
12 
13     /* 高电平数据稳定，低电平数据变化 SCL 时钟线的占空比 */
14     I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
15 
16     I2C_InitStructure.I2C_OwnAddress1 =I2Cx_OWN_ADDRESS7;//这是STM32 IIC自身设备地址，只要是总线上唯一即可
17     I2C_InitStructure.I2C_Ack = I2C_Ack_Enable ; //使能应答
18 
19     /* I2C 的寻址模式 */
20     I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
21 
22     /* 通信速率 */
23     I2C_InitStructure.I2C_ClockSpeed = I2C_Speed; //配置SCL时钟频率
24 
25     /* I2C 初始化 */
26     I2C_Init(EEPROM_I2Cx, &I2C_InitStructure);
27 
28     /* 使能 I2C */
29     I2C_Cmd(EEPROM_I2Cx, ENABLE);
30 }
31 
32 
```

38

(3) 配置 I2C 外设的模式、地址、速率等参数并使能 I2C 外设；

```
    /* I2C 初始化 */
    I2C_Init(EEPROM_I2Cx, &I2C_InitStructure);

    /* 使能 I2C */
    I2C_Cmd(EEPROM_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();

/* 根据头文件 i2c_ee.h 中的定义来选择 EEPROM 要写入的设备地址 */
    /* 选择 EEPROM Block0 来写入 */
    EEPROM_ADDRESS = EEPROM_Block0_ADDRESS;
}
```

39

(4) 编写I2C 按字节收发的基本函数；

见库函数 stm32f10x_i2c.h stm32f10x_i2c.c

```
/**
  * @brief  Sends a data byte through the I2Cx peripheral.
  * @param  I2Cx: where x can be 1 or 2 to select the I2C peripheral.
  * @param  Data: Byte to be transmitted..
  * @retval None
  */
void I2C_SendData(I2C_TypeDef* I2Cx, uint8_t Data)
{
  /* Check the parameters */
  assert_param(IS_I2C_ALL_PERIPH(I2Cx));
  /* Write in the DR register the data to be sent */
  I2Cx->DR = Data;
}
```

```
/**
  * @brief  Transmits the address byte to select the slave device.
  * @param  I2Cx: where x can be 1 or 2 to select the I2C peripheral.
  * @param  Address: specifies the slave address which will be transmitted
  * @param  I2C_Direction: specifies whether the I2C device will be a
  *   Transmitter or a Receiver. This parameter can be one of the following values
  *     @arg I2C_Direction_Transmitter: Transmitter mode
  *     @arg I2C_Direction_Receiver: Receiver mode
  * @retval None.
  */
void I2C_Send7bitAddress(I2C_TypeDef* I2Cx, uint8_t Address, uint8_t I2C_Direction)
```

```
/**
  * @brief  Returns the most recent received data by the I2Cx peripheral.
  * @param  I2Cx: where x can be 1 or 2 to select the I2C peripheral.
  * @retval The value of the received data.
  */
uint8_t I2C_ReceiveData(I2C_TypeDef* I2Cx)
```

40

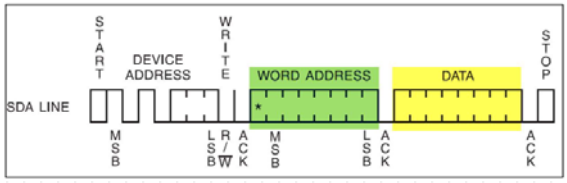
(5) 编写读写EEPROM 存储内容的函数；

I2C通讯过程中，需要检测到某事件后才能继续下一步的操作。
但可能通讯错误或者I2C总线被占用，不能无休止地等待，所以需要设置超时。

```
3  /*通讯等待超时时间*/
4  #define I2CT_FLAG_TIMEOUT   ((uint32_t)0x1000)
5  #define I2CT_LONG_TIMEOUT   ((uint32_t)(10 * I2CT_FLAG_TIMEOUT))
6
7  /**
8  * @brief I2C 等待事件超时的情况下会调用这个函数来处理
9    * @param  errorCode：错误代码，可以用来定位是哪个环节出错.
10 * @retval 返回 0，表示 IIC 读取失败.
11   */
12 static  uint32_t I2C_TIMEOUT_UserCallback(uint8_t errorCode)
13 {
14 /* 使用串口 printf 输出错误信息，方便调试 */
15     EEPROM_ERROR("I2C 等待超时!errorCode = %d",errorCode);
16     return 0;
17 }
```

41

向 EEPROM 写入一个字节的数据



pBuffer:缓冲区指针 ；WriteAddr:写地址

```
24 uint32_t I2C_EE_ByteWrite(u8* pBuffer, u8 WriteAddr)
25 {
26 /* 产生 I2C 起始信号 */
27     I2C_GenerateSTART(EEPROM_I2Cx, ENABLE);
28
29     /*设置超时等待时间*/
30     I2CTimeout = I2CT_FLAG_TIMEOUT;
31    /* 检测 EV5 事件并清除标志*/
32   while(!I2C_CheckEvent(EEPROM_I2Cx,I2C_EVENT_MASTER_MODE_SELECT)){
34       if((I2CTimeout--)==0) return I2C_TIMEOUT_UserCallback(0);
35     }
36
37 /* 发送 EEPROM 设备地址 */
38     I2C_Send7bitAddress(EEPROM_I2Cx, EEPROM_ADDRESS, I2C_Direction_Transmitter);
40
41     I2CTimeout = I2CT_FLAG_TIMEOUT;
42     /* 检测 EV6 事件并清除标志*/
43     while (!I2C_CheckEvent(EEPROM_I2Cx, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED)) {
46       if ((I2CTimeout--)==0) return I2C_TIMEOUT_UserCallback(1);
47     }
48
```

42

向 EEPROM 写入一个字节的数据

```
49 /* 发送要写入的 EEPROM 内部地址(即 EEPROM 内部存储器的地址) */
50     I2C_SendData(EEPROM_I2Cx, WriteAddr);
51
52     I2CTimeout = I2CT_FLAG_TIMEOUT;
53     /* 检测 EV8 事件并清除标志*/
54     while (!I2C_CheckEvent(EEPROM_I2Cx,I2C_EVENT_MASTER_BYTE_TRANSMITTED)) {
57         if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(2);
58     }
59     /* 发送一字节要写入的数据 */
60     I2C_SendData(EEPROM_I2Cx, *pBuffer);
61
62     I2CTimeout = I2CT_FLAG_TIMEOUT;
63     /* 检测 EV8 事件并清除标志*/
64     while (!I2C_CheckEvent(EEPROM_I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED)){
67         if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(3);
68     }
69
70     /* 发送停止信号 */
71     I2C_GenerateSTOP(EEPROM_I2Cx, ENABLE);
72
73     return 1;
74 }
```

43

多字节写入及状态等待

```
1  /*将数据写到I2C EEPROM中，采用单字节写入的方式，速度比页写入慢
4    * @param  pBuffer:缓冲区指针
5    * @param  WriteAddr:写地址
6    * @param  NumByteToWrite:写的字节数
7    * @retval  无
8    */
9  uint8_t I2C_EE_ByetsWrite(uint8_t* pBuffer,
10  uint8_t WriteAddr, uint16_t NumByteToWrite)
11 {
12     uint16_t i;
13     uint8_t res;
14
15 /*每写一个字节调用一次 I2C_EE_ByteWrite 函数*/
16     for (i=0; i<NumByteToWrite; i++)
17     {
18      /*等待 EEPROM 准备完毕*/
19      I2C_EE_WaitEepromStandbyState();
20      /*按字节写入数据*/
21      res = I2C_EE_ByteWrite(pBuffer++,WriteAddr++);
22     }
23     return res;
24 }
```

向EEPROM发送设备地址，检测EEPROM的响应，若EEPROM接收到地址后返回应答信号，则表示EEPROM已准备好，可开始下一次通讯。

```
1  /**
2  * @brief 等待 EEPROM 到准备状态
5   */
6  void I2C_EE_WaitEepromStandbyState(void)
7  {
8      vu16 SR1_Tmp = 0;
9
10     do {
11         /* 发送起始信号 */
12         I2C_GenerateSTART(EEPROM_I2Cx, ENABLE);
13
14         /* 读 I2C1 SR1 寄存器 */
15 SR1_Tmp = I2C_ReadRegister(EEPROM_I2Cx, I2C_Register_SR1);
16
17         /* 发送 EEPROM 地址 + 写方向 */
18         I2C_Send7bitAddress(EEPROM_I2Cx,
19 EEPROM_ADDRESS, 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(EEPROM_I2Cx,
I2C_Register_SR1) & 0x0002));
25     /* 清除 AF 位 */
26     I2C_ClearFlag(EEPROM_I2Cx, I2C_FLAG_AF);
27     /* 发送停止信号 */
28     I2C_GenerateSTOP(EEPROM_I2Cx, ENABLE);
29 }
```

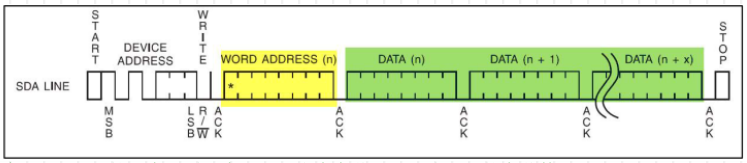
函数中检测响应是通过读取 STM32 的 SR1 寄存器的 ADDR 位及 AF 位来实现的，当 I2C 设备响应了地址的时候，ADDR 会置1，若应答失败，AF 位会置 1。

44

EEPROM的页写入

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。

AT24C02页大小 = 8




```c
/**
* @brief 在 EEPROM 的一个写循环中可以写多个字节，但一次写入的字节数
* 不能超过 EEPROM 页的大小，AT24C02 每页有 8 个字节
  * @param
  * @param    pBuffer:缓冲区指针
  * @param    WriteAddr:写地址
* @param NumByteToWrite:要写的字节数要求 NumByteToWrite 小于页大小
* @retval 正常返回 1，异常返回 0
  */
uint8_t I2C_EE_PageWrite(uint8_t* pBuffer, uint8_t WriteAddr,
                         uint8_t NumByteToWrite)
{
    I2CTimeout = I2CT_LONG_TIMEOUT;

    while (I2C_GetFlagStatus(EEPROM_I2Cx, I2C_FLAG_BUSY))
    {
        if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(4);
    }

/* 产生 I2C 起始信号 */
    I2C_GenerateSTART(EEPROM_I2Cx, ENABLE);
```


EEPROM的页写入

```c
    I2CTimeout = I2CT_FLAG_TIMEOUT;

    /* 检测 EV5 事件并清除标志 */
    while (!I2C_CheckEvent(EEPROM_I2Cx,
I2C_EVENT_MASTER_MODE_SELECT))
    {
        if ((I2CTimeout--) == 0) return
I2C_TIMEOUT_UserCallback(5);
    }

/* 发送 EEPROM 设备地址 */
I2C_Send7bitAddress(EEPROM_I2Cx,
          EEPROM_ADDRESS,I2C_Direction_Transmitter);
    I2CTimeout = I2CT_FLAG_TIMEOUT;

    /* 检测 EV6 事件并清除标志*/
    while (!I2C_CheckEvent(EEPROM_I2Cx,
        I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED))
    {
        if ((I2CTimeout--) == 0) return
I2C_TIMEOUT_UserCallback(6);
    }
/* 发送要写入的 EEPROM 内部地址(即 EEPROM 内部存储器的地址) */
    I2C_SendData(EEPROM_I2Cx, WriteAddr);

    I2CTimeout = I2CT_FLAG_TIMEOUT;

    /* 检测 EV8 事件并清除标志*/
  while (! I2C_CheckEvent(EEPROM_I2Cx,
I2C_EVENT_MASTER_BYTE_TRANSMITTED)) 50     {
        if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(7);
    }
/* 循环发送 NumByteToWrite 个数据 */
    while (NumByteToWrite--)
    {
        /* 发送缓冲区中的数据 */
        I2C_SendData(EEPROM_I2Cx, *pBuffer);

        /* 指向缓冲区中的下一个数据 */
        pBuffer++;

        I2CTimeout = I2CT_FLAG_TIMEOUT;

        /* 检测 EV8 事件并清除标志*/
while(!I2C_CheckEvent(EEPROM_I2Cx,I2C_EVENT_MASTER_BYTE_TRANSMITTED))
        {
         if((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(8);
        }
    }
    /* 发送停止信号 */
    I2C_GenerateSTOP(EEPROM_I2Cx, ENABLE);
    return 1;
}
```

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

利用EEPROM的页写入方式，实现快速写入多字节

```c
// AT24C01/02 每页有 8 个字节
#define I2C_PageSize        8
/**
 * @brief 将缓冲区中的数据写到 I2C EEPROM 中
 * @param
 *    @arg pBuffer:缓冲区指针
 *    @arg WriteAddr:写地址
 *      @arg NumByteToWrite:写的字节数
 * @retval  无
 */
void I2C_EE_BufferWrite(u8* pBuffer, u8 WriteAddr,
                        u16 NumByteToWrite)
{
    u8 NumOfPage=0,NumOfSingle=0,
                    Addr =0,count=0,temp =0;
/*mod 运算求余，若 writeAddr 是 I2C_PageSize 整数倍，
运算结果 Addr 值为 0*/
    Addr = WriteAddr % I2C_PageSize;

/*差 count 个数据值，刚好可以对齐到页地址*/
    count = I2C_PageSize - Addr;

    /*计算出要写多少整数页*/
    NumOfPage =  NumByteToWrite / I2C_PageSize;

/*mod 运算求余，计算出剩余不满一页的字节数*/
    NumOfSingle = NumByteToWrite % I2C_PageSize;
```

表 24-2 首地址对齐到页时的情况

不影响	0	1	2	3	4	5	6	7
不影响	8	9	10	11	12	13	14	15
第 1 页	16	17	18	19	20	21	22	23
第 2 页	24	25	26	27	28	29	30	31
NumOfSingle=6	32	33	34	35	36	37	38	39

```c
// Addr=0,则 WriteAddr 刚好按页对齐 aligned
    // 这样就很简单了，直接写就可以，写完整页后
    // 把剩下的不满一页的写完即可
    if (Addr == 0) {
        /* 如果 NumByteToWrite < I2C_PageSize */
     if (NumOfPage == 0) {
      I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
            I2C_EE_WaitEepromStandbyState();
        }
        /* 如果 NumByteToWrite > I2C_PageSize */
      else {
        /*先把整数页都写了*/
      while (NumOfPage--) {
      I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
                I2C_EE_WaitEepromStandbyState();
                WriteAddr +=  I2C_PageSize;
                pBuffer += I2C_PageSize;
          }
         /*若有多余的不满一页的数据，把它写完*/
      if (NumOfSingle!=0) {
       I2C_EE_PageWrite(pBuffer, WriteAddr,NumOfSingle);
                I2C_EE_WaitEepromStandbyState();
            }
        }
    } // end of if(Addr==0)
```


利用EEPROM的页写入方式，实现快速写入多字节

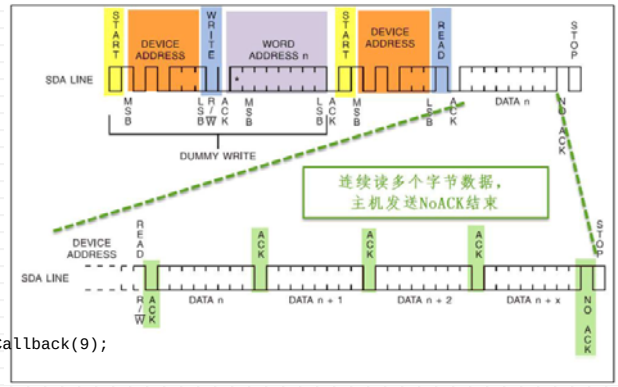
表 24-3 首地址未对齐到页时的情况

不影响	0	1	2	3	4	5	6	7
不影响	8	9	10	11	12	13	14	15
count=7	16	17	18	19	20	21	22	23
第 1 页	24	25	26	27	28	29	30	31
NumOfSingle=7	32	33	34	35	36	37	38	39

```c
    // 如果 WriteAddr不是按 I2C_PageSize 对齐
    // 那就算出对齐到页地址还需要多少个数据，然后
    // 先把这几个数据写完，剩下开始的地址就已经对齐
    // 到页地址了，代码重复上面的即可
    else {
     /* 如果 NumByteToWrite < I2C_PageSize */
     if (NumOfPage== 0) {
     /*若 NumOfSingle>count，当前面写不完，要写到下一页*/
     if (NumOfSingle > count) {
     // temp 的数据要写到写一页
            temp = NumOfSingle - count;

         I2C_EE_PageWrite(pBuffer, WriteAddr, count);
         I2C_EE_WaitEepromStandbyState();
          WriteAddr +=  count;
          pBuffer += count;

          I2C_EE_PageWrite(pBuffer, WriteAddr, temp);
                I2C_EE_WaitEepromStandbyState();
} else { /*若 count 比 NumOfSingle 大*/
 I2C_EE_PageWrite(pBuffer, WriteAddr,NumByteToWrite);
                I2C_EE_WaitEepromStandbyState();
            }
        }
        /* 如果 NumByteToWrite > I2C_PageSize */
        else {
/*地址不对齐多出的 count 分开处理，不加入这个运算*/
        NumByteToWrite -= count;
        NumOfPage =  NumByteToWrite / I2C_PageSize;
        NumOfSingle = NumByteToWrite % I2C_PageSize;

/*先把 WriteAddr 所在页的剩余字节写了*/
       if (count != 0) {
         I2C_EE_PageWrite(pBuffer, WriteAddr, count);
         I2C_EE_WaitEepromStandbyState();

/*WriteAddr 加上 count 后，地址就对齐到页了*/
                WriteAddr += count;
                pBuffer += count;
          }
   /*把整数页都写了*/
 while (NumOfPage--) {
  I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
   I2C_EE_WaitEepromStandbyState();
    WriteAddr +=  I2C_PageSize;
     pBuffer += I2C_PageSize;
 }
  /*若有多余的不满一页的数据，把它写完*/
 if (NumOfSingle != 0) {
  I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
                I2C_EE_WaitEepromStandbyState();
            }
        }
    }
}
```

从EEPROM 读取数据

从 EEPROM 读取数据是一个复合的I2C 时序:
包含一个写过程和一个读过程




```
1
 2 /**
 3 * @brief 从 EEPROM 里面读取一块数据
 4 * @param pBuffer:存放从 EEPROM 读取的数据的缓冲区指针
 5 * @param ReadAddr:接收数据的 EEPROM 的地址
 6 * @param NumByteToRead:要从 EEPROM 读取的字节数
 7 * @retval 正常返回 1，异常返回 0
 8   */
 9 uint8_t I2C_EE_BufferRead(uint8_t* pBuffer,
          uint8_t ReadAddr, u16 NumByteToRead)
11 {
12     I2CTimeout = I2CT_LONG_TIMEOUT;
13
14     while(I2C_GetFlagStatus(EEPROM_I2Cx, I2C_FLAG_BUSY))
15     {
16       if((I2CTimeout--)==0) return I2C_TIMEOUT_UserCallback(9);
17     }
18
19 /* 产生 I2C 起始信号 */
20     I2C_GenerateSTART(EEPROM_I2Cx, ENABLE);
21
22     I2CTimeout = I2CT_FLAG_TIMEOUT;
23
```



```
24     /* 检测 EV5 事件并清除标志*/
25     while (!I2C_CheckEvent(EEPROM_I2Cx, I2C_EVENT_MASTER_MODE_SELECT))
26     {
27         if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(10);
28     }
29
30 /* 发送 EEPROM 设备地址 */
31 I2C_Send7bitAddress(EEPROM_I2Cx,EEPROM_ADDRESS,I2C_Direction_Transmitter); 32
33     I2CTimeout = I2CT_FLAG_TIMEOUT;
34
35     /* 检测 EV6 事件并清除标志*/
36     while (!I2C_CheckEvent(EEPROM_I2Cx,
37                            I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED))
38     {
39         if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(11);
40     }
41 /*通过重新设置 PE 位清除 EV6 事件 */
42     I2C_Cmd(EEPROM_I2Cx, ENABLE);
43
44 /* 发送要读取的 EEPROM 内部地址(即 EEPROM 内部存储器的地址) */
45     I2C_SendData(EEPROM_I2Cx, ReadAddr);
46
47     I2CTimeout = I2CT_FLAG_TIMEOUT;
48
49     /* 检测 EV8 事件并清除标志*/
50     while (!I2C_CheckEvent(EEPROM_I2Cx,I2C_EVENT_MASTER_BYTE_TRANSMITTED)) 51
{
52         if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(12);
53     }
```

```
/* 产生第二次 I2C 起始信号 */
    I2C_GenerateSTART(EEPROM_I2Cx, ENABLE);

    I2CTimeout = I2CT_FLAG_TIMEOUT;

    /* 检测 EV5 事件并清除标志*/
    while (!I2C_CheckEvent(EEPROM_I2Cx, I2C_EVENT_MASTER_MODE_SELECT))
    {
        if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(13);
    }
/* 发送 EEPROM 设备地址 */
 I2C_Send7bitAddress(EEPROM_I2Cx, EEPROM_ADDRESS,
I2C_Direction_Receiver); 66
    I2CTimeout = I2CT_FLAG_TIMEOUT;

    /* 检测 EV6 事件并清除标志*/
    while (!I2C_CheckEvent(EEPROM_I2Cx,
                           I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED)){
        if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(14);
    }
/* 读取 NumByteToRead 个数据*/
    while (NumByteToRead){
/*若 NumByteToRead=1，表示已经接收到最后一个数据了，
        发送非应答信号，结束传输*/
        if (NumByteToRead == 1){
            /* 发送非应答信号 */
            I2C_AcknowledgeConfig(EEPROM_I2Cx, DISABLE);
            /* 发送停止信号 */
            I2C_GenerateSTOP(EEPROM_I2Cx, ENABLE);
        }
        I2CTimeout = I2CT_LONG_TIMEOUT;
```

51

```
    while (I2C_CheckEvent(EEPROM_I2Cx, I2C_EVENT_MASTER_BYTE_RECEIVED)==0)
        {
            if ((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback(3);
        }
        {
/*通过 I2C，从设备中读取一个字节的数据 */
            *pBuffer = I2C_ReceiveData(EEPROM_I2Cx);

            /* 存储数据的指针指向下一个地址 */
            pBuffer++;

             /* 接收数据自减 */
             NumByteToRead--;
        }
    }

/* 使能应答，方便下一次 I2C 传输 */
    I2C_AcknowledgeConfig(EEPROM_I2Cx, ENABLE);
    return 1;
}
```

响应信号通过库函数 I2C_AcknowledgeConfig 来发送，
DISABLE 时为非响应信号，ENABLE 为响应信号。

52

EEPROM读写测试

```c
/**
  * @brief  I2C(AT24C02)读写测试
  * @param  无
* @retval 正常返回 1 , 不正常返回 0
  */
uint8_t I2C_Test(void)
{
    u16 i;
    EEPROM_INFO("写入的数据");

    for ( i=0; i<=255; i++ ) //填充缓冲
    {
        I2c_Buf_Write[i] = i;

        printf("0x%02X ", I2c_Buf_Write[i]);
        if (i%16 == 15)
            printf("\n\r");
    }

//将 I2c_Buf_Write 中顺序递增的数据写入 EERPOM 中
    //页写入方式
//  I2C_EE_BufferWrite( I2c_Buf_Write, EEP_Firstpage, 256);
    //字节写入方式
    I2C_EE_ByetsWrite( I2c_Buf_Write, EEP_Firstpage, 256);

    EEPROM_INFO("写结束");

    EEPROM_INFO("读出的数据");

//将 EEPROM 读出数据顺序保持到 I2c_Buf_Read 中
   I2C_EE_BufferRead(I2c_Buf_Read, EEP_Firstpage, 256);

//将 I2c_Buf_Read 中的数据通过串口打印
    for (i=0; i<256; i++)
    {
        if (I2c_Buf_Read[i] != I2c_Buf_Write[i])
        {
            printf("0x%02X ", I2c_Buf_Read[i]);
EEPROM_ERROR("错误:I2C EEPROM 写入与读出的数据不一致");
            return 0;
        }
        printf("0x%02X ", I2c_Buf_Read[i]);
        if (i%16 == 15)
            printf("\n\r");

    }
    EEPROM_INFO("I2C(AT24C02)读写测试成功");
    return 1;
}
```

EEPROM_INFO， EEPROM_ERROR 宏定义，都是对 printf 函数的封装

EEPROM读写测试

```c

/**
  * @brief  主函数
  * @param  无
  * @retval 无
  */
int main(void)
{
    LED_GPIO_Config();

    LED_BLUE;
    /*初始化 USART1*/
    Debug_USART_Config();
    printf("\r\n 这是一个 I2C 外设
                (AT24C02)读写测试例程 \r\n");
    /* I2C 外设(AT24C02)初始化 */
    I2C_EE_Init();

    if (I2C_Test() ==1)  {
        LED_GREEN;
    }
    else   {
        LED_RED;
    }

    while (1)  {
    }
 }
```