# 28 I2C Controller (I2C)

The I2C (Inter-Integrated Circuit) bus allows ESP32-C3 to communicate with multiple external devices. These external devices can share one bus.

## 28.1 Overview

The I2C bus has two lines, namely a serial data line (SDA) and a serial clock line (SCL). Both SDA and SCL lines are open-drain. The I2C bus can be connected to a single or multiple master devices and a single or multiple slave devices. However, only one master device can access a slave at a time via the bus.

The master initiates communication by generating a START condition: pulling the SDA line low while SCL is high, and sending nine clock pulses via SCL. The first eight pulses are used to transmit a 7-bit address followed by a read/write ($R/\overline{W}$) bit. If the address of an I2C slave matches the 7-bit address transmitted, this matching slave can respond by pulling SDA low on the ninth clock pulse. The master and the slave can send or receive data according to the $R/\overline{W}$ bit. Whether to terminate the data transfer or not is determined by the logic level of the acknowledge (ACK) bit. During data transfer, SDA changes only when SCL is low. Once finishing communication, the master sends a STOP condition: pulling SDA up while SCL is high. If a master both reads and writes data in one transfer, then it should send a RSTART condition, a slave address and a $R/\overline{W}$ bit before changing its operation. The RSTART condition is used to change the transfer direction and the mode of the devices (master mode or slave mode).

## 28.2 Features

The I2C controller has the following features:

- Master mode and slave mode
- Communication between multiple slaves
- Standard mode (100 Kbit/s)
- Fast mode (400 Kbit/s)
- 7-bit addressing and 10-bit addressing
- Continuous data transfer achieved by pulling SCL low
- Programmable digital noise filtering
- Double addressing mode, which uses slave address and slave memory or register address

## 28.3 I2C Architecture

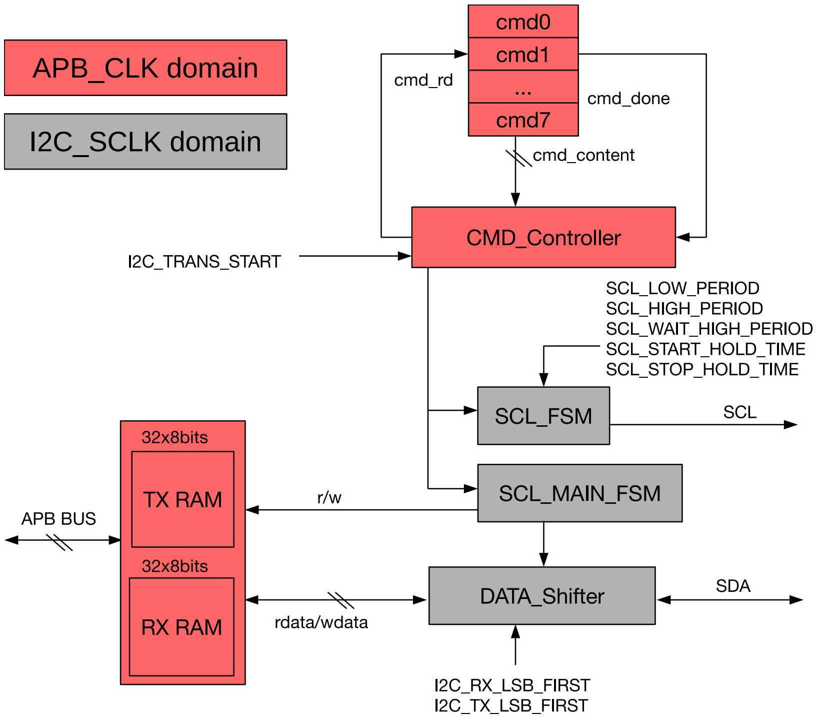


Figure 28-1. I2C Master Architecture

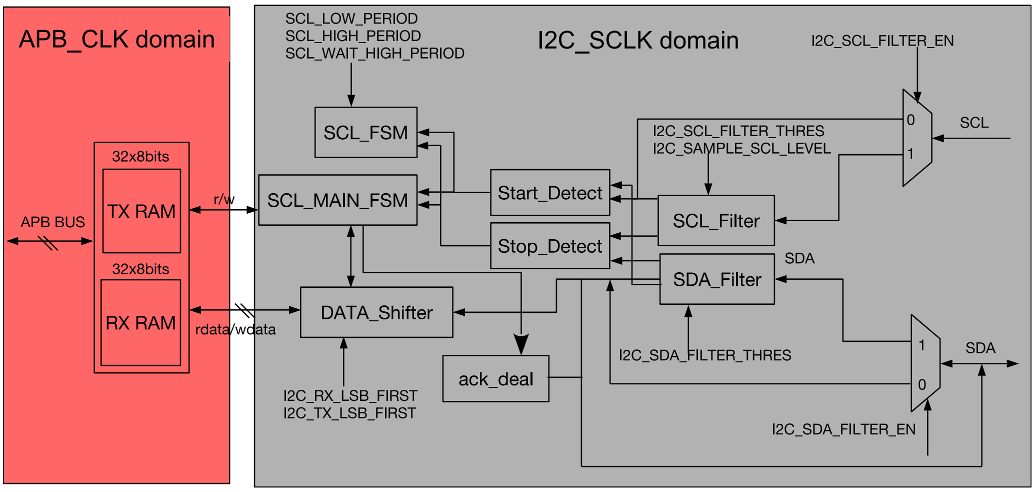


Figure 28-2. I2C Slave Architecture

The I2C controller runs either in master mode or slave mode, which is determined by I2C_MS_MODE. Figure 28-1 shows the architecture of a master, while Figure 28-2 shows that of a slave. The I2C controller has the following main parts:

- transmit and receive memory (TX/RX RAM)
- command controller (CMD_Controller)
- SCL clock controller (SCL_FSM)
- SDA data controller (SCL_MAIN_FSM)

- serial/parallel data converter (DATA_Shifter)
- filter for SCL (SCL_Filter)
- filter for SDA (SDA_Filter)

Besides, the I2C controller also has a clock module which generates I2C clocks, and a synchronization module which synchronizes the APB bus and the I2C controller.

The clock module is used to select clock sources, turn on and off clocks, and divide clocks. SCL_Filter and SDA_Filter remove noises on SCL input signals and SDA input signals respectively. The synchronization module synchronizes signal transfer between different clock domains.

Figure 28-3 and Figure 28-4 are the timing diagram and corresponding parameters of the I2C protocol. SCL_FSM generates the timing sequence conforming to the I2C protocol.

SCL_MAIN_FSM controls the execution of I2C commands and the sequence of the SDA line. CMD_Controller is used for an I2C master to generate (R)START, STOP, WRITE, READ and END commands. TX RAM and RX RAM store data to be transmitted and data received respectively. DATA_Shifter shifts data between serial and parallel form.

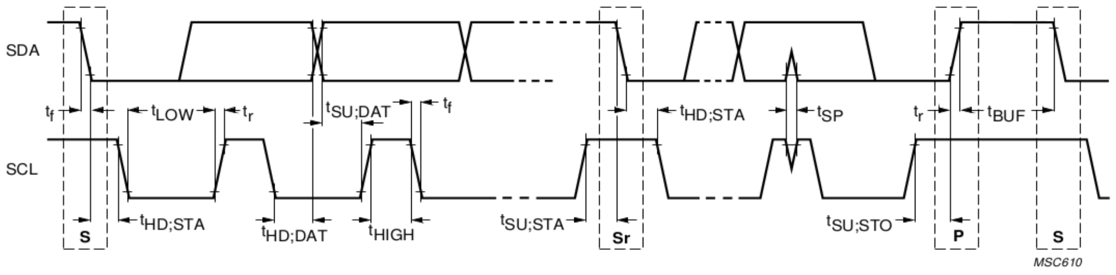
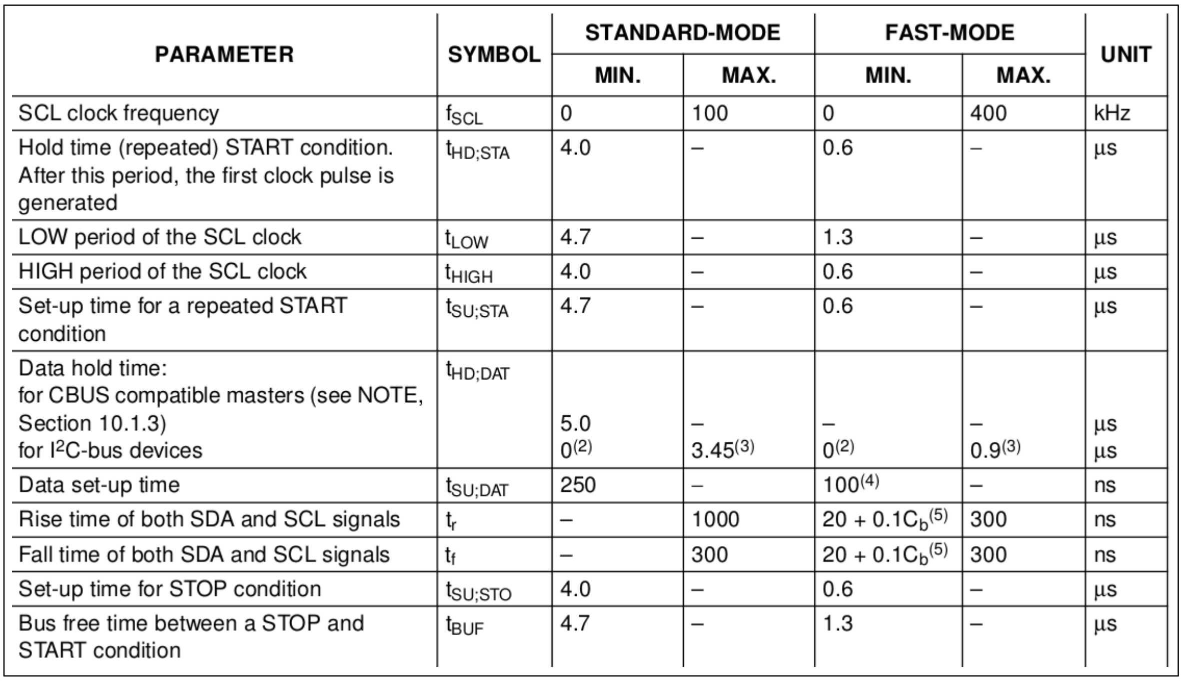


Fig.31 Definition of timing for F/S-mode devices on the I²C-bus.

**Figure 28-3. I2C Protocol Timing (Cited from Fig.31 in The I2C-bus specification Version 2.1)**



| PARAMETER | SYMBOL | STANDARD-MODE | | FAST-MODE | | UNIT |
|---|---|---|---|---|---|---|
| | | MIN. | MAX. | MIN. | MAX. | |
| SCL clock frequency | $f_{SCL}$ | 0 | 100 | 0 | 400 | kHz |
| Hold time (repeated) START condition. After this period, the first clock pulse is generated | $t_{HD;STA}$ | 4.0 | – | 0.6 | – | µs |
| LOW period of the SCL clock | $t_{LOW}$ | 4.7 | – | 1.3 | – | µs |
| HIGH period of the SCL clock | $t_{HIGH}$ | 4.0 | – | 0.6 | – | µs |
| Set-up time for a repeated START condition | $t_{SU;STA}$ | 4.7 | – | 0.6 | – | µs |
| Data hold time:<br>for CBUS compatible masters (see NOTE, Section 10.1.3)<br>for $I^2C$-bus devices | $t_{HD;DAT}$ | 5.0<br>0[2] | –<br>3.45[3] | –<br>0[2] | –<br>0.9[3] | µs<br>µs |
| Data set-up time | $t_{SU;DAT}$ | 250 | – | 100[4] | – | ns |
| Rise time of both SDA and SCL signals | $t_r$ | – | 1000 | $20 + 0.1C_b$[5] | 300 | ns |
| Fall time of both SDA and SCL signals | $t_f$ | – | 300 | $20 + 0.1C_b$[5] | 300 | ns |
| Set-up time for STOP condition | $t_{SU;STO}$ | 4.0 | – | 0.6 | – | µs |
| Bus free time between a STOP and START condition | $t_{BUF}$ | 4.7 | – | 1.3 | – | µs |

Figure 28-4. I2C Timing Parameters (Cited from Table 5 in The I2C-bus specification Version 2.1)

## 28.4 Functional Description

Note that operations may differ between the I2C controller in ESP32-C3 and other masters or slaves on the bus. Please refer to datasheets of individual I2C devices for specific information.

### 28.4.1 Clock Configuration

Registers, TX RAM, and RX RAM are configured and accessed in the APB_CLK clock domain, whose frequency is $1 \sim 80$ MHz. The main logic of the I2C controller, including SCL_FSM, SCL_MAIN_FSM, SCL_FILTER, SDA_FILTER, and DATA_SHIFTER, are in the I2C_SCLK clock domain.

You can choose the clock source for I2C_SCLK from XTAL_CLK or RC_FAST_CLK via I2C_SCLK_SEL. When I2C_SCLK_SEL is cleared, the clock source is XTAL_CLK. When I2C_SCLK_SEL is set, the clock source is RC_FAST_CLK. The clock source is enabled by configuring I2C_SCLK_ACTIVE as high level, and then passes through a fractional divider to generate I2C_SCLK according to the following equation:

$$Divisor = I2C\_SCLK\_DIV\_NUM + 1 + \frac{I2C\_SCLK\_DIV\_A}{I2C\_SCLK\_DIV\_B}$$

The frequency of XTAL_CLK is 40 MHz, while the frequency of RC_FAST_CLK is 17.5 MHz. Limited by timing parameters, the derived clock I2C_SCLK should operate at a frequency 20 timers larger than SCL's frequency.

### 28.4.2 SCL and SDA Noise Filtering

SCL_Filter and SDA_Filter modules are identical and are used to filter signal noises on SCL and SDA, respectively. These filters can be enabled or disabled by configuring I2C_SCL_FILTER_EN and I2C_SDA_FILTER_EN.

Take SCL_Filter as an example. When enabled, SCL_Filter samples input signals on the SCL line continuously. These input signals are valid only if they remain unchanged for consecutive I2C_SCL_FILTER_THRES

I2C_SCLK clock cycles. Given that only valid input signals can pass through the filter, SCL_Filter can remove glitches whose pulse width is shorter than I2C_SCL_FILTER_THRES I2C_SCLK clock cycles, while SDA_Filter can remove glitches whose pulse width is shorter than I2C_SDA_FILTER_THRES I2C_SCLK clock cycles.

### 28.4.3 SCL Clock Stretching

The I2C controller in slave mode (i.e. slave) can hold the SCL line low in exchange for more time to process data. This function called clock stretching is enabled by setting the I2C_SLAVE_SCL_STRETCH_EN bit. The time period to release the SCL line from stretching is configured by setting the I2C_STRETCH_PROTECT_NUM field, in order to avoid timing sequence errors. The slave will hold the SCL line low when one of the following four events occurs:

1. Address match: The address of the slave matches the address sent by the master via the SDA line, and the $R/\overline{W}$ bit is 1.

2. RAM being full: RX RAM of the slave is full. Note that when the slave receives less than 32 bytes, it is not necessary to enable clock stretching; when the slave receives 32 bytes or more, you may interrupt data transmission to wrapped around RAM via the FIFO threshold, or enable clock stretching for more time to process data. When clock stretching is enabled, I2C_RX_FULL_ACK_LEVEL must be cleared, otherwise there will be unpredictable consequences.

3. RAM being empty: The slave is sending data, but its TX RAM is empty.

4. Sending an ACK: If I2C_SLAVE_BYTE_ACK_CTL_EN is set, the slave pulls SCL low when sending an ACK bit. At this stage, software validates data and configures I2C_SLAVE_BYTE_ACK_LVL to control the level of the ACK bit. Note that when RX RAM of the slave is full, the level of the ACK bit to be sent is determined by I2C_RX_FULL_ACK_LEVEL, instead of I2C_SLAVE_BYTE_ACK_LVL. In this case, I2C_RX_FULL_ACK_LEVEL should also be cleared to ensure proper functioning of clock stretching.

After SCL has been stretched low, the cause of stretching can be read from the I2C_STRETCH_CAUSE bit. Clock stretching is disabled by setting the I2C_SLAVE_SCL_STRETCH_CLR bit.

### 28.4.4 Generating SCL Pulses in Idle State

Usually when the I2C bus is idle, the SCL line is held high. The I2C controller in ESP32-C3 can be programmed to generate SCL pulses in idle state. This function only works when the I2C controller is configured as master. If the I2C_SCL_RST_SLV_EN bit is set, hardware will send I2C_SCL_RST_SLV_NUM SCL pulses. When software reads 0 in I2C_SCL_RST_SLV_EN, set I2C_CONF_UPGATE to stop this function.

### 28.4.5 Synchronization

I2C registers are configured in APB_CLK domain, whereas the I2C controller is configured in asynchronous I2C_SCLK domain. Therefore, before being used by the I2C controller, register values should be synchronized by first writing configuration registers and then writing 1 to I2C_CONF_UPGATE. Registers that need synchronization are listed in Table 28-1.

Table 28-1. I2C Synchronous Registers

| Register | Parameter | Address |
|---|---|---|

| | | |
|---|---|---|
| I2C_CTR_REG | I2C_SLV_TX_AUTO_START_EN | 0x0004 |
| | I2C_ADDR_10BIT_RW_CHECK_EN | |
| | I2C_ADDR_BROADCASTING_EN | |
| | I2C_SDA_FORCE_OUT | |
| | I2C_SCL_FORCE_OUT | |
| | I2C_SAMPLE_SCL_LEVEL | |
| | I2C_RX_FULL_ACK_LEVEL | |
| | I2C_MS_MODE | |
| | I2C_TX_LSB_FIRST | |
| | I2C_RX_LSB_FIRST | |
| | I2C_ARBITRATION_EN | |
| I2C_TO_REG | I2C_TIME_OUT_EN | 0x000C |
| | I2C_TIME_OUT_VALUE | |
| I2C_SLAVE_ADDR_REG | I2C_ADDR_10BIT_EN | 0x0010 |
| | I2C_SLAVE_ADDR | |
| I2C_FIFO_CONF_REG | I2C_FIFO_ADDR_CFG_EN | 0x0018 |
| I2C_SCL_SP_CONF_REG | I2C_SDA_PD_EN | 0x0080 |
| | I2C_SCL_PD_EN | |
| | I2C_SCL_RST_SLV_NUM | |
| | I2C_SCL_RST_SLV_EN | |
| I2C_SCL_STRETCH_CONF_REG | I2C_SLAVE_BYTE_ACK_CTL_EN | 0x0084 |
| | I2C_SLAVE_BYTE_ACK_LVL | |
| | I2C_SLAVE_SCL_STRETCH_EN | |
| | I2C_STRETCH_PROTECT_NUM | |
| I2C_SCL_LOW_PERIOD_REG | I2C_SCL_LOW_PERIOD | 0x0000 |
| I2C_SCL_HIGH_PERIOD_REG | I2C_WAIT_HIGH_PERIOD | 0x0038 |
| | I2C_HIGH_PERIOD | |
| I2C_SDA_HOLD_REG | I2C_SDA_HOLD_TIME | 0x0030 |
| I2C_SDA_SAMPLE_REG | I2C_SDA_SAMPLE_TIME | 0x0034 |
| I2C_SCL_START_HOLD_REG | I2C_SCL_START_HOLD_TIME | 0x0040 |
| I2C_SCL_RSTART_SETUP_REG | I2C_SCL_RSTART_SETUP_TIME | 0x0044 |
| I2C_SCL_STOP_HOLD_REG | I2C_SCL_STOP_HOLD_TIME | 0x0048 |
| I2C_SCL_STOP_SETUP_REG | I2C_SCL_STOP_SETUP_TIME | 0x004C |
| I2C_SCL_ST_TIME_OUT_REG | I2C_SCL_ST_TO_I2C | 0x0078 |
| I2C_SCL_MAIN_ST_TIME_OUT_REG | I2C_SCL_MAIN_ST_TO_I2C | 0x007C |
| I2C_FILTER_CFG_REG | I2C_SCL_FILTER_EN | 0x0050 |
| | I2C_SCL_FILTER_THRES | |
| | I2C_SDA_FILTER_EN | |
| | I2C_SDA_FILTER_THRES | |

### 28.4.6 Open-Drain Output

SCL and SDA output drivers must be configured as open drain. There are two ways to achieve this:

1. Set I2C_SCL_FORCE_OUT and I2C_SDA_FORCE_OUT, and configure GPIO_PIN*n*_PAD_DRIVER for corresponding SCL and SDA pads as open-drain.

2. Clear I2C_SCL_FORCE_OUT and I2C_SDA_FORCE_OUT.

Because these lines are configured as open-drain, the low-to-high transition time of each line is longer, determined together by the pull-up resistor and the line capacitance. The output duty cycle of I2C is limited by the SDA and SCL line's pull-up speed, mainly SCL's speed.

In addition, when I2C_SCL_FORCE_OUT and I2C_SCL_PD_EN are set to 1, SCL can be forced low; when I2C_SDA_FORCE_OUT and I2C_SDA_PD_EN are set to 1, SDA can be forced low.

### 28.4.7 Timing Parameter Configuration

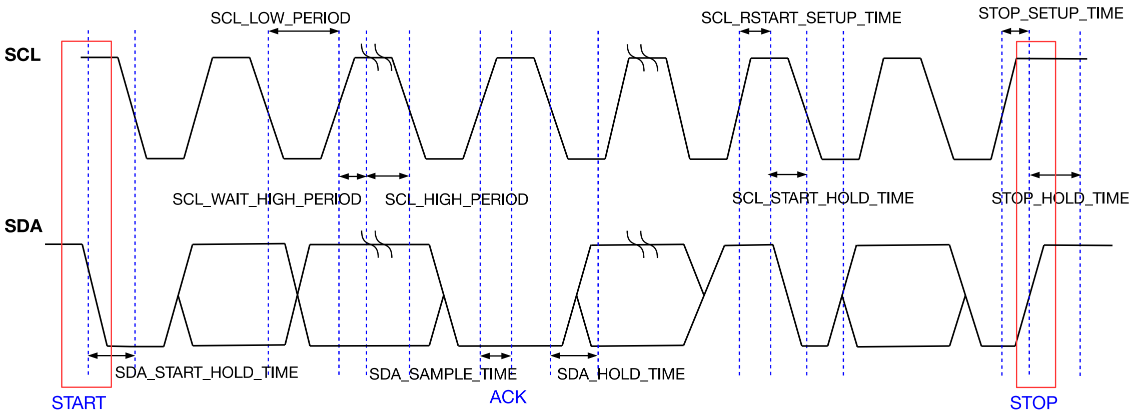


Figure 28-5. I2C Timing Diagram

Figure 28-5 shows the timing diagram of an I2C master. This figure also specifies registers used to configure the START bit, STOP bit, data hold time, data sample time, waiting time on the rising SCL edge, etc. Timing parameters are calculated as follows in I2C_SCLK clock cycles:

1. $t_{LOW} = (I2C\_SCL\_LOW\_PERIOD + 1) \cdot T_{I2C\_SCLK}$
2. $t_{HIGH} = (I2C\_SCL\_HIGH\_PERIOD + 1) \cdot T_{I2C\_SCLK}$
3. $t_{SU:STA} = (I2C\_SCL\_RSTART\_SETUP\_TIME + 1) \cdot T_{I2C\_SCLK}$
4. $t_{HD:STA} = (I2C\_SCL\_START\_HOLD\_TIME + 1) \cdot T_{I2C\_SCLK}$
5. $t_r = (I2C\_SCL\_WAIT\_HIGH\_PERIOD + 1) \cdot T_{I2C\_SCLK}$
6. $t_{SU:STO} = (I2C\_SCL\_STOP\_SETUP\_TIME + 1) \cdot T_{I2C\_SCLK}$
7. $t_{BUF} = (I2C\_SCL\_STOP\_HOLD\_TIME + 1) \cdot T_{I2C\_SCLK}$
8. $t_{HD:DAT} = (I2C\_SDA\_HOLD\_TIME + 1) \cdot T_{I2C\_SCLK}$
9. $t_{SU:DAT} = (I2C\_SCL\_LOW\_PERIOD - I2C\_SDA\_HOLD\_TIME) \cdot T_{I2C\_SCLK}$

Timing registers below are divided into two groups, depending on the mode in which these registers are active:

- Master mode only:
  1. I2C_SCL_START_HOLD_TIME: Specifies the interval between pulling SDA low and pulling SCL low when the master generates a START condition. This interval is (I2C_SCL_START_HOLD_TIME +1) in I2C_SCLK cycles. This register is active only when the I2C controller works in master mode.

2. I2C_SCL_LOW_PERIOD: Specifies the low period of SCL. This period lasts (I2C_SCL_LOW_PERIOD + 1) in I2C_SCLK cycles. However, it could be extended when SCL is pulled low by peripheral devices or by an END command executed by the I2C controller, or when the clock is stretched. This register is active only when the I2C controller works in master mode.
3. I2C_SCL_WAIT_HIGH_PERIOD: Specifies time for SCL to go high in I2C_SCLK cycles. Please make sure that SCL could be pulled high within this time period. Otherwise, the high period of SCL may be incorrect. This register is active only when the I2C controller works in master mode.
4. I2C_SCL_HIGH_PERIOD: Specifies the high period of SCL in I2C_SCLK cycles. This register is active only when the I2C controller works in master mode. When SCL goes high within (I2C_SCL_WAIT_HIGH_PERIOD + 1) in I2C_SCLK cycles, its frequency is:

$$f_{scl} = \frac{f_{\text{I2C\_SCLK}}}{\text{I2C\_SCL\_LOW\_PERIOD + I2C\_SCL\_HIGH\_PERIOD + I2C\_SCL\_WAIT\_HIGH\_PERIOD+3}}$$

- Master mode and slave mode:

1. I2C_SDA_SAMPLE_TIME: Specifies the interval between the rising edge of SCL and the level sampling time of SDA. It is advised to set a value in the middle of SCL's high period, so as to correctly sample the level of SCL. This register is active both in master mode and slave mode.
2. I2C_SDA_HOLD_TIME: Specifies the interval between changing the SDA output level and the falling edge of SCL. This register is active both in master mode and slave mode.

Timing parameters limits corresponding register configuration.

1. $\frac{f_{I2C\_SCLK}}{f_{SCL}} > 20$
2. $3 \times f_{I2C\_SCLK} \leq (I2C\_SDA\_HOLD\_TIME - 4) \times f_{APB\_CLK}$
3. I2C_SDA_HOLD_TIME + I2C_SCL_START_HOLD_TIME > SDA_FILTER_THRES + 3
4. I2C_SCL_WAIT_HIGH_PERIOD < I2C_SDA_SAMPLE_TIME < I2C_SCL_HIGH_PERIOD
5. I2C_SDA_SAMPLE_TIME < I2C_SCL_WAIT_HIGH_PERIOD + I2C_SCL_START_HOLD_TIME + I2C_SCL_RSTART_SETUP_TIME
6. I2C_STRETCH_PROTECT_NUM + I2C_SDA_HOLD_TIME > I2C_SCL_LOW_PERIOD

### 28.4.8 Timeout Control

The I2C controller has three types of timeout control, namely timeout control for SCL_FSM, for SCL_MAIN_FSM, and for the SCL line. The first two are always enabled, while the third is configurable.

When SCL_FSM remains unchanged for more than $2^{I2C\_SCL\_ST\_TO\_I2C}$ clock cycles, an I2C_SCL_ST_TO_INT interrupt is triggered, and then SCL_FSM goes to idle state. The value of I2C_SCL_ST_TO_I2C should be less than or equal to 22, which means SCL_FSM could remain unchanged for $2^{22}$ I2C_SCLK clock cycles at most before the interrupt is generated.

When SCL_MAIN_FSM remains unchanged for more than $2^{I2C\_SCL\_MAIN\_ST\_TO\_I2C}$ clock cycles, an I2C_SCL_MAIN_ST_TO_INT interrupt is triggered, and then SCL_MAIN_FSM goes to idle state. The value of I2C_SCL_MAIN_ST_TO_I2C should be less than or equal to 22, which means SCL_MAIN_FSM could remain unchanged for $2^{22}$ I2C_SCLK clock cycles at most before the interrupt is generated.

Timeout control for SCL is enabled by setting I2C_TIME_OUT_EN. When the level of SCL remains unchanged for more than $2^{I2C\_TIME\_OUT\_VALUE}$ clock cycles, an I2C_TIME_OUT_INT interrupt is triggered, and then the I2C bus goes to idle state.

### 28.4.9 Command Configuration

When the I2C controller works in master mode, CMD_Controller reads commands from 8 sequential command registers and controls SCL_FSM and SCL_MAIN_FSM accordingly.

|  | 31 | 30:14 | 13:11 | 10 | 9 | 8 | 7:0 |
|---|---|---|---|---|---|---|---|
| cmd0 | CMD_DONE | reserved | op_code | ack_value | ack_exp | ack_check_en | byte_num |

⋮

|  | 31 | 30:14 | 13:11 | 10 | 9 | 8 | 7:0 |
|---|---|---|---|---|---|---|---|
| cmd7 | CMD_DONE | reserved | op_code | ack_value | ack_exp | ack_check_en | byte_num |

Figure 28-6. Structure of I2C Command Registers

Command registers, whose structure is illustrated in Figure 28-6, are active only when the I2C controller works in master mode. Fields of command registers are:

1. CMD_DONE: Indicates that a command has been executed. After each command has been executed, the CMD_DONE bit in the corresponding command register is set to 1 by hardware. By reading this bit, software can tell if the command has been executed. When writing new commands, this bit must be cleared by software.

2. op_code: Indicates the command. The I2C controller supports five commands:

   - RSTART: op_code = 6. The I2C controller sends a START bit or a RSTART bit defined by the I2C protocol.

   - WRITE: op_code = 1. The I2C controller sends a slave address, a register address (only in double addressing mode) and data to the slave.

   - READ: op_code = 3. The I2C controller reads data from the slave.

   - STOP: op_code = 2. The I2C controller sends a STOP bit defined by the I2C protocol. This code also indicates that the command sequence has been executed, and the CMD_Controller stops reading commands. After restarted by software, the CMD_Controller resumes reading commands from command register 0.

   - END: op_code = 4. The I2C controller pulls the SCL line down and suspends I2C communication. This code also indicates that the command sequence has completed, and the CMD_Controller stops executing commands. Once software refreshes data in command registers and the RAM, the CMD_Controller can be restarted to execute commands from command register 0 again.

3. ack_value: Used to configure the level of the ACK bit sent by the I2C controller during a read operation. This bit is ignored in RSTART, STOP, END and WRITE conditions.

4. ack_exp: Used to configure the level of the ACK bit expected by the I2C controller during a write operation. This bit is ignored during RSTART, STOP, END and READ conditions.

5. ack_check_en: Used to enable the I2C controller during a write operation to check whether the ACK level sent by the slave matches ack_exp in the command. If this bit is set and the level received does not match ack_exp in the WRITE command, the master will generate an I2C_NACK_INT interrupt and a STOP condition for data transfer. If this bit is cleared, the controller will not check the ACK level sent by the slave. This bit is ignored during RSTART, STOP, END and READ conditions.
6. byte_num: Specifies the length of data (in bytes) to be read or written. Can range from 1 to 255 bytes. This bit is ignored during RSTART, STOP and END conditions.

Each command sequence is executed starting from command register 0 and terminated by a STOP or an END. Therefore, there must be a STOP or an END command in one command sequence.

A complete data transfer on the I2C bus should be initiated by a START and terminated by a STOP. The transfer process may be completed using multiple sequences, separated by END commands. Each sequence may differ in the direction of data transfer, clock frequency, slave addresses, data length, etc. This allows efficient use of available peripheral RAM and also achieves more flexible I2C communication.

### 28.4.10 TX/RX RAM Data Storage

Both TX RAM and RX RAM are 32 × 8 bits, and can be accessed in FIFO or non-FIFO mode. If I2C_NONFIFO_EN bit is cleared, both RAMs are accessed in FIFO mode; if I2C_NONFIFO_EN bit is set, both RAMs are accessed in non-FIFO mode.

TX RAM stores data that the I2C controller needs to send. During communication, when the I2C controller needs to send data (except acknowledgement bits), it reads data from TX RAM and sends them sequentially via SDA. When the I2C controller works in master mode, all data must be stored in TX RAM in the order they will be sent to slaves. The data stored in TX RAM include slave addresses, read/write bits, register addresses (only in double addressing mode) and data to be sent. When the I2C controller works in slave mode, TX RAM only stores data to be sent.

TX RAM can be read and written by the CPU. The CPU writes to TX RAM either in FIFO mode or in non-FIFO mode (direct address). In FIFO mode, the CPU writes to TX RAM via the fixed address I2C_DATA_REG, with addresses for writing in TX RAM incremented automatically by hardware. In non-FIFO mode, the CPU accesses TX RAM directly via address fields (I2C Base Address + 0x100) ~(I2C Base Address + 0x17C). Each byte in TX RAM occupies an entire word in the address space. Therefore, the address of the first byte is I2C Base Address + 0x100, the second byte is I2C Base Address + 0x104, the third byte is I2C Base Address + 0x108, and so on. The CPU can only read TX RAM via direct addresses. Addresses for reading TX RAM are the same with addresses for writing TX RAM.

RX RAM stores data the I2C controller receives during communication. When the I2C controller works in slave mode, neither slave addresses sent by the master nor register addresses (only in double addressing mode) will be stored into RX RAM. Values of RX RAM can be read by software after I2C communication completes.

RX RAM can only be read by the CPU. The CPU reads RX RAM either in FIFO mode or in non-FIFO mode (direct address). In FIFO mode, the CPU reads RX RAM via the fixed address I2C_DATA_REG, with addresses for reading RX RAM incremented automatically by hardware. In non-FIFO mode, the CPU accesses TX RAM directly via address fields (I2C Base Address + 0x180) ~(I2C Base Address + 0x1FC). Each byte in RX RAM occupies an entire word in the address space. Therefore, the address of the first byte is I2C Base Address + 0x180, the second byte is I2C Base Address + 0x184, the third byte is I2C Base Address + 0x188 and so on.

In FIFO mode, TX RAM of a master may wrap around to send data larger than 32 bytes. Set I2C_FIFO_PRT_EN.

If the size of data to be sent is smaller than I2C_TXFIFO_WM_THRHD (master), an I2C_TXFIFO_WM_INT (master) interrupt is generated. After receiving the interrupt, software continues writing to I2C_DATA_REG (master). Please ensure that software writes to or refreshes TX RAM before the master sends data, otherwise it may result in unpredictable consequences.

In FIFO mode, RX RAM of a slave may also wrap around to receive data larger than 32 bytes. Set I2C_FIFO_PRT_EN and clear I2C_RX_FULL_ACK_LEVEL. If data already received (to be overwritten) is larger than I2C_RXFIFO_WM_THRHD (slave), an I2C_RXFIFO_WM_INT (slave) interrupt is generated. After receiving the interrupt, software continues reading from I2C_DATA_REG (slave).

### 28.4.11 Data Conversion

DATA_Shifter is used for serial/parallel conversion, converting byte data in TX RAM to an outgoing serial bitstream or an incoming serial bitstream to byte data in RX RAM. I2C_RX_LSB_FIRST and I2C_TX_LSB_FIRST can be used to select LSB- or MSB-first storage and transmission of data.

### 28.4.12 Addressing Mode

Besides 7-bit addressing, the ESP32-C3 I2C controller also supports 10-bit addressing and double addressing. 10-bit addressing can be mixed with 7-bit addressing.

Define the slave address as SLV_ADDR. In 7-bit addressing mode, the slave address is SLV_ADDR[6:0]; in 10-bit addressing mode, the slave address is SLV_ADDR[9:0].

In 7-bit addressing mode, the master only needs to send one byte of address, which comprises SLV_ADDR[6:0] and a $R/\overline{W}$ bit. In 7-bit addressing mode, there is a special case called general call addressing (broadcast). It is enabled by setting I2C_ADDR_BROADCASTING_EN in a slave. When the slave receives the general call address (0x00) from the master and the $R/\overline{W}$ bit followed is 0, it responds to the master regardless of its own address.

In 10-bit addressing mode, the master needs to send two bytes of address. The first byte is slave_addr_first_7bits followed by a $R/\overline{W}$ bit, and slave_addr_first_7bits should be configured as (0x78 | SLV_ADDR[9:8]). The second byte is slave_addr_second_byte, which should be configured as SLV_ADDR[7:0]. The slave can enable 10-bit addressing by configuring I2C_ADDR_10BIT_EN. I2C_SLAVE_ADDR is used to configure I2C slave address. Specifically, I2C_SLAVE_ADDR[14:7] should be configured as SLV_ADDR[7:0], and I2C_SLAVE_ADDR[6:0] should be configured as (0x78 | SLV_ADDR[9:8]). Since a 10-bit slave address has one more byte than a 7-bit address, byte_num of the WRITE command and the number of bytes in the RAM increase by one.

When working in slave mode, the I2C controller supports double addressing, where the first address is the address of an I2C slave, and the second one is the slave's memory address. When using double addressing, RAM must be accessed in non-FIFO mode. Double addressing is enabled by setting I2C_FIFO_ADDR_CFG_EN.

### 28.4.13 $R/\overline{W}$ Bit Check in 10-bit Addressing Mode

In 10-bit addressing mode, when I2C_ADDR_10BIT_RW_CHECK_EN is set to 1, the I2C controller performs a check on the first byte, which consists of slave_addr_first_7bits and a $R/\overline{W}$ bit. When the $R/\overline{W}$ bit does not indicate a WRITE operation, i.e. not in line with the I2C protocol, the data transfer ends. If the check feature is not enabled, when the $R/\overline{W}$ bit does not indicate a WRITE, the data transfer still continues, but transfer failure may occur.

### 28.4.14 To Start the I2C Controller

To start the I2C controller in master mode, after configuring the controller to master mode and command registers, write 1 to I2C_TRANS_START in order that the master starts to parse and execute command sequences. The master always executes a command sequence starting from command register 0 to a STOP or an END at the end. To execute another command sequence starting from command register 0, refresh commands by writing 1 again to I2C_TRANS_START.

To start the I2C controller in slave mode, there are two ways:

- Set I2C_SLV_TX_AUTO_START_EN, and the slave starts automatic transfer upon an address match;
- Clear I2C_SLV_TX_AUTO_START_EN, and always set I2C_TRANS_START before transfer.

## 28.5 Programming Example

This sections provides programming examples for typical communication scenarios. ESP32-C3 has one I2C controller. For the convenience of description, I2C masters and slaves in all subsequent figures are ESP32-C3 I2C controllers. I2C master is referred to as I2C$_{\text{master}}$, and I2C slave is referred to as I2C$_{\text{slave}}$.

### 28.5.1 I2C$_{\text{master}}$ Writes to I2C$_{\text{slave}}$ with a 7-bit Address in One Command Sequence

### 28.5.1.1 Introduction

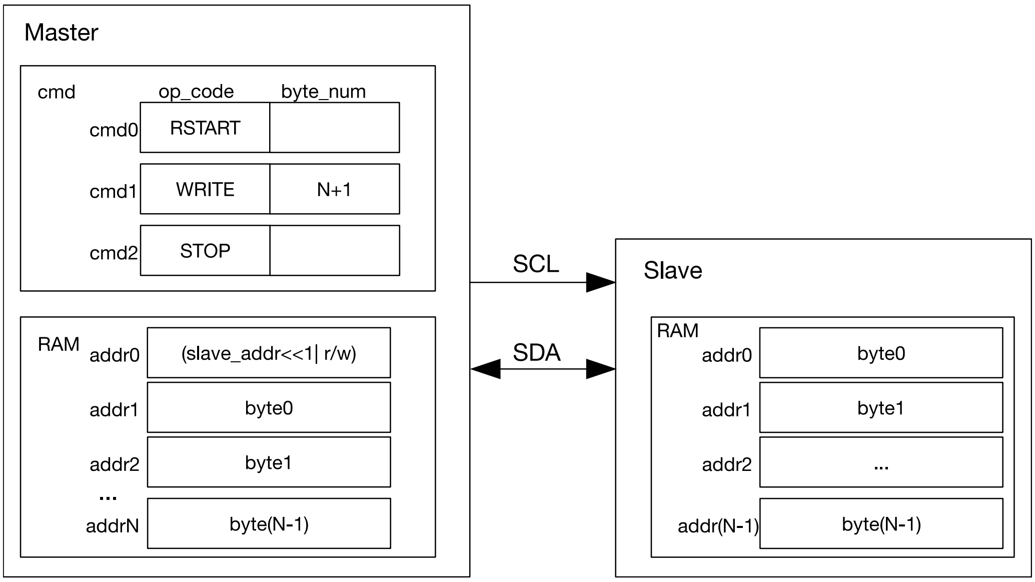


Figure 28-7. I2C$_{\text{master}}$ Writing to I2C$_{\text{slave}}$ with a 7-bit Address

Figure 28-7 shows how I2C$_{\text{master}}$ writes N bytes of data to I2C$_{\text{slave}}$'s RAM using 7-bit addressing. As shown in figure 28-7 , the first byte in the RAM of I2C$_{\text{master}}$ is a 7-bit I2C$_{\text{slave}}$ address followed by a $R/\overline{W}$ bit. When the $R/\overline{W}$ bit is 0, it indicates a WRITE operation. The remaining bytes are used to store data ready for transfer. The cmd box contains related command sequences.

After the command sequence is configured and data in RAM is ready, I2C$_{\text{master}}$ enables the controller and initiates data transfer by setting the I2C_TRANS_START bit. The controller has four steps to take:

1. Wait for SCL to go high, to avoid SCL being used by other masters or slaves.

2. Execute a RSTART command and send a START bit.
3. Execute a WRITE command by taking N+1 bytes from the RAM in order and send them to I2C$_{\text{slave}}$ in the same order. The first byte is the address of I2C$_{\text{slave}}$.
4. Send a STOP. Once the I2C$_{\text{master}}$ transfers a STOP bit, an I2C_TRANS_COMPLETE_INT interrupt is generated.

### 28.5.1.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
3. Configure command registers of I2C$_{\text{master}}$.

| Command register | op_code | ack_value | ack_exp | ack_check_en | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | ack_value | ack_exp | 1 | N+1 |
| I2C_COMMAND2 (master) | STOP | — | — | — | — |

4. Write I2C$_{\text{slave}}$ address and data to be sent to TX RAM of I2C$_{\text{master}}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
5. Write address of I2C$_{\text{slave}}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register.
6. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
7. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start transfer.
8. I2C$_{\text{slave}}$ compares the slave address sent by I2C$_{\text{master}}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in I2C$_{\text{master}}$'s WRITE command is 1, I2C$_{\text{master}}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, I2C$_{\text{master}}$ does not check ACK value and take I2C$_{\text{slave}}$ as a matching slave by default.
    - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), I2C$_{\text{master}}$ continues data transfer.
    - Not match: If the received ACK value does not match ack_exp, I2C$_{\text{master}}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
9. I2C$_{\text{master}}$ sends data, and checks ACK value or not according to ack_check_en (master).
10. If data to be sent (N) is larger than 32 bytes, TX RAM of I2C$_{\text{master}}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.
11. If data to be received (N) is larger than 32 bytes, RX RAM of I2C$_{\text{slave}}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.

    If data to be received (N) is larger than 32 bytes, the other way is to enable clock stretching by setting the I2C_SLAVE_SCL_STRETCH_EN (slave), and clearing I2C_RX_FULL_ACK_LEVEL. When RX RAM is full, an I2C_SLAVE_STRETCH_INT (slave) interrupt is generated. In this way, I2C$_{\text{slave}}$ can hold SCL low, in exchange for more time to read data. After software has finished reading, you can set

I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.

12. After data transfer completes, I2C$_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.5.2 I2C$_{master}$ Writes to I2C$_{slave}$ with a 10-bit Address in One Command Sequence

### 28.5.2.1 Introduction

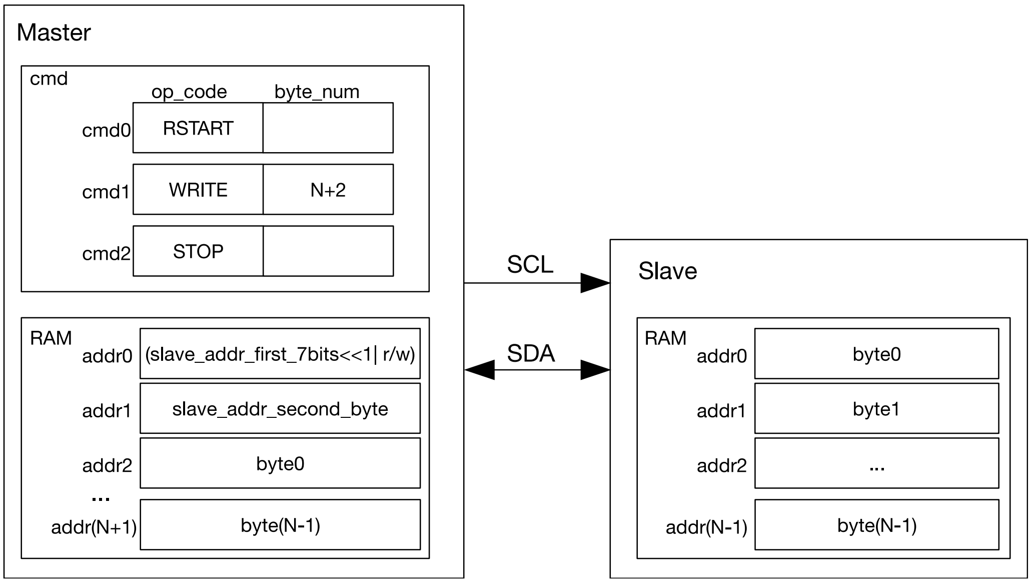


Figure 28-8. I2C$_{master}$ Writing to a Slave with a 10-bit Address

Figure 28-8 shows how I2C$_{master}$ writes N bytes of data using 10-bit addressing to an I2C slave. The configuration and transfer process is similar to what is described in 28.5.1, except that a 10-bit I2C$_{slave}$ address is formed from two bytes. Since a 10-bit I2C$_{slave}$ address has one more byte than a 7-bit I2C$_{slave}$ address, byte_num and length of data in TX RAM increase by 1 accordingly.

### 28.5.2.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
3. Configure command registers of I2C$_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | ack_value | ack_exp | 1 | N+2 |
| I2C_COMMAND2 (master) | STOP | — | — | — | — |

4. Configure I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) as I2C$_{slave}$'s 10-bit address, and

set I2C_ADDR_10BIT_EN (slave) to 1 to enable 10-bit addressing.

5. Write I2C$_{slave}$ address and data to be sent to TX RAM of I2C$_{master}$. The first byte of I2C$_{slave}$ address comprises ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a $R/\overline{W}$ bit. The second byte of I2C$_{slave}$ address is I2C_SLAVE_ADDR[7:0]. These two bytes are followed by data to be sent in FIFO or non-FIFO mode.
6. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
7. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start transfer.
8. I2C$_{slave}$ compares the slave address sent by I2C$_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in I2C$_{master}$'s WRITE command is 1, I2C$_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, I2C$_{master}$ does not check ACK value and take I2C$_{slave}$ as matching slave by default.
   - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), I2C$_{master}$ continues data transfer.
   - Not match: If the received ACK value does not match ack_exp, I2C$_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
9. I2C$_{master}$ sends data, and checks ACK value or not according to ack_check_en (master).
10. If data to be sent is larger than 32 bytes, TX RAM of I2C$_{master}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.
11. If data to be received is larger than 32 bytes, RX RAM of I2C$_{slave}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.

    If data to be received is larger than 32 bytes, the other way is to enable clock stretching by setting I2C_SLAVE_SCL_STRETCH_EN (slave), and clearing I2C_RX_FULL_ACK_LEVEL to 0. When RX RAM is full, an I2C_SLAVE_STRETCH_INT (slave) interrupt is generated. In this way, I2C$_{slave}$ can hold SCL low, in exchange for more time to read data. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
12. After data transfer completes, I2C$_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

### 28.5.3 I2C$_{master}$ Writes to I2C$_{slave}$ with Two 7-bit Addresses in One Command Sequence

#### 28.5.3.1 Introduction

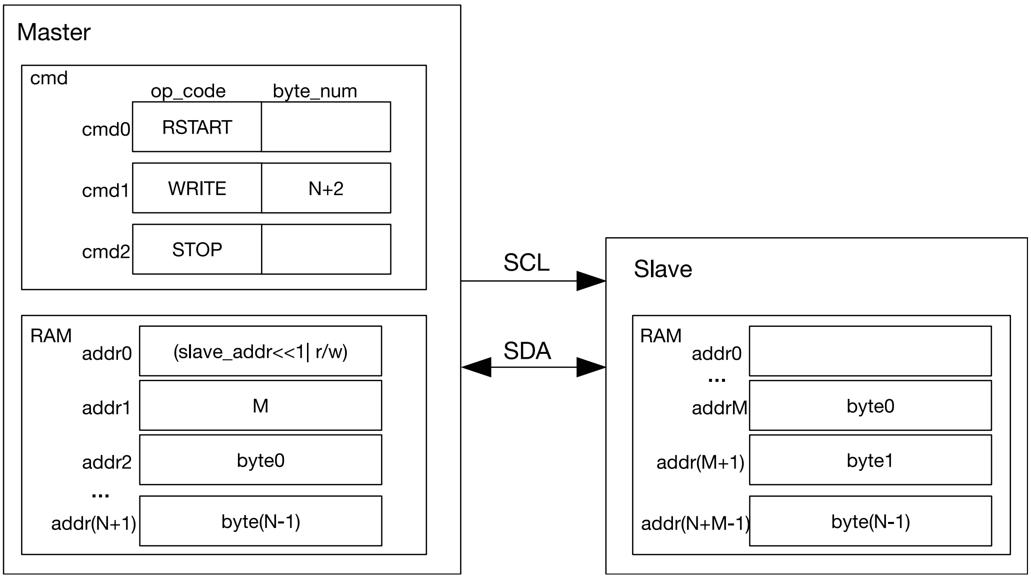


Figure 28-9. I2C$_{master}$ Writing to I2C$_{slave}$ with Two 7-bit Addresses

Figure 28-9 shows how I2C$_{master}$ writes N bytes of data to I2C$_{slave}$'s RAM using 7-bit double addressing. The configuration and transfer process is similar to what is described in Section 28.5.1, except that in 7-bit double addressing mode I2C$_{master}$ sends two 7-bit addresses. The first address is the address of an I2C slave, and the second one is I2C$_{slave}$'s memory address (i.e. addrM in Figure 28-9). When using double addressing, RAM must be accessed in non-FIFO mode. The I2C slave put received byte0 ~ byte(N-1) into its RAM in an order staring from addrM. The RAM is overwritten every 32 bytes.

#### 28.5.3.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. Set I2C_FIFO_ADDR_CFG_EN (slave) to 1 to enable double addressing mode.
3. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
4. Configure command registers of I2C$_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | ack_value | ack_exp | 1 | N+2 |
| I2C_COMMAND2 (master) | STOP | — | — | — | — |

5. Write I2C$_{slave}$ address and data to be sent to TX RAM of I2C$_{master}$ in FIFO or non-FIFO mode.
6. Write address of I2C$_{slave}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register.
7. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.

8. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start transfer.
9. I2C$_{\text{slave}}$ compares the slave address sent by I2C$_{\text{master}}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in I2C$_{\text{master}}$'s WRITE command is 1, I2C$_{\text{master}}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, I2C$_{\text{master}}$ does not check ACK value and take I2C$_{\text{slave}}$ as matching slave by default.
   - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), I2C$_{\text{master}}$ continues data transfer.
   - Not match: If the received ACK value does not match ack_exp, I2C$_{\text{master}}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
10. I2C$_{\text{slave}}$ receives the RX RAM address sent by I2C$_{\text{master}}$ and adds the offset.
11. I2C$_{\text{master}}$ sends data, and checks ACK value or not according to ack_check_en (master).
12. If data to be sent is larger than 32 bytes, TX RAM of I2C$_{\text{master}}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.
13. If data to be received is larger than 32 bytes, you may enable clock stretching by setting I2C_SLAVE_SCL_STRETCH_EN (slave), and clearing I2C_RX_FULL_ACK_LEVEL to 0. When RX RAM is full, an I2C_SLAVE_STRETCH_INT (slave) interrupt is generated. In this way, I2C$_{\text{slave}}$ can hold SCL low, in exchange for more time to read data. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
14. After data transfer completes, I2C$_{\text{master}}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.5.4 I2C$_{master}$ Writes to I2C$_{slave}$ with a 7-bit Address in Multiple Command Sequences

### 28.5.4.1 Introduction

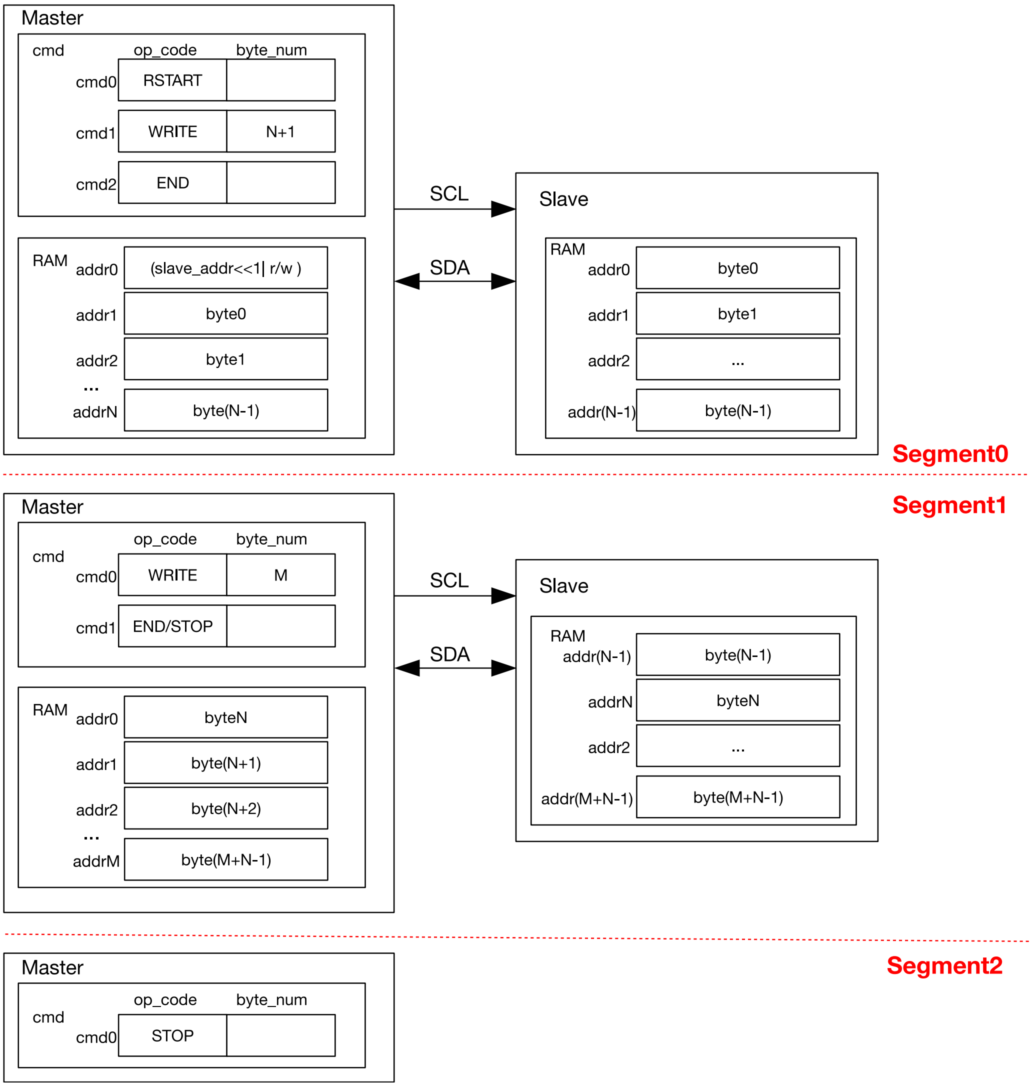


Figure 28-10. I2C$_{master}$ Writing to I2C$_{slave}$ with a 7-bit Address in Multiple Sequences

Given that the I2C Controller RAM holds only 32 bytes, when data are too large to be processed even by the wrapped RAM, it is advised to transmit them in multiple command sequences. At the end of every command sequence is an END command. When the controller executes this END command to pull SCL low, software refreshes command sequence registers and the RAM for next the transfer.

Figure 28-10 shows how I2C$_{master}$ writes to an I2C slave in two or three segments as an example. For the first segment, the CMD_Controller registers are configured as shown in Segment0. Once data in I2C$_{master}$'s RAM is ready and I2C_TRANS_START is set, I2C$_{master}$ initiates data transfer. After executing the END command, I2C$_{master}$ turns off the SCL clock and pulls SCL low to reserve the bus. Meanwhile, the controller generates an I2C_END_DETECT_INT interrupt.

For the second segment, after detecting the I2C_END_DETECT_INT interrupt, software refreshes the CMD_Controller registers, reloads the RAM and clears this interrupt, as shown in Segment1. If cmd1 in the second segment is a STOP, then data is transmitted to I2C$_{slave}$ in two segments. I2C$_{master}$ resumes data transfer after I2C_TRANS_START is set, and terminates the transfer by sending a STOP bit.

For the third segment, after the second data transfer finishes and an I2C_END_DETECT_INT is detected, the CMD_Controller registers of I2C$_{master}$ are configured as shown in Segment2. Once I2C_TRANS_START is set, I2C$_{master}$ generates a STOP bit and terminates the transfer.

Note that other I2C$_{master}$s will not transact on the bus between two segments. The bus is only released after a STOP signal is sent. The I2C controller can be reset by setting I2C_FSM_RST field at any time. This field will later be cleared automatically by hardware.

#### 28.5.4.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
3. Configure command registers of I2C$_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | ack_value | ack_exp | 1 | N+1 |
| I2C_COMMAND2 (master) | END | — | — | — | — |

4. Write I2C$_{slave}$ address and data to be sent to TX RAM of I2C$_{master}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
5. Write address of I2C$_{slave}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register
6. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
7. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start transfer.
8. I2C$_{slave}$ compares the slave address sent by I2C$_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in I2C$_{master}$'s WRITE command is 1, I2C$_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, I2C$_{master}$ does not check ACK value and take I2C$_{slave}$ as matching slave by default.
   - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), I2C$_{master}$ continues data transfer.
   - Not match: If the received ACK value does not match ack_exp, I2C$_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
9. I2C$_{master}$ sends data, and checks ACK value or not according to ack_check_en (master).
10. After the I2C_END_DETECT_INT (master) interrupt is generated, set I2C_END_DETECT_INT_CLR (master) to 1 to clear this interrupt.
11. Update I2C$_{master}$'s command registers.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | WRITE | ack_value | ack_exp | 1 | M |

| | | | | | |
|---|---|---|---|---|---|
| I2C_COMMAND1 (master) | END/STOP | — | — | — | — |

12. Write M bytes of data to be sent to TX RAM of I2C$_{master}$ in FIFO or non-FIFO mode.

13. Write 1 to I2C_TRANS_START (master) bit to start transfer and repeat step 9.

14. If the command is a STOP, I2C stops transfer and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

15. If the command is an END, repeat step 10.

16. Update I2C$_{master}$'s command registers.

| Command registers | op_code | ack_value | ack_exp | ack_check_en | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND1 (master) | STOP | — | — | — | — |

17. Write 1 to I2C_TRANS_START (master) bit to start transfer.

18. I2C$_{master}$ executes the STOP command and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.5.5 I2C$_{master}$ Reads I2C$_{slave}$ with a 7-bit Address in One Command Sequence

### 28.5.5.1 Introduction

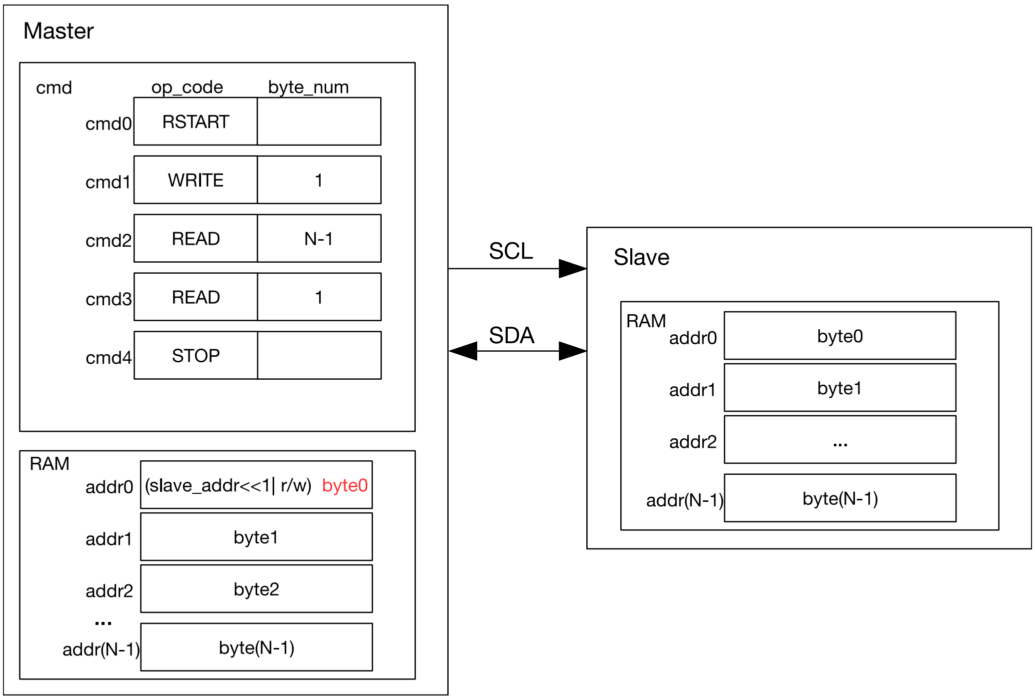


Figure 28-11. I2C$_{master}$ Reading I2C$_{slave}$ with a 7-bit Address

Figure 28-11 shows how I2C$_{master}$ reads N bytes of data from an I2C slave using 7-bit addressing. cmd1 is a WRITE command, and when this command is executed I2C$_{master}$ sends I2C$_{slave}$ address. The byte sent comprises a 7-bit I2C$_{slave}$ address and a $R/\overline{W}$ bit. When the $R/\overline{W}$ bit is 1, it indicates a READ operation. If the address of an I2C slave matches the sent address, this matching slave starts sending data to I2C$_{master}$.

$\text{I2C}_{\text{master}}$ generates acknowledgements according to ack_value defined in the READ command upon receiving a byte.

As illustrated in Figure 28-11, $\text{I2C}_{\text{master}}$ executes two READ commands: it generates ACKs for (N-1) bytes of data in cmd2, and a NACK for the last byte of data in cmd 3. This configuration may be changed as required. $\text{I2C}_{\text{master}}$ writes received data into the controller RAM from addr0, whose original content (a $\text{I2C}_{\text{slave}}$ address and a $R/\overline{W}$ bit) is overwritten by byte0 marked red in Figure 28-11.

### 28.5.5.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. We recommend setting I2C_SLAVE_SCL_STRETCH_EN (slave) to 1, so that SCL can be held low for more processing time when $\text{I2C}_{\text{slave}}$ needs to send data. If this bit is not set, software should write data to be sent to $\text{I2C}_{\text{slave}}$'s TX RAM before $\text{I2C}_{\text{master}}$ initiates transfer. Configuration below is applicable to scenario where I2C_SLAVE_SCL_STRETCH_EN (slave) is 1.
3. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
4. Configure command registers of $\text{I2C}_{\text{master}}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_en | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | 0 | 0 | 1 | 1 |
| I2C_COMMAND2 (master) | READ | 0 | 0 | 1 | N-1 |
| I2C_COMMAND3 (master) | READ | 1 | 0 | 1 | 1 |
| I2C_COMMAND4 (master) | STOP | — | — | — | — |

5. Write $\text{I2C}_{\text{slave}}$ address to TX RAM of $\text{I2C}_{\text{master}}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
6. Write address of $\text{I2C}_{\text{slave}}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register.
7. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
8. Write 1 to I2C_TRANS_START (master) bit to start $\text{I2C}_{\text{master}}$'s transfer.
9. Start $\text{I2C}_{\text{slave}}$'s transfer according to Section 28.4.14.
10. $\text{I2C}_{\text{slave}}$ compares the slave address sent by $\text{I2C}_{\text{master}}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $\text{I2C}_{\text{master}}$'s WRITE command is 1, $\text{I2C}_{\text{master}}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, $\text{I2C}_{\text{master}}$ does not check ACK value and take $\text{I2C}_{\text{slave}}$ as matching slave by default.
    - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $\text{I2C}_{\text{master}}$ continues data transfer.
    - Not match: If the received ACK value does not match ack_exp, $\text{I2C}_{\text{master}}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.

11. After I2C_SLAVE_STRETCH_INT (slave) is generated, the I2C_STRETCH_CAUSE bit is 0. The I2C$_{slave}$ address matches the address sent over SDA, and I2C$_{slave}$ needs to send data.

12. Write data to be sent to TX RAM of I2C$_{slave}$ according to Section 28.4.10.

13. Set I2C_SLAVE_SCL_STRETCH_CLR (slave) to 1 to release SCL.

14. I2C$_{slave}$ sends data, and I2C$_{master}$ checks ACK value or not according to ack_check_en (master) in the READ command.

15. If data to be read by I2C$_{master}$ is larger than 32 bytes, an I2C_SLAVE_STRETCH_INT (slave) interrupt will be generated when TX RAM of I2C$_{slave}$ becomes empty. In this way, I2C$_{slave}$ can hold SCL low, so that software has more time to pad data in TX RAM of I2C$_{slave}$ and read data in RX RAM of I2C$_{master}$. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.

16. After I2C$_{master}$ has received the last byte of data, set ack_value (master) to 1. I2C$_{slave}$ will stop transfer once receiving the I2C_NACK_INT interrupt.

17. After data transfer completes, I2C$_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

### 28.5.6 I2C$_{master}$ Reads I2C$_{slave}$ with a 10-bit Address in One Command Sequence

#### 28.5.6.1 Introduction

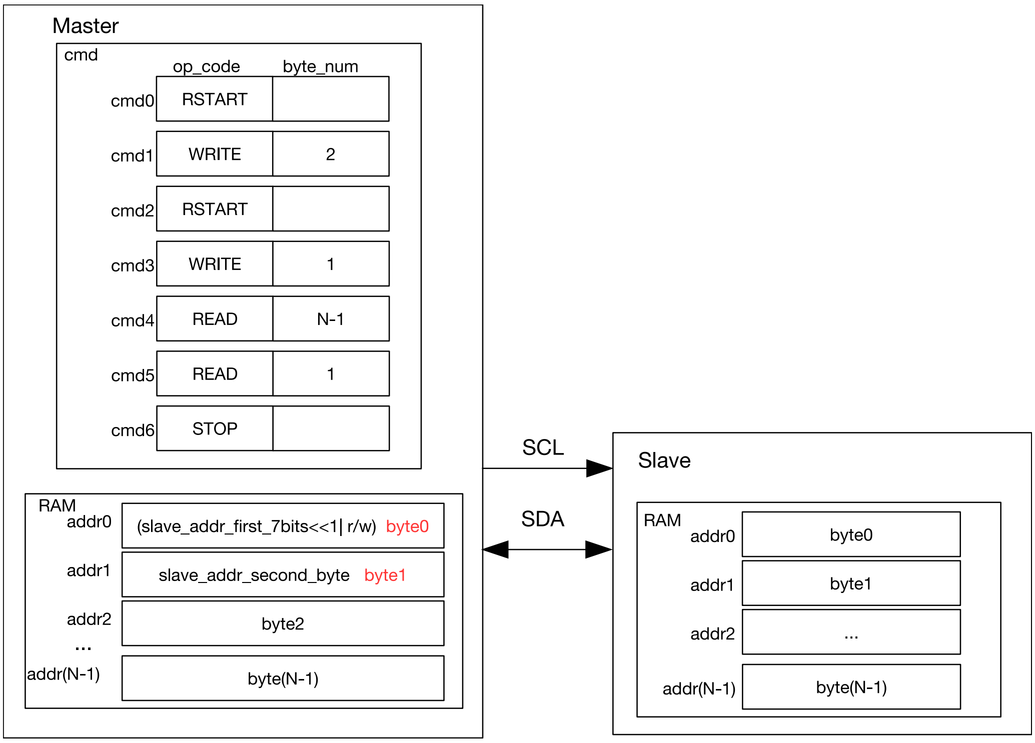


Figure 28-12. I2C$_{master}$ Reading I2C$_{slave}$ with a 10-bit Address

Figure 28-12 shows how I2C$_{master}$ reads data from an I2C slave using 10-bit addressing. Unlike 7-bit addressing, in 10-bit addressing the WRITE command of the I2C$_{master}$ is formed from two bytes, and correspondingly TX RAM of this master stores a 10-bit address of two bytes. The $R/\overline{W}$ bit in the first byte is 0, which indicates a

WRITE operation. After a RSTART condition, I2C$_{master}$ sends the first byte of address again to read data from I2C$_{slave}$, but the $R/\overline{W}$ bit is 1, which indicates a READ operation. The two address bytes can be configured as described in Section 28.5.2.

#### 28.5.6.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. We recommend setting I2C_SLAVE_SCL_STRETCH_EN (slave) to 1, so that SCL can be held low for more processing time when I2C$_{slave}$ needs to send data. If this bit is not set, software should write data to be sent to I2C$_{slave}$'s TX RAM before I2C$_{master}$ initiates transfer. Configuration below is applicable to scenario where I2C_SLAVE_SCL_STRETCH_EN (slave) is 1.
3. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
4. Configure command registers of I2C$_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | 0 | 0 | 1 | 2 |
| I2C_COMMAND2 (master) | RSTART | — | — | — | — |
| I2C_COMMAND3 (master) | WRITE | 0 | 0 | 1 | 1 |
| I2C_COMMAND4 (master) | READ | 0 | 0 | 1 | N-1 |
| I2C_COMMAND5 (master) | READ | 1 | 0 | 1 | 1 |
| I2C_COMMAND6 (master) | STOP | — | — | — | — |

5. Configure I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) as I2C$_{slave}$'s 10-bit address, and set
   I2C_ADDR_10BIT_EN (slave) to 1 to enable 10-bit addressing.
6. Write I2C$_{slave}$ address and data to be sent to TX RAM of I2C$_{master}$ in either FIFO or non-FIFO mode. The first byte of address comprises ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a $R/\overline{W}$ bit, which is 1 and indicates a WRITE operation. The second byte of address is I2C_SLAVE_ADDR[7:0]. The third byte is ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a $R/\overline{W}$ bit, which is 1 and indicates a READ operation.
7. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
8. Write 1 to I2C_TRANS_START (master) to start I2C$_{master}$'s transfer.
9. Start I2C$_{slave}$'s transfer according to Section 28.4.14.
10. I2C$_{slave}$ compares the slave address sent by I2C$_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in I2C$_{master}$'s WRITE command is 1, I2C$_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, I2C$_{master}$ does not check ACK value and take I2C$_{slave}$ as matching slave by default.

- Match: If the received ACK value matches ack_exp (master) (the expected ACK value), I2C$_{master}$ continues data transfer.
- Not match: If the received ACK value does not match ack_exp, I2C$_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.

11. I2C$_{master}$ sends a RSTART and the third byte in TX RAM, which is ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a $R/\overline{W}$ bit that indicates READ.
12. I2C$_{slave}$ repeats step 10. If its address matches the address sent by I2C$_{master}$, I2C$_{slave}$ proceed on to the next steps.
13. After I2C_SLAVE_STRETCH_INT (slave) is generated, the I2C_STRETCH_CAUSE bit is 0. The I2C$_{slave}$ address matches the address sent over SDA, and I2C$_{slave}$ needs to send data.
14. Write data to be sent to TX RAM of I2C$_{slave}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
15. Set I2C_SLAVE_SCL_STRETCH_CLR (slave) to 1 to release SCL.
16. I2C$_{slave}$ sends data, and I2C$_{master}$ checks ACK value or not according to ack_check_en (master) in the READ command.
17. If data to be read by I2C$_{master}$ is larger than 32 bytes, an I2C_SLAVE_STRETCH_INT (slave) interrupt will be generated when TX RAM of I2C$_{slave}$ becomes empty. In this way, I2C$_{slave}$ can hold SCL low, so that software has more time to pad data in TX RAM of I2C$_{slave}$ and read data in RX RAM of I2C$_{master}$. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
18. After I2C$_{master}$ has received the last byte of data, set ack_value (master) to 1. I2C$_{slave}$ will stop transfer once receiving the I2C_NACK_INT interrupt.
19. After data transfer completes, I2C$_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

### 28.5.7 I2C$_{master}$ Reads I2C$_{slave}$ with Two 7-bit Addresses in One Command Sequence

### 28.5.7.1 Introduction

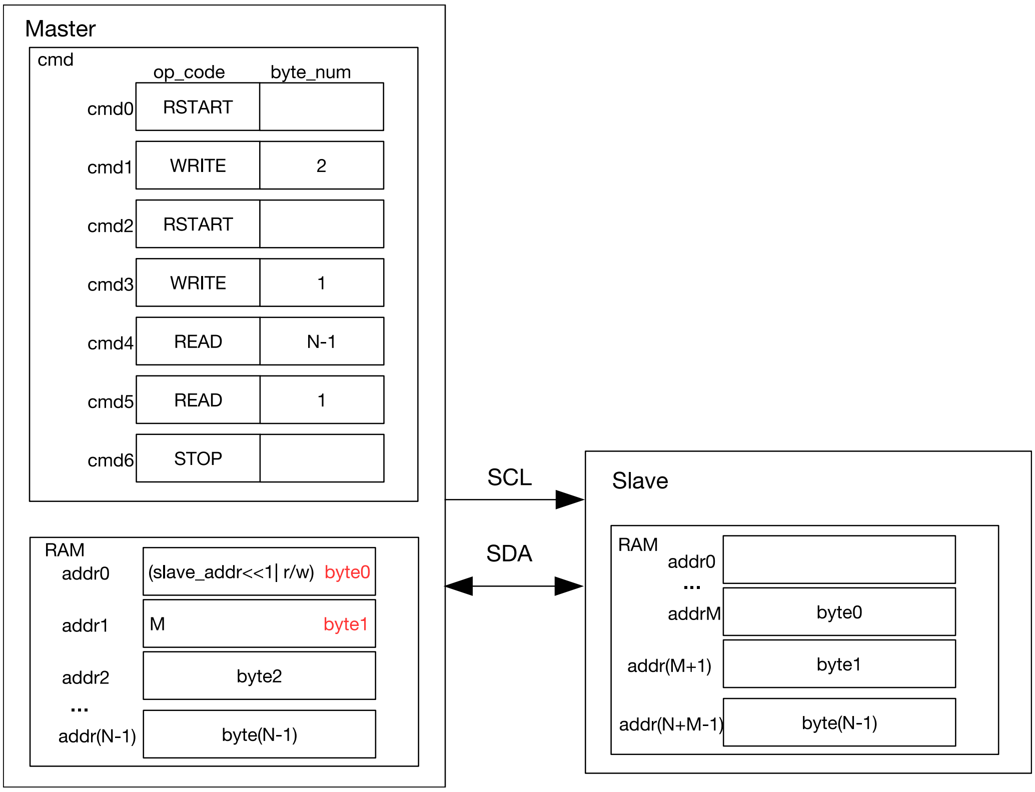


**Figure 28-13. I2C$_{\text{master}}$ Reading N Bytes of Data from addrM of I2C$_{\text{slave}}$ with a 7-bit Address**

Figure 28-13 shows how I2C$_{\text{master}}$ reads data from specified addresses in an I2C slave. I2C$_{\text{master}}$ sends two bytes of addresses: the first byte is a 7-bit I2C$_{\text{slave}}$ address followed by a $R/\overline{W}$ bit, which is 0 and indicates a WRITE; the second byte is I2C$_{\text{slave}}$'s memory address. After a RSTART condition, I2C$_{\text{master}}$ sends the first byte of address again, but the $R/\overline{W}$ bit is 1 which indicates a READ. Then, I2C$_{\text{master}}$ reads data starting from addrM.

### 28.5.7.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. We recommend setting I2C_SLAVE_SCL_STRETCH_EN (slave) to 1, so that SCL can be held low for more processing time when I2C$_{\text{slave}}$ needs to send data. If this bit is not set, software should write data to be sent to I2C$_{\text{slave}}$'s TX RAM before I2C$_{\text{master}}$ initiates transfer. Configuration below is applicable to scenario where I2C_SLAVE_SCL_STRETCH_EN (slave) is 1.
3. Set I2C_FIFO_ADDR_CFG_EN (slave) to 1 to enable double addressing mode.
4. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
5. Configure command registers of I2C$_{\text{master}}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | 0 | 0 | 1 | 2 |

| | | | | | |
|---|---|---|---|---|---|
| I2C_COMMAND2 (master) | RSTART | — | — | — | — |
| I2C_COMMAND3 (master) | WRITE | 0 | 0 | 1 | 1 |
| I2C_COMMAND4 (master) | READ | 0 | 0 | 1 | N-1 |
| I2C_COMMAND5 (master) | READ | 1 | 0 | 1 | 1 |
| I2C_COMMAND6 (master) | STOP | — | — | — | — |

6. Configure I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register as I2C$_{\text{slave}}$'s 7-bit address, and set I2C_ADDR_10BIT_EN (slave) to 0 to enable 7-bit addressing.

7. Write I2C$_{\text{slave}}$ address and data to be sent to TX RAM of I2C$_{\text{master}}$ in either FIFO or non-FIFO mode according to Section 28.4.10. The first byte of address comprises ( I2C_SLAVE_ADDR[6:0])«1) and a $R/\overline{W}$ bit, which is 0 and indicates a WRITE. The second byte of address is memory address M of I2C$_{\text{slave}}$. The third byte is ( I2C_SLAVE_ADDR[6:0])«1) and a $R/\overline{W}$ bit, which is 1 and indicates a READ.

8. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.

9. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start I2C$_{\text{master}}$'s transfer.

10. Start I2C$_{\text{slave}}$'s transfer according to Section 28.4.14.

11. I2C$_{\text{slave}}$ compares the slave address sent by I2C$_{\text{master}}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in I2C$_{\text{master}}$'s WRITE command is 1, I2C$_{\text{master}}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, I2C$_{\text{master}}$ does not check ACK value and take I2C$_{\text{slave}}$ as matching slave by default.

    - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), I2C$_{\text{master}}$ continues data transfer.

    - Not match: If the received ACK value does not match ack_exp, I2C$_{\text{master}}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.

12. I2C$_{\text{slave}}$ receives memory address sent by I2C$_{\text{master}}$ and adds the offset.

13. I2C$_{\text{master}}$ sends a RSTART and the third byte in TX RAM, which is ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a R bit.

14. I2C$_{\text{slave}}$ repeats step 11. If its address matches the address sent by I2C$_{\text{master}}$, I2C$_{\text{slave}}$ proceed on to the next steps.

15. After I2C_SLAVE_STRETCH_INT (slave) is generated, the I2C_STRETCH_CAUSE bit is 0. The I2C$_{\text{slave}}$ address matches the address sent over SDA, and I2C$_{\text{slave}}$ needs to send data.

16. Write data to be sent to TX RAM of I2C$_{\text{slave}}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.

17. Set I2C_SLAVE_SCL_STRETCH_CLR (slave) to 1 to release SCL.

18. I2C$_{\text{slave}}$ sends data, and I2C$_{\text{master}}$ checks ACK value or not according to ack_check_en (master) in the READ command.

19. If data to be read by $\text{I2C}_{\text{master}}$ is larger than 32 bytes, an I2C_SLAVE_STRETCH_INT (slave) interrupt will be generated when TX RAM of $\text{I2C}_{\text{slave}}$ becomes empty. In this way, $\text{I2C}_{\text{slave}}$ can hold SCL low, so that software has more time to pad data in TX RAM of $\text{I2C}_{\text{slave}}$ and read data in RX RAM of $\text{I2C}_{\text{master}}$. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
20. After $\text{I2C}_{\text{master}}$ has received the last byte of data, set ack_value (master) to 1. $\text{I2C}_{\text{slave}}$ will stop transfer once receiving the I2C_NACK_INT interrupt.
21. After data transfer completes, $\text{I2C}_{\text{master}}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

### 28.5.8 $I2C_{master}$ Reads $I2C_{slave}$ with a 7-bit Address in Multiple Command Sequences

#### 28.5.8.1 Introduction

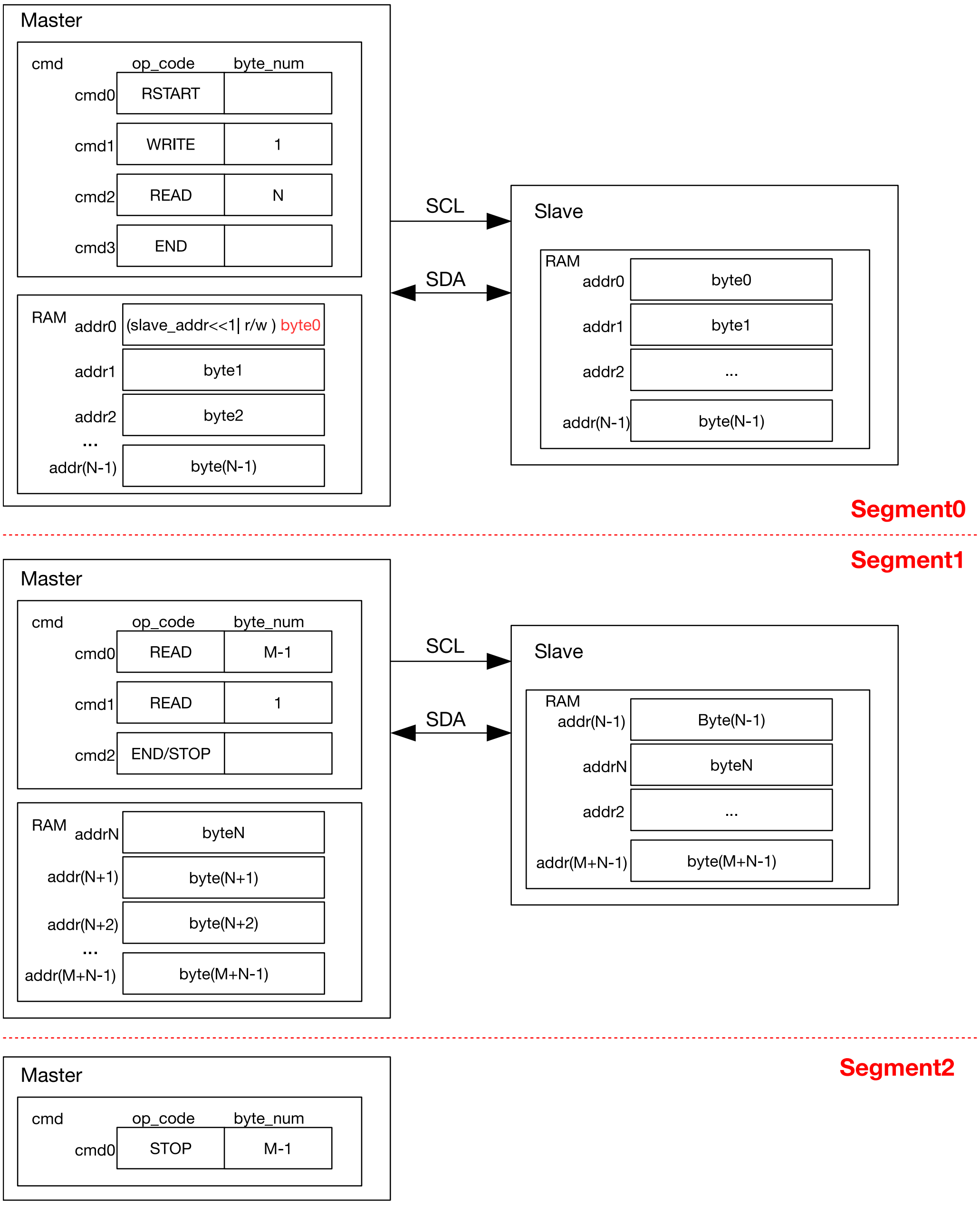


Figure 28-14. $I2C_{master}$ Reading $I2C_{slave}$ with a 7-bit Address in Segments

Figure 28-14 shows how $I2C_{master}$ reads (N+M) bytes of data from an I2C slave in two/three segments separated by END commands. Configuration procedures are described as follows:

1. The procedures for Segment0 is similar to Figure 28-11, except that the last command is an END.

2. Prepare data in the TX RAM of $I2C_{slave}$, and set I2C_TRANS_START to start data transfer. After executing the END command, $I2C_{master}$ refreshes command registers and the RAM as shown in Segment1, and clears the corresponding I2C_END_DETECT_INT interrupt. If cmd2 in Segment1 is a STOP, then data is read from $I2C_{slave}$ in two segments. $I2C_{master}$ resumes data transfer by setting I2C_TRANS_START and terminates the transfer by sending a STOP bit.

3. If cmd2 in Segment1 is an END, then data is read from I2C$_{slave}$ in three segments. After the second data transfer finishes and an I2C_END_DETECT_INT interrupt is detected, the cmd box is configured as shown in Segment2. Once I2C_TRANS_START is set, I2C$_{master}$ terminates the transfer by sending a STOP bit.

### 28.5.8.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. We recommend setting I2C_SLAVE_SCL_STRETCH_EN (slave) to 1, so that SCL can be held low for more processing time when I2C$_{slave}$ needs to send data. If this bit is not set, software should write data to be sent to I2C$_{slave}$'s TX RAM before I2C$_{master}$ initiates transfer. Configuration below is applicable to scenario where I2C_SLAVE_SCL_STRETCH_EN (slave) is 1.
3. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
4. Configure command registers of I2C$_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_en | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | 0 | 0 | 1 | 1 |
| I2C_COMMAND2 (master) | READ | 0 | 0 | 1 | N |
| I2C_COMMAND3 (master) | END | — | — | — | — |

5. Write I2C$_{slave}$ address to TX RAM of I2C$_{master}$ in FIFO or non-FIFO mode.
6. Write address of I2C$_{slave}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register.
7. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
8. Write 1 to I2C_TRANS_START (master) to start I2C$_{master}$'s transfer.
9. Start I2C$_{slave}$'s transfer according to Section 28.4.14.
10. I2C$_{slave}$ compares the slave address sent by I2C$_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in I2C$_{master}$'s WRITE command is 1, I2C$_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, I2C$_{master}$ does not check ACK value and take I2C$_{slave}$ as matching slave by default.
    - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), I2C$_{master}$ continues data transfer.
    - Not match: If the received ACK value does not match ack_exp, I2C$_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
11. After I2C_SLAVE_STRETCH_INT (slave) is generated, the I2C_STRETCH_CAUSE bit is 0. The I2C$_{slave}$ address matches the address sent over SDA, and I2C$_{slave}$ needs to send data.
12. Write data to be sent to TX RAM of I2C$_{slave}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
13. Set I2C_SLAVE_SCL_STRETCH_CLR (slave) to 1 to release SCL.

14. I2C$_{slave}$ sends data, and I2C$_{master}$ checks ACK value or not according to ack_check_en (master) in the READ command.

15. If data to be read by I2C$_{master}$ in one READ command (N or M) is larger than 32 bytes, an I2C_SLAVE_STRETCH_INT (slave) interrupt will be generated when TX RAM of I2C$_{slave}$ becomes empty. In this way, I2C$_{slave}$ can hold SCL low, so that software has more time to pad data in TX RAM of I2C$_{slave}$ and read data in RX RAM of I2C$_{master}$. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.

16. Once finishing reading data in the first READ command, I2C$_{master}$ executes the END command and triggers an I2C_END_DETECT_INT (master) interrupt, which is cleared by setting I2C_END_DETECT_INT_CLR (master) to 1.

17. Update I2C$_{master}$'s command registers using one of the following two methods:

| Command registers | op_code | ack_value | ack_exp | ack_check_en | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | READ | ack_value | ack_exp | 1 | M |
| I2C_COMMAND1 (master) | END | — | — | — | — |

Or

| Command registers | op_code | ack_value | ack_exp | ack_check_en | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | READ | 0 | 0 | 1 | M-1 |
| I2C_COMMAND0 (master) | READ | 1 | 0 | 1 | 1 |
| I2C_COMMAND1 (master) | STOP | — | — | — | — |

18. Write M bytes of data to be sent to TX RAM of I2C$_{slave}$. If M is larger than 32, then repeat step 14 in FIFO or non-FIFO mode.

19. Write 1 to I2C_TRANS_START (master) bit to start transfer and repeat step 14.

20. If the last command is a STOP, then set ack_value (master) to 1 after I2C$_{master}$ has received the last byte of data. I2C$_{slave}$ stops transfer upon the I2C_NACK_INT interrupt. I2C$_{master}$ executes the STOP command to stop transfer and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

21. If the last command is an END, then repeat step 16 and proceed on to the next steps.

22. Update I2C$_{master}$'s command registers.

| Command registers | op_code | ack_value | ack_exp | ack_check_en | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND1 (master) | STOP | — | — | — | — |

23. Write 1 to I2C_TRANS_START (master) bit to start transfer.

24. I2C$_{master}$ executes the STOP command to stop transfer, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.6 Interrupts

- I2C_SLAVE_STRETCH_INT: Generated when one of the four stretching events occurs in slave mode.
- I2C_DET_START_INT: Triggered when the master or the slave detects a START bit.
- I2C_SCL_MAIN_ST_TO_INT: Triggered when the main state machine SCL_MAIN_FSM remains unchanged for over I2C_SCL_MAIN_ST_TO_I2C[23:0] clock cycles.
- I2C_SCL_ST_TO_INT: Triggered when the state machine SCL_FSM remains unchanged for over I2C_SCL_ST_TO_I2C[23:0] clock cycles.
- I2C_RXFIFO_UDF_INT: Triggered when the I2C controller reads RX FIFO via the APB bus, but RX FIFO is empty.
- I2C_TXFIFO_OVF_INT: Triggered when the I2C controller writes TX FIFO via the APB bus, but TX FIFO is full.
- I2C_NACK_INT: Triggered when the ACK value received by the master is not as expected, or when the ACK value received by the slave is 1.
- I2C_TRANS_START_INT: Triggered when the I2C controller sends a START bit.
- I2C_TIME_OUT_INT: Triggered when SCL stays high or low for more than $2^{I2C\_TIME\_OUT\_VALUE}$ clock cycles during data transfer.
- I2C_TRANS_COMPLETE_INT: Triggered when the I2C controller detects a STOP bit.
- I2C_MST_TXFIFO_UDF_INT: Triggered when TX FIFO of the master underflows.
- I2C_ARBITRATION_LOST_INT: Triggered when the SDA's output value does not match its input value while the master's SCL is high.
- I2C_BYTE_TRANS_DONE_INT: Triggered when the I2C controller sends or receives a byte.
- I2C_END_DETECT_INT: Triggered when op_code of the master indicates an END command and an END condition is detected.
- I2C_RXFIFO_OVF_INT: Triggered when RX FIFO of the I2C controller overflows.
- I2C_TXFIFO_WM_INT: I2C TX FIFO watermark interrupt. Triggered when I2C_FIFO_PRT_EN is 1 and the pointers of TX FIFO are less than I2C_TXFIFO_WM_THRHD[4:0].
- I2C_RXFIFO_WM_INT: I2C RX FIFO watermark interrupt. Triggered when I2C_FIFO_PRT_EN is 1 and the pointers of RX FIFO are greater than I2C_RXFIFO_WM_THRHD[4:0].

## 28.7 Register Summary

The addresses in this section are relative to I2C Controller base address provided in Table 3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

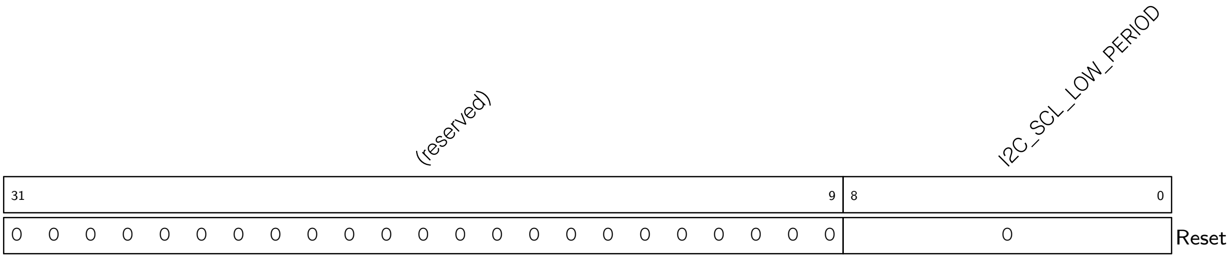
| Name | Description | Address | Access |
|---|---|---|---|
| **Timing registers** | | | |
| I2C_SCL_LOW_PERIOD_REG | Configures the low level width of SCL | 0x0000 | R/W |
| I2C_SDA_HOLD_REG | Configures the hold time after a negative SCL edge | 0x0030 | R/W |
| I2C_SDA_SAMPLE_REG | Configures the sample time after a positive SCL edge | 0x0034 | R/W |
| I2C_SCL_HIGH_PERIOD_REG | Configures the high level width of SCL | 0x0038 | R/W |
| I2C_SCL_START_HOLD_REG | Configures the delay between the SDA and SCL negative edge for a START condition | 0x0040 | R/W |
| I2C_SCL_RSTART_SETUP_REG | Configures the delay between the positive edge of SCL and the negative edge of SDA | 0x0044 | R/W |
| I2C_SCL_STOP_HOLD_REG | Configures the delay after the SCL clock edge for a STOP condition | 0x0048 | R/W |
| I2C_SCL_STOP_SETUP_REG | Configures the delay between the SDA and SCL positive edge for a STOP condition | 0x004C | R/W |
| I2C_SCL_ST_TIME_OUT_REG | SCL status timeout register | 0x0078 | R/W |
| I2C_SCL_MAIN_ST_TIME_OUT_REG | SCL main status timeout register | 0x007C | R/W |
| **Configuration registers** | | | |
| I2C_CTR_REG | Transmission configuration register | 0x0004 | varies |
| I2C_TO_REG | Timeout control register | 0x000C | R/W |
| I2C_SLAVE_ADDR_REG | Slave address configuration register | 0x0010 | R/W |
| I2C_FIFO_CONF_REG | FIFO configuration register | 0x0018 | R/W |
| I2C_FILTER_CFG_REG | SCL and SDA filter configuration register | 0x0050 | R/W |
| I2C_CLK_CONF_REG | I2C clock configuration register | 0x0054 | R/W |
| I2C_SCL_SP_CONF_REG | Power configuration register | 0x0080 | varies |
| I2C_SCL_STRETCH_CONF_REG | Configures SCL clock stretching | 0x0084 | varies |
| **Status registers** | | | |
| I2C_SR_REG | Describes I2C work status | 0x0008 | RO |
| I2C_FIFO_ST_REG | FIFO status register | 0x0014 | RO |
| I2C_DATA_REG | Read/write FIFO register | 0x001C | R/W |
| **Interrupt registers** | | | |
| I2C_INT_RAW_REG | Raw interrupt status | 0x0020 | R/SS/WTC |
| I2C_INT_CLR_REG | Interrupt clear bits | 0x0024 | WT |
| I2C_INT_ENA_REG | Interrupt enable bits | 0x0028 | R/W |
| I2C_INT_STATUS_REG | Status of captured I2C communication events | 0x002C | RO |
| **Command registers** | | | |
| I2C_COMD0_REG | I2C command register 0 | 0x0058 | varies |
| I2C_COMD1_REG | I2C command register 1 | 0x005C | varies |

| Name | Description | Address | Access |
|---|---|---|---|
| I2C_COMD2_REG | I2C command register 2 | 0x0060 | varies |
| I2C_COMD3_REG | I2C command register 3 | 0x0064 | varies |
| I2C_COMD4_REG | I2C command register 4 | 0x0068 | varies |
| I2C_COMD5_REG | I2C command register 5 | 0x006C | varies |
| I2C_COMD6_REG | I2C command register 6 | 0x0070 | varies |
| I2C_COMD7_REG | I2C command register 7 | 0x0074 | varies |
| **Version register** | | | |
| I2C_DATE_REG | Version control register | 0x00F8 | R/W |

## 28.8 Registers

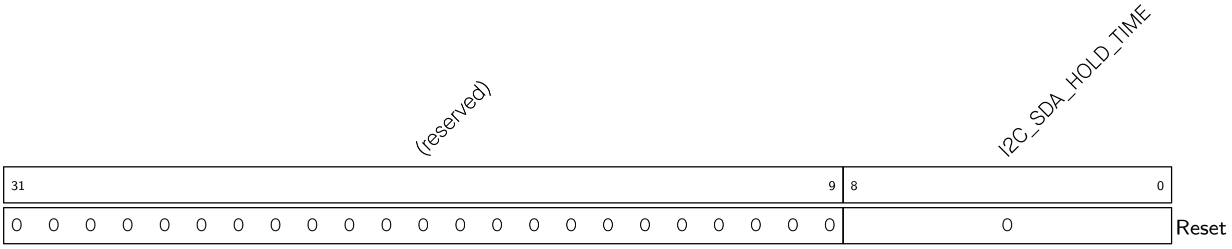
The addresses in this section are relative to I2C Controller base address provided in Table 3-3 in Chapter 3 *System and Memory*.

Register 28.1. I2C_SCL_LOW_PERIOD_REG (0x0000)



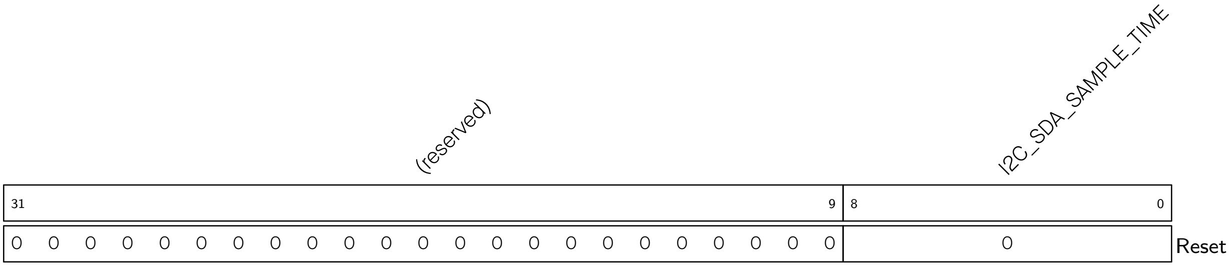
**I2C_SCL_LOW_PERIOD** This field is used to configure how long SCL remains low in master mode, in I2C module clock cycles. (R/W)

Register 28.2. I2C_SDA_HOLD_REG (0x0030)



**I2C_SDA_HOLD_TIME** This field is used to configure the time to hold the data after the falling edge of SCL, in I2C module clock cycles. (R/W)

Register 28.3. I2C_SDA_SAMPLE_REG (0x0034)



**I2C_SDA_SAMPLE_TIME** This field is used to configure how long SDA is sampled, in I2C module clock cycles. (R/W)

### Register 28.4. I2C_SCL_HIGH_PERIOD_REG (0x0038)

| (reserved) | I2C_SCL_WAIT_HIGH_PERIOD | I2C_SCL_HIGH_PERIOD | |
|---|---|---|---|
| 31 16 | 15 9 | 8 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**I2C_SCL_HIGH_PERIOD** This field is used to configure how long SCL remains high in master mode, in I2C module clock cycles. (R/W)

**I2C_SCL_WAIT_HIGH_PERIOD** This field is used to configure the SCL_FSM's waiting period for SCL high level in master mode, in I2C module clock cycles. (R/W)

### Register 28.5. I2C_SCL_START_HOLD_REG (0x0040)

| (reserved) | I2C_SCL_START_HOLD_TIME | |
|---|---|---|
| 31 9 | 8 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 8 | Reset |

**I2C_SCL_START_HOLD_TIME** This field is used to configure the time between the falling edge of SDA and the falling edge of SCL for a START condition, in I2C module clock cycles. (R/W)

### Register 28.6. I2C_SCL_RSTART_SETUP_REG (0x0044)

| (reserved) | I2C_SCL_RSTART_SETUP_TIME | |
|---|---|---|
| 31 9 | 8 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 8 | Reset |

**I2C_SCL_RSTART_SETUP_TIME** This field is used to configure the time between the rising edge of SCL and the falling edge of SDA for a RSTART condition, in I2C module clock cycles. (R/W)

### Register 28.7. I2C_SCL_STOP_HOLD_REG (0x0048)

| (reserved) | I2C_SCL_STOP_HOLD_TIME | |
|---|---|---|
| 31 9 | 8 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 8 | Reset |

**I2C_SCL_STOP_HOLD_TIME** This field is used to configure the delay after the STOP condition, in I2C module clock cycles. (R/W)

### Register 28.8. I2C_SCL_STOP_SETUP_REG (0x004C)

| (reserved) | I2C_SCL_STOP_SETUP_TIME | |
|---|---|---|
| 31 9 | 8 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 8 | Reset |

**I2C_SCL_STOP_SETUP_TIME** This field is used to configure the time between the rising edge of SCL and the rising edge of SDA, in I2C module clock cycles. (R/W)

### Register 28.9. I2C_SCL_ST_TIME_OUT_REG (0x0078)

| (reserved) | I2C_SCL_ST_TO_I2C | |
|---|---|---|
| 31 5 | 4 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x10 | Reset |

**I2C_SCL_ST_TO_I2C** The maximum time that SCL_FSM remains unchanged. It should be no more than 23. (R/W)

### Register 28.10. I2C_SCL_MAIN_ST_TIME_OUT_REG (0x007C)

| (reserved) | I2C_SCL_MAIN_ST_TO_I2C | |
|---|---|---|
| 31 5 | 4 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x10 | Reset |

**I2C_SCL_MAIN_ST_TO_I2C** The maximum time that SCL_MAIN_FSM remains unchanged. It should be no more than 23. (R/W)

**Register 28.11. I2C_CTR_REG (0x0004)**

| (reserved) | I2C_ADDR_BROADCASTING_EN | I2C_ADDR_10BIT_RW_CHECK_EN | I2C_SLV_TX_AUTO_START_EN | I2C_CONF_UPGATE | I2C_FSM_RST | I2C_ARBITRATION_EN | I2C_CLK_EN | I2C_RX_LSB_FIRST | I2C_TX_LSB_FIRST | I2C_TRANS_START | I2C_MS_MODE | I2C_RX_FULL_ACK_LEVEL | I2C_SAMPLE_SCL_LEVEL | I2C_SCL_FORCE_OUT | I2C_SDA_FORCE_OUT | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | Reset |

**I2C_SDA_FORCE_OUT** Configures the SDA output mode.

0: Open drain output

1: Direct output

(R/W)

**I2C_SCL_FORCE_OUT** Configures the SCL output mode.

0: Open drain output

1: Direct output

(R/W)

**I2C_SAMPLE_SCL_LEVEL** This bit is used to select the sampling mode. 0: samples SDA data on the SCL high level; 1: samples SDA data on the SCL low level. (R/W)

**I2C_RX_FULL_ACK_LEVEL** This bit is used to configure the ACK value that need to be sent by master when I2C_RXFIFO_CNT has reached the threshold. (R/W)

**I2C_MS_MODE** Set this bit to configure the I2C controller as an I2C Master. Clear this bit to configure the I2C controller as a slave. (R/W)

**I2C_TRANS_START** Set this bit to start sending the data in TX FIFO. (WT)

**I2C_TX_LSB_FIRST** This bit is used to control the order to send data. 0: sends data from the most significant bit; 1: sends data from the least significant bit. (R/W)

**I2C_RX_LSB_FIRST** This bit is used to control the order to receive data. 0: receives data from the most significant bit; 1: receives data from the least significant bit. (R/W)

**I2C_CLK_EN** This field controls APB_CLK clock gating. 0: APB_CLK is gated to save power; 1: APB_CLK is always on. (R/W)

**I2C_ARBITRATION_EN** This is the enable bit for I2C bus arbitration function. (R/W)

**I2C_FSM_RST** This bit is used to reset the SCL_FSM. (WT)

**I2C_CONF_UPGATE** Synchronization bit. (WT)

**I2C_SLV_TX_AUTO_START_EN** This is the enable bit for slave to send data automatically. (R/W)

**I2C_ADDR_10BIT_RW_CHECK_EN** This is the enable bit to check if the R/W bit of 10-bit addressing is consistent with the I2C protocol. (R/W)

**I2C_ADDR_BROADCASTING_EN** This is the enable bit for 7-bit general call addressing. (R/W)

### Register 28.12. I2C_TO_REG (0x000C)

| (reserved) | I2C_TIME_OUT_EN | I2C_TIME_OUT_VALUE | |
|---|---|---|---|
| 31 6 | 5 | 4 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0x10 | Reset |

**I2C_TIME_OUT_VALUE** This field is used to configure the timeout value for receiving a data bit in I2C_SCLK clock cycles. The configured timeout value equals $2^{I2C\_TIME\_OUT\_VALUE}$ clock cycles. (R/W)

**I2C_TIME_OUT_EN** This is the enable bit for timeout control. (R/W)

### Register 28.13. I2C_SLAVE_ADDR_REG (0x0010)

| I2C_ADDR_10BIT_EN | (reserved) | I2C_SLAVE_ADDR | |
|---|---|---|---|
| 31 | 30 15 | 14 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_SLAVE_ADDR** When the I2C controller is in slave mode, this field is used to configure the slave address. (R/W)

**I2C_ADDR_10BIT_EN** This field is used to enable the 10-bit addressing mode in master mode. (R/W)

### Register 28.14. I2C_FIFO_CONF_REG (0x0018)

| (reserved) | I2C_FIFO_PRT_EN | I2C_TX_FIFO_RST | I2C_RX_FIFO_RST | I2C_FIFO_ADDR_CFG_EN | I2C_NONFIFO_EN | I2C_TXFIFO_WM_THRHD | I2C_RXFIFO_WM_THRHD | |
|---|---|---|---|---|---|---|---|---|
| 31 15 | 14 | 13 | 12 | 11 | 10 | 9 5 | 4 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 | 0 | 0 | 0 | 0 | 0x4 | 0xb | Reset |

**I2C_RXFIFO_WM_THRHD** The watermark threshold of RX FIFO in non-FIFO mode. When I2C_FIFO_PRT_EN is 1 and RX FIFO counter is bigger than I2C_RXFIFO_WM_THRHD[4:0], I2C_RXFIFO_WM_INT_RAW bit is valid. (R/W)

**I2C_TXFIFO_WM_THRHD** The watermark threshold of TX FIFO in non-FIFO mode. When I2C_FIFO_PRT_EN is 1 and TX FIFO counter is smaller than I2C_TXFIFO_WM_THRHD[4:0], I2C_TXFIFO_WM_INT_RAW bit is valid. (R/W)

**I2C_NONFIFO_EN** Set this bit to enable APB non-FIFO mode. (R/W)

**I2C_FIFO_ADDR_CFG_EN** When this bit is set to 1, the byte received after the I2C address byte represents the offset address in the I2C Slave RAM. (R/W)

**I2C_RX_FIFO_RST** Set this bit to reset RX FIFO. (R/W)

**I2C_TX_FIFO_RST** Set this bit to reset TX FIFO. (R/W)

**I2C_FIFO_PRT_EN** The control enable bit of FIFO pointer in non-FIFO mode. This bit controls the valid bits and TX/RX FIFO overflow, underflow, full and empty interrupts. (R/W)

### Register 28.15. I2C_FILTER_CFG_REG (0x0050)

| (reserved) | I2C_SDA_FILTER_EN | I2C_SCL_FILTER_EN | I2C_SDA_FILTER_THRES | I2C_SCL_FILTER_THRES | |
|---|---|---|---|---|---|
| 31 10 | 9 | 8 | 7 4 | 3 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 | 1 | 0 | 0 | Reset |

**I2C_SCL_FILTER_THRES** When a pulse on the SCL input has smaller width than the value of this field in I2C module clock cycles, the I2C controller ignores that pulse. (R/W)

**I2C_SDA_FILTER_THRES** When a pulse on the SDA input has smaller width than the value of this field in I2C module clock cycles, the I2C controller ignores that pulse. (R/W)

**I2C_SCL_FILTER_EN** This is the filter enable bit for SCL. (R/W)

**I2C_SDA_FILTER_EN** This is the filter enable bit for SDA. (R/W)

### Register 28.16. I2C_CLK_CONF_REG (0x0054)

| (reserved) | I2C_SCLK_ACTIVE | I2C_SCLK_SEL | I2C_SCLK_DIV_B | I2C_SCLK_DIV_A | I2C_SCLK_DIV_NUM | |
|---|---|---|---|---|---|---|
| 31 22 | 21 | 20 | 19 14 | 13 8 | 7 0 | |
| 0 0 0 0 0 0 0 0 0 0 | 1 | 0 | 0 | 0 | 0 | Reset |

**I2C_SCLK_DIV_NUM** The integral part of the divisor. (R/W)

**I2C_SCLK_DIV_A** The numerator of the divisor's fractional part. (R/W)

**I2C_SCLK_DIV_B** The denominator of the divisor's fractional part. (R/W)

**I2C_SCLK_SEL** The clock selection bit for the I2C controller. 0: XTAL_CLK; 1: RC_FAST_CLK. (R/W)

**I2C_SCLK_ACTIVE** The clock switch bit for the I2C controller. (R/W)

### Register 28.17. I2C_SCL_SP_CONF_REG (0x0080)

| (reserved) | I2C_SDA_PD_EN | I2C_SCL_PD_EN | I2C_SCL_RST_SLV_NUM | I2C_SCL_RST_SLV_EN | |
|---|---|---|---|---|---|
| 31 8 | 7 | 6 | 5 1 | 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | Reset |

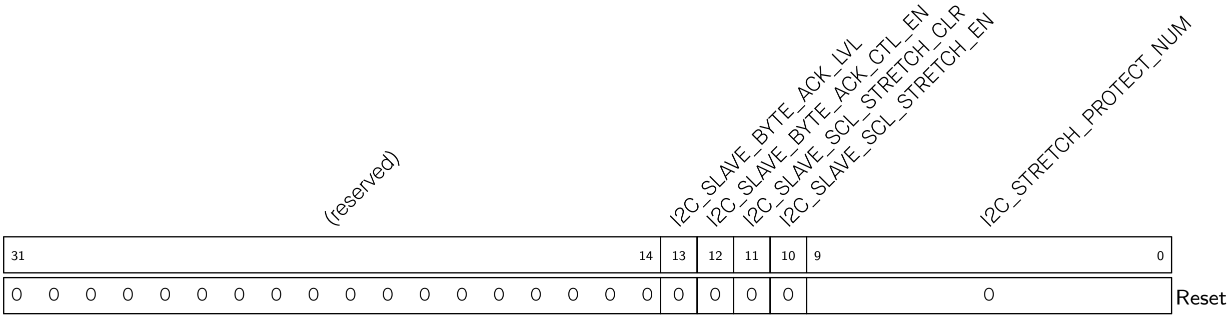
**I2C_SCL_RST_SLV_EN** When the master is idle, set this bit to send out SCL pulses. The number of pulses equals to I2C_SCL_RST_SLV_NUM[4:0]. (R/W/SC)

**I2C_SCL_RST_SLV_NUM** Configures the pulses of SCL generated in master mode. Valid when I2C_SCL_RST_SLV_EN is 1. (R/W)

**I2C_SCL_PD_EN** The power down enable bit for the I2C output SCL line. 0: Not power down; 1: Power down. Set I2C_SCL_FORCE_OUT and I2C_SCL_PD_EN to 1 to stretch SCL low. (R/W)

**I2C_SDA_PD_EN** The power down enable bit for the I2C output SDA line. 0: Not power down; 1: Power down. Set I2C_SDA_FORCE_OUT and I2C_SDA_PD_EN to 1 to stretch SDA low. (R/W)

### Register 28.18. I2C_SCL_STRETCH_CONF_REG (0x0084)



| (reserved) | I2C_SLAVE_BYTE_ACK_LVL | I2C_SLAVE_BYTE_ACK_CTL_EN | I2C_SLAVE_SCL_STRETCH_CLR | I2C_SLAVE_SCL_STRETCH_EN | I2C_STRETCH_PROTECT_NUM | |
|---|---|---|---|---|---|---|
| 31 14 | 13 | 12 | 11 | 10 | 9 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**I2C_STRETCH_PROTECT_NUM** Configures the time period to release the SCL line from stretching to avoid timing violation. Usually it should be larger than the SDA steup time. (R/W)

**I2C_SLAVE_SCL_STRETCH_EN** The enable bit for SCL clock stretching. 0: Disable; 1: Enable. The SCL output line will be stretched low when I2C_SLAVE_SCL_STRETCH_EN is 1 and one of the four stretching events occurs. The cause of stretching can be seen in I2C_STRETCH_CAUSE. (R/W)

**I2C_SLAVE_SCL_STRETCH_CLR** Set this bit to clear SCL clock stretching. (WT)

**I2C_SLAVE_BYTE_ACK_CTL_EN** The enable bit for slave to control the level of the ACK bit. (R/W)

**I2C_SLAVE_BYTE_ACK_LVL** Set the level of the ACK bit when I2C_SLAVE_BYTE_ACK_CTL_EN is set. (R/W)

**Register 28.19. I2C_SR_REG (0x0008)**

| (reserved) | I2C_SCL_STATE_LAST | (reserved) | I2C_SCL_MAIN_STATE_LAST | I2C_TXFIFO_CNT | (reserved) | I2C_STRETCH_CAUSE | I2C_RXFIFO_CNT | (reserved) | I2C_SLAVE_ADDRESSED | I2C_BUS_BUSY | I2C_ARB_LOST | (reserved) | I2C_SLAVE_RW | I2C_RESP_REC | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 28 | 27 | 26 24 | 23 18 | 17 16 | 15 14 | 13 8 | 7 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 0 | 0x3 | 0 | 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**I2C_RESP_REC** The received ACK value in master mode or slave mode. 0: ACK; 1: NACK. (RO)

**I2C_SLAVE_RW** When in slave mode, 0: master writes to slave; 1: master reads from slave. (RO)

**I2C_ARB_LOST** When the I2C controller loses control of the SCL line, this bit changes to 1. (RO)

**I2C_BUS_BUSY** 0: The I2C bus is in idle state; 1: The I2C bus is busy transferring data. (RO)

**I2C_SLAVE_ADDRESSED** When the I2C controller is in slave mode, and the address sent by the master matches the address of the slave, this bit is at high level. (RO)

**I2C_RXFIFO_CNT** This field represents the number of data bytes to be sent. (RO)

**I2C_STRETCH_CAUSE** The cause of SCL clock stretching in slave mode. 0: stretching SCL low when the master starts to read data; 1: stretching SCL low when TX FIFO is empty in slave mode; 2: stretching SCL low when RX FIFO is full in slave mode. (RO)

**I2C_TXFIFO_CNT** This field stores the number of data bytes received in RAM. (RO)

**I2C_SCL_MAIN_STATE_LAST** This field indicates the status of the state machine. 0: idle; 1: address shift; 2: ACK address; 3: receive data; 4: transmit data; 5: send ACK; 6: wait for ACK. (RO)

**I2C_SCL_STATE_LAST** This field indicates the status of the state machine used to produce SCL. 0: idle; 1: start; 2: falling edge; 3: low; 4: rising edge; 5: high; 6: stop. (RO)

### Register 28.20. I2C_FIFO_ST_REG (0x0014)

| (reserved) | I2C_SLAVE_RW_POINT | (reserved) | I2C_TXFIFO_WADDR | I2C_TXFIFO_RADDR | I2C_RXFIFO_WADDR | I2C_RXFIFO_RADDR | |
|---|---|---|---|---|---|---|---|
| 31 30 | 29 22 | 21 20 | 19 15 | 14 10 | 9 5 | 4 0 | |
| 0 0 | 0 | 0 0 | 0 | 0 | 0 | 0 | Reset |

**I2C_RXFIFO_RADDR** This is the offset address of the APB reading from RX FIFO. (RO)

**I2C_RXFIFO_WADDR** This is the offset address of the I2C controller receiving data and writing to RX FIFO. (RO)

**I2C_TXFIFO_RADDR** This is the offset address of the I2C controller reading from TX FIFO. (RO)

**I2C_TXFIFO_WADDR** This is the offset address of APB bus writing to TX FIFO. (RO)

**I2C_SLAVE_RW_POINT** The received data in I2C slave mode. (RO)

### Register 28.21. I2C_DATA_REG (0x001C)

| (reserved) | I2C_FIFO_RDATA | |
|---|---|---|
| 31 8 | 7 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_FIFO_RDATA** This field is used to read data from RX FIFO, or write data to TX FIFO. (R/W)

### Register 28.22. I2C_INT_RAW_REG (0x0020)

| (reserved) | I2C_GENERAL_CALL_INT_RAW | I2C_SLAVE_STRETCH_INT_RAW | I2C_DET_START_INT_RAW | I2C_SCL_MAIN_ST_TO_INT_RAW | I2C_SCL_ST_TO_INT_RAW | I2C_RXFIFO_UDF_INT_RAW | I2C_TXFIFO_OVF_INT_RAW | I2C_NACK_INT_RAW | I2C_TRANS_START_INT_RAW | I2C_TIME_OUT_INT_RAW | I2C_TRANS_COMPLETE_INT_RAW | I2C_MST_TXFIFO_UDF_INT_RAW | I2C_ARBITRATION_LOST_INT_RAW | I2C_BYTE_TRANS_DONE_INT_RAW | I2C_END_DETECT_INT_RAW | I2C_RXFIFO_OVF_INT_RAW | I2C_TXFIFO_WM_INT_RAW | I2C_RXFIFO_WM_INT_RAW | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Reset |

**I2C_RXFIFO_WM_INT_RAW** The raw interrupt bit for the I2C_RXFIFO_WM_INT interrupt. (R/SS/WTC)

**I2C_TXFIFO_WM_INT_RAW** The raw interrupt bit for the I2C_TXFIFO_WM_INT interrupt. (R/SS/WTC)

**I2C_RXFIFO_OVF_INT_RAW** The raw interrupt bit for the I2C_RXFIFO_OVF_INT interrupt. (R/SS/WTC)

**I2C_END_DETECT_INT_RAW** The raw interrupt bit for the I2C_END_DETECT_INT interrupt. (R/SS/WTC)

**I2C_BYTE_TRANS_DONE_INT_RAW** The raw interrupt bit for the I2C_BYTE_TRANS_DONE_INT interrupt. (R/SS/WTC)

**I2C_ARBITRATION_LOST_INT_RAW** The raw interrupt bit for the I2C_ARBITRATION_LOST_INT interrupt. (R/SS/WTC)

**I2C_MST_TXFIFO_UDF_INT_RAW** The raw interrupt bit for the I2C_MST_TXFIFO_UDF_INT interrupt. (R/SS/WTC)

**I2C_TRANS_COMPLETE_INT_RAW** The raw interrupt bit for the I2C_TRANS_COMPLETE_INT interrupt. (R/SS/WTC)

**I2C_TIME_OUT_INT_RAW** The raw interrupt bit for the I2C_TIME_OUT_INT interrupt. (R/SS/WTC)

**I2C_TRANS_START_INT_RAW** The raw interrupt bit for the I2C_TRANS_START_INT interrupt. (R/SS/WTC)

**I2C_NACK_INT_RAW** The raw interrupt bit for the I2C_NACK_INT interrupt. (R/SS/WTC)

**I2C_TXFIFO_OVF_INT_RAW** The raw interrupt bit for the I2C_TXFIFO_OVF_INT interrupt. (R/SS/WTC)

**I2C_RXFIFO_UDF_INT_RAW** The raw interrupt bit for the I2C_RXFIFO_UDF_INT interrupt. (R/SS/WTC)

**Continued on the next page...**

### Register 28.22. I2C_INT_RAW_REG (0x0020)

Continued from the previous page...

**I2C_SCL_ST_TO_INT_RAW** The raw interrupt bit for the I2C_SCL_ST_TO_INT interrupt. (R/SS/WTC)

**I2C_SCL_MAIN_ST_TO_INT_RAW** The raw interrupt bit for the I2C_SCL_MAIN_ST_TO_INT interrupt. (R/SS/WTC)

**I2C_DET_START_INT_RAW** The raw interrupt bit for the I2C_DET_START_INT interrupt. (R/SS/WTC)

**I2C_SLAVE_STRETCH_INT_RAW** The raw interrupt bit for the I2C_SLAVE_STRETCH_INT interrupt. (R/SS/WTC)

**I2C_GENERAL_CALL_INT_RAW** The raw interrupt bit for the I2C_GENARAL_CALL_INT interrupt. (R/SS/WTC)

### Register 28.23. I2C_INT_CLR_REG (0x0024)

| (reserved) | I2C_GENERAL_CALL_INT_CLR | I2C_SLAVE_STRETCH_INT_CLR | I2C_DET_START_INT_CLR | I2C_SCL_MAIN_ST_TO_INT_CLR | I2C_SCL_ST_TO_INT_CLR | I2C_RXFIFO_UDF_INT_CLR | I2C_TXFIFO_OVF_INT_CLR | I2C_NACK_INT_CLR | I2C_TRANS_START_INT_CLR | I2C_TIME_OUT_INT_CLR | I2C_TRANS_COMPLETE_INT_CLR | I2C_MST_TXFIFO_UDF_INT_CLR | I2C_ARBITRATION_LOST_INT_CLR | I2C_BYTE_TRANS_DONE_INT_CLR | I2C_END_DETECT_INT_CLR | I2C_RXFIFO_OVF_INT_CLR | I2C_TXFIFO_WM_INT_CLR | I2C_RXFIFO_WM_INT_CLR | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**I2C_RXFIFO_WM_INT_CLR** Set this bit to clear the I2C_RXFIFO_WM_INT interrupt. (WT)

**I2C_TXFIFO_WM_INT_CLR** Set this bit to clear the I2C_TXFIFO_WM_INT interrupt. (WT)

**I2C_RXFIFO_OVF_INT_CLR** Set this bit to clear the I2C_RXFIFO_OVF_INT interrupt. (WT)

**I2C_END_DETECT_INT_CLR** Set this bit to clear the I2C_END_DETECT_INT interrupt. (WT)

**I2C_BYTE_TRANS_DONE_INT_CLR** Set this bit to clear the I2C_BYTE_TRANS_DONE_INT interrupt. (WT)

**I2C_ARBITRATION_LOST_INT_CLR** Set this bit to clear the I2C_ARBITRATION_LOST_INT interrupt. (WT)

**I2C_MST_TXFIFO_UDF_INT_CLR** Set this bit to clear the I2C_MST_TXFIFO_UDF_INT interrupt. (WT)

**I2C_TRANS_COMPLETE_INT_CLR** Set this bit to clear the I2C_TRANS_COMPLETE_INT interrupt. (WT)

**I2C_TIME_OUT_INT_CLR** Set this bit to clear the I2C_TIME_OUT_INT interrupt. (WT)

**I2C_TRANS_START_INT_CLR** Set this bit to clear the I2C_TRANS_START_INT interrupt. (WT)

**I2C_NACK_INT_CLR** Set this bit to clear the I2C_NACK_INT interrupt. (WT)

**I2C_TXFIFO_OVF_INT_CLR** Set this bit to clear the I2C_TXFIFO_OVF_INT interrupt. (WT)

**I2C_RXFIFO_UDF_INT_CLR** Set this bit to clear the I2C_RXFIFO_UDF_INT interrupt. (WT)

**I2C_SCL_ST_TO_INT_CLR** Set this bit to clear the I2C_SCL_ST_TO_INT interrupt. (WT)

**I2C_SCL_MAIN_ST_TO_INT_CLR** Set this bit to clear the I2C_SCL_MAIN_ST_TO_INT interrupt. (WT)

**I2C_DET_START_INT_CLR** Set this bit to clear the I2C_DET_START_INT interrupt. (WT)

**I2C_SLAVE_STRETCH_INT_CLR** Set this bit to clear the I2C_SLAVE_STRETCH_INT interrupt. (WT)

**I2C_GENERAL_CALL_INT_CLR** Set this bit for the I2C_GENARAL_CALL_INT interrupt. (WT)

### Register 28.24. I2C_INT_ENA_REG (0x0028)

| (reserved) | I2C_GENERAL_CALL_INT_ENA | I2C_SLAVE_STRETCH_INT_ENA | I2C_DET_START_INT_ENA | I2C_SCL_MAIN_ST_TO_INT_ENA | I2C_SCL_ST_TO_INT_ENA | I2C_RXFIFO_UDF_INT_ENA | I2C_TXFIFO_OVF_INT_ENA | I2C_NACK_INT_ENA | I2C_TRANS_START_INT_ENA | I2C_TIME_OUT_INT_ENA | I2C_TRANS_COMPLETE_INT_ENA | I2C_MST_TXFIFO_UDF_INT_ENA | I2C_ARBITRATION_LOST_INT_ENA | I2C_BYTE_TRANS_DONE_INT_ENA | I2C_END_DETECT_INT_ENA | I2C_RXFIFO_OVF_INT_ENA | I2C_TXFIFO_WM_INT_ENA | I2C_RXFIFO_WM_INT_ENA | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**I2C_RXFIFO_WM_INT_ENA** The interrupt enable bit for the I2C_RXFIFO_WM_INT interrupt. (R/W)

**I2C_TXFIFO_WM_INT_ENA** The interrupt enable bit for the I2C_TXFIFO_WM_INT interrupt. (R/W)

**I2C_RXFIFO_OVF_INT_ENA** The interrupt enable bit for the I2C_RXFIFO_OVF_INT interrupt. (R/W)

**I2C_END_DETECT_INT_ENA** The interrupt enable bit for the I2C_END_DETECT_INT interrupt. (R/W)

**I2C_BYTE_TRANS_DONE_INT_ENA** The interrupt enable bit for the I2C_BYTE_TRANS_DONE_INT interrupt. (R/W)

**I2C_ARBITRATION_LOST_INT_ENA** The interrupt enable bit for the I2C_ARBITRATION_LOST_INT interrupt. (R/W)

**I2C_MST_TXFIFO_UDF_INT_ENA** The interrupt enable bit for the I2C_MST_TXFIFO_UDF_INT interrupt. (R/W)

**I2C_TRANS_COMPLETE_INT_ENA** The interrupt enable bit for the I2C_TRANS_COMPLETE_INT interrupt. (R/W)

**I2C_TIME_OUT_INT_ENA** The interrupt enable bit for the I2C_TIME_OUT_INT interrupt. (R/W)

**I2C_TRANS_START_INT_ENA** The interrupt enable bit for the I2C_TRANS_START_INT interrupt. (R/W)

**I2C_NACK_INT_ENA** The interrupt enable bit for the I2C_NACK_INT interrupt. (R/W)

**I2C_TXFIFO_OVF_INT_ENA** The interrupt enable bit for the I2C_TXFIFO_OVF_INT interrupt. (R/W)

**I2C_RXFIFO_UDF_INT_ENA** The interrupt enable bit for the I2C_RXFIFO_UDF_INT interrupt. (R/W)

**I2C_SCL_ST_TO_INT_ENA** The interrupt enable bit for the I2C_SCL_ST_TO_INT interrupt. (R/W)

**I2C_SCL_MAIN_ST_TO_INT_ENA** The interrupt enable bit for the I2C_SCL_MAIN_ST_TO_INT interrupt. (R/W)

**I2C_DET_START_INT_ENA** The interrupt enable bit for the I2C_DET_START_INT interrupt. (R/W)

**I2C_SLAVE_STRETCH_INT_ENA** The interrupt enable bit for the I2C_SLAVE_STRETCH_INT interrupt. (R/W)

**I2C_GENERAL_CALL_INT_ENA** The interrupt enable bit for the I2C_GENARAL_CALL_INT interrupt. (R/W)

### Register 28.25. I2C_INT_STATUS_REG (0x002C)

| (reserved) | I2C_GENERAL_CALL_INT_ST | I2C_SLAVE_STRETCH_INT_ST | I2C_DET_START_INT_ST | I2C_SCL_MAIN_ST_TO_INT_ST | I2C_SCL_ST_TO_INT_ST | I2C_RXFIFO_UDF_INT_ST | I2C_TXFIFO_OVF_INT_ST | I2C_NACK_INT_ST | I2C_TRANS_START_INT_ST | I2C_TIME_OUT_INT_ST | I2C_TRANS_COMPLETE_INT_ST | I2C_MST_TXFIFO_UDF_INT_ST | I2C_ARBITRATION_LOST_INT_ST | I2C_BYTE_TRANS_DONE_INT_ST | I2C_END_DETECT_INT_ST | I2C_RXFIFO_OVF_INT_ST | I2C_TXFIFO_WM_INT_ST | I2C_RXFIFO_WM_INT_ST | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**I2C_RXFIFO_WM_INT_ST** The masked interrupt status bit for the I2C_RXFIFO_WM_INT interrupt. (RO)

**I2C_TXFIFO_WM_INT_ST** The masked interrupt status bit for the I2C_TXFIFO_WM_INT interrupt. (RO)

**I2C_RXFIFO_OVF_INT_ST** The masked interrupt status bit for the I2C_RXFIFO_OVF_INT interrupt. (RO)

**I2C_END_DETECT_INT_ST** The masked interrupt status bit for the I2C_END_DETECT_INT interrupt. (RO)

**I2C_BYTE_TRANS_DONE_INT_ST** The masked interrupt status bit for the I2C_BYTE_TRANS_DONE_INT interrupt. (RO)

**I2C_ARBITRATION_LOST_INT_ST** The masked interrupt status bit for the I2C_ARBITRATION_LOST_INT interrupt. (RO)

**I2C_MST_TXFIFO_UDF_INT_ST** The masked interrupt status bit for the I2C_MST_TXFIFO_UDF_INT interrupt. (RO)

**I2C_TRANS_COMPLETE_INT_ST** The masked interrupt status bit for the I2C_TRANS_COMPLETE_INT interrupt. (RO)

**I2C_TIME_OUT_INT_ST** The masked interrupt status bit for the I2C_TIME_OUT_INT interrupt. (RO)

**I2C_TRANS_START_INT_ST** The masked interrupt status bit for the I2C_TRANS_START_INT interrupt. (RO)

**I2C_NACK_INT_ST** The masked interrupt status bit for the I2C_NACK_INT interrupt. (RO)

**I2C_TXFIFO_OVF_INT_ST** The masked interrupt status bit for the I2C_TXFIFO_OVF_INT interrupt. (RO)

**I2C_RXFIFO_UDF_INT_ST** The masked interrupt status bit for the I2C_RXFIFO_UDF_INT interrupt. (RO)

**Continued on the next page...**

### Register 28.25. I2C_INT_STATUS_REG (0x002C)

**Continued from the previous page...**

**I2C_SCL_ST_TO_INT_ST** The masked interrupt status bit for the I2C_SCL_ST_TO_INT interrupt. (RO)

**I2C_SCL_MAIN_ST_TO_INT_ST** The masked interrupt status bit for the I2C_SCL_MAIN_ST_TO_INT interrupt. (RO)

**I2C_DET_START_INT_ST** The masked interrupt status bit for the I2C_DET_START_INT interrupt. (RO)

**I2C_SLAVE_STRETCH_INT_ST** The masked interrupt status bit for the I2C_SLAVE_STRETCH_INT interrupt. (RO)

**I2C_GENERAL_CALL_INT_ST** The masked interrupt status bit for the I2C_GENARAL_CALL_INT interrupt. (RO)

### Register 28.26. I2C_COMD0_REG (0x0058)

| I2C_COMMAND0_DONE | (reserved) | I2C_COMMAND0 | |
|---|---|---|---|
| 31 | 30 14 | 13 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

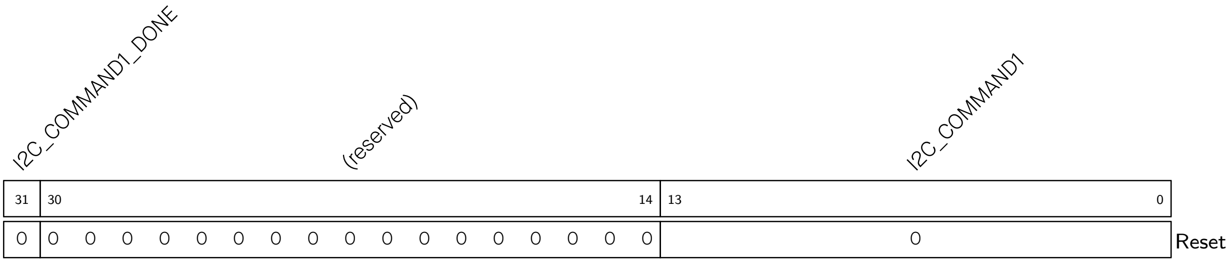
**I2C_COMMAND0** This is the content of command register 0. It consists of three parts:

- op_code is the command. 1: WRITE; 2: STOP; 3: READ; 4: END; 6: RSTART.
- Byte_num represents the number of bytes that need to be sent or received.
- ack_check_en, ack_exp and ack are used to control the ACK bit. For more information, see Section 28.4.9.

(R/W)

**I2C_COMMAND0_DONE** When command 0 has been executed in master mode, this bit changes to high level. (R/W/SS)

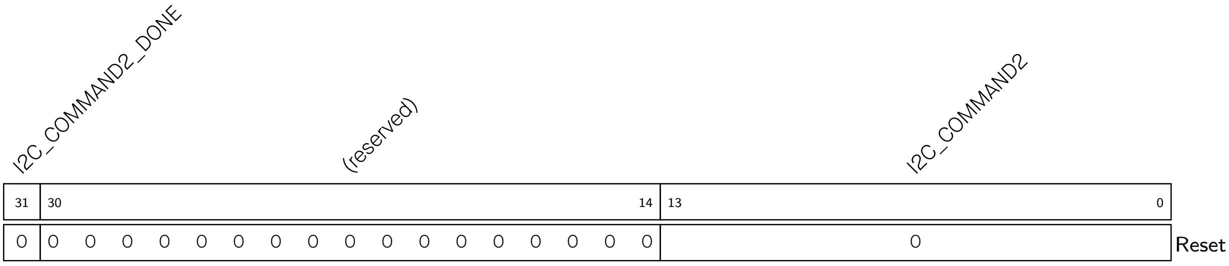
### Register 28.27. I2C_COMD1_REG (0x005C)



**I2C_COMMAND1** This is the content of command register 1. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND1_DONE** When command 1 has been executed in master mode, this bit changes to high level. (R/W/SS)

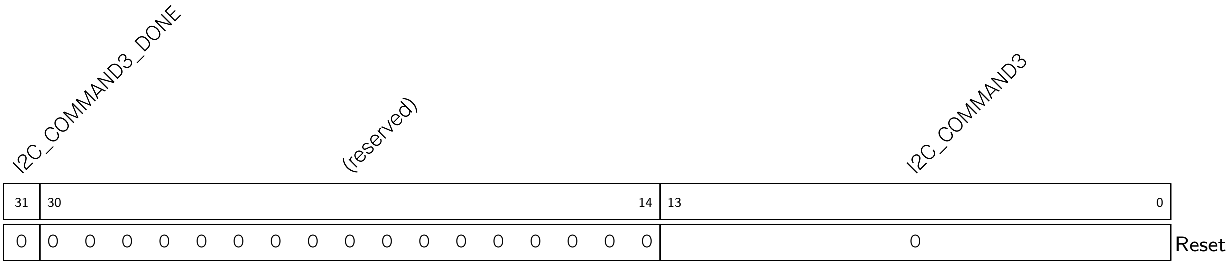
### Register 28.28. I2C_COMD2_REG (0x0060)



**I2C_COMMAND2** This is the content of command register 2. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND2_DONE** When command 2 has been executed in master mode, this bit changes to high Level. (R/W/SS)

### Register 28.29. I2C_COMD3_REG (0x0064)



**I2C_COMMAND3** This is the content of command register 3. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND3_DONE** When command 3 has been executed in master mode, this bit changes to high level. (R/W/SS)

### Register 28.30. I2C_COMD4_REG (0x0068)

| I2C_COMMAND4_DONE | (reserved) | I2C_COMMAND4 | |
|---|---|---|---|
| 31 | 30 14 | 13 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND4** This is the content of command register 4. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND4_DONE** When command 4 has been executed in master mode, this bit changes to high level. (R/W/SS)

### Register 28.31. I2C_COMD5_REG (0x006C)

| I2C_COMMAND5_DONE | (reserved) | I2C_COMMAND5 | |
|---|---|---|---|
| 31 | 30 14 | 13 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND5** This is the content of command register 5. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND5_DONE** When command 5 has been executed in master mode, this bit changes to high level. (R/W/SS)

### Register 28.32. I2C_COMD6_REG (0x0070)

| I2C_COMMAND6_DONE | (reserved) | I2C_COMMAND6 | |
|---|---|---|---|
| 31 | 30 14 | 13 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND6** This is the content of command register 6. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND6_DONE** When command 6 has been executed in master mode, this bit changes to high level. (R/W/SS)

Register 28.33. I2C_COMD7_REG (0x0074)

| I2C_COMMAND7_DONE | (reserved) | I2C_COMMAND7 | |
|---|---|---|---|
| 31 | 30 14 | 13 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND7** This is the content of command register 7. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND7_DONE** When command 7 has been executed in master mode, this bit changes to high level. (R/W/SS)

Register 28.34. I2C_DATE_REG (0x00F8)

| I2C_DATE | |
|---|---|
| 31 0 | |
| 0x20070201 | Reset |

**I2C_DATE** This is the version control register. (R/W)