LC26G (AB)&LC76G Series
I2C Application Note

GNSS Module Series

Version: 1.0

Date: 2022-09-16

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)&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

Contents

Table Index

Figure Index

1 Introduction

This document outlines the I2C function and its usage on Quectel LC26G (AB) and LC76G series modules. The modules always operate as slave device 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:

- Works in slave mode
- Standard mode (100 kbps) and fast mode (400 kbps)
- 7-bit address
- Supports sending and receiving variable length messages
- I2C pins: I2C_SDA and I2C_SCL

In addition, this document also provides a detailed introduction as well as a flow chart and sample code to illustrate how the master reads NMEA messages and transmits/receives **$PQTM** or **$PAIR** messages via the I2C bus.

2 NMEA Data Reading

This chapter provides a detailed explanation on how the master reads NMEA data packets via the I2C bus. The length of NMEA data is not fixed, and needs to be read from the module to determine. The capacity of the slave's I2C buffer is 4096 bytes, which means that the master can read one I2C data packet of a maximum size of 4096 bytes at a time. To read a complete NMEA packet of one second, the master needs to read several I2C data packets and then extract valid NMEA data from them.

2.1. NMEA Data Reading Flow of Master

The master reads NMEA data as follows:

Step 1 The master reads the NMEA data length in the slave transmit buffer.

The master sends a configuration read command to the slave.

- 7-bit slave address is 0x50.
- The sent data is two words: 0xAA510008 and 0x00000004 (little-endian transmission).

The master receives the NMEA data length from the slave transmit buffer.

- 7-bit slave address is 0x54.
- The master receives the NMEA data length from the slave transmit buffer.

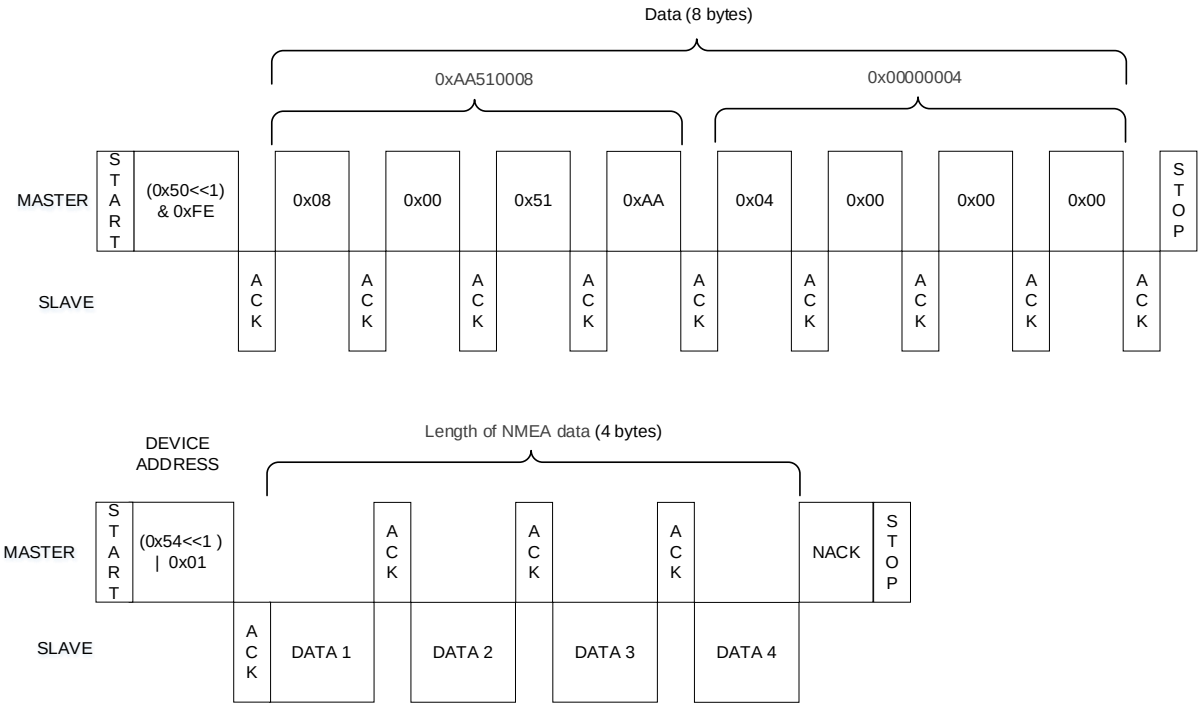

Figure 1: NMEA Data Reading Flow of the Master in Step 1

Step 2 The master reads the **data_read_len** [1)] bytes of the NMEA data.

The master sends configuration read command to the slave.

- 7-bit slave address is 0x50.
- The sent data is two words: 0xAA512000 and **data_read_len** (little-endian transmission).

The master receives the **data_read_len** bytes of the NMEA data.

- 7-bit slave address is 0x54.
- The master receives the **data_read_len** bytes of the NMEA data.

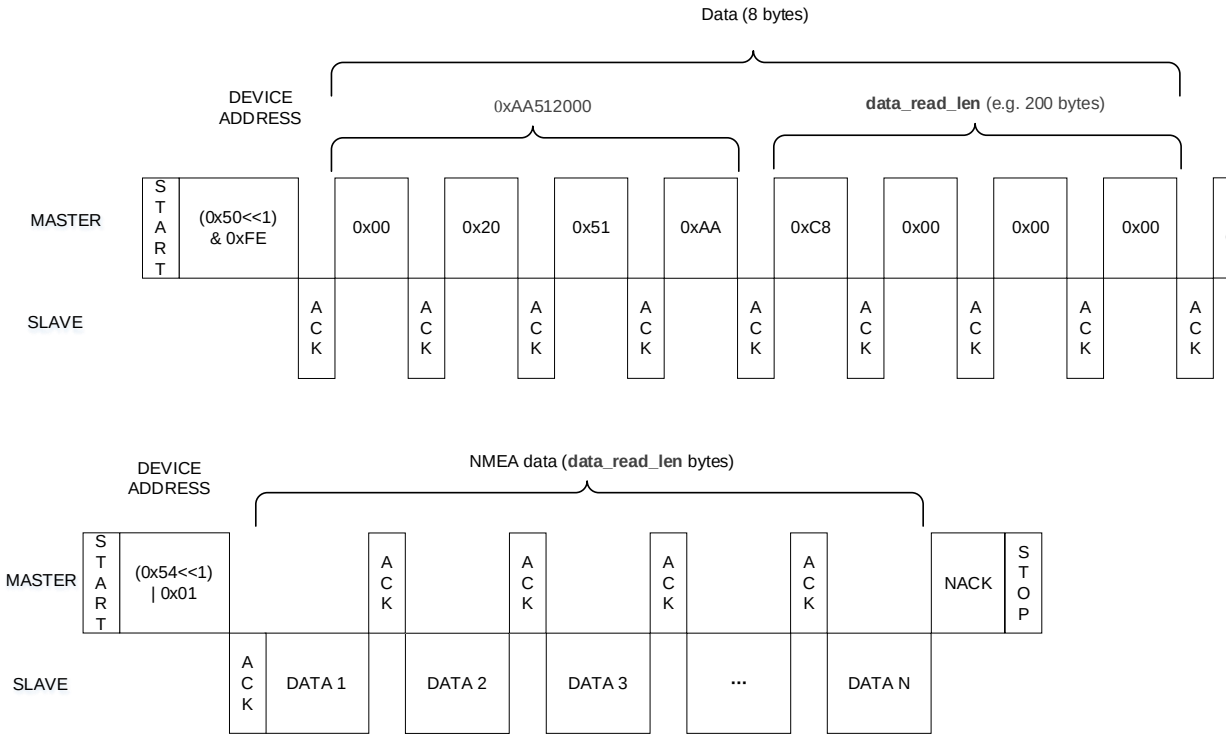

Figure 2: NMEA Data Reading Flow of Master in Step 2

NOTE

1. [1)] Unsigned int **data_read_len** defines the length of NMEA data that master needs to receive. The **data_read_len** should be less than or equal to the length read in the ***Step 1***.
2. If user does not read the data on time, the transmitter buffer will be full and the I2C transmitter will enter sleep state. Sending any command 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.

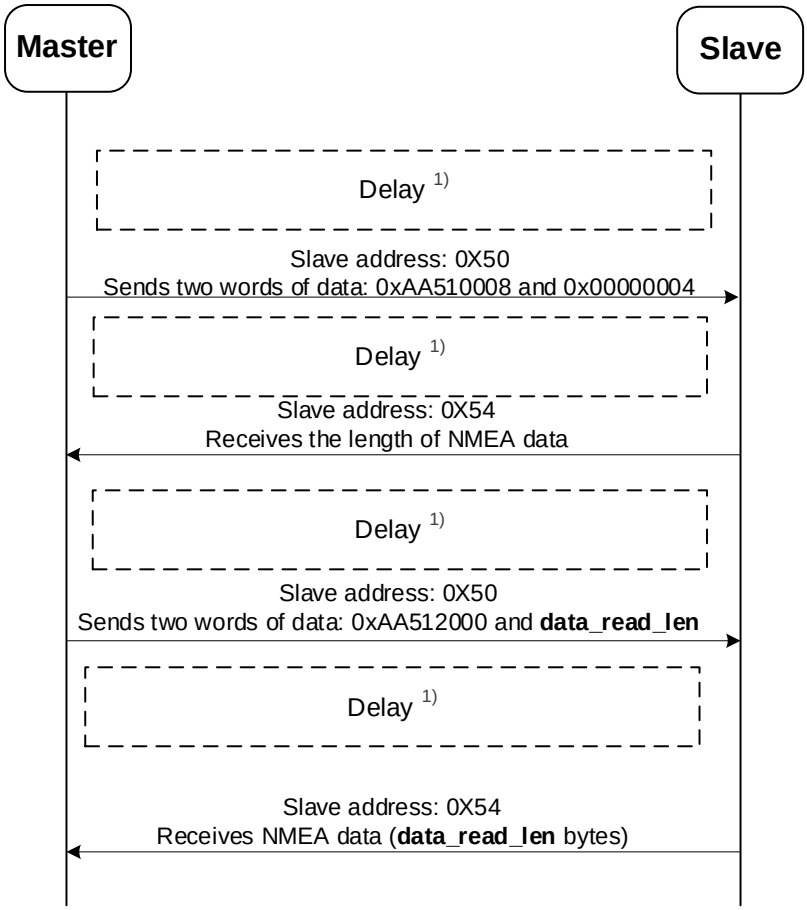

Figure 3: NMEA Data Reading Flow of Master

NOTE

1) The time of delay is about 10 ms.

3 NEMA Data Writing

The master can send messages to the slave via I2C bus. See ***document [1]*** for detailed information on the messages.

As the maximum capacity of the slave's I2C receive buffer is 4096 bytes, each message input by the master should be 4096 bytes at most. 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. NEMA Data Writing Flow of Master

The master writes NEMA data as following:

Step 1 The master reads the free length in the slave receive buffer.

a) The master sends a configuration read command to the slave.
- 7-bit slave address is 0x50.
- The sent data is two words: 0xAA510004 and 0x00000004 (little-endian transmission).

b) The master receives the free length from the slave receive buffer.
- 7-bit slave address is 0x54.
- The master receives the free length in the slave receive buffer.

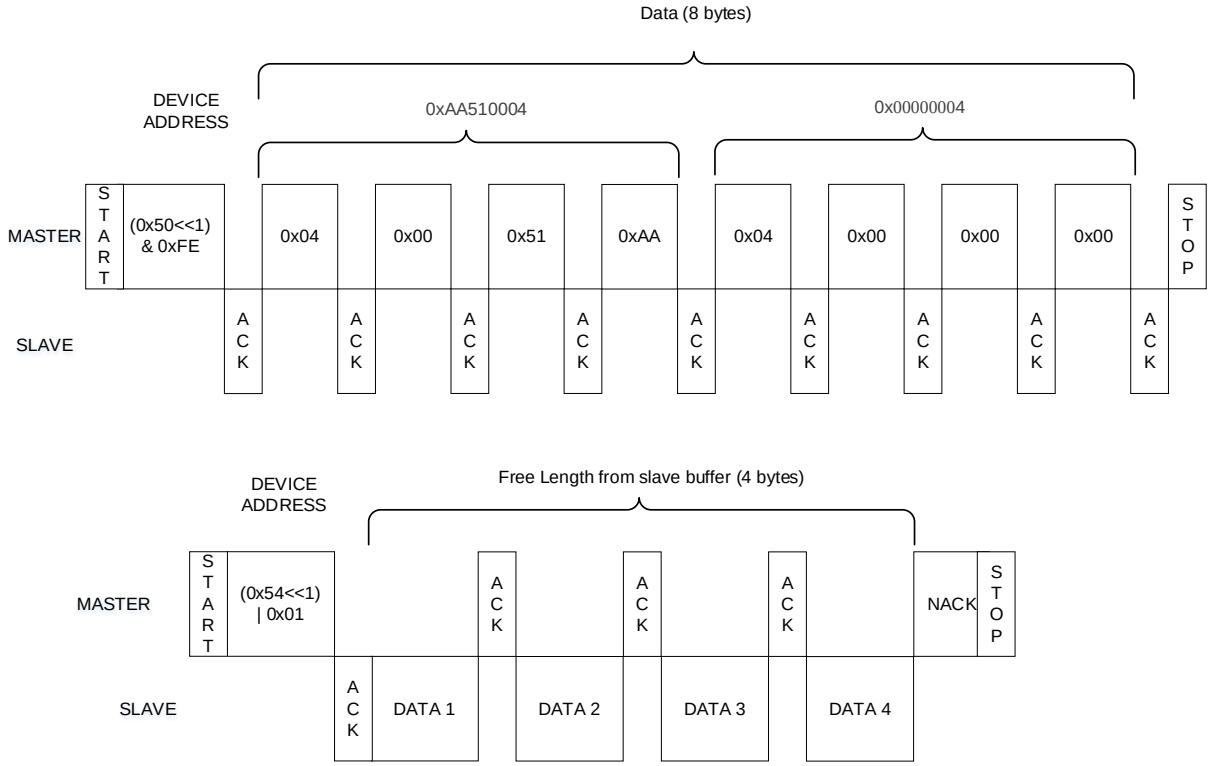

Figure 4: Data Writing Flow of Master in Step 1

Step 2 The master writes **data_written_len**[1)] bytes of data.

a) The master sends configuration write command to the slave.

- 7-bit slave address is 0x50.
- The sent data is two words: 0xAA531000 and **data_written_len** (little-endian transmission).

b) The master writes **data_written_len** bytes of data.

- 7-bit slave address is 0x58.
- The master writes **data_written_len** bytes of data.

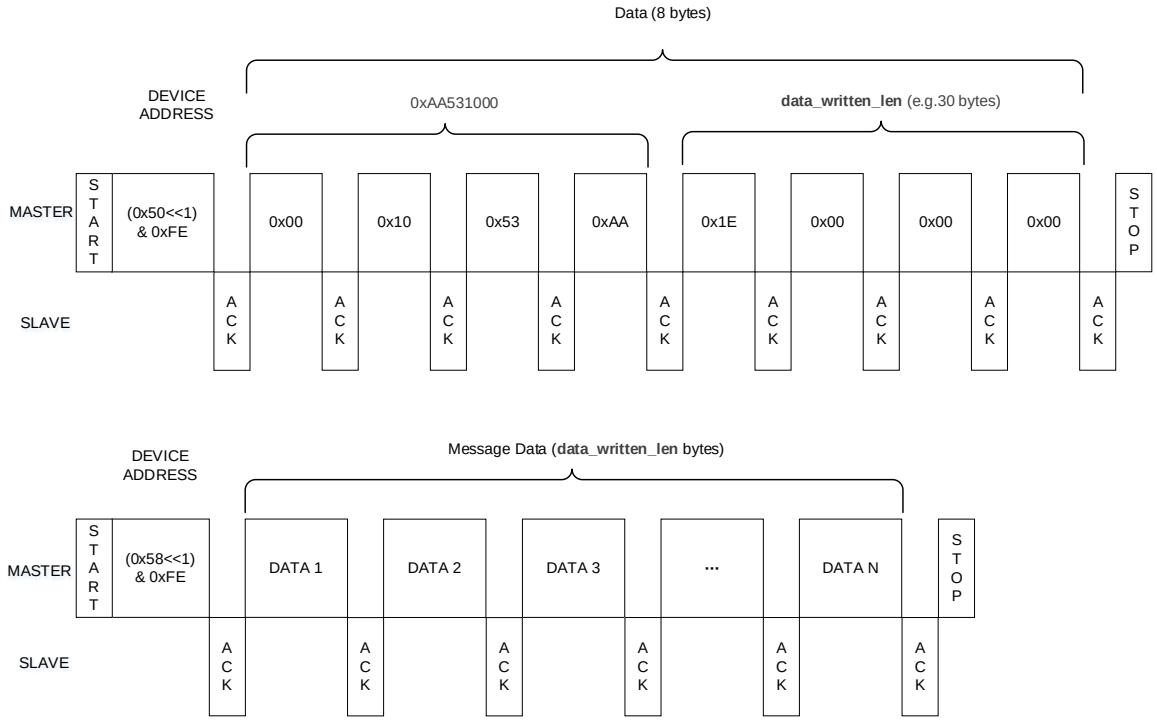

Figure 5: Data Writing Flow of Master in Step 2

NOTE

1. [1)] Unsigned int **data_written_len** defines the length of NEMA data that master needs to write.
2. The free length in the salve receiver buffer is the maximum length master can write. Sent data that exceeds the free length in the slave receive buffer can not be written in by the module.
3. 1 word = 4 bytes.
4. The module transmits data in little-endian format.

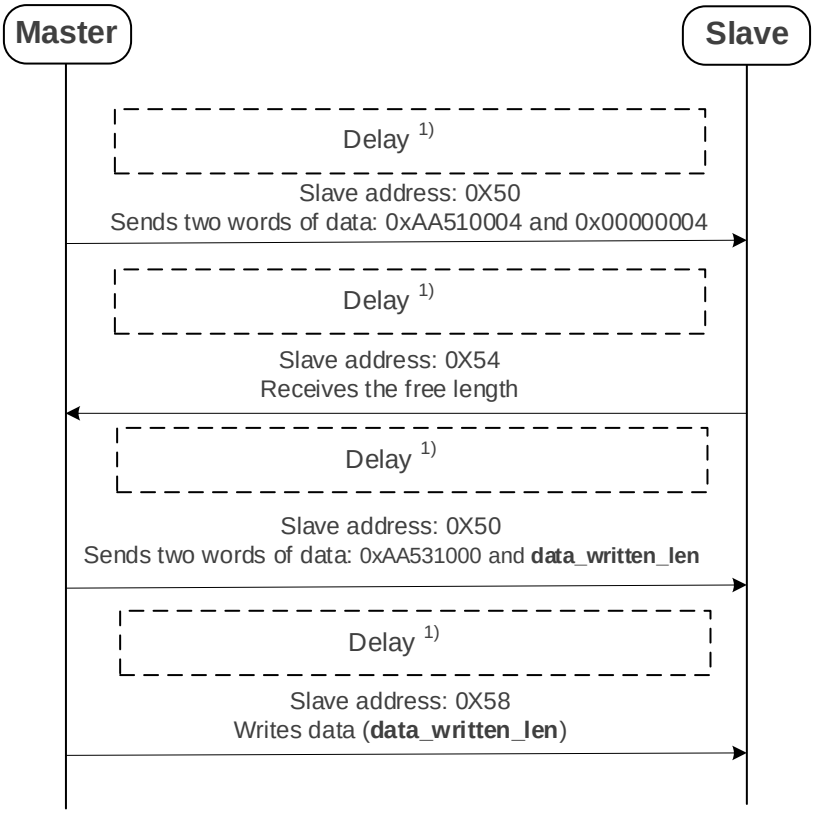

Figure 6: Data Writing Flow of Master

NOTE

1) The time of delay is about 10 ms.

4 Sample Code for I2C Reading/Writing Sequence

4.1. Sample Code

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;

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)
        {
            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)
        {
            return DEV_REP_ERROR;
        }
     }

    if(*pRxLength == 0)
    {
        return DEV_REP_ERROR;
    }
    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)
        {
            *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)
        {
            *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)
        {
            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)
        {
            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)
        {
            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)
        {
            return DEV_REP_ERROR;
        }
     }
     return DEV_REP_SUCCESS;
}
```

5 Appendix References

Table 1: Related Document

Document Name
[1] Quectel_LC26G&LC76G&LC86G_Series_GNSS_Protocol_Specification

Table 2: Terms and Abbreviations

Abbreviation	Description
ACK	Acknowledge
GNSS	Global Navigation Satellite System
I2C	Inter-Integrated Circuit
MCU	Microcontroller Unit
NMEA	National Marine Electronics Association
SCL	Serial Clock
SDA	Serial Data