AN0091
入门指南

AT32F435/437 I²C使用指南

前言

AT32 的 I²C 总线接口用于处理微控制器和串行 I²C 总线之间的通信，支持主机和从机模式，最大通信速度为 1Mbit/s(增强快速模式 fast mode plus)。本文主要就 I²C 总线接口的基本功能进行讲解和案列解析。

支持型号列表：

支持型号	AT32F435 系列
	AT32F437 系列

目录

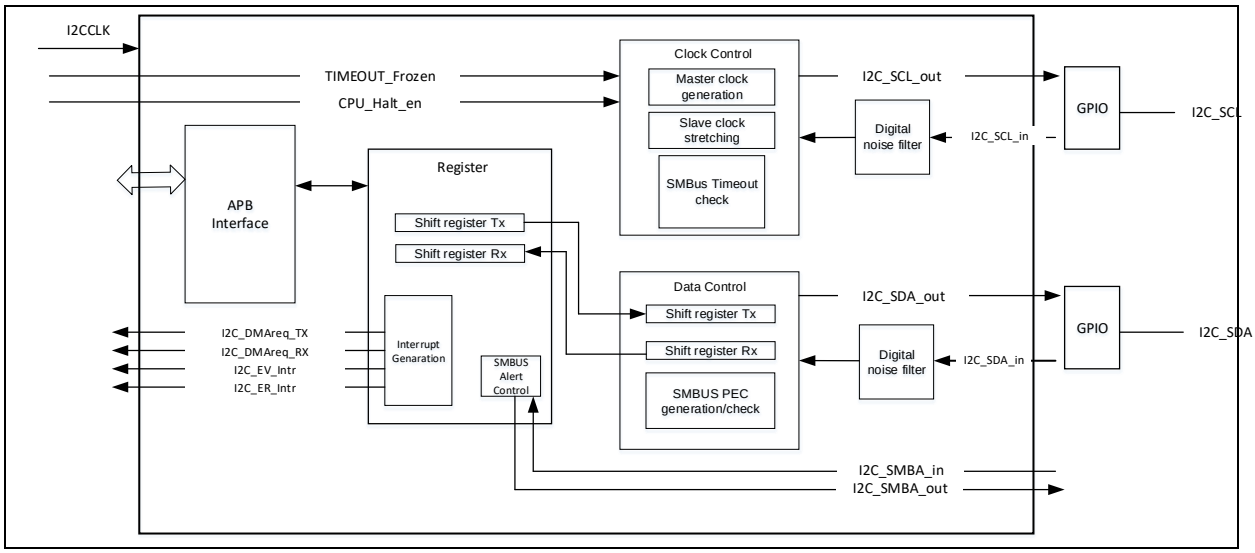
表目录

图目录

1 I²C 接口简介

I²C 接口是由数据线 SDA 和时钟线 SCL 构成，在标准模式下通信速度可达到 100kHz，快速模式下则可以达到 400kHz，增强快速模式可达到 1MHz。一帧数据传输从开始信号开始，在结束信号后停止，在收到开始信号后总线被认为是繁忙的，当收到结束信号后，总线被认为再次空闲。I²C 接口具有主机和从机模式、多主机功能、可编程建立和保持时间、时钟延展功能、DMA 存取数据、支持 SMBus 2.0 协议等特点。

图 1. I²C 框图



2 I²C 接口通信

2.1 主机通信流程

2.1.1 主机通信初始化

1. 主机时钟初始化

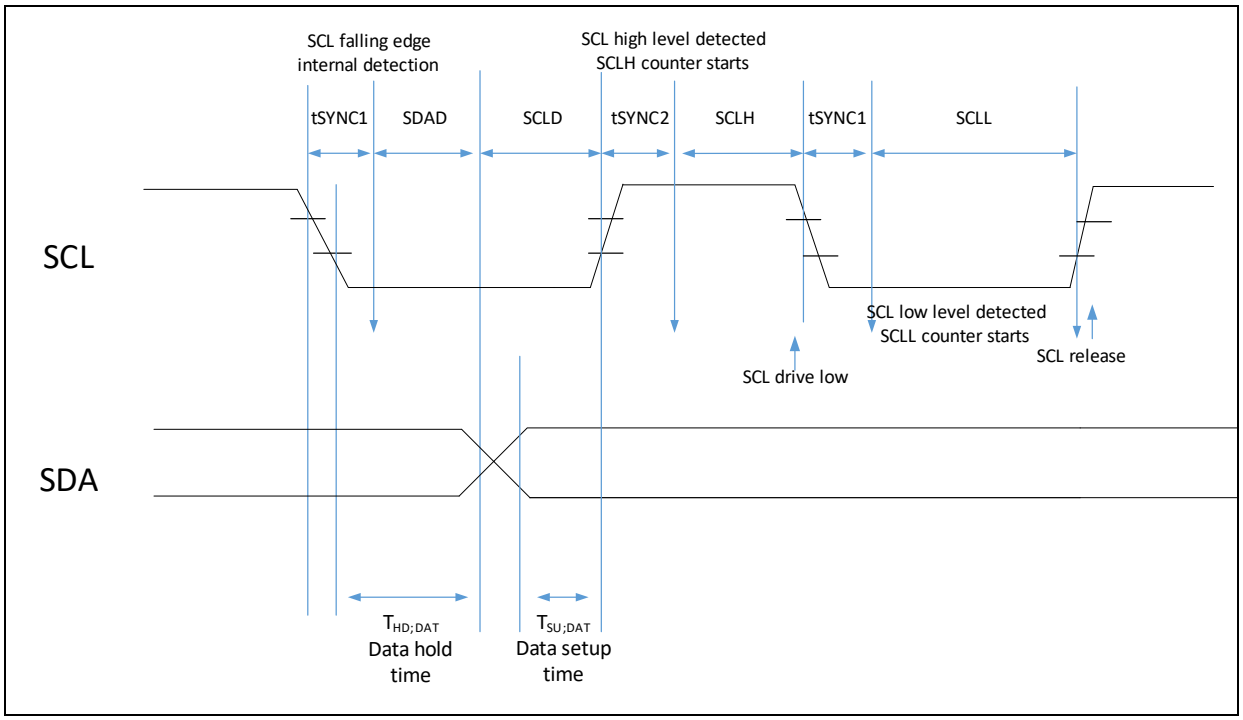
在启动外设(I2CEN)之前，必须先设置 I2Cx_CLKCTRL 寄存器的各个位用以配置 I²C 主时钟。

- DIV[7：0]：I²C 时钟分频；
- SDAD[3：0]：数据保持时间（$t_{HD;DAT}$）
- SCLD[3：0]：数据建立时间（$t_{SU;DAT}$）
- SCLH[7：0]：SCL 高电平时间
- SCLL[7：0]：SCL 低电平时间

该寄存器的配置可以使用 Artery_I2C_Timing_Configuration 时钟配置工具计算，见第三章节。
低电平控制：当检测到 SCL 总线为低电平时，内部 SCLL 计数器开始计数，当计数值达到 SCLL 值时，释放 SCL 线，SCL 线变为高电平。

高电平控制：当检测到 SCL 总线为高电平时，内部 SCLH 计数器开始计数，当计数值达到 SCLH 值时，拉低 SCL 线，SCL 线变为低电平，当在高电平期间，如果被外部总线拉低，那么内部 SCLH 计数器停止计数，并开始低电平计数，这为时钟同步提供了条件。

图 2. 主机时钟的产生



2. 主机通信初始化

在启动通讯前须先设定 I2C_CTRL2 寄存器中的几项参数：

1) 设置传输字节数

- ≤255 字节

  配置 I2C_CTRL2 的 RLDEN=0，关闭重载模式

  配置 I2C_CTRL2 的 CNT[7:0]=N

— ＞255 字节

配置 I2C_CTRL2 的 RLDEN=1，使能重载模式

配置 I2C_CTRL2 的 CNT[7:0]=255

剩余传输字节数 N=N-255

2) 设置传输结束模式

— ASTOPEN=0：软件结束模式，当数据传输完成后，I2C_STS 的 TDC 标志置 1，软件设置 GENSTOP=1 或者 GENSTART=1，发送 STOP 条件或者 START 条件。

— ASTOPEN=1：自动结束模式，当数据传输完成后，自动发送 STOP 条件。

3) 设置从机地址

— 设置寻址的从机地址值（I2C_CTRL2 的 SADDR）

— 设置从机地址模式（I2C_CTRL2 的 ADDR10）

ADDR10=0：7 位地址模式

ADDR10=1：10 位地址模式

4) 设置传输方向（I2C_CTRL2 的 DIR）

— DIR=0：主机接收数据

— DIR=1：主机发送数据

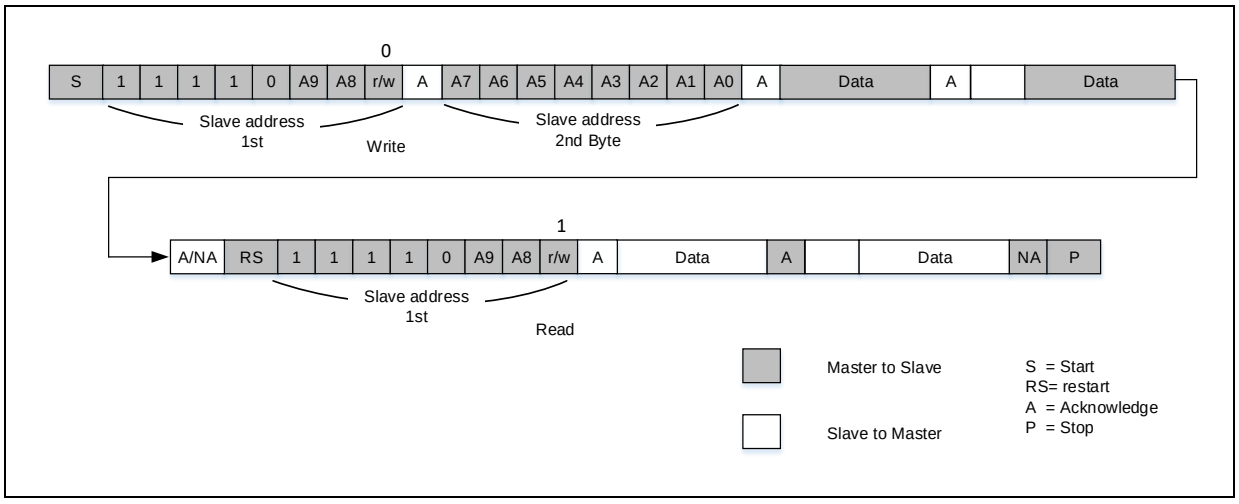
5) 开始传输

设置 I2C_CTRL2 的 GENSTART=1，主机开始在总线上发送 START 条件和从机地址。

3. 主机 10 bits 寻址的特殊时序初始化

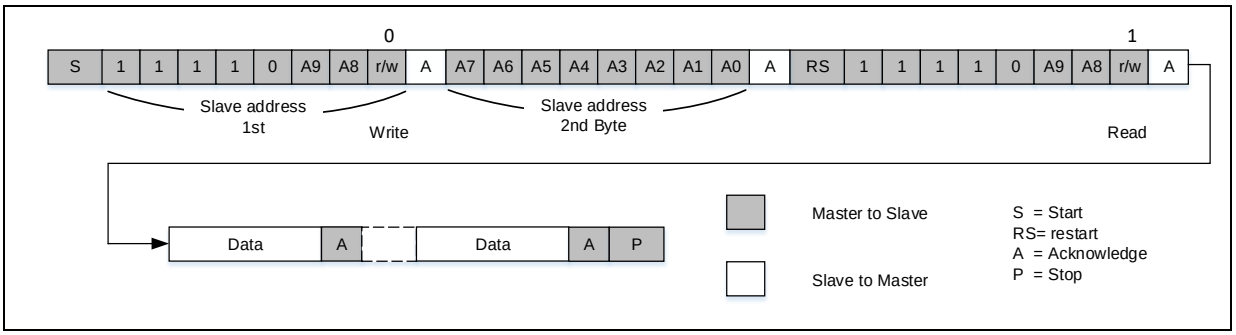
在 10 位地址传输模式下，I2C_CTRL2 的 READH10 用于产生特殊时序，当 READH10=1 时，支持如下传输序：主机先发送数据给从机，然后再从从机读取数据，传输时序图如下图所示：

图 3. 10 位地址的读访问 READH10 = 1



主机在软件结束模式（ASTOPEN = 0）下，发送数据到从机，当数据发送完成后设置 READH10=1，然后再从从机接收数据。

图 4. 10 位地址的读访问 READH10 = 0



2.1.2 主机通信初始化软件接口

主机通信初始化所用到的软件接口通过独立的函数接口实现，如下：

```
void i2c_init(i2c_type *i2c_x, uint8_t dfilters, uint32_t clk); /* 主机时钟初始化 */
void i2c_transmit_set(i2c_type *i2c_x, uint16_t address, uint8_t cnt, i2c_reload_stop_mode_type rld_stop,
i2c_start_stop_mode_type start_stop); /* 主机通信初始化 */
void i2c_addr10_mode_enable(i2c_type *i2c_x, confirm_state new_state); /* 10 位地址使能 */
void i2c_addr10_header_enable(i2c_type *i2c_x, confirm_state new_state); /* 10 位地址头读取时序使能 */
```

i2c_init 函数三个参数分别为：所使用的 I2C、数字滤波值和主机时钟配置值。

i2c_transmit_set 函数用于初始化通信参数，包括：所使用的 I2C、从机地址、传输字节数、停止条件产生模式和起始条件产生模式。

i2c_addr10_mode_enable 函数用于使能 10 位地址模式。

i2c_addr10_header_enable 函数用于使能 10 位地址头读取时序，即主机发送完整的 10 位从机地址读序列或主机只发送 10 位地址的前 7 位。

2.1.3 主机发送流程

1) I2C_TXDT 数据寄存器为空，I2C_STS 的 TDIS=1；
2) 向 TXDT 数据寄存器写入数据，数据开始发送；
3) 重复 1、2 步骤直到发送 CNT[7:0]个数据；
4) 如果此时 I2C_STS 的 TCRLD=1（重载模式），分为以下两种情况：
   - 剩余字节数 N>255：向 CNT 写入 255，N=N-255，TCRLD 被自动清 0，传输继续；
   - 剩余字节数 N≤255：关闭重载模式（RLDEN=0），向 CNT 写入 N，TCRLD 被自动清 0，传输继续。
5) 结束时序
   - 停止条件产生：
     软件结束模式（ASTOPEN=0）：此时 I2C_STS 的 TDC 置 1，设置 GENSTOP=1 产生 STOP 条件；
     自动结束模式（ASTOPEN=1）：自动产生 STOP 条件。
   - 等待产生 STOP 条件，当 STOP 条件产生时，I2C_STS 的 STOPF 置 1，将 I2C_CLR 的 STOPC 写 1，清除 STOPF 标志，传输结束。

图 5. I²C 主机发送流程图

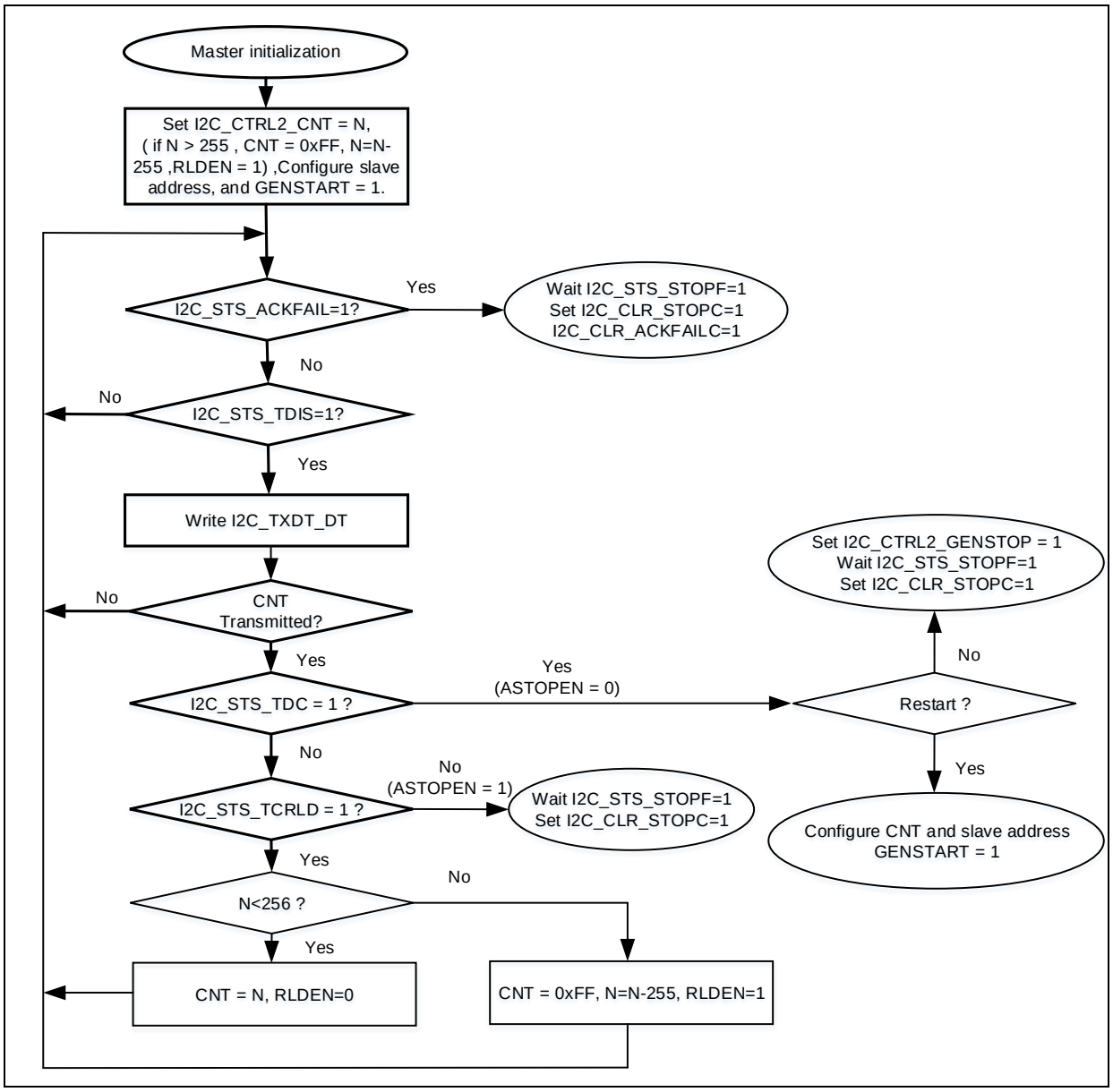
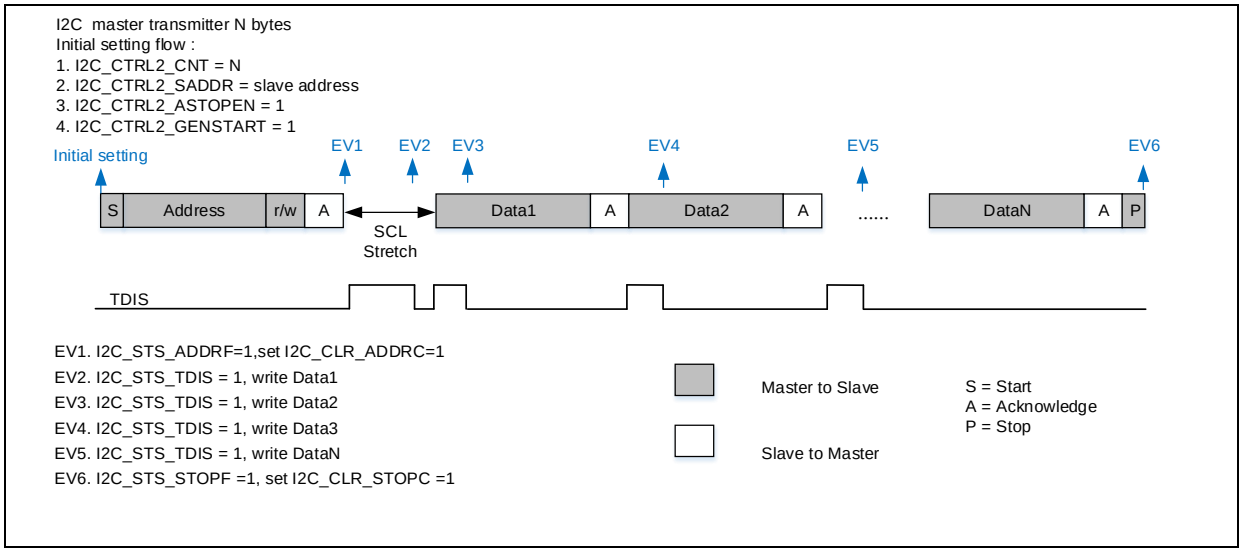


图 6. I²C 主机发送时序图



2.1.4 主机发送流程软件接口

主机发送通过独立的函数接口实现，如下：

```
i2c_status_type i2c_master_transmit(i2c_handle_type* hi2c, uint16_t address, uint8_t* pdata, uint16_t size,
uint32_t timeout);
```

i2c_master_transmit 函数为 i2c_application.c 文件所提供的应用层接口函数，参数包括：I2C 结构体指针、从机地址、发送数据指针、发送数据字节数和函数超时时间。

注：此函数为Artery 所提供的标准主机发送函数。用户也可根据前述主机发送流程，自行编写主机发送函数。

2.1.5 主机接收流程

1) 当收到数据后，RDBF=1，读取 RXDT 数据寄存器，RDBF 被自动清零；
2) 重复步骤 2 直到接收 CNT[7:0]个数据；
3) 如果此时 I2C_STS 的 TCRLD=1（重载模式），分为以下两种情况：
   - 剩余字节数 N>255：向 CNT 写入 255，N=N-255，TCRLD 被自动清 0，传输继续；
   - 剩余字节数 N≤255：关闭重载模式（RLDEN=0），向 CNT 写入 N，TCRLD 被自动清 0，传输继续。
4) 当在接收到最后一个字节时，主机会自动发送一个 NACK。
5) 结束时序
   - 停止条件产生：

     软件结束模式（ASTOPEN=0）：此时 I2C_STS 的 TDC 置 1，设置 GENSTOP=1 产生 STOP 条件；

     自动结束模式（ASTOPEN=1）：自动产生 STOP 条件。
   - 等待产生 STOP 条件，当 STOP 条件产生时，I2C_STS 的 STOPF 置 1，将 I2C_CLR 的 STOPC 写 1，清除 STOPF 标志，传输结束。

图 7. I²C 主机接收流程图

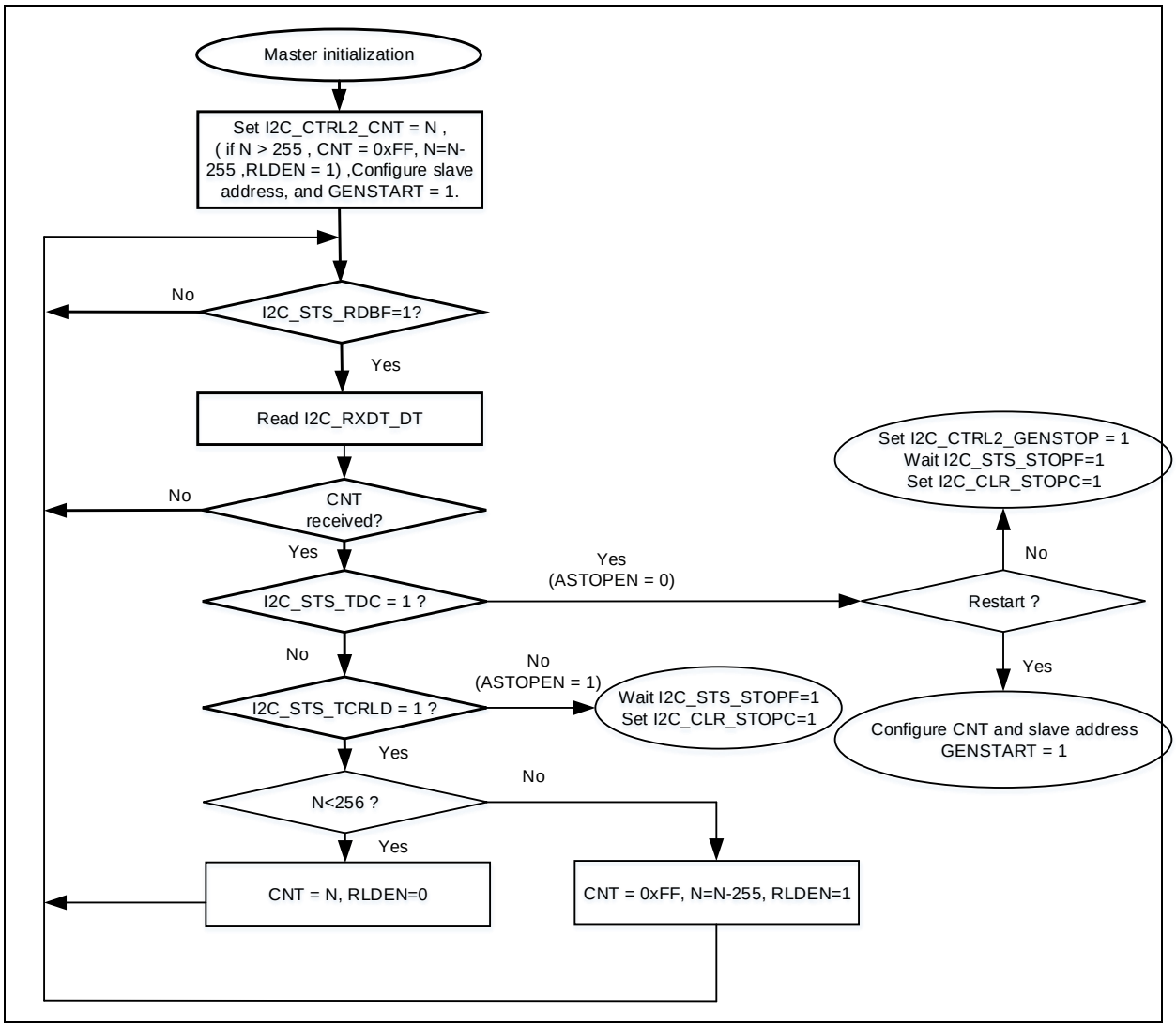
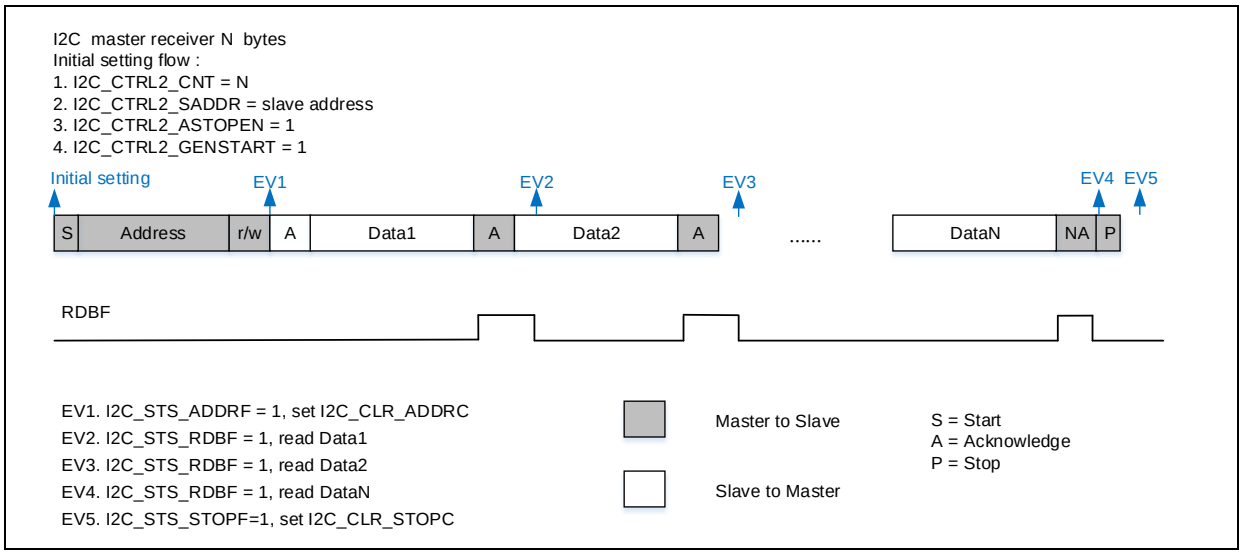


图 8. I²C 主机接收时序图



2.1.6 主机接收流程软件接口

主机接收通过独立的函数接口实现，如下：

```
i2c_status_type i2c_master_receive(i2c_handle_type* hi2c, uint16_t address, uint8_t* pdata, uint16_t size,
uint32_t timeout);
```

i2c_master_receive 函数为 i2c_application.c 文件所提供的应用层接口函数，参数包括：I2C 结构体指针、从机地址、接收数据指针、接收数据字节数和函数超时时间。

注：此函数为Artery 所提供的标准主机接收函数。用户也可根据前述主机接收流程，自行编写主机接收函数。

2.2 从机通信流程

2.2.1 从机通信初始化

1. 从机地址配置

每个 I²C 从设备可同时支持 2 个从设备地址，由 OADDR1 和 OADDR2 指定

I2C_OADDR1

- 通过 ADDR1EN 使能
- 通过 ADDR1MODE 配置为 7 位（默认）或 10 位地址

I2C_OADDR2

- 通过 ADDR2EN 使能
- 固定 7 位地址模式
- 可通过 ADDR2MASK [2:0]来在进行地址匹配比较时屏蔽掉 0~7 个 LSB 地址位
  ADDR2MASK = 0 表示 7 位地址中的每一位都要参与匹配比较
  ADDR2MASK = 7 表示任何非保留地址的 7 位地址都会被该从设备应答

2. 从机地址匹配

当 I²C 启用的地址选中匹配时，ADDRF 中断状态标志会被置 1，如果 ADDRIEN 位为 1，就会产生一个中断。如果两个从地址都使能，在地址匹配产生 ADDR 中断时，可以查看状态寄存器中的 ADDR [6:0]来得知是 OADDR1 还是 OADDR2 被寻址了。

3. 从机字节控制模式（通常 SMBus 模式下才使用）

从设备可以对每个收到的字节进行应答控制。

所需配置：SCTRL = 1 & RLDEN =1 & STRETCH = 0 & CNT ≥ 1

从机字节控制流程：

1) 每收到一个字节 TCRLD 置位，时钟延展于第 8 和第 9 个脉冲之间
2) 软件读取 RXDT 中的值，并决定是否置位 ACK
3) 软件重装载 CNT = 1 来停止时钟延展
4) 应答或非应答信号在第 9 个脉冲时刻出现在总线上

注意：

置位 SCTRL 时，必须开启时钟延展，即 STRETCH = 0

CNT 可以是大于 1 的值，来实现多个字节以自动 ACK 接收完毕后再启动应答控制，从设备发送时推荐关闭 SCTRL，此时无需字节应答控制。

2.2.2 从机通信初始化软件接口

从机通信初始化所用到的软件接口通过独立的函数接口实现，如下：

```
void i2c_own_address1_set(i2c_type *i2c_x, i2c_address_mode_type mode, uint16_t address);
void i2c_own_address2_set(i2c_type *i2c_x, uint8_t address, i2c_addr2_mask_type mask);
void i2c_own_address2_enable(i2c_type *i2c_x, confirm_state new_state);
void i2c_slave_data_ctrl_enable(i2c_type *i2c_x, confirm_state new_state);
void i2c_clock_stretch_enable(i2c_type *i2c_x, confirm_state new_state);
void i2c_reload_enable(i2c_type *i2c_x, confirm_state new_state);
```

i2c_own_address1_set 函数用于配置 OADDR1 地址模式以及 ADDR1 地址值。

i2c_own_address2_set 函数用于配置 ADDR2 地址值以及 ADDR2 屏蔽位。

i2c_own_address2_enable 函数用于使能 ADDR2 地址。

i2c_slave_data_ctrl_enable 函数用于使能从机字节控制模式。

i2c_clock_stretch_enable 函数用于使能从机时钟延展功能。

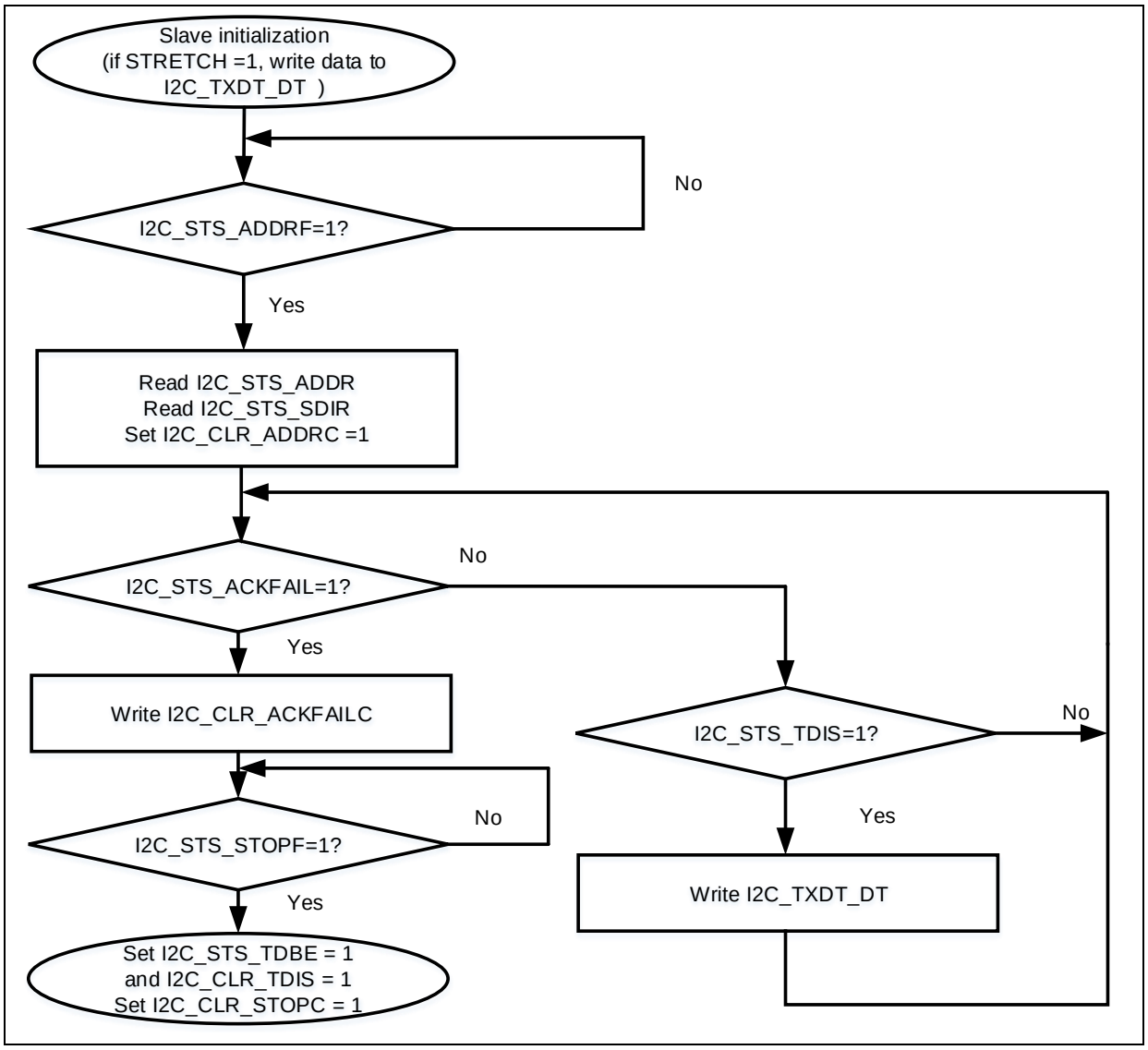
i2c_reload_enable 函数用于使能发送数据重载模式。

2.2.3 从机发送流程

1) 响应主机地址，匹配时回复 ACK；
2) TXDT 为空时，置位 TDIS，从设备写入发送数据；
3) 每发送一个字节会收到 ACK，且置位 TDIS；
4) 如果收到 NACK 位：
   — 置位 NACKF，产生中断；
   — 从设备自动释放 SCL 和 SDA（以便主设备发送 STOP 或 RESTART）；
5) 如果收到 STOP 位：
   — 置位 STOPF，产生中断；

当从机发送开启时钟延展（STRETCH = 0）时，在等待 ADDRF 标志时和发送前一个数据的第 9 个时钟脉冲后，会把 TXDT 中的数据拷贝到移位寄存器中，如果此时 TDIS 还是置位，表示 TXDT 没有写进待发送数据，将发生时钟延展，如下流程图：

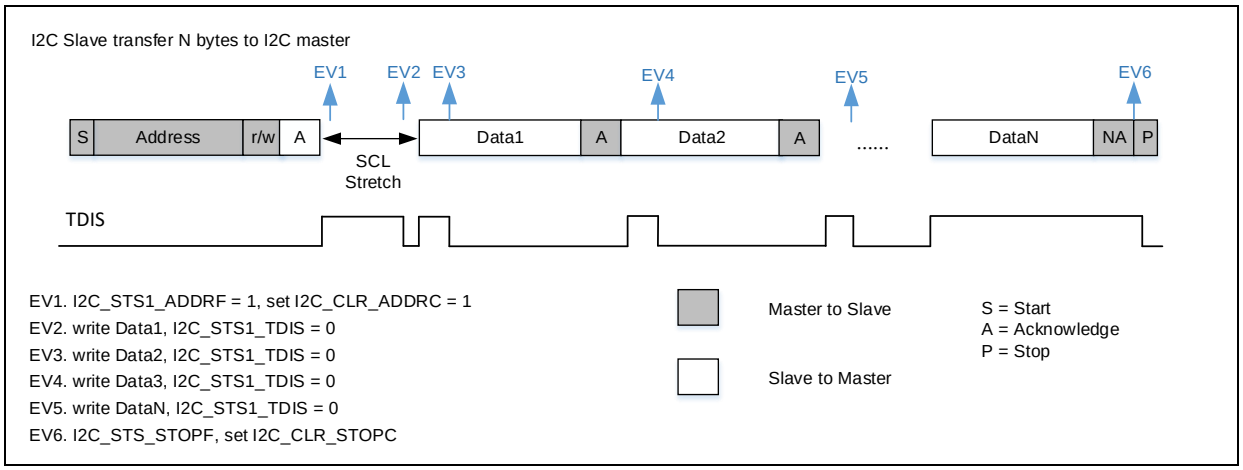
图 9. I²C 从机发送流程图



需要注意的是，在时钟延展关闭（STRETCH=1）的情况下，如果在将要传输数据的第一个 Bit 位开始发送之前，也就是 SDA 边沿产生之前，如果数据还未写入 TXDT 数据寄存器，那么会发生欠载错误，此时 I2C_STS 的 OUF 将会置 1，并将 0xFF 发送到总线。

为了能及时的写入数据，可以在通信开始前，先将数据写入到 DT 寄存器：软件先将 TDBE 置 1，目的是为了清空 TXDT 寄存器的数据，然后将第一个数据写入 TXDT 寄存器，此时 TDBE 清零。

图 10. I²C 从机发送时序图



2.2.4 从机发送流程软件接口

从机发送通过独立的函数接口实现，如下：

```
i2c_status_type i2c_slave_transmit(i2c_handle_type* hi2c, uint8_t* pdata, uint16_t size, uint32_t timeout);
```

i2c_slave_transmit 函数为 i2c_application.c 文件所提供的应用层接口函数，参数包括：I2C 结构体指针、发送数据指针、发送数据字节数和函数超时时间。

注：此函数为Artery 所提供的标准从机发送函数。用户也可根据前述从机发送流程，自行编写从机发送函数。

2.2.5 从机接收流程

1) 当收到数据后，RDBF=1，读取 RXDT 数据寄存器，RDBF 被自动清零；
2) 重复步骤 2 直到所有数据接收完成；
3) 等待收到 STOP 条件，当收到 STOP 条件时，I2C_STS 的 STOPF 置 1，将 I2C_CLR 的 STOPC 写 1，清除 STOPF 标志，传输结束。

图 11. I²C 从机接收流程图

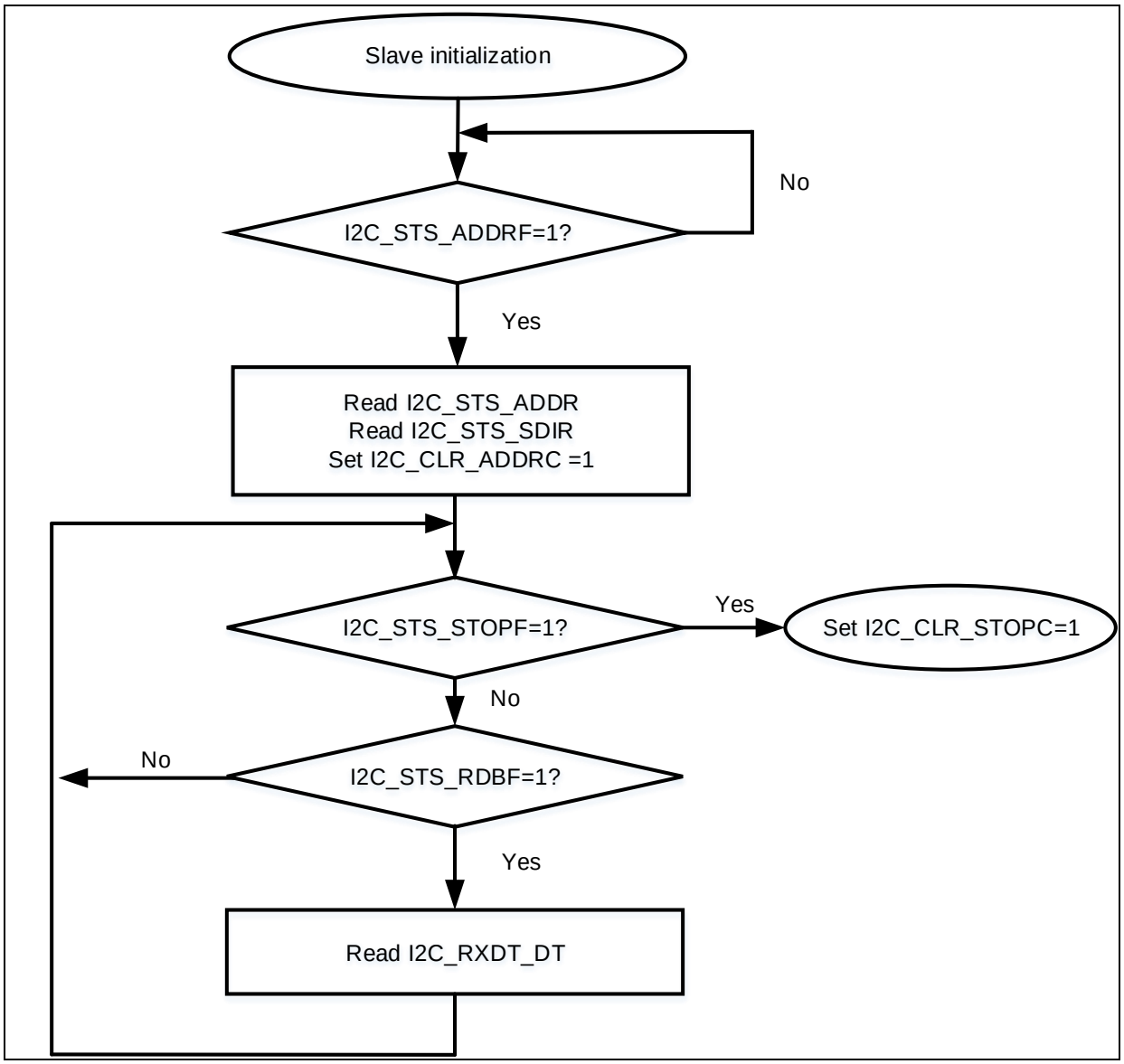
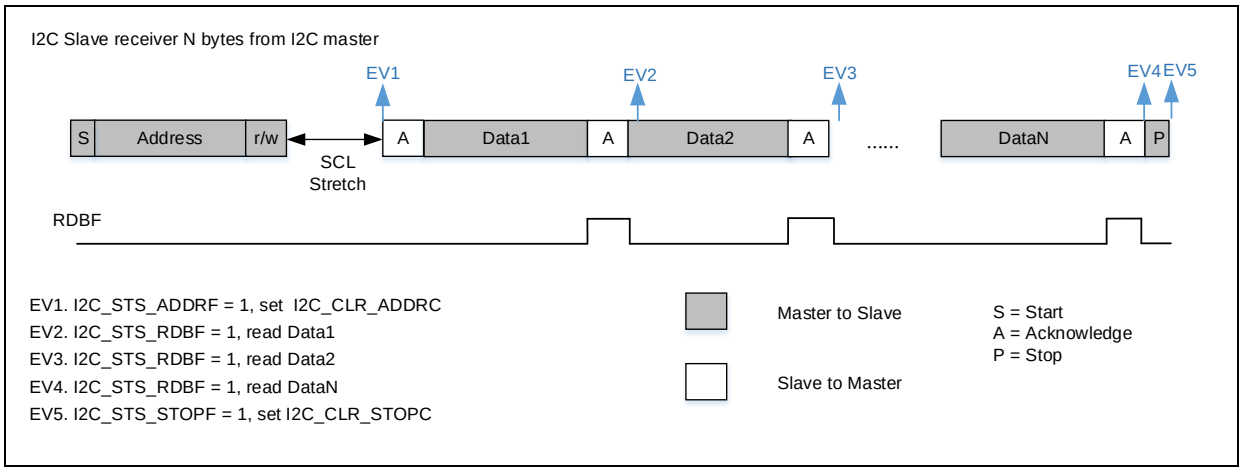


图 12. I²C 从机接收时序图



2.2.6 从机接收流程软件接口

从机接收通过独立的函数接口实现，如下：

```
i2c_status_type i2c_slave_receive(i2c_handle_type* hi2c, uint8_t* pdata, uint16_t size, uint32_t timeout);
```

i2c_slave_receive 函数为 i2c_application.c 文件所提供的应用层接口函数，参数包括：I2C 结构体指针、接收数据指针、接收数据字节数和函数超时时间。

注：此函数为Artery 所提供的标准从机接收函数。用户也可根据前述从机接收流程，自行编写从机接收函数。

3 I²C 配置工具

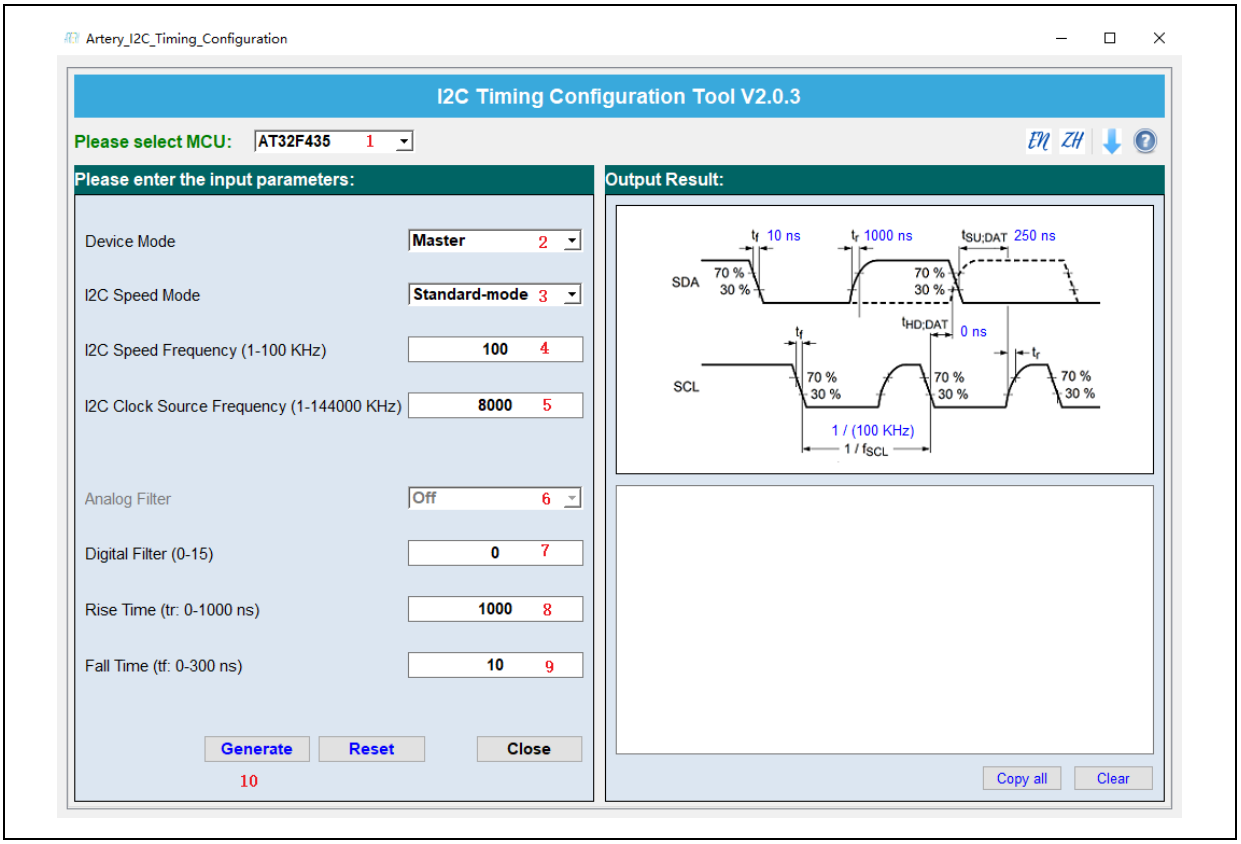
3.1 功能简介

I²C 配置工具 Artery_I2C_Timing_Configuration.exe 可以实现对主机和从机的时钟、数字滤波、模拟滤波配置。

3.2 资源准备

1) 软件环境 Artery_I2C_Timing_Configuration.exe

图 13. Artery I2C Timing Configuration



3.3 使用步骤

1) 选择芯片型号

选择当前使用的芯片型号，例如可以选择 AT32F435、AT32F437。

2) 选择设备模式

- Master：主模式，I²C 作为主机；
- Slave：从模式，I²C 作为从机。

3) 选择 I²C 速度模式

- Standard-mode：标准模式，范围 0~100kHz；
- Fast-mode：快速模式，范围 0~400kHz；
- Fast-mode Plus：增强快速模式，范围 0~1000kHz。

4) 设置 I²C 速度（单位 kHz）

根据实际需求设置 I²C 通信速度，例如需要通信速度为 10kHz，那么这里设置为 10。

5) 设置 I²C 时钟源频率（单位 kHz）

根据实际使用的 I²C 时钟源频率来配置，例如 AT32435 I²C 时钟源为 PCLK1，当 AT32435 主频为 288 MHz，APB1 为 144MHz 时，这里设置为 144000。

6) 模拟滤波使能

- On：打开；
- Off：关闭。

模拟滤波使能后，将过滤 50ns 以下的脉冲。

7) 数字滤波（范围 0~15）

数字滤波时间 = 数字滤波值 x T_{I2C_CLK}；

其中 T_{I2C_CLK} = 1 / I2C 时钟源频率。

当值为 0 时，数字滤波关闭，当值>0 时将过滤小于数字滤波时间的脉冲。

8) 上升时间（t_r 单位 ns）

SCL 和 SDA 总线的上升沿，如图 18 所示。I²C 协议中规定了在标准模式（Standard-mode）、快速模式（Fast-mode）、增强快速模式（Fast-mode Plus）下的范围，详情请参照表 1。上升时间和上拉电阻的阻值关系很大，上拉电阻越小，上升时间越短，可以支持的通信速度就越快，但是功耗也越高。

表 2 中给出了一些常用上拉电阻阻值所对应的上升沿时间，实际可能会因为总线挂的设备数量、布线等差异而有所不同，仅供参考。

9) 下降时间（t_f 单位 ns）

SCL 和 SDA 总线的下降沿，如图 18 所示。I²C 协议中规定了在标准模式（Standard-mode）、快速模式（Fast-mode）、增强快速模式（Fast-mode Plus）下的范围，详情请参照表 1。

图 14.上升沿（t_r）下降沿（t_f）规范

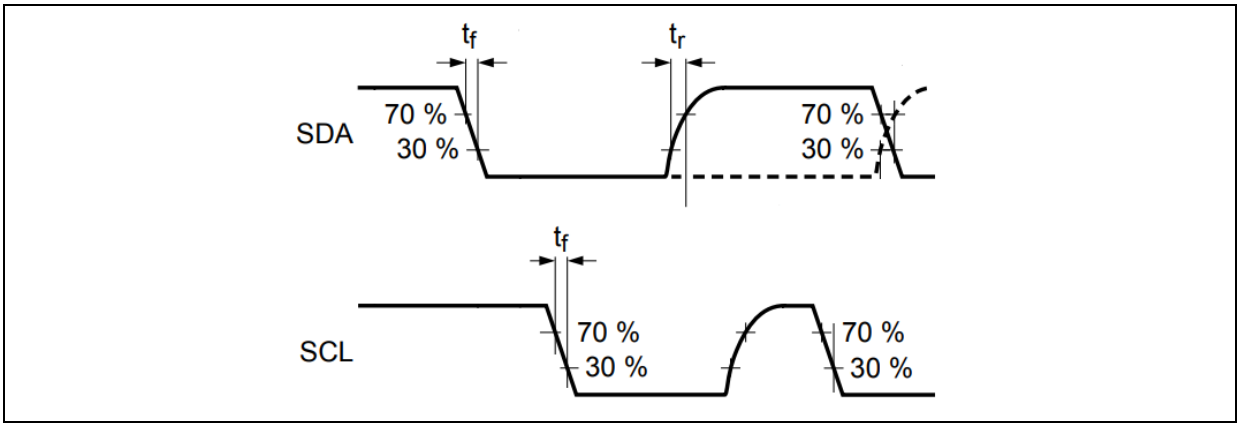


表 1. I²C 时间规范

参数		标志模式 Standard-mode		快速模式 Fast-mode		超快速模式 Fast-mode Plus	
		最小值	最大值	最小值	最大值	最小值	最大值
f_{SCL}(kHz)	SCL 频率	0	100	0	400	0	1000
tr(ns)	SCL、SDA 上升沿	-	1000	-	300	-	120
tf(ns)	SCL、SDA 上升沿	-	300	-	300	-	120

表 2. 常用上拉电阻阻值的 t_r、t_f 参考值(VDD=3.3V)

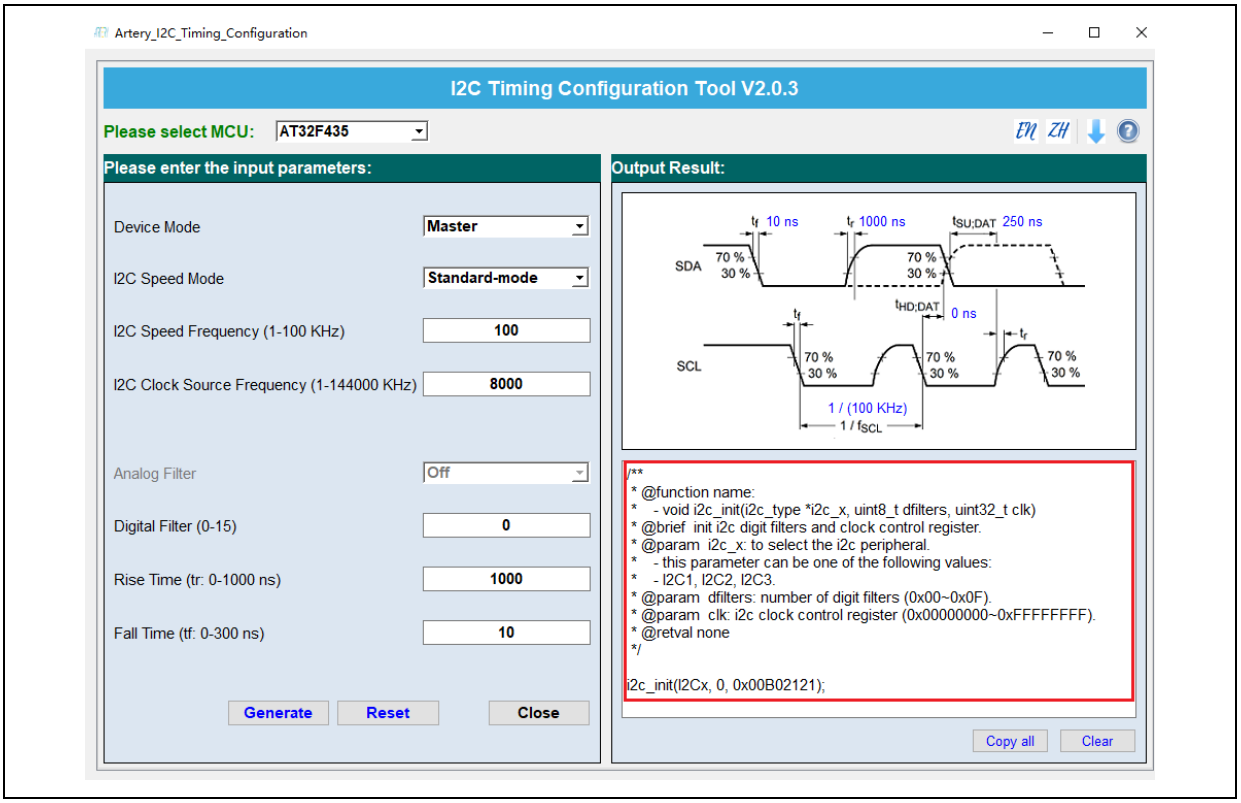
上拉电阻阻值（Ω）	上升沿 tr(ns)	下降沿 tf(ns)	推荐最大速度（kHz）
510	100	9	1000
1K	200	8	500
2K	390	8	300
4.7K	960	8	100
10K	1900	8	50

注：该值是总线上连接两片 AT32 MCU，一个作为主机，一个作为从机测试出来的值，实际可能会因为总线挂的设备数量、布线等差异而有所不同。

10) 产生代码

点击产生代码，上诉配置的值，将会以代码的形式产生出来，如下图红框所示，只需要将右侧输出的代码替换到自己的程序即可。

图 15.代码产生



4 案例 读写 EEPROM

4.1 功能简介

使用硬件 I²C 接口对 EEPROM 存储设备进行读写访问。

4.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

4.7K 上拉电阻

EEPROM 存储设备

2) 软件环境

project\at_start_f4xx\examples\i2c\eeprom

4.3 软件设计

1) 配置流程

- 开启 I²C 外设时钟
- 配置 I²C 所复用的 GPIO
- 配置 I²C 所用的 DMA 通道
- 使能 I²C 外设接口
- 写入 EEPROM 并读取写入的数据
- 比较读写数据内容是否正确

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  i2c_status_type i2c_status;

  /* 初始化系统时钟 */
  system_clock_config();

  /* 配置 NVIC 优先级组 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* at-start board 初始化 */
  at32_board_init();

  hi2cx.i2cx = I2Cx_PORT;

  /* 配置 I2C */
  i2c_config(&hi2cx);

  while(1)
```

```
  {
    /* wait for key USER_BUTTON press before starting the communication */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* 写数据到 EEPROM */
    if((i2c_status = i2c_memory_write(&hi2cx, I2Cx_ADDRESS, 0, tx_buf1, BUF_SIZE, I2C_TIMEOUT)) !=
I2C_OK)
    {
      error_handler(i2c_status);
    }

    delay_ms(5);

    /* 读 EEPROM 数据 */
    if((i2c_status = i2c_memory_read(&hi2cx, I2Cx_ADDRESS, 0, rx_buf1, BUF_SIZE, I2C_TIMEOUT)) !=
I2C_OK)
    {
      error_handler(i2c_status);
    }

    （省略部分代码，完整代码请查看 BSP）

    /* 等待通讯完成 */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* 比较读写数据 */
    if((buffer_compare(tx_buf1, rx_buf1, BUF_SIZE) == 0) &&
       (buffer_compare(tx_buf2, rx_buf2, BUF_SIZE) == 0) &&
       (buffer_compare(tx_buf3, rx_buf3, BUF_SIZE) == 0))
    {
      at32_led_on(LED3);
    }
    else
    {
      error_handler(i2c_status);
    }

  }
}
```

4.4 实验效果

- 如若读写数据完全相同，则 LED3 会被点亮。

5 案例 轮询方式通信

5.1 功能简介

通过轮询方式让两块 AT-START BOARD 的 I²C 接口进行通信，测试作为主机或从机发送和接收数据。

5.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD 两块

4.7K 上拉电阻

2) 软件环境

project\at_start_f4xx\examples\i2c\communication_poll

5.3 软件设计

1) 配置流程

- 开启 I²C 外设时钟
- 配置 I²C 所复用的 GPIO
- 使能 I²C 外设接口
- 从机准备接收数据
- 主机发送数据
- 从机准备发送数据
- 主机接收数据
- 主机从机对比发送和接收到的数据是否正确

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  i2c_status_type i2c_status;

  /* 初始化系统时钟 */
  system_clock_config();

  /* 配置 NVIC 优先级组 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* at-start board 初始化 */
  at32_board_init();

  hi2cx.i2cx = I2Cx_PORT;

  /* 配置 I2C */
  i2c_config(&hi2cx);
```

```
  while(1)
  {

  #if defined (MASTER_BOARD)

    /* 等待 USER_BUTTON 按键按下 */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* 主机发送数据 */
    if((i2c_status = i2c_master_transmit(&hi2cx, I2Cx_ADDRESS, tx_buf, BUF_SIZE, I2C_TIMEOUT)) !=
I2C_OK)
    {
      error_handler(i2c_status);
    }

    delay_ms(10);

    /* 主机接收数据 */
    if((i2c_status = i2c_master_receive(&hi2cx, I2Cx_ADDRESS, rx_buf, BUF_SIZE, I2C_TIMEOUT)) !=
I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* 主机比较读写数据 */
    if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
    {
      at32_led_on(LED3);
    }
    else
    {
      error_handler(i2c_status);
    }
  #else

    /* 等待 USER_BUTTON 按键按下 */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* 从机接收数据 */
    if((i2c_status = i2c_slave_receive(&hi2cx, rx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
    {
```

```
        error_handler(i2c_status);
      }

      /* 从机发送数据 */
      if((i2c_status = i2c_slave_transmit(&hi2cx, tx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
      {
        error_handler(i2c_status);
      }

      /* 从机比较读写数据 */
      if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
      {
        at32_led_on(LED3);
      }
      else
      {
        error_handler(i2c_status);
      }
    #endif

    }
}
```

5.4 实验效果

- 通过宏定义：#define MASTER_BOARD 来选择两块板子的主从关系；
- 如若主机或从机的读写数据完全相同，则 LED3 会被点亮，否则 LED2 会不停闪烁。

6 案例 中断方式通信

6.1 功能简介

通过中断方式让两块 AT-START BOARD 的 I²C 接口进行通信，测试作为主机或从机发送和接收数据。

6.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD 两块

4.7K 上拉电阻

2) 软件环境

project\at_start_f4xx\examples\i2c\communication_int

6.3 软件设计

1) 配置流程

- 开启 I²C 外设时钟
- 配置 I²C 所复用的 GPIO
- 使能 I²C 外设接口
- 使能 I²C 中断
- 从机准备接收数据
- 主机发送数据
- 从机准备发送数据
- 主机接收数据
- 主机从机对比发送和接收到的数据是否正确

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  i2c_status_type i2c_status;

  /* 初始化系统时钟 */
  system_clock_config();

  /* 配置 NVIC 优先级组 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* at-start board 初始化 */
  at32_board_init();

  hi2cx.i2cx = I2Cx_PORT;

  /* 配置 I2C */
```

```
  i2c_config(&hi2cx);

  while(1)
  {

  #if defined (MASTER_BOARD)

    /* 等待 USER_BUTTON 按键按下 */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* 主机发送数据 */
    if((i2c_status = i2c_master_transmit_int(&hi2cx, I2Cx_ADDRESS, tx_buf, BUF_SIZE,
I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* 等待操作完成 */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    delay_ms(10);

    /* 主机接收数据 */
    if((i2c_status = i2c_master_receive_int(&hi2cx, I2Cx_ADDRESS, rx_buf, BUF_SIZE,
I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* 等待操作完成 */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* 主机比较读写数据 */
    if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
    {
      at32_led_on(LED3);
    }
    else
```

```
  {
    error_handler(i2c_status);
  }

#else

  /* 等待 USER_BUTTON 按键按下 */
  while(at32_button_press() != USER_BUTTON)
  {
  }

  /* 从机接收数据 */
  if((i2c_status = i2c_slave_receive_int(&hi2cx, rx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* 等待操作完成 */
  if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* 从机发送数据 */
  if((i2c_status = i2c_slave_transmit_int(&hi2cx, tx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* 等待操作完成 */
  if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* 从机比较读写数据 */
  if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
  {
    at32_led_on(LED3);
  }
  else
  {
    error_handler(i2c_status);
  }
#endif
```

```
  }
}
```

■ 主机中断处理函数代码描述

```
i2c_status_type i2c_master_irq_handler_int(i2c_handle_type* hi2c)
{
  if (i2c_flag_get(hi2c->i2cx, I2C_ACKFAIL_FLAG) != RESET)
  {
    /* 清除 ackfail 标志 */
    i2c_flag_clear(hi2c->i2cx, I2C_ACKFAIL_FLAG);

    /* 刷新 TXDT 数据寄存器 */
    i2c_refresh_txdt_register(hi2c);

    if(hi2c->pcount != 0)
    {
      hi2c->error_code = I2C_ERR_ACKFAIL;
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_TDIS_FLAG) != RESET)
  {
    /* 发送数据 */
    i2c_data_send(hi2c->i2cx, *hi2c->pbuff++);
    hi2c->pcount--;
    hi2c->psize--;
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_TCRLD_FLAG) != RESET)
  {
    if ((hi2c->psize == 0) && (hi2c->pcount != 0))
    {
      /* 继续传输 */
      i2c_start_transfer(hi2c, i2c_transfer_addr_get(hi2c->i2cx), I2C_WITHOUT_START);
    }
    else
    {
      return I2C_ERR_TCRLD;
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_RDBF_FLAG) != RESET)
  {
    /* 接收数据 */
    (*hi2c->pbuff++) = i2c_data_receive(hi2c->i2cx);
    hi2c->pcount--;
    hi2c->psize--;
  }
```

```
  else if (i2c_flag_get(hi2c->i2cx, I2C_TDC_FLAG) != RESET)
  {
    if (hi2c->pcount == 0)
    {
      if (hi2c->i2cx->ctrl2_bit.astopen == 0)
      {
        /* 产生停止条件 */
        i2c_stop_generate(hi2c->i2cx);
      }
    }
    else
    {
      return I2C_ERR_TDC;
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_STOPF_FLAG) != RESET)
  {
    /* 清除停止标志 */
    i2c_flag_clear(hi2c->i2cx, I2C_STOPF_FLAG);

    /* 复位 ctrl2 寄存器 */
    i2c_reset_ctrl2_register(hi2c);

    if (i2c_flag_get(hi2c->i2cx, I2C_ACKFAIL_FLAG) != RESET)
    {
      /* 清除 ackfail 标志 */
      i2c_flag_clear(hi2c->i2cx, I2C_ACKFAIL_FLAG);
    }

    /* 刷新 TXDT 寄存器 */
    i2c_refresh_txdt_register(hi2c);

    /* 关闭中断 */
    i2c_interrupt_enable(hi2c->i2cx, I2C_ERR_INT | I2C_TDC_INT | I2C_STOP_INT | I2C_ACKFIAL_INT |
I2C_TD_INT | I2C_RD_INT, FALSE);

    /* 传输完成 */
    hi2c->status = I2C_END;
  }

  return I2C_OK;
}
```

■ 从机中断处理函数代码描述

```
i2c_status_type i2c_slave_irq_handler_int(i2c_handle_type* hi2c)
```

```
{
  if (i2c_flag_get(hi2c->i2cx, I2C_ACKFAIL_FLAG) != RESET)
  {
    /* 传输完成 */
    if (hi2c->pcount == 0)
    {
      i2c_refresh_txdt_register(hi2c);

      /* 清除 ackfail 标志 */
      i2c_flag_clear(hi2c->i2cx, I2C_ACKFAIL_FLAG);
    }
    /* the transfer has not been completed */
    else
    {
      /* 清除 ackfail 标志 */
      i2c_flag_clear(hi2c->i2cx, I2C_ACKFAIL_FLAG);
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_ADDRF_FLAG) != RESET)
  {
    /* 清除 addrf 标志 */
    i2c_flag_clear(hi2c->i2cx, I2C_ADDRF_FLAG);
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_TDIS_FLAG) != RESET)
  {
    if (hi2c->pcount > 0)
    {
      /* 发送数据 */
      hi2c->i2cx->txdt = (*(hi2c->pbuff++));
      hi2c->psize--;
      hi2c->pcount--;
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_RDBF_FLAG) != RESET)
  {
    if (hi2c->pcount > 0)
    {
      /* 接收数据 */
      (*hi2c->pbuff++) = i2c_data_receive(hi2c->i2cx);
      hi2c->pcount--;
      hi2c->psize--;
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_STOPF_FLAG) != RESET)
  {
    /* 清除停止条件 */
```

```
      i2c_flag_clear(hi2c->i2cx, I2C_STOPF_FLAG);

      /* 关闭中断 */
      i2c_interrupt_enable(hi2c->i2cx, I2C_ADDR_INT | I2C_STOP_INT | I2C_ACKFIAL_INT | I2C_ERR_INT
| I2C_TDC_INT | I2C_TD_INT | I2C_RD_INT, FALSE);

      /* 复位 ctrl2 寄存器 */
      i2c_reset_ctrl2_register(hi2c);

      /* 刷新 TXDT 寄存器 */
      i2c_refresh_txdt_register(hi2c);

      if (i2c_flag_get(hi2c->i2cx, I2C_RDBF_FLAG) != RESET)
      {
        /* 接收数据 */
        (*hi2c->pbuff++) = i2c_data_receive(hi2c->i2cx);

        if ((hi2c->psize > 0))
        {
          hi2c->pcount--;
          hi2c->psize--;
        }
      }

      /* 传输完成 */
      hi2c->status = I2C_END;
    }

    return I2C_OK;
}
```

6.4 实验效果

- 通过宏定义：#define MASTER_BOARD 来选择两块板子的主从关系；
- 如若主机或从机的读写数据完全相同，则 LED3 会被点亮，否则 LED2 会不停闪烁。

7 案例 DMA 方式通信

7.1 功能简介

通过 DMA 方式让两块 AT-START BOARD 的 I²C 接口进行通信，测试作为主机或从机发送和接收数据。

7.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD 两块

4.7K 上拉电阻

2) 软件环境

project\at_start_f4xx\examples\ i2c\communication_dma

7.3 软件设计

1) 配置流程

- 开启 I²C 外设时钟
- 配置 I²C 所复用的 GPIO
- 配置 I²C 所使用的 DMA 通道
- 使能 I²C 外设接口
- 从机准备接收数据
- 主机发送数据
- 从机准备发送数据
- 主机接收数据
- 主机从机对比发送和接收到的数据是否正确

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  i2c_status_type i2c_status;

  /* 初始化系统时钟 */
  system_clock_config();

  /* 配置 NVIC 优先级组 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* at-start board 初始化 */
  at32_board_init();

  hi2cx.i2cx = I2Cx_PORT;

  /* 配置 I2C */
```

```
  i2c_config(&hi2cx);

  while(1)
  {

  #if defined (MASTER_BOARD)

    /* 等待 USER_BUTTON 按键按下 */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* 主机发送数据 */
    if((i2c_status = i2c_master_transmit_dma(&hi2cx, I2Cx_ADDRESS, tx_buf, BUF_SIZE,
I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* 等待操作完成 */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    delay_ms(10);

    /* 主机接收数据 */
    if((i2c_status = i2c_master_receive_dma(&hi2cx, I2Cx_ADDRESS, rx_buf, BUF_SIZE,
I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* 等待操作完成 */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* 主机比较读写数据 */
    if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
    {
      at32_led_on(LED3);
    }
    else
```

```
  {
    error_handler(i2c_status);
  }

#else

  /* 等待 USER_BUTTON 按键按下 */
  while(at32_button_press() != USER_BUTTON)
  {
  }

  /* 从机接收数据 */
  if((i2c_status = i2c_slave_receive_dma(&hi2cx, rx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* 等待操作完成 */
  if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* 从机发送数据 */
  if((i2c_status = i2c_slave_transmit_dma(&hi2cx, tx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* 等待操作完成 */
  if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* 从机比较读写数据 */
  if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
  {
    at32_led_on(LED3);
  }
  else
  {
    error_handler(i2c_status);
  }

#endif
```

```
  }
}
```

■ 主机 DMA 发送接收完成中断处理函数代码描述

```
void i2c_dma_tx_rx_irq_handler(i2c_handle_type* hi2c, dma_channel_type* dma_channel)
{
  /* 传输完成 */
  if (dma_flag_get(DMA_GET_TC_FLAG(dma_channel)) != RESET)
  {
    /* 关闭 DMA 传输完成中断 */
    dma_interrupt_enable(dma_channel, DMA_FDT_INT, FALSE);

    /* 清除传输完成标志 */
    dma_flag_clear(DMA_GET_TC_FLAG(dma_channel));

    /* 关闭 DMA 请求 */
    i2c_dma_enable(hi2c->i2cx, DMA_GET_REQUEST(dma_channel), FALSE);

    /* 关闭 DMA 通道 */
    dma_channel_enable(dma_channel, FALSE);

    switch(hi2c->mode)
    {
      case I2C_DMA_MA_TX:
      case I2C_DMA_MA_RX:
      {
        /* 更新传输字节数 */
        hi2c->pcount -= hi2c->psize;

        /* 传输完成 */
        if (hi2c->pcount == 0)
        {
          /* 打开 STOP 中断 */
          i2c_interrupt_enable(hi2c->i2cx, I2C_STOP_INT, TRUE);
        }
        /* 传输未完成 */
        else
        {
          /* 更新传输缓存指针 */
          hi2c->pbuff += hi2c->psize;

          /* 设置传输字节数 */
          if (hi2c->pcount > MAX_TRANSFER_CNT)
          {
```

```
                hi2c->psize = MAX_TRANSFER_CNT;
              }
              else
              {
                hi2c->psize = hi2c->pcount;
              }

              /* 配置 DMA 通道，继续传输 */
              i2c_dma_config(hi2c, dma_channel, hi2c->pbuff, hi2c->psize);

              /* 打开 TDC 中断 */
              i2c_interrupt_enable(hi2c->i2cx, I2C_TDC_INT, TRUE);
            }
          }break;
          case I2C_DMA_SLA_TX:
          case I2C_DMA_SLA_RX:
          {

          }break;

          default:break;
        }
      }
}
```

7.4 实验效果

- 通过宏定义：#define MASTER_BOARD 来选择两块板子的主从关系；
- 如若主机或从机的读写数据完全相同，则 LED3 会被点亮，否则 LED2 会不停闪烁。

8 文档版本历史

表 3. 文档版本历史

日期	版本	变更
2021.10.19	2.0.0	最初版本
2022.08.24	2.0.1	修改图14

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。