AN0122
Application Note

AT32F425 I²C Application Note

Introduction

The I²C (inter-integrated circuit) bus interface of AT32 series manages the communication between the microcontroller and serial I²C bus. It supports master and slave modes, with up to 1 Mbit/s of communication speed (fast mode plus). This document introduces main features and applications of I²C bus interface.

Applicable products:

Part number	AT32F425xx

Contents

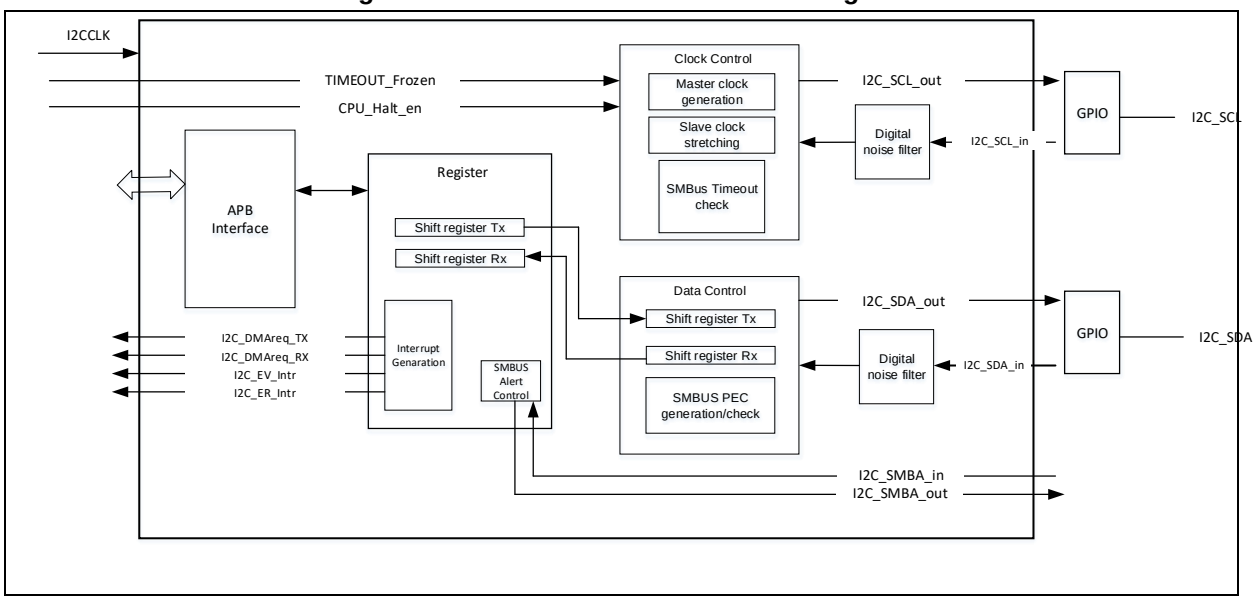
List of Tables

List of Figures

1 I²C interface introduction

I²C bus consists of a data line (SDA) and a clock line (SCL). It can achieve a maximum of 100 kHz communication speed in standard mode, up to 400 kHz in fast mode and 1 MHz in fast mode plus. A frame of data transmission begins with a Start condition and ends with a Stop condition. The bus is kept in busy state after receiving the Start condition, and becomes idle as long as it receives the Stop condition. I²C bus is featured with master and slave modes, multimaster capability, programmable data setup and hold time, clock stretching capability, and DMA data access, and it supports SMBus 2.0 protocol.

Figure 1. I²C interface function block diagram



2 I²C interface communication

2.1 Master communication

2.1.1 Initialization

1. Master clock initialization

Before enabling the peripheral (I2CEN), set the following bits of the I2Cx_CLKCTRL register to configure the I²C master clock.

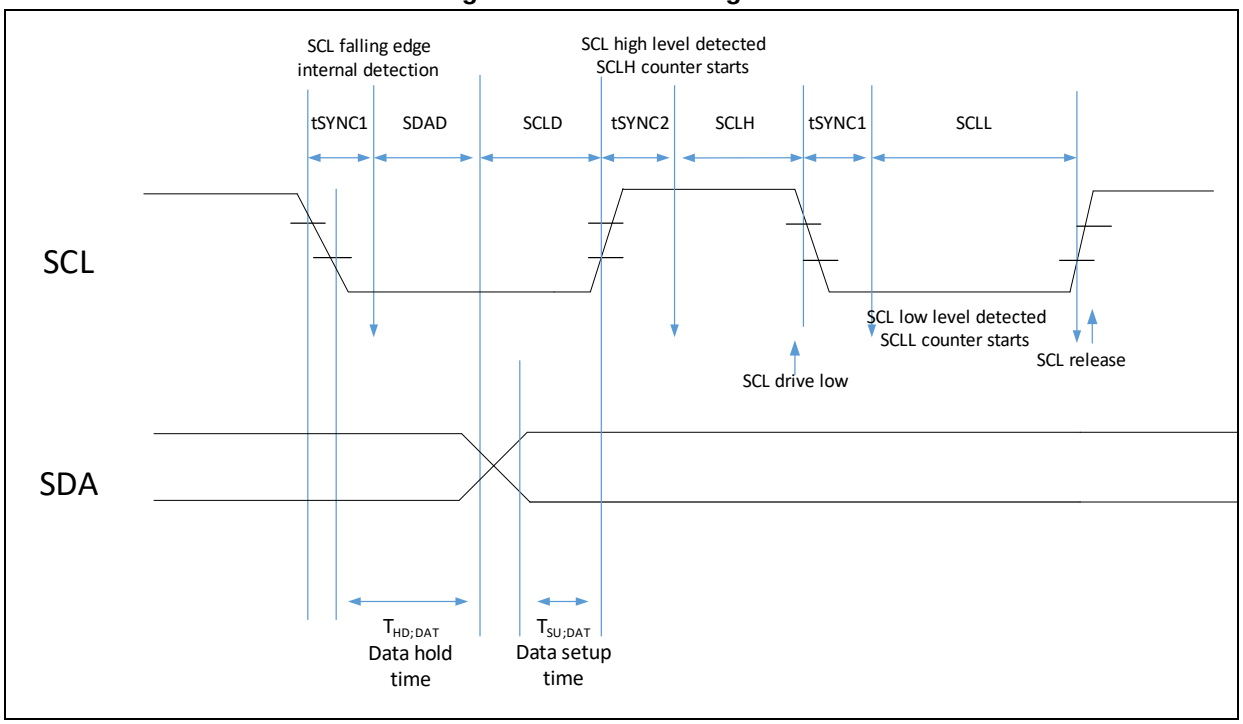
- DIV[7:0]: I²C clock divider
- SDAD[3:0]: Data hold time ($t_{HD;DAT}$)
- SCLD[3:0]: Data setup time ($t_{SU;DAT}$)
- SCLH[7:0]: SCL high
- SCLL[7:0]: SCL low

This register can be configured by means of Artery_I2C_Timing_Configuration tool. For details, refer to Section 3.

SCL low: When the SCL low signal is detected, the internal SCLL counter starts counting until it reaches the SCLL value. At this point, the SCL line is released and becomes high.

SCL high: When the SCL high signal is detected, the internal SCLH counter starts counting. When the counter value reaches the SCLH value, the SCL line is pulled low. In the process of SCL remaining high, if it is pulled low by external bus, the internal SCLH counter will stop counting and start counting in SCL low mode, laying the foundation for clock synchronization.

Figure 2. Master clock generation



2. Master communication initialization

Set the following parameters in the I2C_CTRL2 register before enabling communication:

1) Number of bytes to be transferred
   - ≤255 bytes

Disable reload mode by setting RLDEN=0 in the I2C_CTRL2 register.

Set CNT[7:0]=N in the I2C_CTRL2 register.

- >255 bytes

  Enable reload mode by setting RLDEN=1 in the I2C_CTRL2 register.

  Set CNT[7:0]=255 in the I2C_CTRL2 register.

  Remaining bytes N=N-255.

2) End of data transfer
   - ASTOPEN=0: stop data transfer by software. After the completion of data transfer, the TDC in the I2C_STS register is set to 1, and GENSTOP=1 or GENSTART=1 is written by software to send a STOP or START condition.
   - ASTOPEN=1: data transfer is stopped automatically. A STOP condition is sent at the end of data transfer.
3) Slave address
   - Set slave address value (by setting the SADDR bit in the I2C_CTRL2 register).
   - Set slave address mode (by setting the ADDR10 bit in the I2C_CTRL2 register).

   ADDR10=0:7 bit address mode

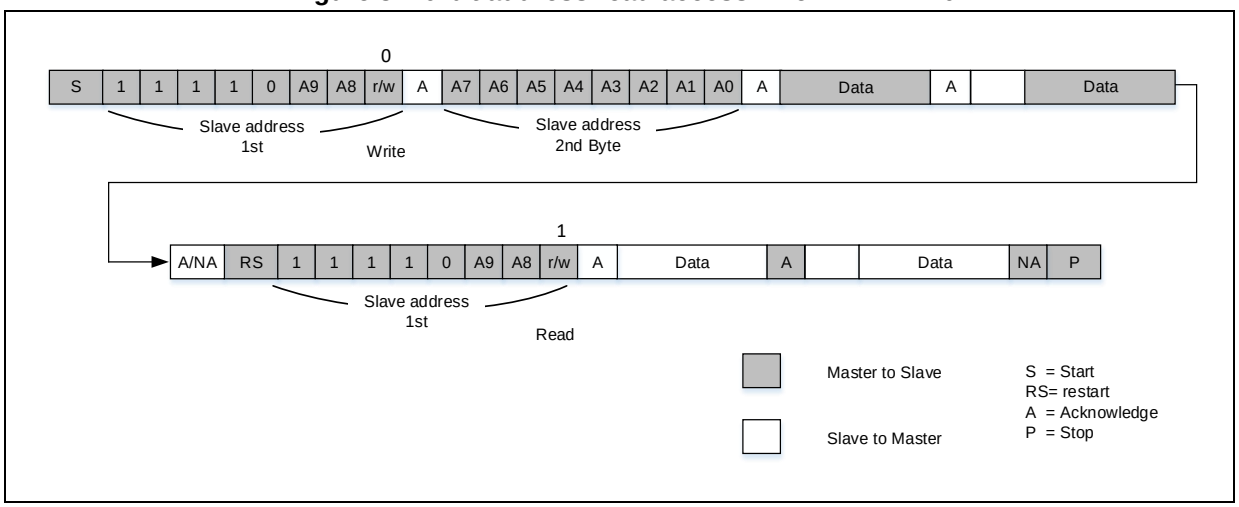
   ADDR10=1:10 bit address mode
4) Data transfer direction (by setting the DIR bit in the I2C_CTRL2 register)
   - DIR=0: Master receiving
   - DIR=1: Master transmission
5) Start data transfer

When GENSTART=1 in the I2C_CTRL2 register, the master starts sending a START condition and Slave address.

3. Special timing initialization for master 10-bit addressing

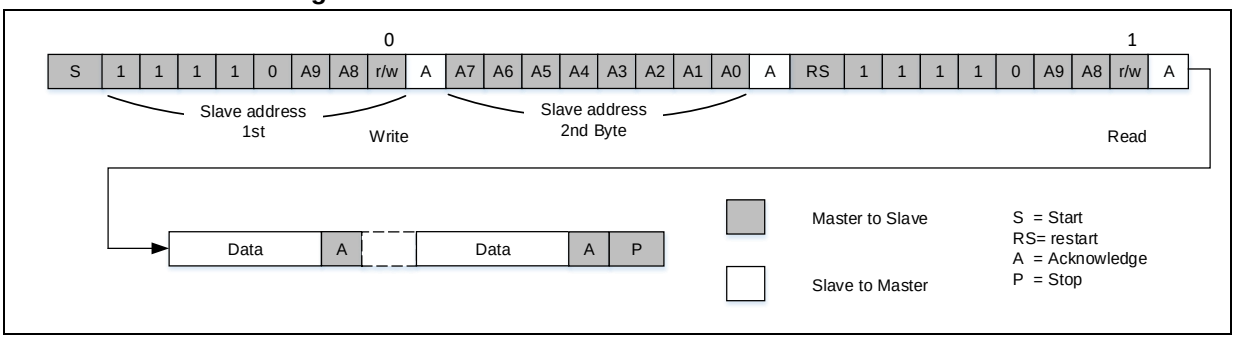
In 10-bit addressing mode, the READH10 bit of the I2C_CTRL2 register is used to generate a special timing. When READH10=1, the master sends data to the slave before read access to the slave, as shown in the figure below.

Figure 3. 10-bit address read access when READH10 = 1



When ASTOPEN = 0, data is transferred from master to slave. At the end of data transfer, set READH10=1, and then the master starts receiving data from the slave.

Figure 4. 10-bit address read access when READH10 = 0



2.1.2 Master communication initialization software interface

The software interface for master communication initialization is implemented by independent functions, as shown below.

```
void i2c_init(i2c_type *i2c_x, uint8_t dfilters, uint32_t clk); /* master clock initialization */
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); /* master communication initialization */
void i2c_addr10_mode_enable(i2c_type *i2c_x, confirm_state new_state); /* 10-bit addressing mode enable*/
void i2c_addr10_header_enable(i2c_type *i2c_x, confirm_state new_state); /* 10-bit address header read access
timing enable */
```

The i2c_init function includes three parameters, i.e., I2C, digital filter value and master clock configuration value.

The i2c_transmit_set function is used to initialize communication parameters, including I2C, slave address, number of bytes to be transferred, and STOP/START condition generation mode.

The i2c_addr10_mode_enable function is used to enable the 10-bit addressing mode.

The i2c_addr10_header_enable function is used to enable the 10-bit address header read access timing, that is, the master sending a complete 10-bit slave address read sequence or only the first 7 bits of the 10-bit address.

2.1.3 Master transmit

1) I2C_TXDT data register is empty, and TDIS=1 in the I2C_STS register;
2) Write data to the TXDT register, and data transfer starts;
3) Repeat step 1 and step 2 until the data in the CNT[7:0] is sent;
4) When TCRLD=1 (reload mode) in the I2C_STS register, the following two circumstances should be noted:
   - Remaining bytes N>255: write 255 to the CNT bit, N=N-255, TCRLD is cleared, and data transfer continues;
   - Remaining bytes N≤255: disable the reload mode (RLDEN=0), write N to the CNT bit, TCRLD is cleared, and data transfer continues.
5) STOP condition
   - STOP condition generation:
     ASTOPEN=0: TDC=1 in the I2C_STS register, set GENSTOP=1 to generate a STOP condition;
     ASTOPEN=1: A STOP condition is generated automatically;

— Wait for the generation of a STOP condition. When a STOP condition is generated, set STOPF=1 in the I2C_STS register. The STOPF flag can be cleared by setting STOPC=1 in the I2C_CLR register, and the transfer stops.

Figure 5. I²C master transmission flow

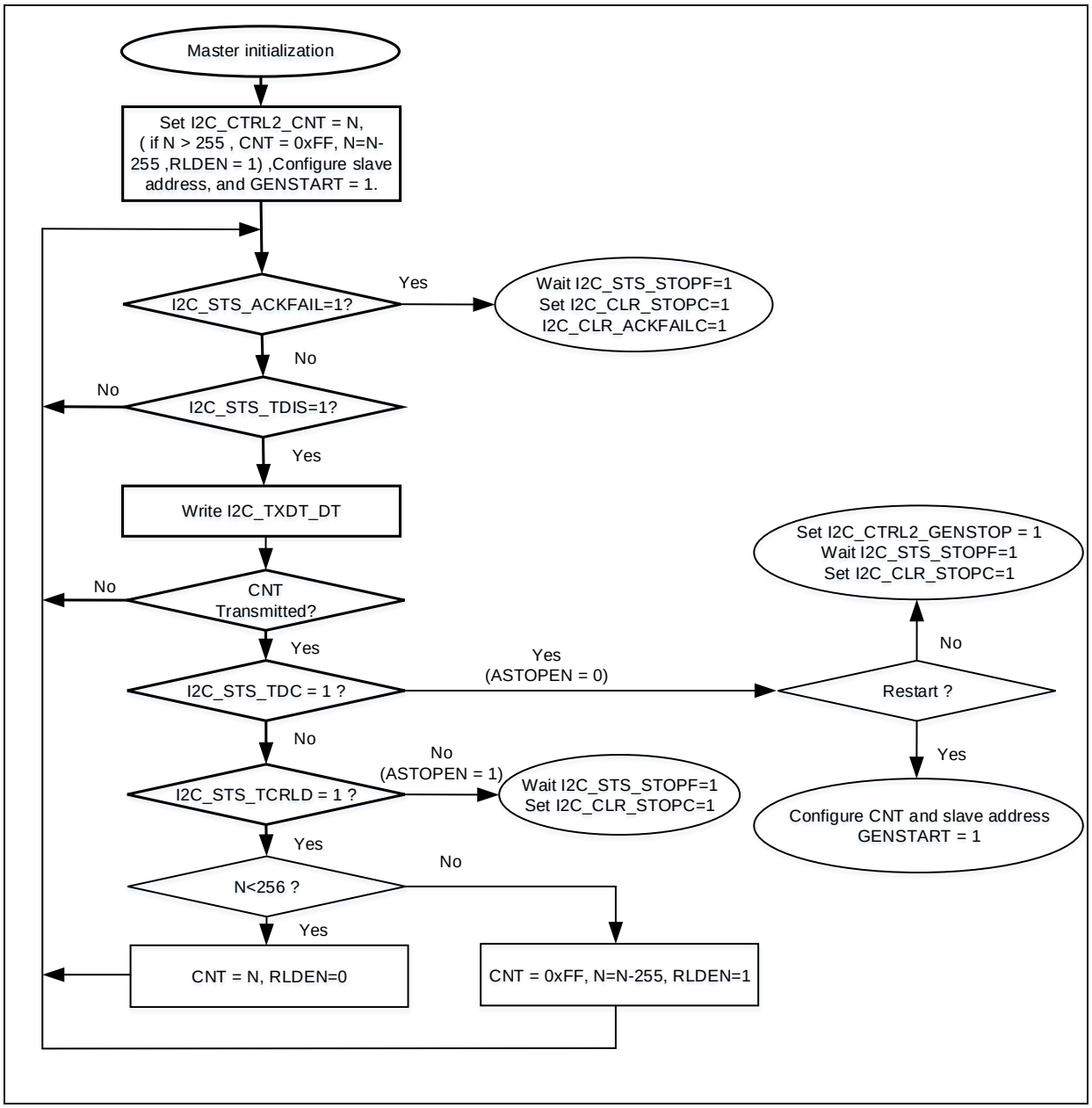
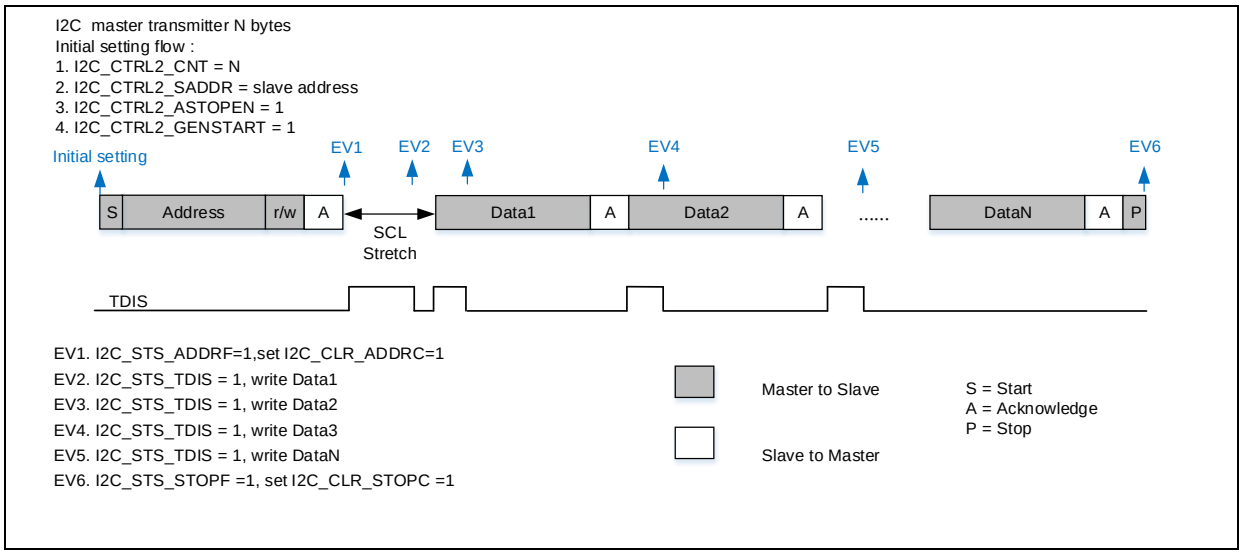


Figure 6. I²C master transmission timing



2.1.4 Master transmission software interface

Master transmission is implemented by independent functions, as shown below.

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

The i2c_master_transmit is an application-level interface function provided by i2c_application.c, and it includes I2C structure pointer, slave address, transmit data pointer, number of bytes to be transferred, and function timeout.

Note: This function is a standard master transmission function provided by Artery. Users can write a master transmission function according to the above-mentioned master transmission flow.

2.1.5 Master receive

1) After the data is received, RDBF=1, read the RXDT register, and the RDBF flag is cleared automatically;
2) Repeat step 2 until the data in CNT[7:0] bit is received;
3) When TCRLD=1 (reload mode) in the I2C_STS register, the following circumstances should be noted:
   - Remaining bytes N>255: write 255 to the CNT bit, N=N-255, TCRLD is cleared automatically, and data transfer continues;
   - Remaining bytes N≤255: disable the reload mode (RLDEN=0), write N to the CNT bit, TCRLD is cleared automatically, and data transfer continues.
4) After receiving the last data, an NACK signal will be sent by the master automatically.
5) STOP condition
   - STOP condition generation:

     ASTOPEN=0: TDC=1 in the I2C_STS register, set GENSTOP=1 to generate a STOP condition;

     ASTOPEN=1: A STOP condition is generated automatically.
   - Wait for the generation of a STOP condition. When a STOP condition is generated, set STOPF=1 in the I2C_STS register, and set STOPC=1 in the I2C_CLR register. The STOPF is cleared, and then transfer stops.

Figure 7. I²C master receiving flow

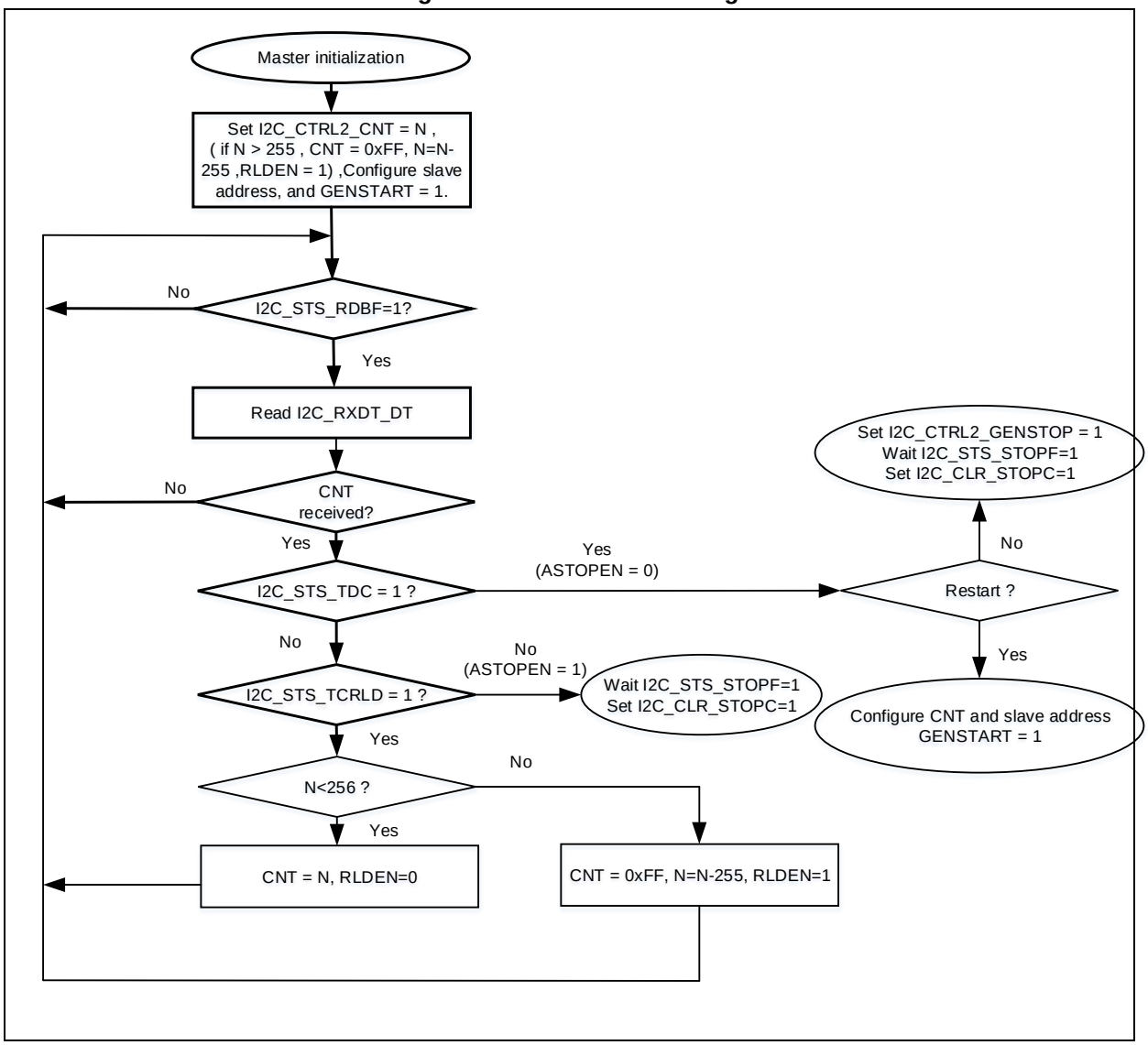
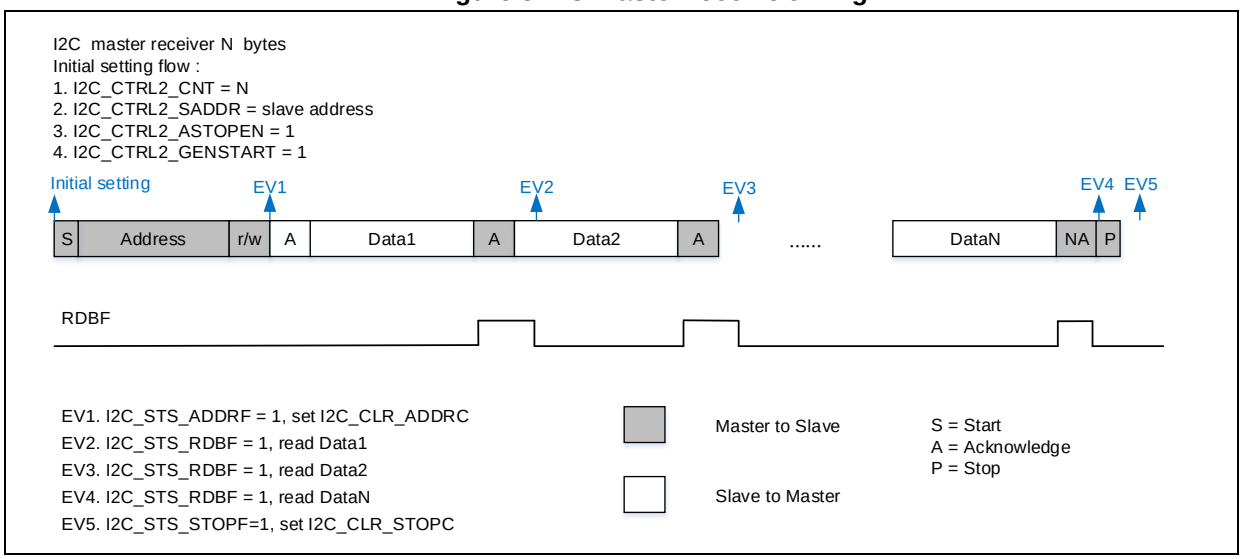


Figure 8. I²C master receive timing



2.1.6 Master receive software interface

The master receive is implemented by independent functions, as shown below.

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

The i2c_master_receive is an application-level interface function provided by i2c_application.c, and it includes I2C structure pointer, slave address, receive data pointer, number of bytes to be received, and function timeout.

Note: This function is a standard master receive function provided by Artery. Users can write a master receive function according to the above-mentioned master receive flow.

2.2 Slave communication

2.2.1 Initialization

1. Slave address configuration

Each I²C slave device supports two slave addresses simultaneously, specified by OADDR1 and OADDR2.

I2C_OADDR1

- Enable by setting the ADDR1EN bit;
- Configure as 7-bit (default) or 10-bit address by setting the ADDR1MODE bit.

I2C_OADDR2

- Enable by setting the ADDR2EN bit;
- Fix 7-bit address mode;
- Mask 0~7 LSB address bits during address matching by setting the ADDR2MASK [2:0] bit:

  ADDR2MASK = 0: each bit of the 7-bit address takes part in address matching

  ADDR2MASK = 7: any non-reserved 7-bit address will be acknowledged by the slave device

2. Slave address matching

When an I²C enabled address is selected for matching, the ADDRF interrupt status flag is set to 1. At this point, if ADDRIEN=1, an interrupt will be generated. If both slave addresses are enabled, when an ADDR interrupt is generated during address matching, check the ADDR [6:0] bit in the status register to confirm whether OADDR1 or OADDR2 is addressed.

3. Slave byte control mode (typically in SMBus mode)

The slave device can perform acknowledgement control for each byte received.

Required configuration: SCTRL = 1 & RLDEN =1 & STRETCH = 0 & CNT ≥ 1

Slave byte control flow:

1) When a byte is received, set the TCRLD bit, and the clock stretches between the 8th and 9th pulse;
2) Read the value in RXDT by software, and confirm whether to set the ACK;
3) Reload CNT = 1 by software to stop clock stretching;
4) Acknowledgement or non-acknowledgement signal appears on the bus at the 9th pulse.

Note:

When set the SCTRL bit, enable the clock stretching, i.e., STRETCH = 0.

The CNT value can be larger than 1, to realize that multiple bytes are received with automatic ACK and then enable acknowledgement control. It is recommended to disable SCTRL during slave

transmission, and no byte acknowledgement control is required at this point.

2.2.2 Slave communication initialization software interface

The software interface for slave communication initialization is implemented by independent functions, as shown below.

```
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);
```

The i2c_own_address1_set function is used to configure OADDR1 address mode and ADDR1 address value.

The i2c_own_address2_set function is used to configure ADDR2 address value and ADDR2 mask bit.

The i2c_own_address2_enable function is used to enable the ADDR2 address.

The i2c_slave_data_ctrl_enable function is used to enable slave byte control mode.

The i2c_clock_stretch_enable function is used to enable slave clock stretching.

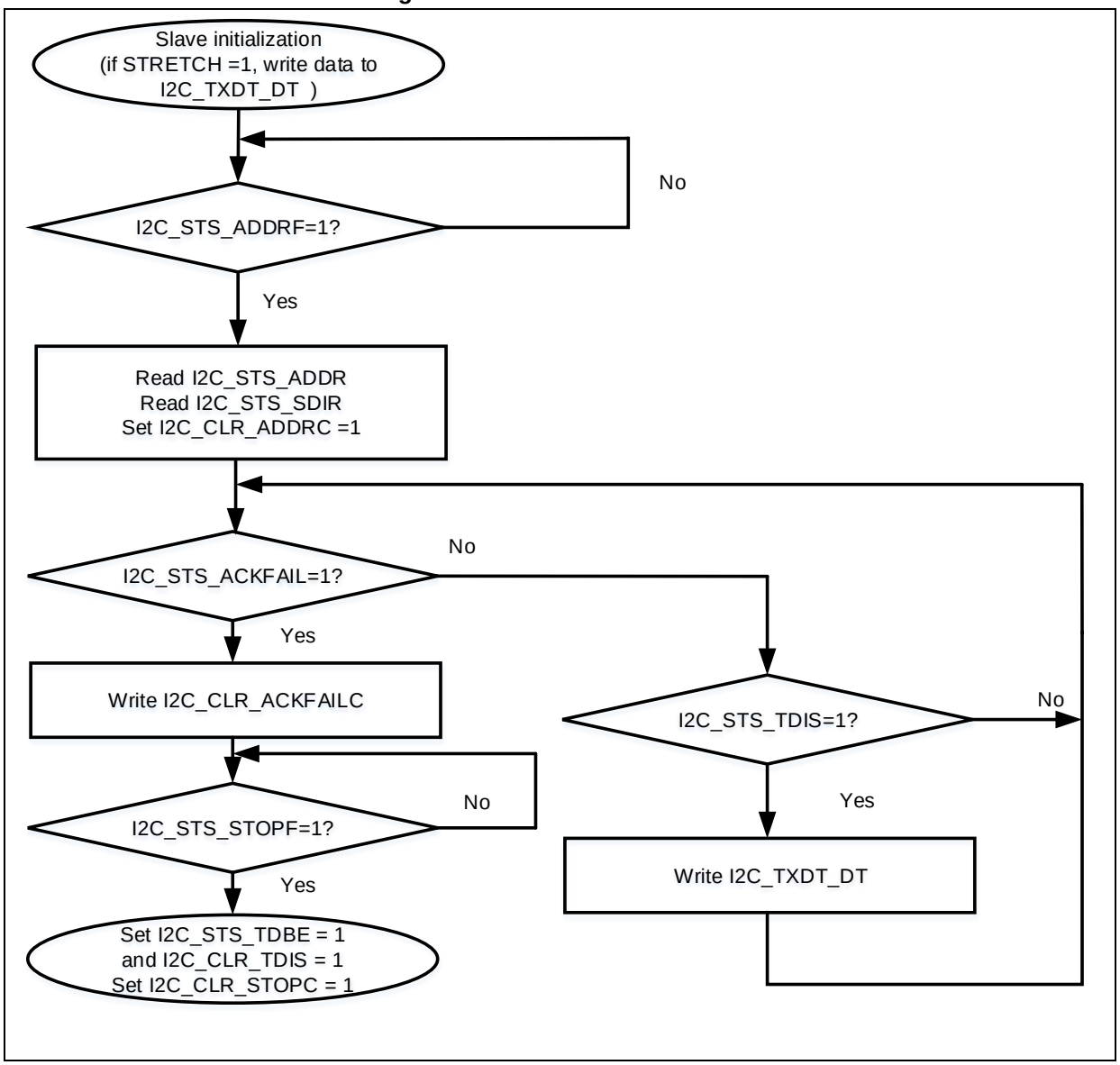
The i2c_reload_enable function is used to enable transmit data reload mode.

2.2.3 Slave transmit

1) Respond the master address, and return ACK when matching;
2) When the TXDT is empty, set the TDIS bit, and write the data to be transferred to the slave device;
3) When a byte is sent, the ACK is received, and set the TDIS bit;
4) If the NACK bit is received:
   - Set the NACKF bit and generate an interrupt;
   - Slave device release SCL and SDA (for the master to send STOP or RESTART) automatically.
5) If the STOP bit is received:
   - Set the STOPF bit and generate an interrupt.

When the clock stretching is enabled (STRETCH = 0), the data in TXDT is copied to the shift register when waiting for ADDRF flag and after the transmission of the 9th clock pulse of the previous data. If the TDIS bit is set at this point, it indicates that the data to be sent is not written to the TXDT register, thus resulting in clock stretching. This process is shown in the figure below.

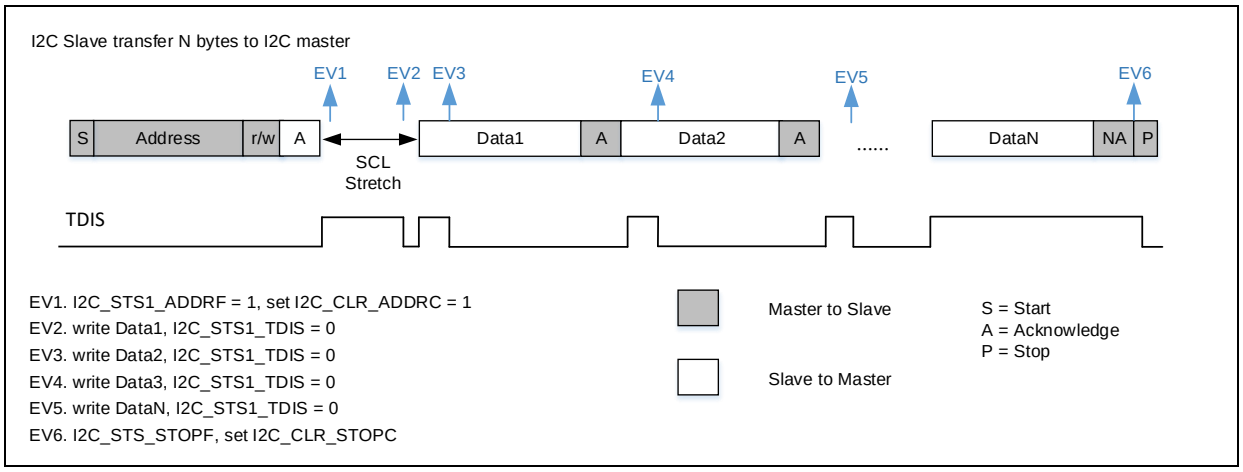
Figure 9. I²C slave transmission flow



In case of the clock stretching being disabled (STRETCH=1), if data has not yet been written to the TXDT register before the transmission of the first bit of the to-be-transferred data (i.e., before the generation of SDA edge), an underrun error may occur, and the OUF bit is set to 1 in the I2C_STS register, sending 0xFF to the bus.

In order to write data in time, data must be written to the DT register first before communication: set TDBE to 1 by software to clear the TXDT register, and then write the first data to the TXDT register to clear the TDBE bit.

Figure 10. I²C slave transmission timing



2.2.4 Slave transmission software interface

Slave transmission is implemented by independent functions, as shown below.

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

The i2c_slave_transmit is an application-level interface function provided by i2c_application.c, and it includes I2C structure pointer, transmit data pointer, number of bytes to be transferred, and function timeout.

Note: This function is a standard slave transmission function provided by Artery. Users can write a slave transmission function according to the above-mentioned slave transmission flow.

2.2.5 Slave receive

1) After the data is received, RDBF=1, read the RXDT register, and the RDBF bit is cleared automatically;
2) Repeat step 2 until all data is received;
3) Wait for the generation of a STOP condition. Once received, set the STOPF bit to 1 in the I2C_STS register, and clear the STOPF flag by writing 1 to the STOPC bit in the I2C_CLR register, and then transfer ends.

Figure 11. I²C slave receiving flow

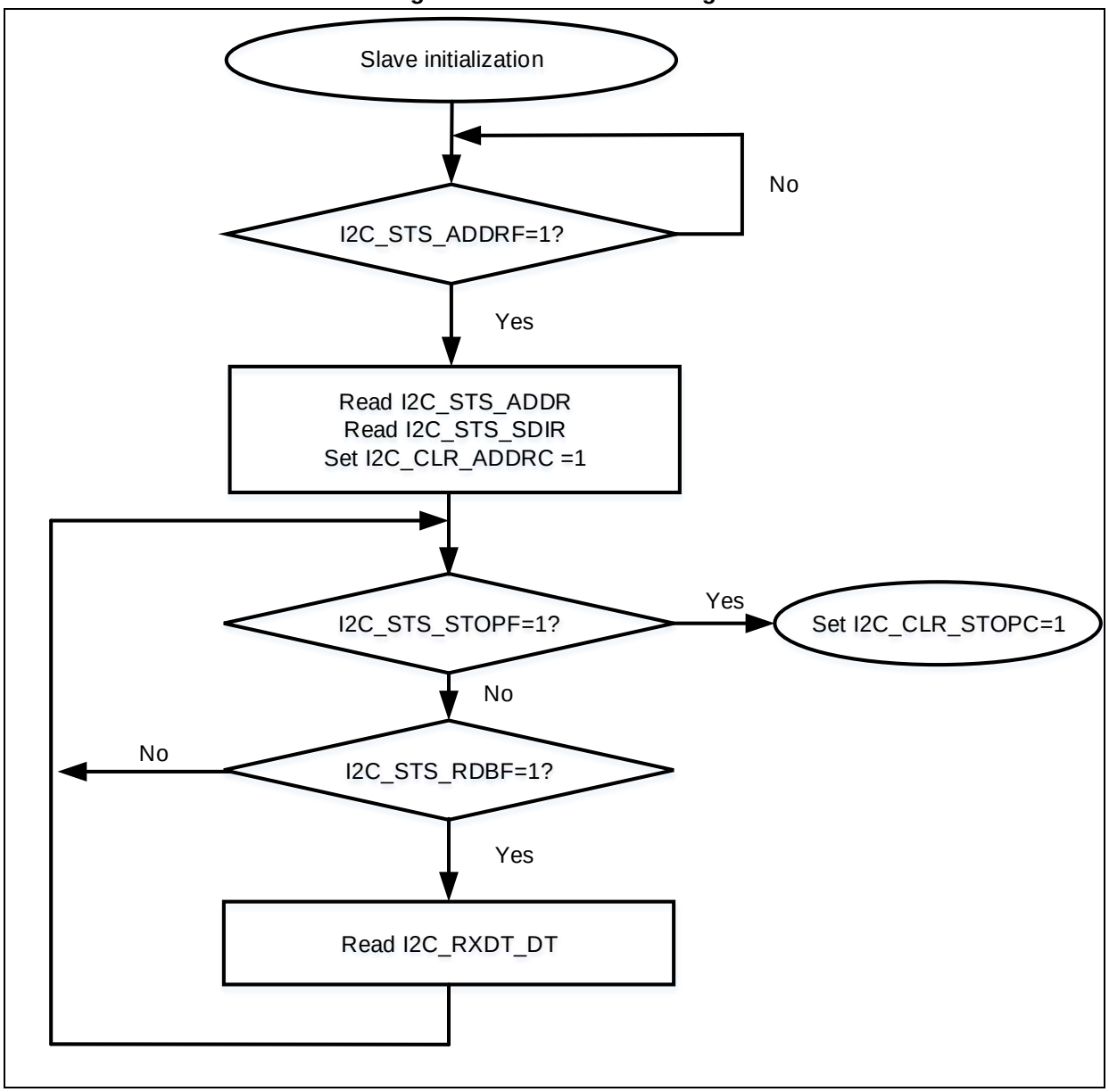
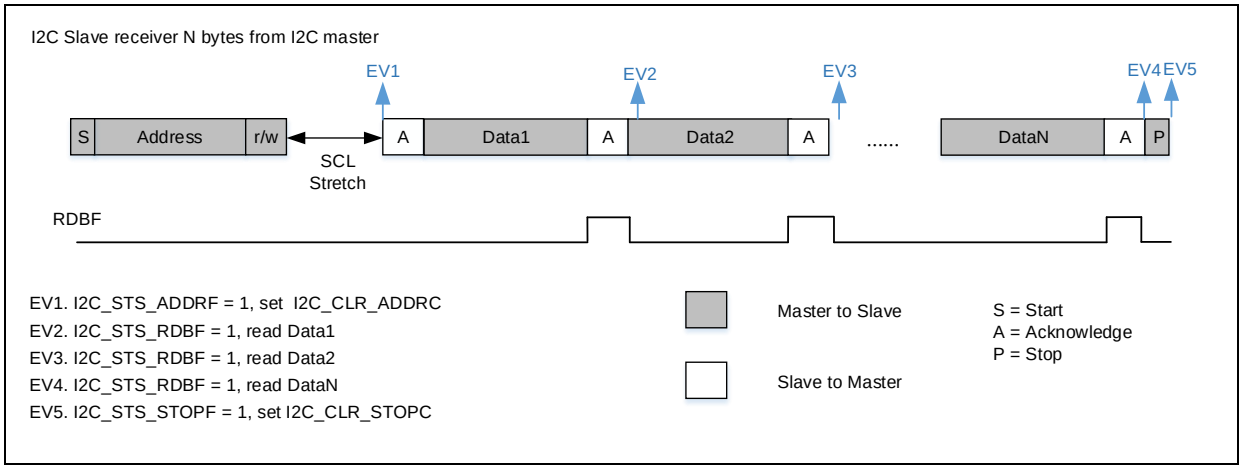


Figure 12. I²C slave receive timing



2.2.6 Slave receive software interface

The slave receive is implemented by independent functions, as shown below.

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

The i2c_slave_receive is an application-level interface function provided by i2c_application.c, and it includes I2C structure pointer, receive data pointer, number of bytes to be received, and function timeout.

Note: This function is a standard slave receive function provided by Artery. Users can write a slave receive function according to the above-mentioned slave receive flow.

3 I²C configuration tool

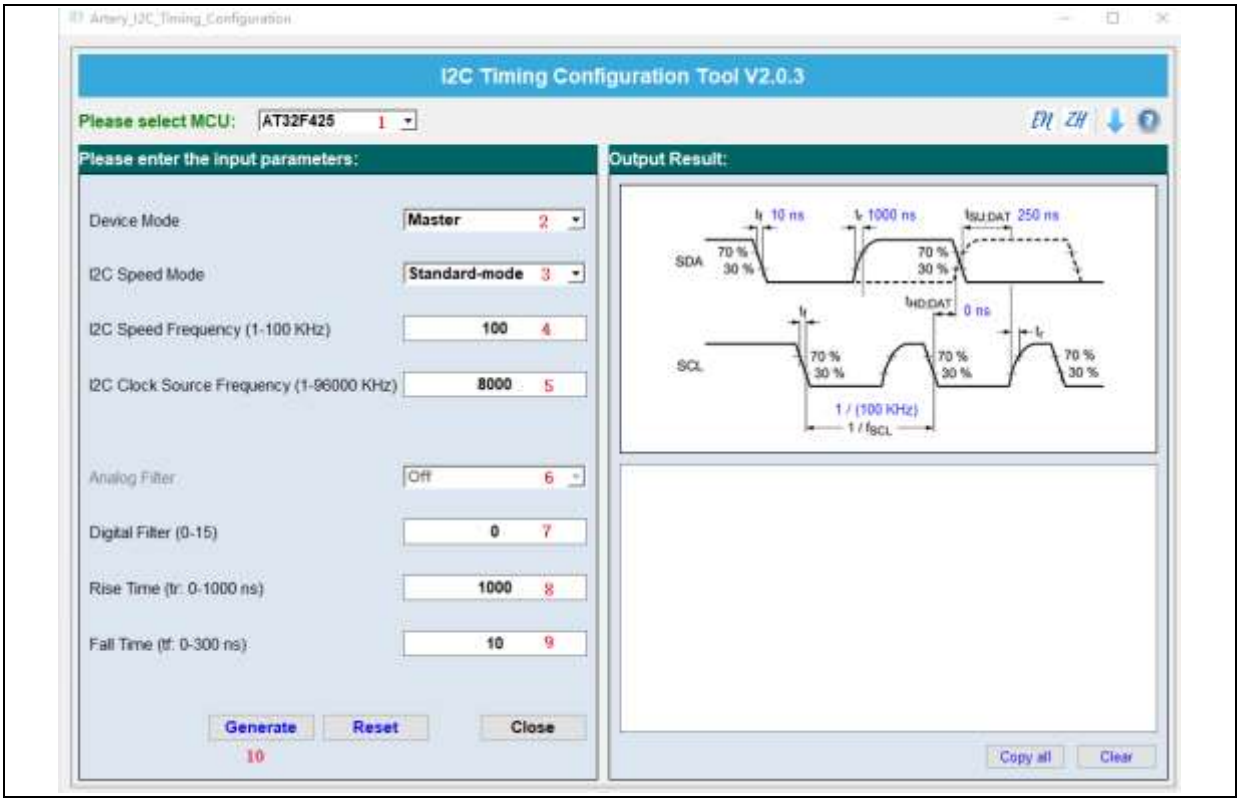
3.1 Function overview

Artery_I2C_Timing_Configuration.exe can be used to configure the master and slave clocks, digital filter and analog filter.

3.2 Resource preparation

1) Software environment: Artery_I2C_Timing_Configuration.exe

Figure 13. Artery I2C Timing Configuration



3.3 Operation procedure

1) Select MCU

   Select the corresponding part number, e.g. AT32F425.

2) Select device mode

   - Master: I²C used as the master

- Slave: I²C used as the slave

3) Select I²C speed mode
   - Standard-mode: 0~100 kHz
   - Fast-mode: 0~400 kHz
   - Fast-mode Plus: 0~1000 kHz
4) Set I²C speed frequency (unit: kHz)

   Set I²C communication frequency. For example, if the communication speed is required to be 10 kHz, key in 10.
5) Set I²C clock source frequency (unit: kHz)

   Set I²C clock source frequency. For example, the I²C clock source of AT32F425 is PCLK1, and when the main frequency is 144 MHz and APB1 is 144 MHz, key in 14400.
6) Enable analog filter
   - On: Analog filter enabled
   - Off: Analog filter disabled

   When the analog filter is enabled, the pulse below 50 ns is filtered.
7) Digital filter (0~15)

   Digital filter time = Digital filter value x T_{I2C_CLK}

   Where, T_{I2C_CLK} = 1 / I2C clock source frequency

   When the value is 0, the digital filter is disabled. When the value is larger than 0, the pulse below the digital filter time is filtered.
8) Rise time (t_r, unit: ns)

   Rising edges of SCL and SDA buses, as shown in Figure 18. I²C protocol specifies the rise time range in standard mode, fast mode and fast mode plus (for details, refer to Table 1). 上升

   Rise time is related to the resistance value of the pull-up resistor. The smaller the pull-up resistance, the shorter the rise time and the faster communication speed, but the higher the power consumption.

   Table 2 listed the rise time corresponding to some commonly used pull-up resistors, which may vary due to the number of devices connected to the bus, wiring, etc. The rise time in Table 2 is for reference only.
9) Fall time (t_f, unit: ns)

   Falling edges of SCL and SDA buses, as shown in Figure 18. I²C protocol specifies the fall time range in standard mode, fast mode and fast mode plus (for details, refer to Table 1).

Figure 14. Rising edge (t_r) and failing edge (t_f)

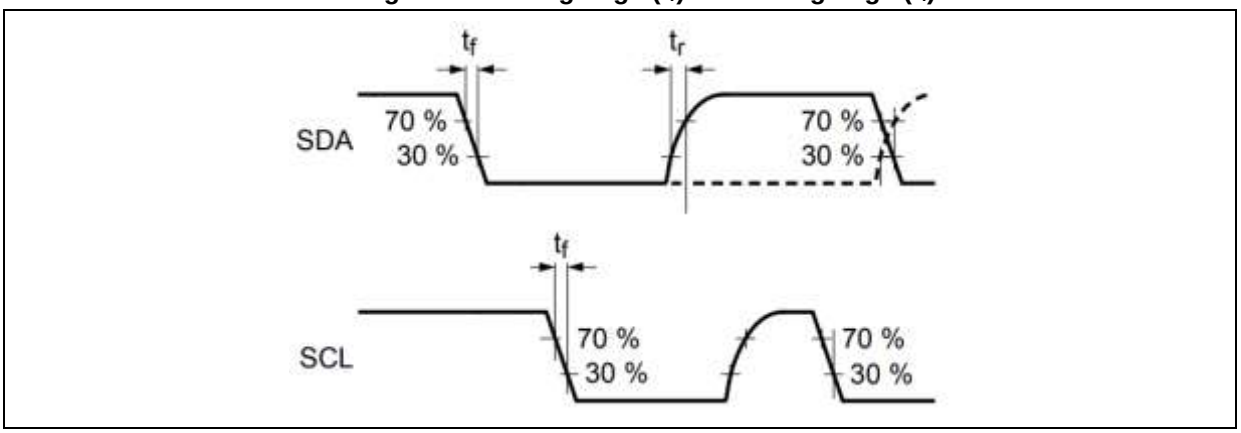


Table 1. Time specification for I²C

Parameter		Standard mode		Fast mode		Fast mode plus	
		Min	Max	Min	Max	Min	Max
f_{SCL}(kHz)	SCL frequency	0	100	0	400	0	1000
tr(ns)	SCL and SDA rising edges	-	1000	-	300	-	120
tf(ns)	SCL and SDA falling edges	-	300	-	300	-	120

Table 2. Reference values of t_r and t_f (VDD=3.3 V)

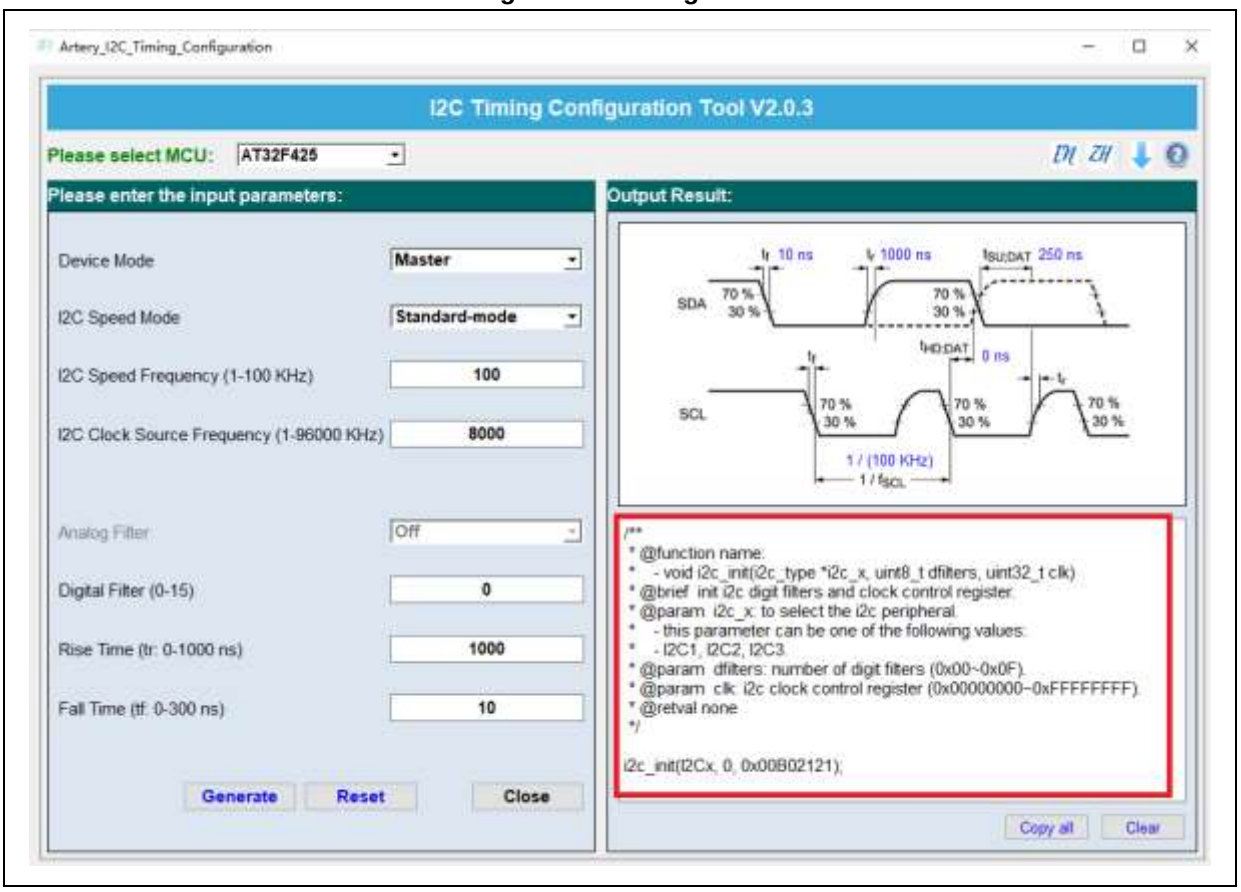
Pull-up resistor (Ω)	Rising edge t_r (ns)	Falling edge t_f (ns)	Recommended max. speed (kHz)
510	100	9	1000
1K	200	8	500
2K	390	8	300
4.7K	960	8	100
10K	1900	8	50

Note: Reference values in Table 2 are tested in the condition that two AT32 MCUs are connected to the bus (one as the master and the other as the slave). The actual values may vary due to the number of devices connected to the bus, wiring, etc.

10) Generate code

Click “Generate”, and the above configured values will be generated in the form of code, as shown in the red box below. Users only need to replace the code output on the right into their own routines.

Figure 15. Code generation



4 EEPROM read/write access

4.1 Function overview

Use hardware I²C interface for EEPROM read/write access.

4.2 Resource preparation

1) Hardware environment

   AT-START BOARD of the corresponding model

   4.7 K pull-up resistor

   EEPROM

2) Software environment

   project\at_start_f4xx\examples\i2c\eeprom

4.3 Software programming

1) Configuration process

   - Enable I²C peripheral clock;
   - Configure I²C multiplexed GPIO;
   - Configure I²C DMA channel;
   - Enable I²C peripheral interface;
   - Write EEPROM and read the data being written;

- Compare the write and read data to check whether it is correct.

2) Code

- main function code

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

  /* System clock initialization */
  system_clock_config();

  /* Configure NVIC priority group */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* at-start board initialization */
  at32_board_init();

  hi2cx.i2cx = I2Cx_PORT;

  /* Configure I2C */
  i2c_config(&hi2cx);

  while(1)
  {
    /* wait for key USER_BUTTON press before starting the communication */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* Write data to 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);

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

    (Part of the code is omitted. See BSP for the complete codes)
```

```
    /* Wait for the completion of communication */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Compare the read and write data */
    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 Test result

- If the write data and read data are exactly the same, LED3 will be on.

5 Communication through polling mode

5.1 Function overview

The I²C interface of two AT-START BOARDs can communicate with each other through polling mode, and test the data transmitted and received by the master or slave.

5.2 Resource preparation

1) Hardware environment

   Two AT-START BOARDs of the corresponding model

   4.7 K pull-up resistor

2) Software environment

   project\at_start_f4xx\examples\i2c\communication_poll

5.3 Software programming

1) Configuration process
   - Enable I²C peripheral clock;
   - Configure I²C multiplexed GPIO;
   - Enable I²C peripheral interface;
   - Slave is ready to receive data;

- Master transmits data;
- Slave is ready to transmit data;
- Master receives data;
- Compare the data transmitted and received to check whether they are correct.

2) Code

- main function code

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

  /* System clock initialization */
  system_clock_config();

  /* Configure NVIC priority group */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* at-start board initialization */
  at32_board_init();

  hi2cx.i2cx = I2Cx_PORT;

  /* Configure I2C */
  i2c_config(&hi2cx);

  while(1)
  {

  #if defined (MASTER_BOARD)

    /* Wait for pressing USER_BUTTON */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* Master transmits data */
    if((i2c_status = i2c_master_transmit(&hi2cx, I2Cx_ADDRESS, tx_buf, BUF_SIZE, I2C_TIMEOUT)) !=
I2C_OK)
    {
      error_handler(i2c_status);
    }

    delay_ms(10);

    /* Master receives data */
    if((i2c_status = i2c_master_receive(&hi2cx, I2Cx_ADDRESS, rx_buf, BUF_SIZE, I2C_TIMEOUT)) !=
```

```
I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Master compares the read and write data */
    if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
    {
      at32_led_on(LED3);
    }
    else
    {
      error_handler(i2c_status);
    }
  #else

    /* Wait for pressing USER_BUTTON */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* Slave receives data */
    if((i2c_status = i2c_slave_receive(&hi2cx, rx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Slave transmits data */
    if((i2c_status = i2c_slave_transmit(&hi2cx, tx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Slave compares the read and write data */
    if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
    {
      at32_led_on(LED3);
    }
    else
    {
      error_handler(i2c_status);
    }
  #endif

  }
}
```

5.4 Test result

- Set the master-slave relationship of two boards through the macro definition #define MASTER_BOARD;
- If the read and write data of the master or slave are exactly the same, LED3 will be on; otherwise, LED2 will flash.

6 Communication through interrupt mode

6.1 Function overview

The I²C interface of two AT-START BOARDs can communicate with each other through interrupt mode, and test the data transmitted and received by the master or slave.

6.2 Resource preparation

1) Hardware environment

   Two AT-START BOARDs of the corresponding model

   4.7 K pull-up resistor

2) Software environment

   project\at_start_f4xx\examples\i2c\communication_int

6.3 Software programming

1) Configuration process
   - Enable I²C peripheral clock;
   - Configure I²C multiplexed GPIO;
   - Enable I²C peripheral interface;
   - Enable I²C interrupt;
   - Slave is ready to receive data;
   - Master transmits data;
   - Slave is ready to transmit data;
   - Master receives data;
   - Compare the data transmitted and received to check whether they are correct.
2) Code
   - main function code

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

  /* System clock initialization */
  system_clock_config();

  /* Configure NVICpriority group */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
```

```
  /* at-start board initialization */
  at32_board_init();

  hi2cx.i2cx = I2Cx_PORT;

  /* Configure I2C */
  i2c_config(&hi2cx);

  while(1)
  {

  #if defined (MASTER_BOARD)

    /* Wait for pressing USER_BUTTON */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* Master transmits data */
    if((i2c_status = i2c_master_transmit_int(&hi2cx, I2Cx_ADDRESS, tx_buf, BUF_SIZE,
I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Wait for completion */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    delay_ms(10);

    /* Master receives data */
    if((i2c_status = i2c_master_receive_int(&hi2cx, I2Cx_ADDRESS, rx_buf, BUF_SIZE,
I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Wait for completion */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }
```

```
  /* Master compares read and write data */
  if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
  {
    at32_led_on(LED3);
  }
  else
  {
    error_handler(i2c_status);
  }

#else

  /* Wait for pressing USER_BUTTON */
  while(at32_button_press() != USER_BUTTON)
  {
  }

  /* Slave receives data */
  if((i2c_status = i2c_slave_receive_int(&hi2cx, rx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* Wait for completion */
  if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* Slave transmits data */
  if((i2c_status = i2c_slave_transmit_int(&hi2cx, tx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* Wait for completion */
  if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* Slave compares read and write data */
  if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
  {
    at32_led_on(LED3);
```

```
    }
    else
    {
      error_handler(i2c_status);
    }
  #endif

  }
}
```

■ Master interrupt handler code

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

    /* Refresh TXDT register */
    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)
  {
    /* Send data */
    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))
    {
      /* Continue transfer */
      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)
```

```
  {
    /* Receive data */
    (*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)
      {
        /* Generate a STOP condition */
        i2c_stop_generate(hi2c->i2cx);
      }
    }
    else
    {
      return I2C_ERR_TDC;
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_STOPF_FLAG) != RESET)
  {
    /* Clear STOP flag */
    i2c_flag_clear(hi2c->i2cx, I2C_STOPF_FLAG);

    /* Reset ctrl2 register */
    i2c_reset_ctrl2_register(hi2c);

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

    /* Refresh TXDT register */
    i2c_refresh_txdt_register(hi2c);

    /* Disable interrupt */
    i2c_interrupt_enable(hi2c->i2cx, I2C_ERR_INT | I2C_TDC_INT | I2C_STOP_INT | I2C_ACKFIAL_INT |
I2C_TD_INT | I2C_RD_INT, FALSE);

    /* Transfer complete */
    hi2c->status = I2C_END;
  }
```

```
  return I2C_OK;
}
```

■ Slave interrupt handler code

```
i2c_status_type i2c_slave_irq_handler_int(i2c_handle_type* hi2c)
{
  if (i2c_flag_get(hi2c->i2cx, I2C_ACKFAIL_FLAG) != RESET)
  {
    /* Transfer complete */
    if (hi2c->pcount == 0)
    {
      i2c_refresh_txdt_register(hi2c);

      /* Clear ackfail flag */
      i2c_flag_clear(hi2c->i2cx, I2C_ACKFAIL_FLAG);
    }
    /* the transfer has not been completed */
    else
    {
      /* Clear ackfail flag */
      i2c_flag_clear(hi2c->i2cx, I2C_ACKFAIL_FLAG);
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_ADDRF_FLAG) != RESET)
  {
    /* Clear addrf flag */
    i2c_flag_clear(hi2c->i2cx, I2C_ADDRF_FLAG);
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_TDIS_FLAG) != RESET)
  {
    if (hi2c->pcount > 0)
    {
      /* Transmit data */
      hi2c->i2cx->txdt = (*(hi2c->pbuff++));
      hi2c->psize--;
      hi2c->pcount--;
    }
  }
  else if (i2c_flag_get(hi2c->i2cx, I2C_RDBF_FLAG) != RESET)
  {
    if (hi2c->pcount > 0)
    {
      /* Receive data */
      (*hi2c->pbuff++) = i2c_data_receive(hi2c->i2cx);
      hi2c->pcount--;
      hi2c->psize--;
```

```
      }
    }
    else if (i2c_flag_get(hi2c->i2cx, I2C_STOPF_FLAG) != RESET)
    {
      /* Clear STOP condition */
      i2c_flag_clear(hi2c->i2cx, I2C_STOPF_FLAG);

      /* Disable interrupt */
      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);

      /* Reset ctrl2 register */
      i2c_reset_ctrl2_register(hi2c);

      /* Refresh TXDT register */
      i2c_refresh_txdt_register(hi2c);

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

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

      /* Transfer complete */
      hi2c->status = I2C_END;
    }

    return I2C_OK;
}
```

6.4 Test result

- Set the master-slave relationship of two boards through the macro definition #define MASTER_BOARD.
- If the read and write data of the master or slave are exactly the same, LED3 will be on; otherwise, LED2 will flash.

7 Communication through DMA mode

7.1 Function overview

The I²C interface of two AT-START BOARDs can communicate with each other through DMA mode, and test the data transmitted and received by the master or slave.

7.2 Resource preparation

1) Hardware environment

   Two AT-START BOARDs of the corresponding model

   4.7 K pull-up resistor

2) Software environment

   project\at_start_f4xx\examples\ i2c\communication_dma

7.3 Software programming

1) Configuration process

   - Enable I²C peripheral clock;
   - Configure I²C multiplexed GPIO;
   - Configure I²C DMA channel;
   - Enable I²C peripheral interface;
   - Slave is ready to receive data;
   - Master transmits data;
   - Slave is ready to transmit data;
   - Master receives data;
   - Compare the data transmitted and received to check whether they are correct.

2) Code

   - main function code

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

  /* System clock initialization */
  system_clock_config();

  /* Configure NVIC priority group */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* at-start board initialization */
  at32_board_init();

  hi2cx.i2cx = I2Cx_PORT;

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

```
  while(1)
  {

  #if defined (MASTER_BOARD)

    /* Wait for pressing USER_BUTTON */
    while(at32_button_press() != USER_BUTTON)
    {
    }

    /* Master transmits data */
    if((i2c_status = i2c_master_transmit_dma(&hi2cx, I2Cx_ADDRESS, tx_buf, BUF_SIZE,
I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Wait for completion */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    delay_ms(10);

    /* Master receives data */
    if((i2c_status = i2c_master_receive_dma(&hi2cx, I2Cx_ADDRESS, rx_buf, BUF_SIZE,
I2C_TIMEOUT)) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Wait for completion */
    if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
    {
      error_handler(i2c_status);
    }

    /* Master compares read and write data */
    if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
    {
      at32_led_on(LED3);
    }
    else
    {
```

```
    error_handler(i2c_status);
  }

#else

  /* Wait for pressing USER_BUTTON */
  while(at32_button_press() != USER_BUTTON)
  {
  }

  /* Slave receives data */
  if((i2c_status = i2c_slave_receive_dma(&hi2cx, rx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* Wait for completion */
  if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* Slave transmits data */
  if((i2c_status = i2c_slave_transmit_dma(&hi2cx, tx_buf, BUF_SIZE, I2C_TIMEOUT)) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* Wait for completion */
  if(i2c_wait_end(&hi2cx, I2C_TIMEOUT) != I2C_OK)
  {
    error_handler(i2c_status);
  }

  /* Slave compares read and write data */
  if(buffer_compare(tx_buf, rx_buf, BUF_SIZE) == 0)
  {
    at32_led_on(LED3);
  }
  else
  {
    error_handler(i2c_status);
  }

#endif
```

```
  }
}
```

■ Master DMA transfer complete interrupt handler code

```
void i2c_dma_tx_rx_irq_handler(i2c_handle_type* hi2c, dma_channel_type* dma_channel)
{
  /* Transfer complete */
  if (dma_flag_get(DMA_GET_TC_FLAG(dma_channel)) != RESET)
  {
    /* Disable DMA transfer complete interrupt */
    dma_interrupt_enable(dma_channel, DMA_FDT_INT, FALSE);

    /* Clear transfer complete flag */
    dma_flag_clear(DMA_GET_TC_FLAG(dma_channel));

    /* Disable DMA request */
    i2c_dma_enable(hi2c->i2cx, DMA_GET_REQUEST(dma_channel), FALSE);

    /* Disable DMA channel */
    dma_channel_enable(dma_channel, FALSE);

    switch(hi2c->mode)
    {
      case I2C_DMA_MA_TX:
      case I2C_DMA_MA_RX:
      {
        /* Update the number of bytes transferred */
        hi2c->pcount -= hi2c->psize;

        /* Transfer complete */
        if (hi2c->pcount == 0)
        {
          /* Enable STOP interrupt */
          i2c_interrupt_enable(hi2c->i2cx, I2C_STOP_INT, TRUE);
        }
        /* Transfer not complete */
        else
        {
          /* Update transfer buffer pointer */
          hi2c->pbuff += hi2c->psize;

          /* Set the number of bytes transferred */
          if (hi2c->pcount > MAX_TRANSFER_CNT)
          {
            hi2c->psize = MAX_TRANSFER_CNT;
          }
```

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

            /* Configure DMA channel and continue transfer */
            i2c_dma_config(hi2c, dma_channel, hi2c->pbuff, hi2c->psize);

            /* Enable TDC interrupt */
            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 Test result

- Set the master-slave relationship of two boards through the macro definition #define MASTER_BOAR.
- If the read and write data of the master or slave are exactly the same, LED3 will be on; otherwise, LED2 will flash.

8 Revision history

Table 3. Document revision history

Date	Version	Revision note
2022.01.21	2.0.0	Initial release
2022.08.22	2.0.1	Modified Figure 13.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services; ARTERY assumes no liability for purchasers' selection or use of the products and the relevant services.

No license, express or implied, to any intellectual property right is granted by ARTERY herein regardless of the existence of any previous representation in any forms. If any part of this document involves third party's products or services, it does NOT imply that ARTERY authorizes the use of the third party's products or services, or permits any of the intellectual property, or guarantees any uses of the third party's products or services or intellectual property in any way.

Except as provided in ARTERY's terms and conditions of sale for such products, ARTERY disclaims any express or implied warranty, relating to use and/or sale of the products, including but not restricted to liability or warranties relating to merchantability, fitness for a particular purpose (based on the corresponding legal situation in any unjudicial districts), or infringement of any patent, copyright, or other intellectual property right.

ARTERY's products are not designed for the following purposes, and thus not intended for the following uses: (A) Applications that have specific requirements on safety, for example: life-support applications, active implant devices, or systems that have specific requirements on product function safety; (B) Aviation applications; (C) Auto-motive application or environment; (D) Aerospace applications or environment, and/or (E) weapons. Since ARTERY products are not intended for the above-mentioned purposes, if purchasers apply ARTERY products to these purposes, purchasers are solely responsible for any consequences or risks caused, even if any written notice is sent to ARTERY by purchasers; in addition, purchasers are solely responsible for the compliance with all statutory and regulatory requirements regarding these uses.

Any inconsistency of the sold ARTERY products with the statement and/or technical features specification described in this document will immediately cause the invalidity of any warranty granted by ARTERY products or services stated in this document by ARTERY, and ARTERY disclaims any responsibility in any form.