LC26G (AB)&LC26G-T (AA)&LC76G Series
I2C Application Note

GNSS Module Series

Version: 1.2

Date: 2025-01-07

Status: Released

At Quectel, our aim is to provide timely and comprehensive services to our customers. If you require any assistance, please contact our headquarters:

Quectel Wireless Solutions Co., Ltd.
Building 5, Shanghai Business Park Phase III (Area B), No.1016 Tianlin Road, Minhang District, Shanghai 200233, China
Tel: +86 21 5108 6236
Email: info@quectel.com

Or our local offices. For more information, please visit:
http://www.quectel.com/support/sales.htm.

For technical support, or to report documentation errors, please visit:
http://www.quectel.com/support/technical.htm.
Or email us at: support@quectel.com.

Legal Notices

We offer information as a service to you. The provided information is based on your requirements and we make every effort to ensure its quality. You agree that you are responsible for using independent analysis and evaluation in designing intended products, and we provide reference designs for illustrative purposes only. Before using any hardware, software or service guided by this document, please read this notice carefully. Even though we employ commercially reasonable efforts to provide the best possible experience, you hereby acknowledge and agree that this document and related services hereunder are provided to you on an “as available” basis. We may revise or restate this document from time to time at our sole discretion without any prior notice to you.

Use and Disclosure Restrictions

License Agreements

Documents and information provided by us shall be kept confidential, unless specific permission is granted. They shall not be accessed or used for any purpose except as expressly provided herein.

Copyright

Our and third-party products hereunder may contain copyrighted material. Such copyrighted material shall not be copied, reproduced, distributed, merged, published, translated, or modified without prior written consent. We and the third party have exclusive rights over copyrighted material. No license shall be granted or conveyed under any patents, copyrights, trademarks, or service mark rights. To avoid ambiguities, purchasing in any form cannot be deemed as granting a license other than the normal non-exclusive, royalty-free license to use the material. We reserve the right to take legal action for noncompliance with abovementioned requirements, unauthorized use, or other illegal or malicious use of the material.

Trademarks

Except as otherwise set forth herein, nothing in this document shall be construed as conferring any rights to use any trademark, trade name or name, abbreviation, or counterfeit product thereof owned by Quectel or any third party in advertising, publicity, or other aspects.

Third-Party Rights

This document may refer to hardware, software and/or documentation owned by one or more third parties ("third-party materials"). Use of such third-party materials shall be governed by all restrictions and obligations applicable thereto.

We make no warranty or representation, either express or implied, regarding the third-party materials, including but not limited to any implied or statutory, warranties of merchantability or fitness for a particular purpose, quiet enjoyment, system integration, information accuracy, and non-infringement of any third-party intellectual property rights with regard to the licensed technology or use thereof. Nothing herein constitutes a representation or warranty by us to either develop, enhance, modify, distribute, market, sell, offer for sale, or otherwise maintain production of any our products or any other hardware, software, device, tool, information, or product. We moreover disclaim any and all warranties arising from the course of dealing or usage of trade.

Privacy Policy

To implement module functionality, certain device data are uploaded to Quectel's or third-party's servers, including carriers, chipset suppliers or customer-designated servers. Quectel, strictly abiding by the relevant laws and regulations, shall retain, use, disclose or otherwise process relevant data for the purpose of performing the service only or as permitted by applicable laws. Before data interaction with third parties, please be informed of their privacy and data security policy.

Disclaimer

a) We acknowledge no liability for any injury or damage arising from the reliance upon the information.
b) We shall bear no liability resulting from any inaccuracies or omissions, or from the use of the information contained herein.
c) While we have made every effort to ensure that the functions and features under development are free from errors, it is possible that they could contain errors, inaccuracies, and omissions. Unless otherwise provided by valid agreement, we make no warranties of any kind, either implied or express, and exclude all liability for any loss or damage suffered in connection with the use of features and functions under development, to the maximum extent permitted by law, regardless of whether such loss or damage may have been foreseeable.
d) We are not responsible for the accessibility, safety, accuracy, availability, legality, or completeness of information, advertising, commercial offers, products, services, and materials on third-party websites and third-party resources.

About the Document

Document Information	
Title	**LC26G (AB)&LC26G-T (AA)&LC76G Series I2C Application Note**
Subtitle	GNSS Module Series
Document Type	Application Note
Document Status	Released

Revision History

Revision	Date	Description
-	2022-07-21	Creation of the document
1.0	2022-09-16	First official release
1.1	2024-08-14	1. Added applicable module LC26G-T (AA). 2. Added a note indicating that I2C interface cannot be used for firmware upgrade (*Chapter 1*). 3. Added the following chapters: ● I2C communication protocol (*Chapter 2*); ● I2C multi-slave operation (*Chapter 4*); ● I2C read/write example (*Chapter 5*). 4. Updated the maximum capacity of the slave's transmit buffer (*Chapter 3*). 5. Updated the note on length of data read by the master and added a note on master data read timeout and recovery mechanism (*Chapter 3.1*). 6. Updated the note on length of data to be written (*Chapter 3.2*). 7. Updated data reading flow of master (*Figure 8*). 8. Updated the sample code by adding the action to recover the I2C bus (*Chapter 6*).
1.2	2025-01-07	Updated the note on master data read timeout and recovery mechanism (*Chapter 3.1*).

Contents

Table Index

Figure Index

1 Introduction

This document outlines the I2C function and its usage on Quectel LC26G (AB), LC26G-T (AA) and LC76G series modules. The modules always operate as slave devices when communicating with the master (client-side MCU). The master can read/write any data via I2C bus.

The features of the modules' I2C interface:

- Slave mode.
- Standard mode (100 kbps) and fast mode (400 kbps).
- 7-bit address format. Three device addresses are provided, namely configuration read/write address 0x50, read address 0x54, and write address 0x58.
- Supports sending and receiving variable length messages.
- I2C pins: I2C_SDA and I2C_SCL.

This document mainly explains how the master performs read and write operations on the modules via the I2C bus, including the I2C communication protocol, read/write process, example, and sample code.

NOTE

The I2C interface cannot be used for firmware upgrade.

2 I2C Communication Protocol

2.1. START and STOP Signals

The bus transaction begins with the transmission of a START (S) signal. A START signal is initiated by a HIGH to LOW transition on the SDA line while the SCL line is kept HIGH (see figure below). The bus is considered busy until the master generates a STOP signal (P) on the bus, which is defined as a LOW to HIGH transition on the SDA line while SCL is kept HIGH (see figure below). In addition, the bus remains busy if a repeated START (S) is generated instead of a STOP signal.

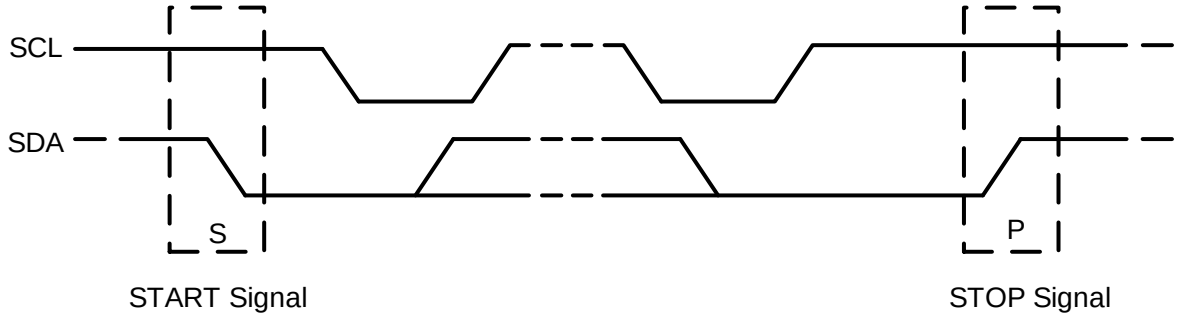


Figure 1: START and STOP Signals

2.2. Data Transfer and Acknowledge Signals

I2C data bytes are 8 bits long. Any number of bytes can be transmitted per transfer. Each transferred byte must be followed by an acknowledge signal. The clock for the acknowledge signal is generated by the master, while the receiver generates the acknowledge (ACK) signal by pulling down SDA and keeping it low (or the negative acknowledge (NAK) signal by pulling up SDA and keeping it high) during the HIGH phase of the acknowledge clock pulse.

If the slave is busy and unable to transmit or receive another byte of data until the ongoing processing task is performed, it can hold the SCL line LOW. This action forces the master into a wait state, effectively pausing the data transfer. Once the slave completes its task and is ready for another byte of data, it releases the clock line, allowing normal data transfer to resume.

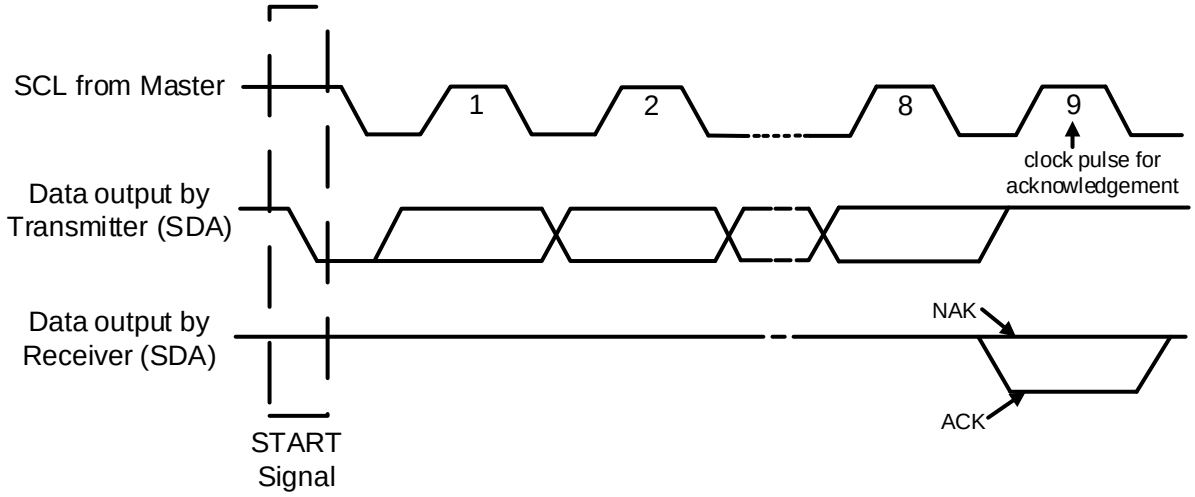


Figure 2: Acknowledge on I2C Bus

2.3. Communication

After initiating the communication with the START signal (S), the master sends a 7-bit slave address followed by an extra 8th bit, known as the read/write bit, to inform the slave if the master intends to write to it or read from it. If the bit is zero it indicates a write operation, whereas 1 indicates a read operation. Data is transferred on the SDA line, starting with the Most Significant bit (MSB). Then, the master releases the SDA line and waits for the acknowledge signal (ACK) from the slave device. The slave device must return an acknowledge bit for each transferred byte by pulling the SDA line LOW and keeping it LOW during the high period of the SCL line. To terminate data transmission, the master generates a STOP signal (P), thus freeing the communication line. However, the master can generate a repeated START signal (S) to address another slave without first generating a STOP signal (P). It is important to note that all SDA changes should take place when SCL is low, with the exception of START and STOP signals.

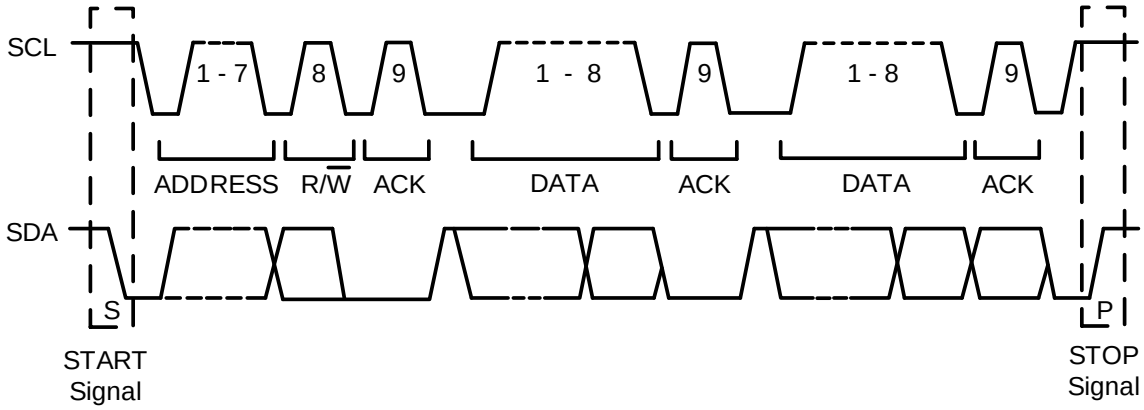


Figure 3: Complete I2C Data Transfer

2.4. Read Sequence

The sequence charts for reading data from the I2C slave are shown below.

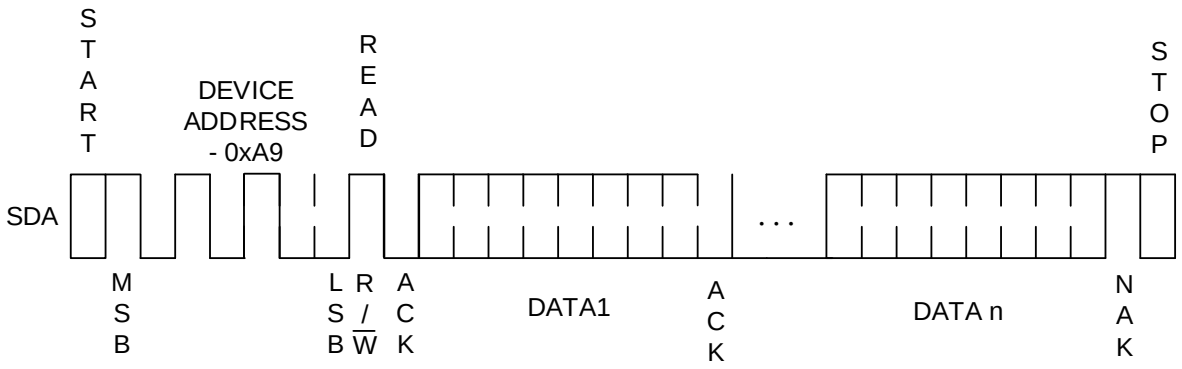


Figure 4: Read Sequence

To read the slave data, the master initiates the process by sending a START signal, followed by the I2C slave address and a read bit. As a result, the slave sends an ACK signal and the requested data. The communication ends with a NAK signal and a stop bit from the master. The above figures illustrate the sequences for reading multiple bytes of data.

NOTE

If the SDA line is accidentally locked in a low-level state, the master should first release the SDA line and then send 9 clock pulses through the SCL line. The slave releases control of the SDA line upon receiving any of the 9 clock pulses, thereby allowing it to return to a normal state. If this operation fails to successfully restore the SDA line to a normal state, a hardware reset is required to clear the erroneous state on the bus.

2.5. Write Sequence

The sequence charts for writing data to the I2C slave are shown below.

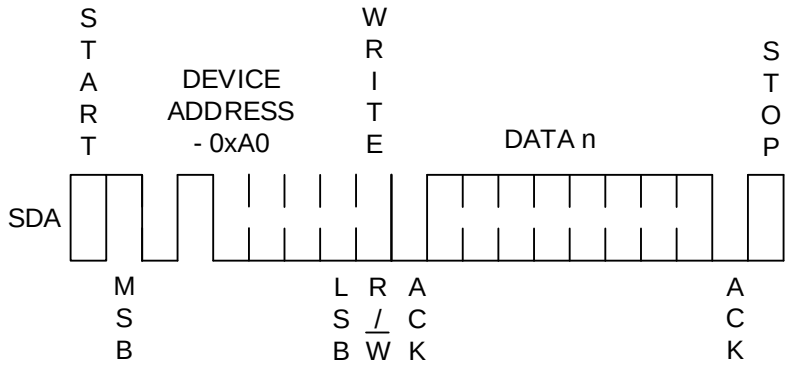


Figure 5: Write Sequence

To write data to the module, the master initiates the process by sending a start signal, followed by the I2C slave address and a write bit. At the 9th clock cycle (when the clock is high), the slave acknowledges the master's request. Then the master transmits the data onto the bus. After every 8 bits of data transfer, the slave responds with an ACK to indicate successful reception. If it generates an ACK bit, it signifies that it has received the data and is ready to accept another byte. On the other hand, if it generates a NAK bit, it indicates it cannot accept any further data, and the master should terminate the transfer by sending a STOP signal (P).

3 I2C Read/Write Operation

The following chapter provides a detailed explanation on how the master reads and writes messages via I2C bus. See *document [1] protocol specification* for detailed information on the messages.

Note that the I2C receive buffer of the slave has a maximum capacity of 4096 bytes, while the transmit buffer can accommodate up to 9216 bytes. The interval between two input messages cannot be less than 10 ms because the slave needs 10 ms to process the input data.

3.1. Master Data Reading Flow

The master reads data as follows:

Step 1 Master reads data length from the slave transmit buffer.

a) Master sends a configuration read command to the slave.
- 7-bit slave address (0x50) and write bit.
- Two data words: 0xAA510008 and 0x00000004 (little-endian transmission).

b) Master reads the data length from the slave transmit buffer.
- 7-bit slave address (0x54) and read bit.
- Master retrieves the data length from the slave transmit buffer.

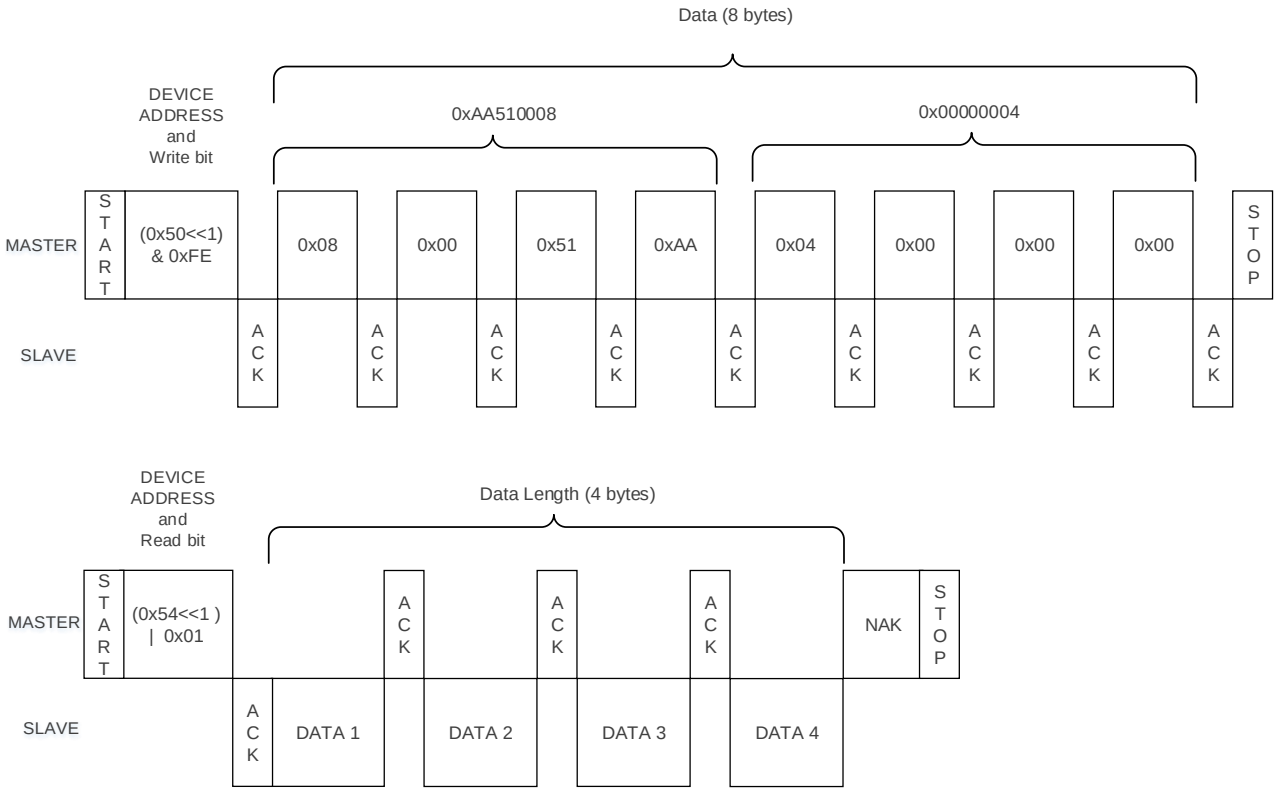


Figure 6: Master Data Reading Flow Step 1

Step 2 Master reads the **data_read_len** [1] bytes of data.

a) Master sends configuration read command to the slave.
- 7-bit slave address (0x50) and write bit.
- Two data words: 0xAA512000 and **data_read_len** [1] (little-endian transmission).

b) Master reads the **data_read_len** [1] bytes of data.
- 7-bit slave address (0x54) and read bit.
- Master receives the **data_read_len** [1] bytes of data.

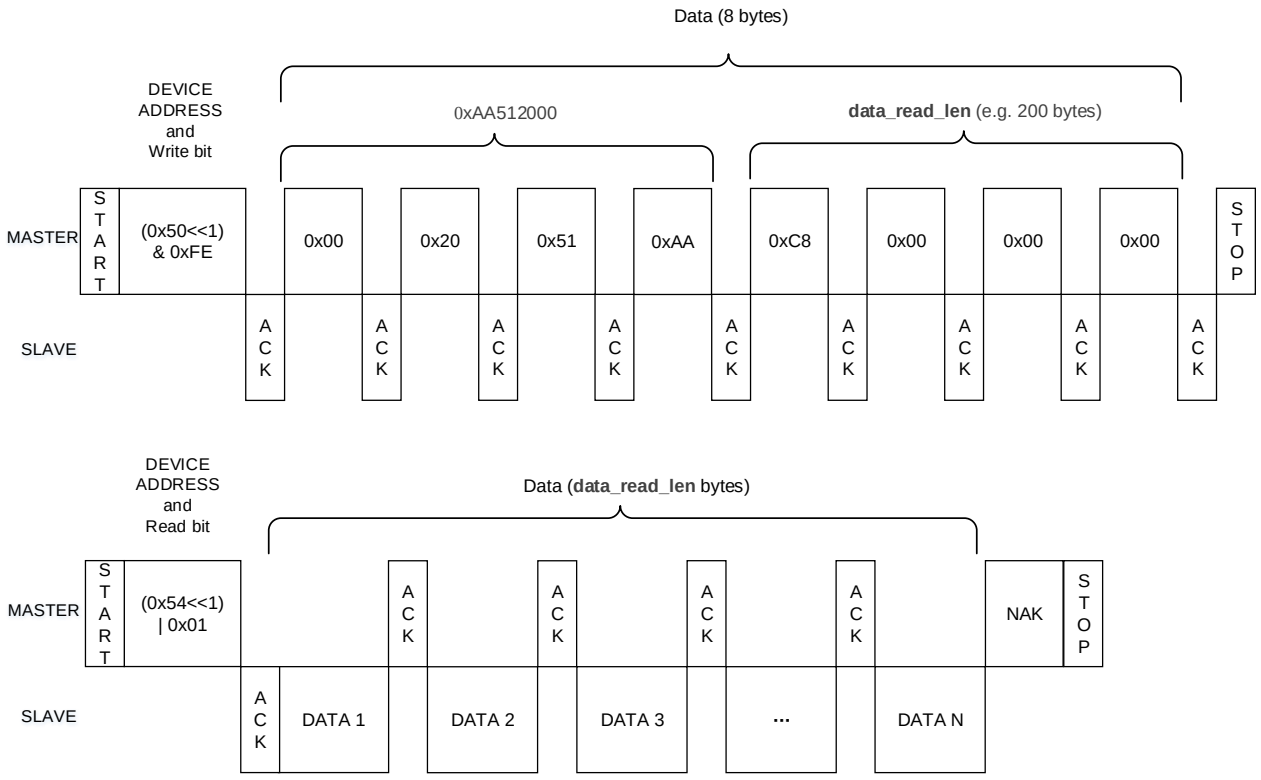


Figure 7: Master Data Reading Flow Step 2

NOTE

1. [1)] Unsigned int **data_read_len** represents the data length that the master intends to read. The value of **data_read_len** should be less than or equal to the length read in ***Step 1***. If the data length read from the module I2C transmit buffer exceeds **data_read_len**, you can repeat ***Step 2*** until all data have been read. However, the total length of the data read by the master cannot exceed the data length read in ***Step 1***.
2. If the master fails to read all data generated by the module within every epoch in time, causing the I2C transmit buffer of the module to become full, the I2C will enter sleep state. Writing any data (ensuring a complete data writing flow is executed, i.e., ***Step 1*** and ***Step 2***; see *Chapter 3.2 Master Data Writing Flow* for details) to the module via I2C port can wake up the I2C transmitter.
3. 1 word = 4 bytes.
4. The module transmits data in little-endian format.

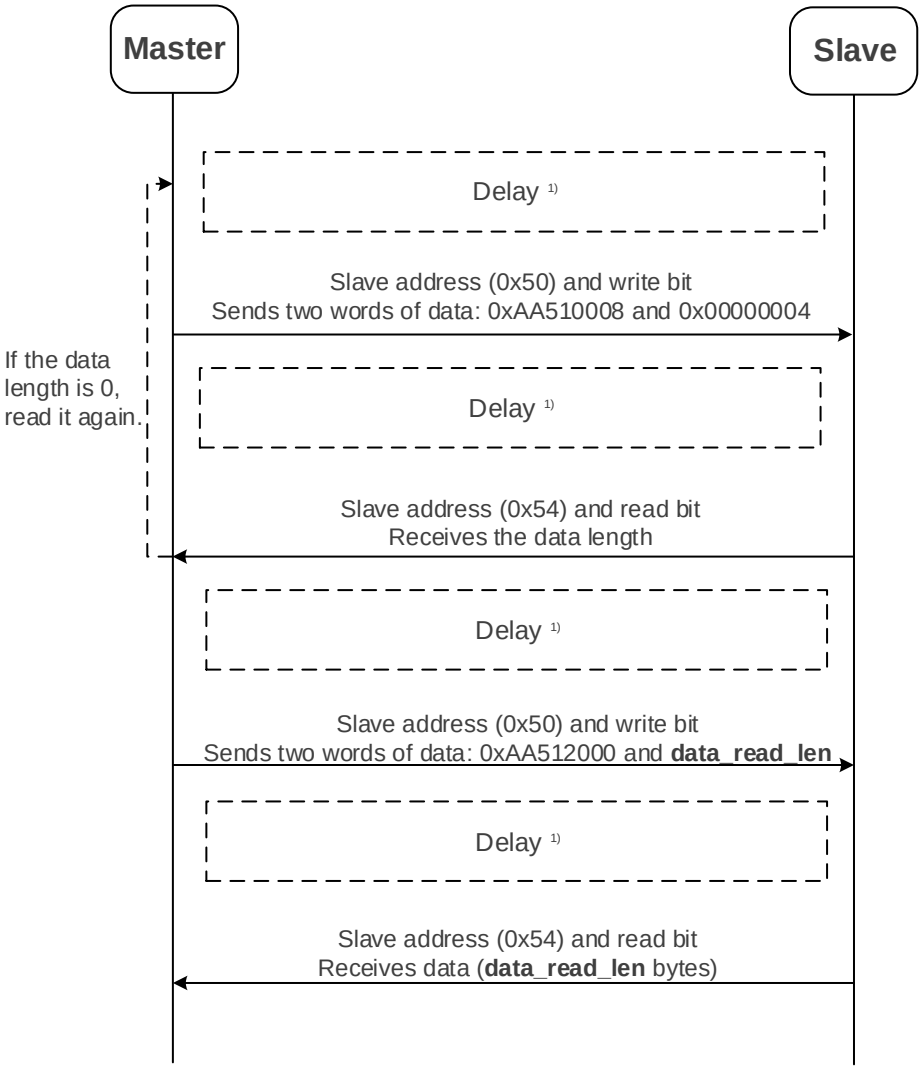


Figure 8: Data Reading Flow of Master

NOTE

1) The time of delay is about 10 ms.

3.2. Master Data Writing Flow

The master writes data as follows:

Step 1 Master reads the available free length in the slave receive buffer.

a) Master sends a configuration read command to the slave.
- 7-bit slave address (0x50) and write bit.
- Two data words: 0xAA510004 and 0x00000004 (little-endian transmission).

b) Master reads the free length from the slave receive buffer.
- 7-bit slave address (0x54) and read bit.
- Master receives the free length from the slave receive buffer.

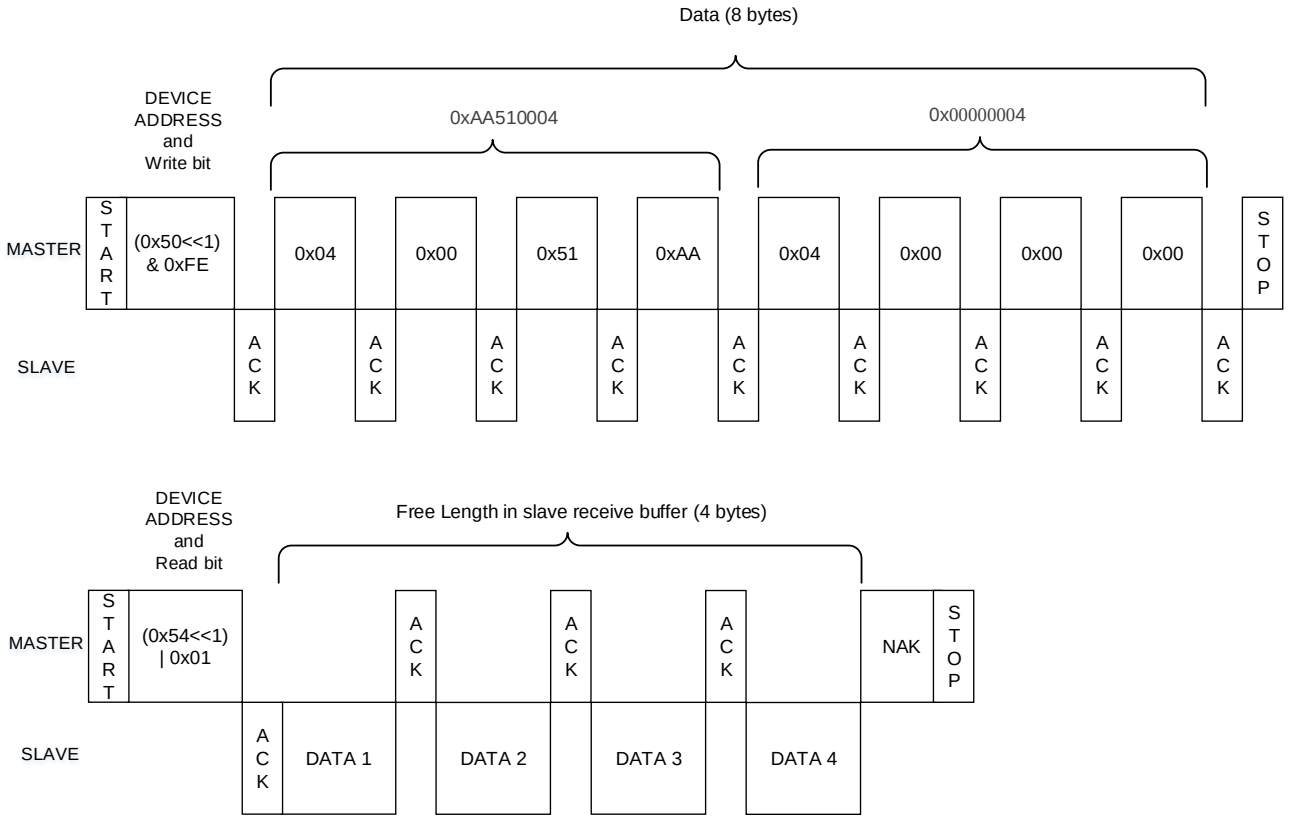


Figure 9: Master Data Writing Flow Step 1

Step 2 Master writes **data_written_len** [1] bytes of data.

a) Master sends configuration write command to the slave.
- 7-bit slave address (0x50) and write bit.
- Two data words: 0xAA531000 and **data_written_len** [1] (little-endian transmission).

b) Master writes **data_written_len** [1] bytes of data.
- 7-bit slave address (0x58) and write bit.
- Master writes **data_written_len** [1] bytes of data.

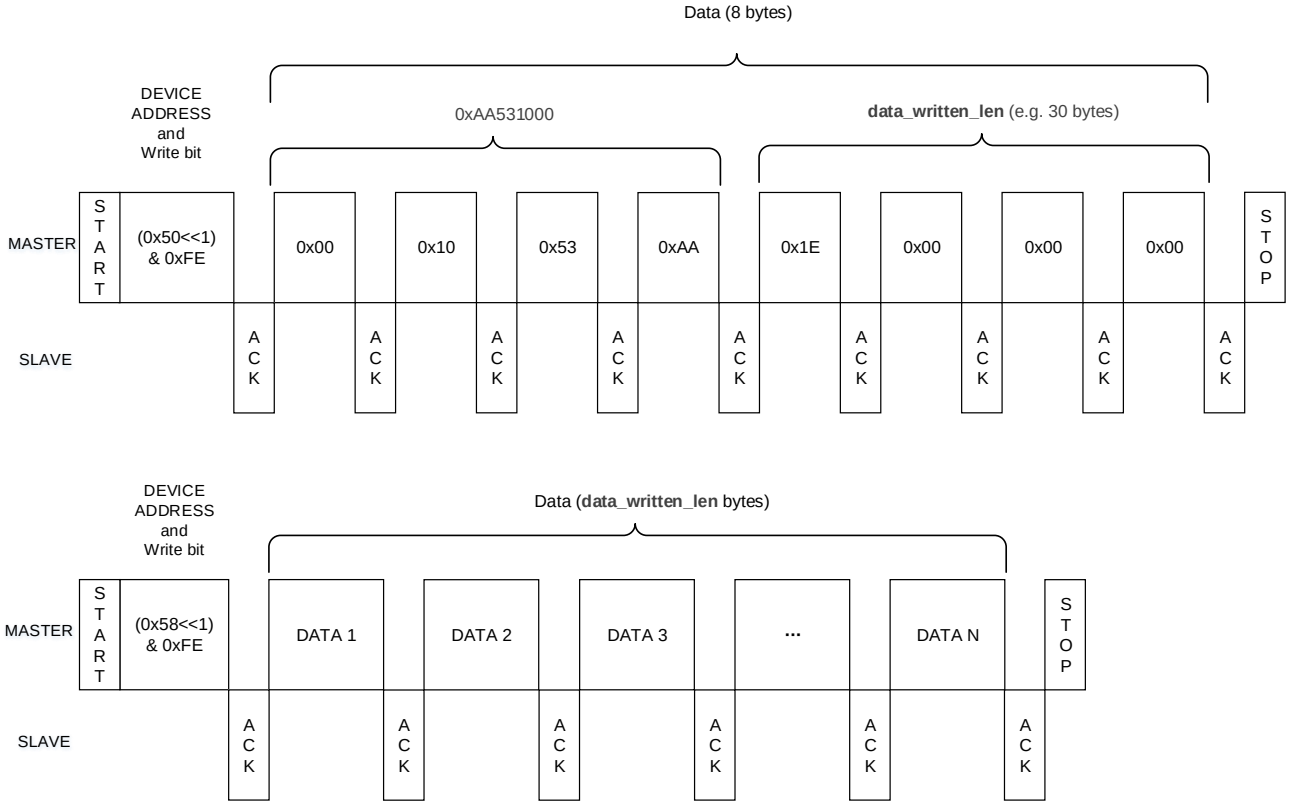


Figure 10: Master Data Writing Flow Step 2

NOTE

1. [1] Unsigned int **data_written_len** represents the data length that the master intends to write.
2. The free length in the slave receive buffer indicates the maximum length of data the master can write. If the data to be sent exceeds the free length in the slave receive buffer, you need to divide the data into multiple parts and send them separately.
3. 1 word = 4 bytes.
4. The module transmits data in little-endian format.

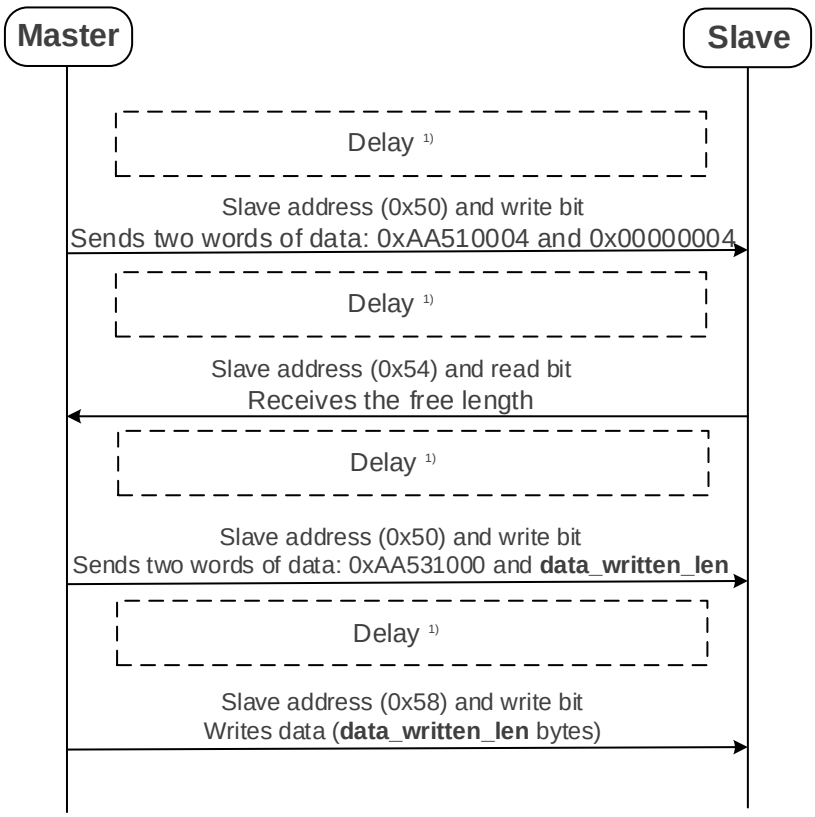


Figure 11: Data Writing Flow of Master

NOTE

1) The time of delay is about 10 ms.

4 I2C Multi-Slave Operation

If there is more than one slave device on the I2C bus, the master must consider the following:

1. To ensure proper communication on the I2C bus, it is important to follow a specific sequence when reading/writing data from/to the GNSS module and other slave devices. If the master intends to interact with other slave devices before the GNSS module, you should write GNSS module's 7-bit slave address (0x50) first, then go on the steps outlined in *Step 1-a)* to *Step 2-b)* in *Chapter 3.1 Master Data Reading Flow* (or *Step 1-a)*to *Step 2-b)* in *Chapter 3.2 Master Data Writing Flow*). Otherwise, a **NAK** can be generated on the I2C bus.

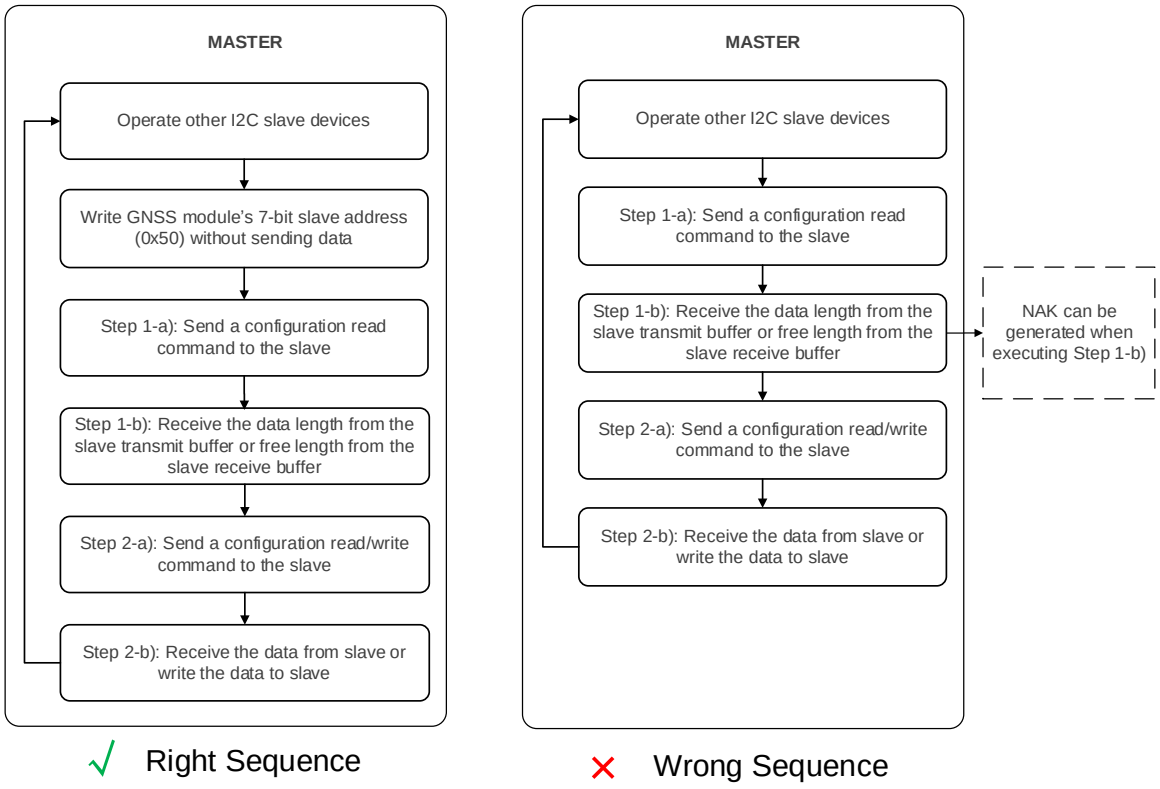
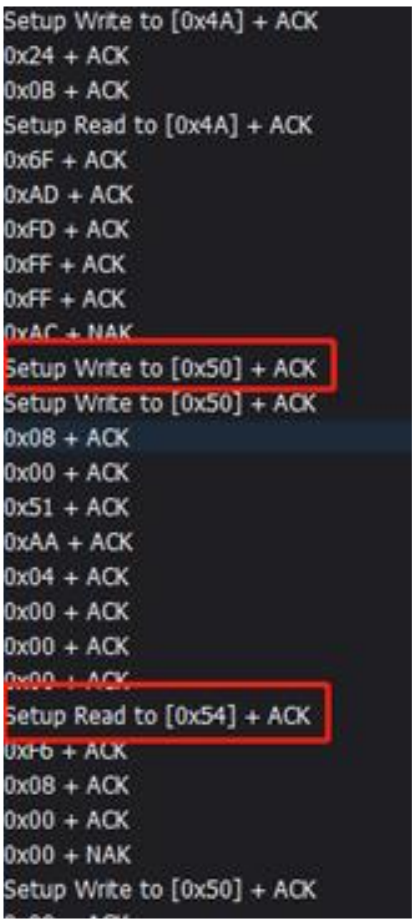


Figure 12: Multi-Slave Operation Sequence on I2C Bus

Correct Sequence Example



Incorrect Sequence Example

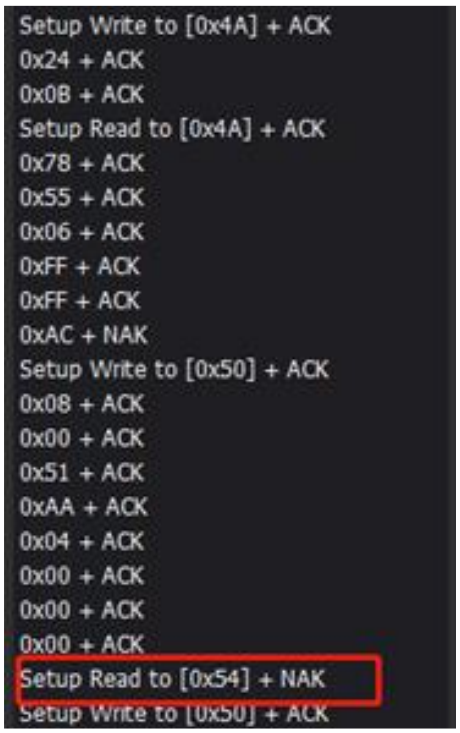


Figure 13: Multi-Slave Operation on I2C Bus Example 1

2. Master cannot operate other I2C slave devices simultaneously when performing I2C data transfer with the GNSS module.

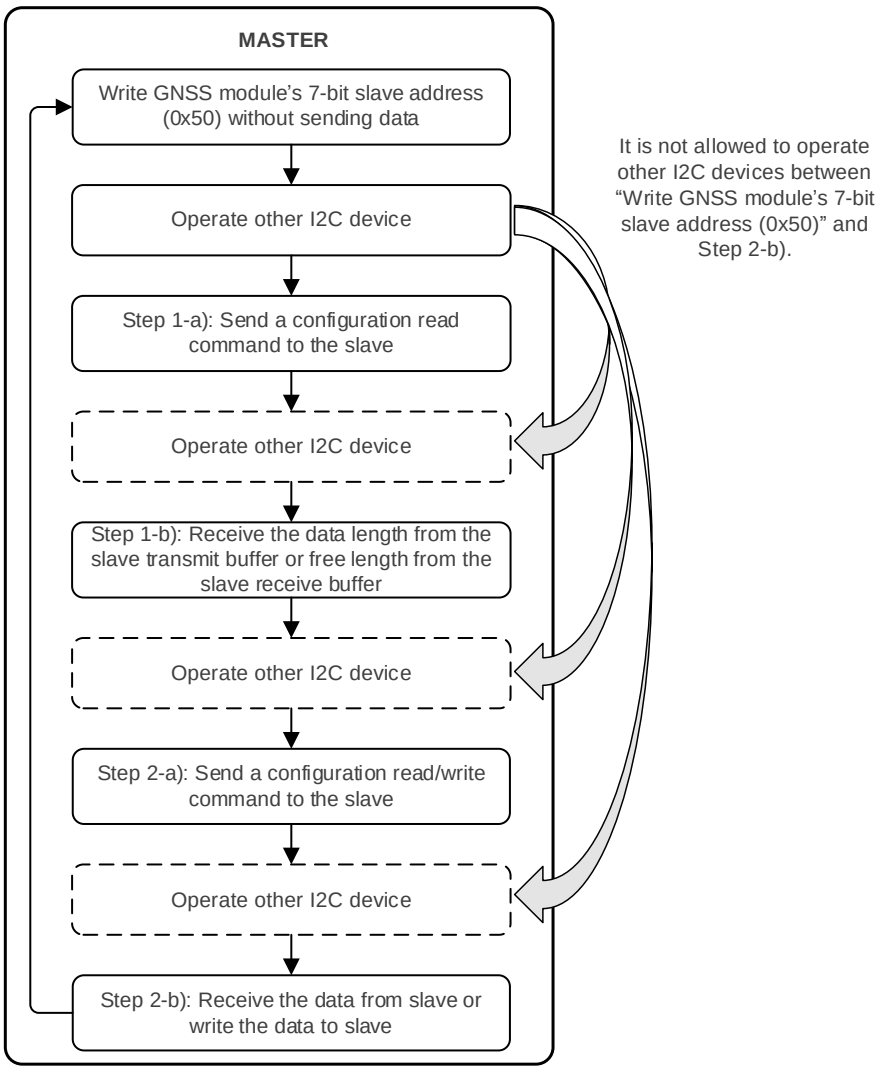
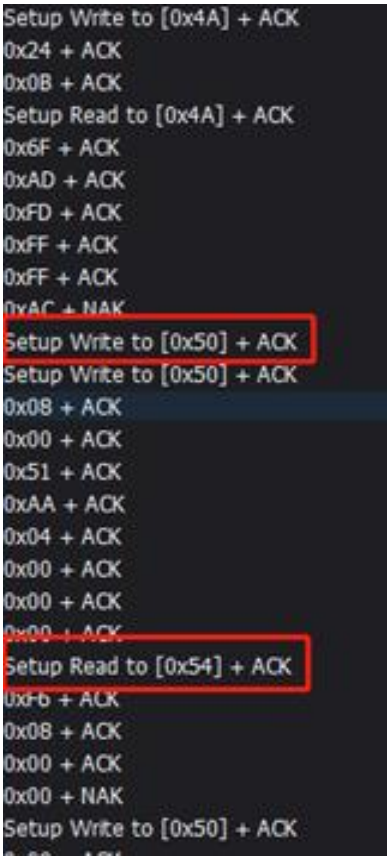


Figure 14: Incorrect Multi-Slave Operation Sequence on I2C Bus

Correct Sequence Example



Incorrect Sequence Example

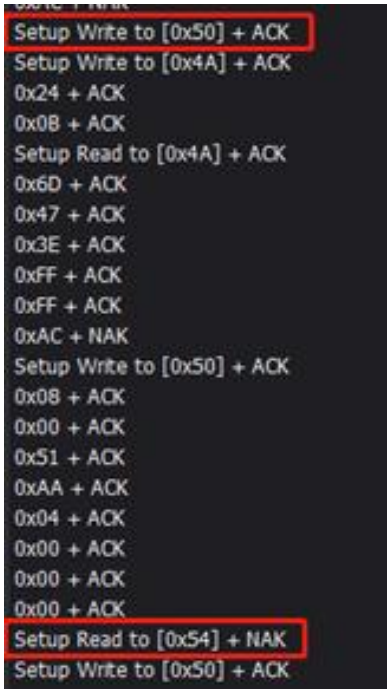


Figure 15: Multi-Slave Operation on I2C Bus Example 2

5 I2C Read/Write Example

An example of reading NMEA data and writing data through the I2C interface on the LC76G (AB) module is as follows.

//Data Reading Example.
//Step 1-a: Master sends a configuration read command to the slave.
A0 08 00 51 AA 04 00 00 00
//Step 1-b: Master receives the data length from the slave transmit buffer. If the data length is 0, return to Step 1-a.
A9 FC 09 00 00
//Step 2-a: Master sends a configuration read command to the slave. The read length is configured to 0x00000400.
A0 00 20 51 AA 00 04 00 00
//Step 2-b: Master reads NMEA data, and the length is 0x00000400.
A9 24 50 51 54 4D 56 45 52 2C 4D 4F 44 55 4C 45 5F……
//Step 2-a: Master sends a configuration read command to the slave. The read length is configured to 0x00000400.
A0 00 20 51 AA 00 04 00 00
//Step 2-b: Master reads NMEA data, and the length is 0x00000400.
A9 43 0D 0A 24 47 4E 56 54 47 2C 2C 54 2C 2C 4D 2C……
//Step 2-a: Master sends a configuration read command to the slave. The read length is configured to 0x000001FC.
A0 00 20 51 AA FC 01 00 00
//Step 2-b: Master reads NMEA data, and the length is 0x000001FC.
A9 42 47 53 56 2C 31 2C 31 2C 30 30 2C 35 2A 37 32……

//Data Writing Example: 15-byte message **$PQTMVERNO*58\r\n**.
//Step 1-a: Master sends a configuration read command to the slave.
A0 04 00 51 AA 04 00 00 00
//Step 1-b: Master receives the free length from the slave receive buffer. As the read free length (0x00000003) is less than the length of message to be sent (0x0000000F), split the message before sending.
A9 03 00 00 00
//Step 2-a: Master sends a configuration write command to the slave to configure the write length as 0x00000003.
A0 00 10 53 AA 03 00 00 00
//Step 2-b: Master writes data to the slave, starting from "$PQ".
B0 24 50 51

//Step 1-a: Master sends a read command to the slave.

A0 04 00 51 AA 04 00 00 00

//Step 1-b: Master receives the free length from the slave receive buffer.

A9 FD F0 00 00

//Step 2-a: Master sends a configuration write command to the slave to configure the write length as 0x0000000C.

A0 00 10 53 AA 0C 00 00 00

//Step 2-b: Master writes data to the slave, i.e., writing "TMVERNO*58\r\n" into it.

B0 54 4D 56 45 52 4E 4F 2A 35 38 0D 0A

6 Sample Code for I2C Reading/Writing Sequence

The sample code for reading data from and writing data to the I2C buffer is shown below.

```
#define QUECTEL_I2C_SLAVE_CR_CMD 0xaa51
#define QUECTEL_I2C_SLAVE_CW_CMD 0xaa53

#define QUECTEL_I2C_SLAVE_CMD_LEN              8
#define QUECTEL_I2C_SLAVE_TX_LEN_REG_OFFSET    0x08
#define QUECTEL_I2C_SLAVE_TX_BUF_REG_OFFSET    0x2000

#define QUECTEL_I2C_SLAVE_RX_LEN_REG_OFFSET    0x04
#define QUECTEL_I2C_SLAVE_RX_BUF_REG_OFFSET    0x1000

#define QUECTEL_I2C_SLAVE_ADDRESS_CR_OR_CW     0x50
#define QUECTEL_I2C_SLAVE_ADDRESS_R            0x54
#define QUECTEL_I2C_SLAVE_ADDRESS_W            0x58

#define MAX_ERROR_NUMBER                      20
#define MAX_I2C_BUFFER                        1024

typedef enum
{
    I2C_ACK = 0,
    I2C_NACK = 1
}I2c_Resp_FlagStatus;
typedef enum
{
    DEV_REP_SUCCESS = 0,
    DEV_REP_ERROR = 1
}Dev_Resp_FlagStatus;

int Recovery_I2c(void)
{
    uint8_t dummy_data = 0;
    if(I2c_Master_Transmit(QUECTEL_I2C_SLAVE_ADDRESS_CR_OR_CW << 1, &dummy_data, 1)
== I2C_ACK)
```

```
        {
            return 1;
        }
        if(I2c_Master_Transmit(QUECTEL_I2C_SLAVE_ADDRESS_R << 1, &dummy_data, 1) ==
I2C_ACK)
        {
            return 2;
        }
        if(I2c_Master_Transmit(QUECTEL_I2C_SLAVE_ADDRESS_W << 1, &dummy_data, 1) ==
I2C_ACK)
        {
            return 3;
        }
        return 0;
    }


    I2c_Resp_FlagStatus I2c_Master_Receive(uint8_t addr, uint8_t *Data, uint16_t Length)
    {
        mcu_i2c_start();
        mcu_i2c_send_byte(addr|0x01);
        if(mcu_i2c_wait_ack() != I2C_ACK)
        {
            mcu_i2c_stop();
            return I2C_NACK;
        }
        for(int i = 0; i < Length; i++)
        {
            *(Data + i) = mcu_i2c_receive_byte();
            if(i != (Length - 1))
            {
                mcu_i2c_ack();
            }
        }
        mcu_i2c_no_ack();
        mcu_i2c_stop();
        return I2C_ACK;
    }

    I2c_Resp_FlagStatus I2c_Master_Transmit(uint8_t addr, uint8_t *Data, uint8_t Length)
    {
        uint8_t i = 0;
        uint8_t flag=0;
        mcu_i2c_start();
```

```
        mcu_i2c_send_byte(addr);
        if(mcu_i2c_wait_ack() == I2C_NACK)
        {
            mcu_i2c_stop();
            return I2C_NACK;
        }
        for(i = 0; i < Length; i++)
        {
            mcu_i2c_send_byte(*(Data+i));
            if(mcu_i2c_wait_ack() == I2C_NACK)
            {
                mcu_i2c_stop();
                return I2C_NACK;
            }
        }
        mcu_i2c_stop();
        return I2C_ACK;
    }

    Dev_Resp_FlagStatus Quectel_Dev_Receive(uint8_t* pData, uint16_t maxLength,
uint16_t* pRecLength)
    {
        uint32_t request_cmd[2];
        uint16_t* pRxLength = pRecLength;
        uint8_t* pBuff = pData;
        uint8_t i2c_master_receive_error_counter = 0;
        I2c_Resp_FlagStatus status;

        //step 1_a
        request_cmd[0] = (uint32_t)((uint32_t)(QUECTEL_I2C_SLAVE_CR_CMD << 16) |
QUECTEL_I2C_SLAVE_TX_LEN_REG_OFFSET);
        request_cmd[1] = 4;

        i2c_master_receive_error_counter = 0;
        while(1)
        {
            delay_ms(10);
            status = I2c_Master_Transmit(QUECTEL_I2C_SLAVE_ADDRESS_CR_OR_CW << 1,
(uint8_t *)request_cmd, QUECTEL_I2C_SLAVE_CMD_LEN);
            if(status == I2C_ACK)
            {
                break;
            }
```

```
            i2c_master_receive_error_counter++;
            if(i2c_master_receive_error_counter > MAX_ERROR_NUMBER)
            {
                Recovery_i2c();
                return DEV_REP_ERROR;
            }
        }

        //step 1_b
        i2c_master_receive_error_counter = 0;
        while(1)
        {
            delay_ms(10);
            status = I2c_Master_Receive(QUECTEL_I2C_SLAVE_ADDRESS_R << 1,
(uint8_t*)pRxLength, 4);
            if(status == I2C_ACK)
            {
                break;
            }

            i2c_master_receive_error_counter++;
            if(i2c_master_receive_error_counter > MAX_ERROR_NUMBER)
            {
                Recovery_i2c();
                return DEV_REP_ERROR;
            }
        }

        if(*pRxLength == 0)
        {
            return DEV_REP_SUCCESS;
        }
        if(*pRxLength > MAX_I2C_BUFFER)
        {
            *pRxLength = MAX_I2C_BUFFER;
        }

        //step 2_a
        request_cmd[0] = (uint32_t)(QUECTEL_I2C_SLAVE_CR_CMD << 16) |
QUECTEL_I2C_SLAVE_TX_BUF_REG_OFFSET;
        request_cmd[1] = *pRxLength;
        i2c_master_receive_error_counter = 0;
        while(1)
        {
```

```
            delay_ms(10);
            status = I2c_Master_Transmit(QUECTEL_I2C_SLAVE_ADDRESS_CR_OR_CW << 1,
(uint8_t *)request_cmd, QUECTEL_I2C_SLAVE_CMD_LEN);
            if(status == I2C_ACK)
            {
                break;
            }

            i2c_master_receive_error_counter++;
            if(i2c_master_receive_error_counter > MAX_ERROR_NUMBER)
            {
                Recovery_i2c();
                *pRxLength = 0;
                return DEV_REP_ERROR;
            }
        }

        //step 2_b
        i2c_master_receive_error_counter = 0;
        while(1)
        {
            delay_ms(10);
            status = I2c_Master_Receive(QUECTEL_I2C_SLAVE_ADDRESS_R << 1, pBuff,
*pRxLength);
            if(status == I2C_ACK)
            {
                return DEV_REP_SUCCESS;
            }

            i2c_master_receive_error_counter++;
            if(i2c_master_receive_error_counter > MAX_ERROR_NUMBER)
            {
                Recovery_i2c();
                *pRxLength = 0;
                return DEV_REP_ERROR;
            }
        }

        return DEV_REP_SUCCESS;
    }

    Dev_Resp_FlagStatus Quectel_Dev_Transmit(uint8_t *pData, uint16_t dataLength)
    {
        uint32_t request_cmd[2];
```

```
        uint16_t rxBuffLength = 0;

        uint8_t  i2c_master_receive_error_counter = 0;
        I2c_Resp_FlagStatus status;

        //step 1_a
        request_cmd[0] = (uint32_t)((QUECTEL_I2C_SLAVE_CR_CMD << 16) |
QUECTEL_I2C_SLAVE_RX_LEN_REG_OFFSET);
        request_cmd[1] = 4;

        i2c_master_receive_error_counter = 0;
        while(1)
        {
            delay_ms(10);
            status = I2c_Master_Transmit(QUECTEL_I2C_SLAVE_ADDRESS_CR_OR_CW << 1,
(uint8_t *)request_cmd, QUECTEL_I2C_SLAVE_CMD_LEN);
            if(status == I2C_ACK)
            {
                break;
            }

            i2c_master_receive_error_counter++;
            if(i2c_master_receive_error_counter > MAX_ERROR_NUMBER)
            {
                Recovery_i2c();
                return DEV_REP_ERROR;
            }
        }

        //step 1_b
        i2c_master_receive_error_counter = 0;
        while(1)
        {
            delay_ms(10);
            status = I2c_Master_Receive(QUECTEL_I2C_SLAVE_ADDRESS_R << 1,
(uint8_t*)&rxBuffLength, 4);
            if(status == I2C_ACK)
            {
                break;
            }

            i2c_master_receive_error_counter++;
            if(i2c_master_receive_error_counter > MAX_ERROR_NUMBER)
            {
```

```
                Recovery_i2c();
                return DEV_REP_ERROR;
            }
        }

        if(dataLength > rxBuffLength)
        {
            return DEV_REP_ERROR;
        }

        //step 2_a
        request_cmd[0] = (uint32_t)(QUECTEL_I2C_SLAVE_CW_CMD << 16) |
QUECTEL_I2C_SLAVE_RX_BUF_REG_OFFSET;
        request_cmd[1] = dataLength;
        i2c_master_receive_error_counter = 0;
        while(1)
        {
            delay_ms(10);
            status = I2c_Master_Transmit(QUECTEL_I2C_SLAVE_ADDRESS_CR_OR_CW << 1,
(uint8_t *)request_cmd, QUECTEL_I2C_SLAVE_CMD_LEN);
            if(status == I2C_ACK)
            {
                break;
            }

            i2c_master_receive_error_counter++;
            if(i2c_master_receive_error_counter > MAX_ERROR_NUMBER)
            {
                Recovery_i2c();
                return DEV_REP_ERROR;
            }
        }

        //step 2_b
        i2c_master_receive_error_counter = 0;
        while(1)
        {
            delay_ms(10);
            status = I2c_Master_Transmit(QUECTEL_I2C_SLAVE_ADDRESS_W << 1, pData,
dataLength);
            if(status == I2C_ACK)
            {
                return DEV_REP_SUCCESS;
            }
```

```
            i2c_master_receive_error_counter++;
            if(i2c_master_receive_error_counter > MAX_ERROR_NUMBER)
            {
                Recovery_i2c();
                return DEV_REP_ERROR;
            }
        }
        return DEV_REP_SUCCESS;
    }
```

7 Appendix References

Table 1: Related Document

Document Name
[1] Quectel_LC26G&LC26G-T&LC76G&LC86G_Series_GNSS_Protocol_Specification

Table 2: Terms and Abbreviations

Abbreviation	Description
ACK	Acknowledge
GNSS	Global Navigation Satellite System
I2C	Inter-Integrated Circuit
LSB	Least Significant Bit
MCU	Microcontroller Unit
MSB	Most Significant Bit
NAK/NACK	Negative Acknowledgement
NMEA	NMEA (National Marine Electronics Association) 0183 Interface Standard
SCL	Serial Clock
SDA	Serial Data