QUECTEL

LC76G (AB)
SPI Application Note

GNSS Module Series

Version: 1.0

Date: 2024-01-23

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.

Copyright © Quectel Wireless Solutions Co., Ltd. 2024. All rights reserved.

About the Document

Document Information	
Title	**LC76G (AB) SPI Application Note**
Subtitle	GNSS Module Series
Document Type	Application Note
Document Status	Released

Revision History

Version	Date	Description
-	2023-06-30	Creation of the document
1.0	2024-01-23	First official release

Contents

About the Document ... 3
Contents ... 4
Table Index ... 5
Figure Index ... 6

1 Introduction ... 7

2 Module SPI Characteristics ... 8
- 2.1. Host Connects with Module ... 8
- 2.2. Module SPI Reading/Writing Timing ... 8
- 2.3. Module SPI Timing Diagram ... 9

3 Module SPI Specification ... 10
- 3.1. Module SPI Registers ... 10
- 3.2. Module SPI Commands and Usage ... 10
  - 3.2.1. Module SPI Commands ... 10
  - 3.2.2. Usage of Module SPI Commands ... 11
    - 3.2.2.1. PWON_CMD and PWOFF_CMD ... 11
    - 3.2.2.2. CFG_RD_CMD and CFG_WR_CMD ... 12
    - 3.2.2.3. Other Commands ... 12
- 3.3. Module SPI Status Flags ... 13

4 Data Reading and Data Writing ... 14
- 4.1. Data Reading ... 15
- 4.2. Data Writing ... 17

5 Example for SPI Reading/Writing Sequence ... 19

6 Sample Code for SPI Reading/Writing Sequence ... 21

7 Appendix References ... 32

Table Index

Table 1: Module SPI Characteristics 9
Table 2: Module SPI Registers 10
Table 3: Module SPI Commands 10
Table 4: Module SPI Status Flags 13
Table 5: Related Documents 32
Table 6: Terms and Abbreviations 32

Figure Index

Figure 1: Connection Between Host and Module 8
Figure 2: Module SPI Reading/Writing Timing 9
Figure 3: Module SPI Timing Diagram 9
Figure 4: Host Sends PWON_CMD or PWOFF_CMD 11
Figure 5: Host Sends CFG_RD_CMD or CFG_WR_CMD 12
Figure 6: Host Sends Other Module SPI Commands 12
Figure 7: Module SPI Read/Write Flow 14
Figure 8: Host Data Reading Step 1-a) 15
Figure 9: Host Data Reading Step 1-b) 15
Figure 10: Host Data Reading Step 1-c) 15
Figure 11: Host Data Reading Step 1-d) 15
Figure 12: Host Data Reading Step 2-a) 16
Figure 13: Host Data Reading Step 2-b) 16
Figure 14: Host Data Reading Step 2-c) 16
Figure 15: Host Data Reading Step 2-d) 16
Figure 16: Host Data Writing Step 1-a) 17
Figure 17: Host Data Writing Step 1-b) 17
Figure 18: Host Data Writing Step 1-c) 17
Figure 19: Host Data Writing Step 1-d) 17
Figure 20: Host Data Writing Step 2-a) 18
Figure 21: Host Data Writing Step 2-b) 18
Figure 22: Host Data Writing Step 2-c) 18
Figure 23: Host Data Writing Step 2-d) 18

1 Introduction

This document outlines the SPI function and its usage on the LC76G (AB) module. The module always operates as a slave device when communicating to the host (client-side MCU). The host can read/write any message via the SPI bus. When using the SPI to read/write messages, you must pull up the D_SEL pin before powering on. See *document [1] hardware design* for details. For details about the messages that the host can read and write through the SPI function, such as NMEA and RTCM messages, see *document [2] protocol specification*.

The features of the module's SPI:

- Slave mode
- Clock up to 48 MHz
- LSB first
- SPI pins: SPI_CS, SPI_CLK, SPI_MOSI, SPI_MISO
- Mode 0 (CPOL = 0, CPHA = 0)

NOTE

Currently, only LC76G (AB) with LC76GABNR12A02S or higher versions supports the SPI function.

2 Module SPI Characteristics

2.1. Host Connects with Module

The SPI communicates to the host using 4 wires: SPI_CS, SPI_CLK, SPI_MOSI and SPI_MISO.

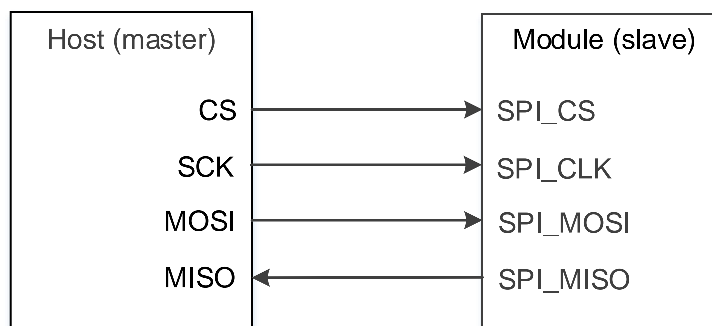

Figure 1: Connection Between Host and Module

SPI_CS is the chip select pin controlled by the SPI master. Pull it low to start a transmission and pull it high to indicate the end of the transmission.
SPI_CLK is the serial clock pin controlled by the master.
SPI_MOSI is the master out slave in pin.
SPI_MISO is the master in slave out pin.

2.2. Module SPI Reading/Writing Timing

The module's SPI works in Mode 0 (CPOL = 0, CPHA = 0). In Mode 0 SPI_CLK is at a low level when SPI is idle, and data is captured starting from the first edge of SPI_CLK. SPI_MOSI and SPI_MISO data changes occur while SPI_CLK is low, and the data is captured on the rising edge of SPI_CLK. The data does not change while SPI_CLK is high.

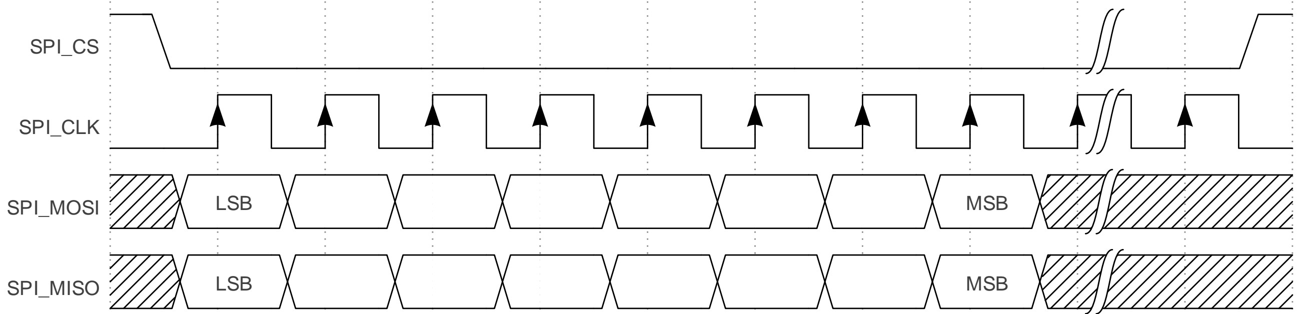

Figure 2: Module SPI Reading/Writing Timing

2.3. Module SPI Timing Diagram

The operating characteristics of module SPI at the typical temperature (25 °C) are show in *Figure 3: Module SPI Timing Diagram* and *Table 1: Module SPI Characteristics*.

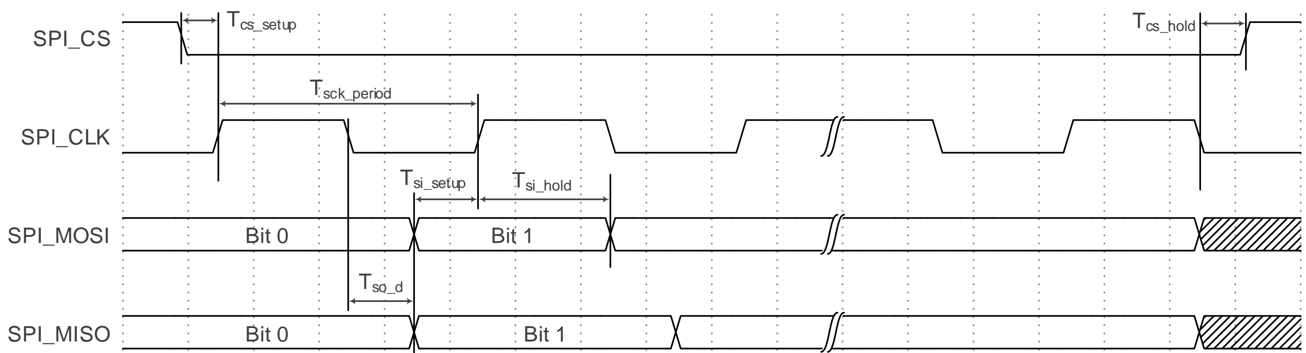

Figure 3: Module SPI Timing Diagram

Table 1: Module SPI Characteristics

Signal	Parameter	Description	Min.	Max.	Unit
SPI_CS	T_{cs_setup}	Chip select setup time	9.6	-	ns
	T_{cs_hold}	Chip select hold time	28.8	-	ns
SPI_CLK	T_{sck_period}	Serial clock period	19.2	-	ns
SPI_MOSI	T_{si_setup}	SPI slave data input valid time	4.8	-	ns
	T_{si_hold}	SPI slave data input hold time	4.8	-	ns
SPI_MISO	T_{so_d}	SPI slave data output delay time	0.0	15.0	ns

3 Module SPI Specification

This chapter explains the registers, commands and status flags of the module SPI.

3.1. Module SPI Registers

The following table shows the register addresses of the module SPI.

Table 2: Module SPI Registers

Register Name	Register Address	Description
SLAVE_TX_LEN_REG	0x00000008	Module SPI transmit buffer available data length register.
SLAVE_TX_BUF_REG	0x00002000	Module SPI transmit buffer register.
SLAVE_RX_LEN_REG	0x00000004	Module SPI receive buffer free length register.
SLAVE_RX_BUF_REG	0x00001000	Module SPI receive buffer register.

3.2. Module SPI Commands and Usage

3.2.1. Module SPI Commands

The following table lists the module SPI commands.

Table 3: Module SPI Commands

Command Name	Command Value	Description
CFG_RD_CMD	0x0A	Configures register address and data length of the data to be read.
RD_CMD	0x81	Reads data.
CFG_WR_CMD	0x0C	Configures register address and data length of the data to be

Command Name	Command Value	Description
		written.
WR_CMD	0x0E	Writes data.
RS_CMD	0x06	Reads module SPI status.
WS_CMD	0x08	Erases module SPI read/write error flag.
PWON_CMD	0x04	Powers on module SPI.
PWOFF_CMD	0x02	Powers off module SPI.

NOTE

PWON_CMD must be sent every reset or restart.

3.2.2. Usage of Module SPI Commands

The module SPI commands are categorized into three groups according to their usage:

- **PWON_CMD** and **PWOFF_CMD**, which are sent without being followed by a data field;
- **CFG_RD_CMD** and **CFG_WR_CMD**, which are sent with a data field;
- Other commands, i.e., **RD_CMD**, **WR_CMD**, **RS_CMD** and **WS_CMD**, which are sent with data.

3.2.2.1. PWON_CMD and PWOFF_CMD

Host does not have a data field when sending **PWON_CMD** or **PWOFF_CMD**.

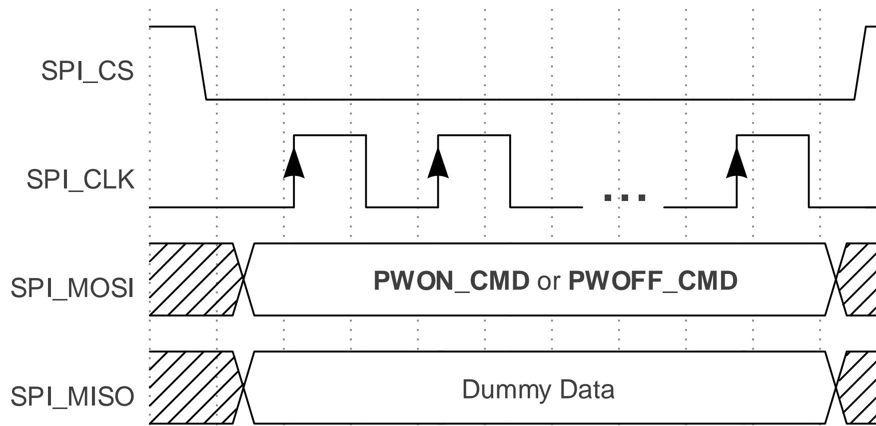

Figure 4: Host Sends PWON_CMD or PWOFF_CMD

3.2.2.2. CFG_RD_CMD and CFG_WR_CMD

Host sends the **CFG_RD_CMD** or **CFG_WR_CMD** command followed by a data field. The data field consists of a 4-byte module SPI register address and a 4-byte data length of the data to be read or written.

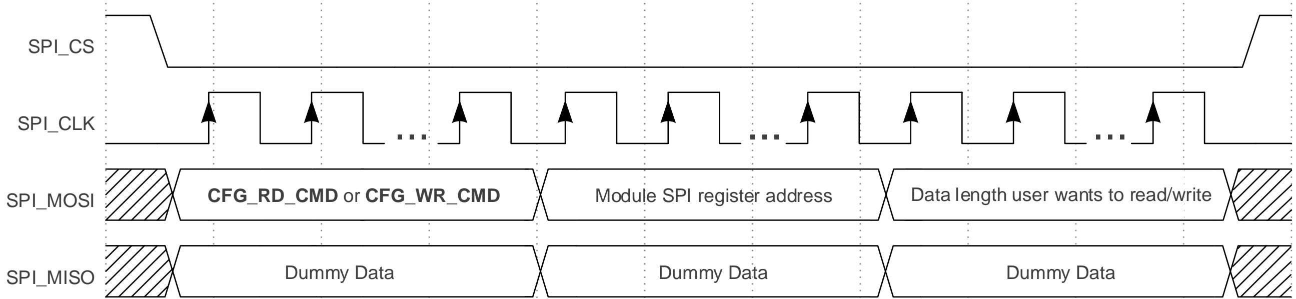

Figure 5: Host Sends CFG_RD_CMD or CFG_WR_CMD

NOTE

The data length must be decremented by 1 when sending, because the length is calculated from the SPI register address.

3.2.2.3. Other Commands

The host sends other module SPI commands followed by data.

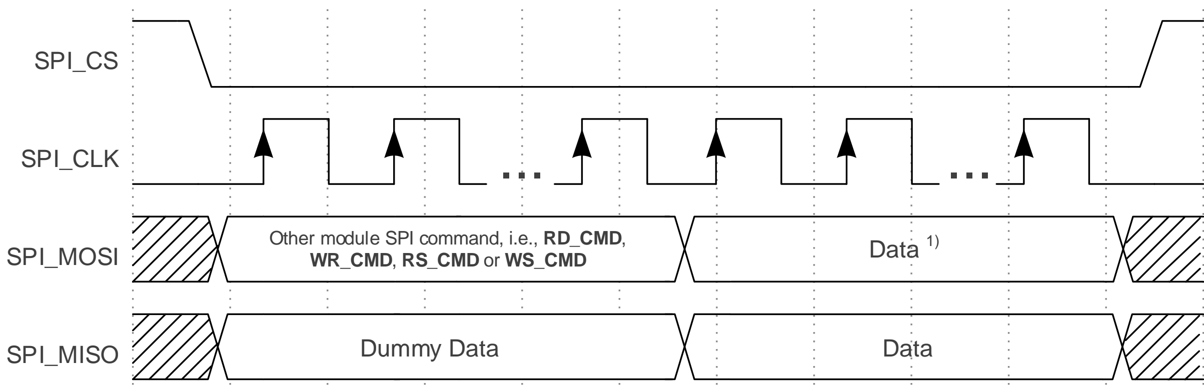

Figure 6: Host Sends Other Module SPI Commands

NOTE

[1] For **WR_CMD** or **WS_CMD** sent via SPI_MOSI, this data is what the user wants to send. For **RD_CMD** or **RS_CMD**, it is dummy data.

3.3. Module SPI Status Flags

The following table lists the status flags of the module SPI.

Table 4: Module SPI Status Flags

Flag Name	Flag Value	Description
STA_SLV_ON	0x01	Module SPI power on flag.
STA_TXRX_FIFO_RDY	0x04	Module SPI transmit/receive buffer with FIFO ready flag.
STA_RD_ERR	0x08	Module SPI read error flag.
STA_WR_ERR	0x10	Module SPI write error flag.
STA_RDWR_FINISH	0x20	Module SPI read/write finish flag.

4 Data Reading and Data Writing

The following chapter provides a detailed explanation on how the host reads/write data via the SPI bus and describes the registers, commands and status used in the reading and writing processes.

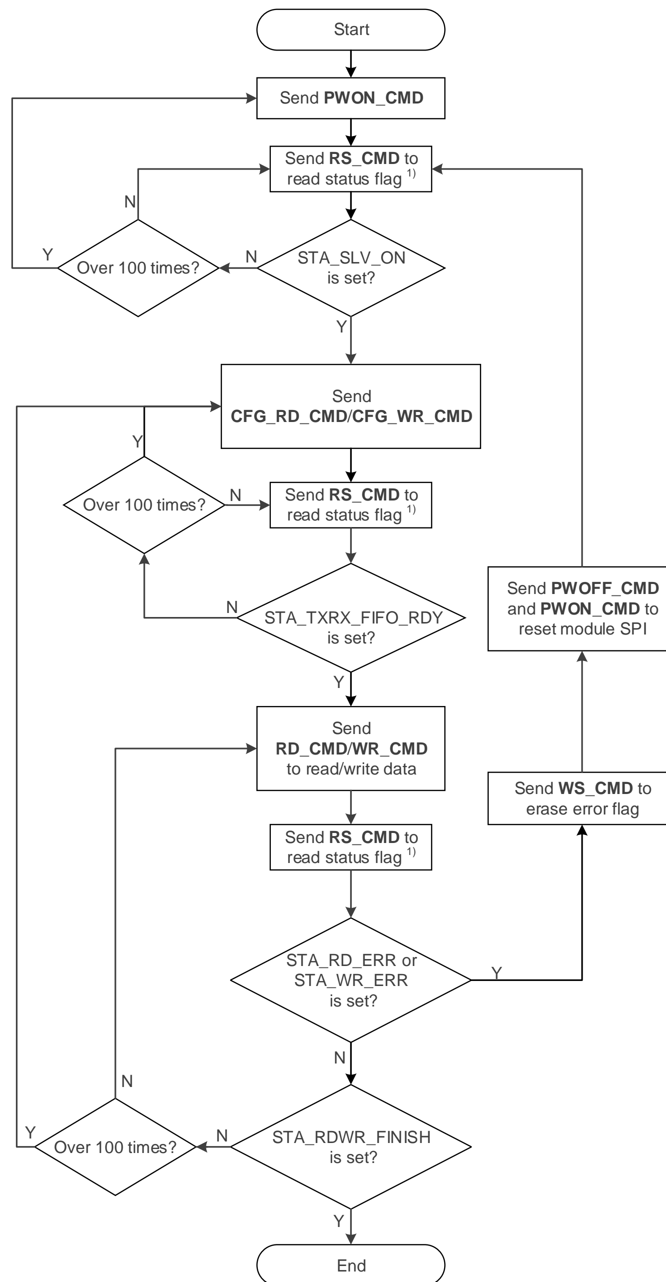

Figure 7: Module SPI Read/Write Flow

NOTE

[1)] Each time **RS_CMD** is sent, there must be a 1 ms delay.

4.1. Data Reading

The host reads data as follows.

Step 1 Read available data length from the module SPI transmit buffer.

a) Host sends **CFG_RD_CMD**, SLAVE_TX_LEN_REG and 0x00000003, 9 bytes in total.

Host	**CFG_RD_CMD**	SLAVE_TX_LEN_REG	0x00000003
Module	Dummy Data		

Figure 8: Host Data Reading Step 1-a)

b) Host sends **RS_CMD** and checks STA_TXRX_FIFO_RDY.

Host	**RS_CMD**	1 byte Dummy Data
Module	1 byte Dummy Data	Data (including STA_TXRX_FIFO_RDY)

Figure 9: Host Data Reading Step 1-b)

c) Host sends **RD_CMD** and receives 4 bytes of available data length from the module SPI transmit buffer [1)].

Host	**RD_CMD**	4 bytes Dummy Data
Module	1 byte Dummy Data	Available data length from module SPI transmit buffer [1)]

Figure 10: Host Data Reading Step 1-c)

d) Host sends **RS_CMD** and checks STA_RDWR_FINISH.

Host	**RS_CMD**	1 byte Dummy Data
Module	1 byte Dummy Data	Data (including STA_RDWR_FINISH)

Figure 11: Host Data Reading Step 1-d)

Step 2 Read module SPI transmit buffer data.

a) Host sends **CFG_RD_CMD**, SLAVE_TX_BUF_REG and data_read_len [2] -1, 9 bytes in total.

Host	**CFG_RD_CMD**	SLAVE_TX_BUF_REG	data_read_len [2] - 1
Module	Dummy Data		

Figure 12: Host Data Reading Step 2-a)

b) Host sends **RS_CMD** and checks STA_TXRX_FIFO_RDY.

Host	**RS_CMD**	1 byte Dummy Data
Module	1 byte Dummy Data	Data (including STA_TXRX_FIFO_RDY)

Figure 13: Host Data Reading Step 2-b)

c) Host sends **RD_CMD** and receives data simultaneously.

Host	**RD_CMD**	data_read_len [2] bytes Dummy Data
Module	1 byte Dummy Data	Data (read_data_len [2] bytes)

Figure 14: Host Data Reading Step 2-c)

d) Host sends **RS_CMD** and checks STA_RDWR_FINISH.

Host	**RS_CMD**	1 byte Dummy Data
Module	1 byte Dummy Data	Data (including STA_RDWR_FINISH)

Figure 15: Host Data Reading Step 2-d)

NOTE

1. [1] If the available data length from the module SPI transmit buffer in *Step 1* is 0, it means that the module currently has no available data, and the host must return to *Step 1-a)* to re-execute the reading process.
2. [2] Unsigned integer data_read_len represents the data length that the host intends to receive. If the data length from module SPI transmit buffer read in *Step 1* exceeds data_read_len, you can repeat *Step 2* to read the data until all data have been read, but the total read length cannot exceed the data length from module SPI transmit buffer. Otherwise the status of STA_TXRX_FIFO_RDY cannot be read in *Step 2*.
3. If the host fails to read the data on time, the transmit buffer becomes full, causing the SPI transmitter to enter sleep state. Sending any data to the module via SPI can wake up the SPI

transmitter, see *Chapter 4.2 Data Writing* for details.
4. The module transmits data in little-endian format.

4.2. Data Writing

The host can send messages to the module via the SPI bus. See *document [2] protocol specification* for detailed information on the messages. The host writes data as follows.

Step 1 Read free length in the module SPI receive buffer.

a) Host sends **CFG_RD_CMD**, SLAVE_RX_LEN_REG and 0×00000003, 9 bytes in total.

Host	**CFG_RD_CMD**	SLAVE_RX_LEN_REG	0×00000003
Module	Dummy Data		

Figure 16: Host Data Writing Step 1-a)

b) Host sends **RS_CMD** and checks STA_TXRX_FIFO_RDY.

Host	**RS_CMD**	1 byte Dummy Data
Module	1 byte Dummy Data	Data (including STA_TXRX_FIFO_RDY)

Figure 17: Host Data Writing Step 1-b)

c) Host sends **RD_CMD** and receives 4 bytes of free length from the module SPI receive buffer [1)].

Host	**RD_CMD**	4 bytes Dummy Data
Module	1 byte Dummy Data	Free length from module SPI receive buffer [1)]

Figure 18: Host Data Writing Step 1-c)

d) Host sends **RS_CMD** and checks STA_RDWR_FINISH.

Host	**RS_CMD**	1 byte Dummy Data
Module	1 byte Dummy Data	Data (including STA_RDWR_FINISH)

Figure 19: Host Data Writing Step 1-d)

Step 2 Write data to the module SPI receive buffer.

a) Host sends **CFG_WR_CMD**, SLAVE_RX_BUF_REG and data_written_len [2] - 1, 9 bytes in total.

Host	**CFG_WR_CMD**	SLAVE_RX_BUF_REG	data_written_len [2] - 1
Module	Dummy Data		

Figure 20: Host Data Writing Step 2-a)

b) Host sends **RS_CMD**, checks STA_TXRX_FIFO_RDY.

Host	**RS_CMD**	1 byte Dummy Data
Module	1 byte Dummy Data	Data (including STA_TXRX_FIFO_RDY)

Figure 21: Host Data Writing Step 2-b)

c) Host sends **WR_CMD** and data_written_length [2] bytes of data.

Host	WR_CMD	User's data (data_written_length [2] bytes)
Module	Dummy Data	

Figure 22: Host Data Writing Step 2-c)

d) Host sends **RS_CMD** and checks STA_RDWR_FINISH.

Host	**RS_CMD**	1 byte Dummy Data
Module	1 byte Dummy Data	Data (including STA_RDWR_FINISH)

Figure 23: Host Data Writing Step 2-d)

NOTE

1. [1] The free length in the slave receive buffer indicates the maximum length of data the host can write. If the data to be sent exceeds the free length in the slave receive buffer, the user must divide the data into segments and send them separately.
2. [2] Unsigned integer data_written_len represents the data length that the host intends to write.
3. The module transmits data in little-endian format.

5 Example for SPI Reading/Writing Sequence

Below is an example of the module reading NMEA messages and writing data via the SPI. The **XX** below represents dummy data.

```
//Ensure module SPI is on.
//Master sends PWON_CMD.
04
//Master sends RS_CMD.
06 XX
//Module responds with STA_SLV_ON status flag.
XX 03

//Read data.
//Step1-a): Master sends CFG_RD_CMD, SLAVE_TX_LEN_REG and data length.
0A 08 00 00 00 03 00 00 00
//Step1-b): Master sends RS_CMD.
06 XX
//Module responds with STA_TXRX_FIFO_RDY status flag.
XX 07
//Step1-c): Master sends RD_CMD.
81 XX XX XX XX
//Module responds with data length from slave SPI transmit buffer.
XX F0 04 00 00
//Step1-d): Master sends RS_CMD.
06 XX
//Module responds with STA_RDWR_FINISH status flag.
XX 27
//Step2-a): Master sends CFG_RD_CMD, SLAVE_TX_BUF_REG and data length minus 1 (see Chapter
3.2.2.2 CFG_RD_CMD and CFG_WR_CMD for details).
0A 00 20 00 00 EF 04 00 00
//Step2-b): Master sends RS_CMD.
06 XX
//Module responds with STA_TXRX_FIFO_RDY status flag.
XX 07
//Step2-c): Master sends RD_CMD.
81 XX XX XX XX … XX XX
```

```
//Module responds with NMEA data.
XX 24 50 51 54 … 0D 0A
//Step2-d): Master sends RS_CMD.
06 XX
//Module responds with STA_RDWR_FINISH status flag.
XX 27

//Write data $PQTMVERNO*58\r\n.
//Step1-a): Master sends CFG_RD_CMD, SLAVE_RX_LEN_REG and data length.
0A 04 00 00 00 03 00 00 00
//Step1-b): Master sends RS_CMD.
06 XX
//Module responds with STA_TXRX_FIFO_RDY status flag.
XX 07
//Step1-c): Master sends RD_CMD.
81 XX XX XX XX
//Module responds with data length from slave SPI transmit buffer.
XX 00 01 00 00
//Step1-d): Master sends RS_CMD.
06 XX
//Module responds with STA_RDWR_FINISH status flag.
XX 27
//Step2-a): Master sends CFG_WR_CMD, SLAVE_RX_BUF_REG and data length minus 1 (see Chapter 3.2.2.2 CFG_RD_CMD and CFG_WR_CMD for details).
0C 00 10 00 00 0E 00 00 00
//Step2-b): Master sends RS_CMD.
06 XX
//Module responds with STA_TXRX_FIFO_RDY status flag.
XX 07
//Step2-c): Master sends WR_CMD and data $PQTMVERNO*58\r\n.
0E 24 50 51 54 4D 56 45 52 4E 4F 2A 35 38 0D 0A
//Step2-d): Master sends RS_CMD.
06 XX
//Module responds with STA_RDWR_FINISH status flag.
XX 27
```

6 Sample Code for SPI Reading/Writing Sequence

The sample code for reading data from and writing data to the SPI buffer:

```
#define MAX_ERROR_NUMBER                      100
#define FAIL_MAX_TIME                         5

#define SPI_MAX_BUFF_LENGTH                   0x1000

#define CFG_RD_CMD                            0x0A
#define RD_CMD                                0x81
#define CFG_WR_CMD                            0x0C
#define WR_CMD                                0x0E
#define RS_CMD                                0x06
#define WS_CMD                                0x08
#define PWON_CMD                              0x04
#define PWOFF_CMD                             0x02

#define SLAVE_TX_LEN_REG                      0x08
#define SLAVE_TX_BUF_REG                      0x2000

#define SLAVE_RX_LEN_REG                      0x04
#define SLAVE_RX_BUF_REG                      0x1000

#define SPIS_STA_SLV_ON_OFFSET                (0)
#define SPIS_STA_SLV_ON                       (0x1<<SPIS_STA_SLV_ON_OFFSET)

#define SPIS_STA_TXRX_FIFO_RDY_OFFSET         (2)
#define SPIS_STA_TXRX_FIFO_RDY                (0x1<<SPIS_STA_TXRX_FIFO_RDY_OFFSET)

#define SPIS_STA_RD_ERR_OFFSET                (3)
#define SPIS_STA_RD_ERR                       (0x1<<SPIS_STA_RD_ERR_OFFSET)

#define SPIS_STA_WR_ERR_OFFSET                (4)
#define SPIS_STA_WR_ERR                       (0x1<<SPIS_STA_WR_ERR_OFFSET)

#define SPIS_STA_RDWR_FINISH_OFFSET           (5)
```

```
#define SPIS_STA_RDWR_FINISH                (0x1<<SPIS_STA_RDWR_FINISH_OFFSET)

#define Ql_SPI_SlavePowerOn()               Ql_SPI_Write_Power_Cmd(PWON_CMD)
#define Ql_SPI_SlavePowerOff()              Ql_SPI_Write_Power_Cmd(PWOFF_CMD)
#define QL_SPI_Reset()                      Ql_SPI_SlavePowerOff();\
                                            Ql_SPI_SlavePowerOn()
#define Is_SPI_Slave_Error_Status(Status)   ((Status == SPIS_STA_TXRX_FIFO_RDY) || \
                                          (Status == SPIS_STA_RDWR_FINISH)   && \
                                          (Status == SPIS_STA_SLV_ON))

typedef enum
{
    STATUS_OK,
    STATUS_ERROR,
} Ql_Host_Status_TypeDef;

typedef struct
{
    uint16_t ReceiveLength;
    uint16_t SendLength;
    uint8_t* SendData;
    uint8_t* ReceiveBuffer;
} Ql_SPI_Send_Receive_TypeDef;

uint8_t Ql_SPI_RecvBuffer[SPI_MAX_BUFF_LENGTH+1];
uint8_t Ql_SPI_SendBuffer[SPI_MAX_BUFF_LENGTH+1];

void Ql_SPI_Write_Power_Cmd(uint8_t Cmd);

static Ql_Host_Status_TypeDef Ql_SPI_Power_State(uint32_t BitMask, uint32_t
BitValue, uint32_t RetryCounter)
{
    uint8_t status_cmd[2] = {RS_CMD,0x00};
    uint8_t status = 0;
    uint8_t status_receive[2] = {0};
    uint8_t clear_cmd_buf[2] = {0};
    uint8_t temp_cmd = PWOFF_CMD;
    uint8_t recv_buf[2] = {0};
    Ql_SPI_Send_Receive_TypeDef spi_send_and_receive_config = {0};
    for(int i = 0; i < RetryCounter; i ++)
    {
        status_receive[1] = 0;
        spi_send_and_receive_config.ReceiveLength = 2;
        spi_send_and_receive_config.SendLength = 1;
```

```
        spi_send_and_receive_config.SendData = status_cmd;
        spi_send_and_receive_config.ReceiveBuffer = status_receive;
        HAL_SPI_WriteRead(spi_send_and_receive_config.SendData,       \
                         spi_send_and_receive_config.ReceiveBuffer,  \
                         spi_send_and_receive_config.ReceiveLength);
        status = status_receive[1];
        if (status != 0xff)
        {
            if (status & (SPIS_STA_RD_ERR|SPIS_STA_WR_ERR))
            {
                clear_cmd_buf[0] = WS_CMD;
                clear_cmd_buf[1] = status;
                HAL_SPI_WriteRead(clear_cmd_buf, status_receive, 2);
                return STATUS_ERROR;
            }
            else if ((BitMask & status)== BitValue)
            {
                return STATUS_OK;
            }
            else
            {
                return STATUS_ERROR;
            }
        }
    }
    return STATUS_ERROR;
}

void Ql_SPI_Write_Power_Cmd(uint8_t Cmd)
{
    uint8_t temp_cmd = Cmd;
    uint8_t recv_buf[2] = {0};
    switch(temp_cmd)
    {
        case PWON_CMD:
        {
            while(1)
            {
                HAL_SPI_WriteRead(&temp_cmd, recv_buf, 1);
                if (STATUS_OK == Ql_SPI_Power_State(SPIS_STA_SLV_ON, SPIS_STA_SLV_ON,
MAX_ERROR_NUMBER))
                {
                    break;
                }
```

```
            }
        }
        break;
        case PWOFF_CMD:
        {
            while(1)
            {
                HAL_SPI_WriteRead(&temp_cmd, recv_buf, 1);
                if (STATUS_OK == Ql_SPI_Ql_SPI_Power_State(SPIS_STA_SLV_ON, 0,
MAX_ERROR_NUMBER))
                {
                    break;
                }
            }
        }
        break;
    }
}

static Ql_Host_Status_TypeDef Ql_SPI_Query_Slave_Status(uint32_t BitMask, uint32_t
BitValue, uint32_t RetryCounter)
{
    uint8_t status_cmd[2] = {RS_CMD,0x00};
    uint8_t status = 0;
    uint8_t status_receive[2] = {0};
    uint8_t clear_cmd_buf[2] = {0};
    Ql_SPI_Send_Receive_TypeDef spi_send_and_receive_config = {0};
    for(int i = 0; i < RetryCounter; i ++)
    {
        HAL_Delay(1);
        status_receive[1] = 0;
        spi_send_and_receive_config.ReceiveLength = 2;
        spi_send_and_receive_config.SendLength = 1;
        spi_send_and_receive_config.SendData = status_cmd;
        spi_send_and_receive_config.ReceiveBuffer = status_receive;
        HAL_SPI_WriteRead(spi_send_and_receive_config.SendData,     \
                          spi_send_and_receive_config.ReceiveBuffer,\
                          spi_send_and_receive_config.ReceiveLength);
        status = status_receive[1];
        if (status != 0xff)
        {
            if (status & (SPIS_STA_RD_ERR|SPIS_STA_WR_ERR))
            {
                clear_cmd_buf[0]= WS_CMD;
```

```
                clear_cmd_buf[1] = status;
                HAL_SPI_WriteRead(clear_cmd_buf, status_receive, 2);
                if(BitMask != SPIS_STA_TXRX_FIFO_RDY)
                {
                    QL_SPI_Reset();
                    HAL_Delay(1);
                }
                return STATUS_ERROR;
            }
            else if((BitMask & status)== BitValue)
            {
                return STATUS_OK;
            }
            else if (Is_SPI_Slave_Error_Status(BitMask))
            {
                QL_SPI_Reset();
                return STATUS_ERROR;
            }
        }
        else
        {
            return STATUS_ERROR;
        }
    }
    return STATUS_ERROR;
}

uint8_t Ql_SPI_WriteAndRead(uint32_t Offset, uint8_t *Buffer, uint32_t *Length)
{
    uint8_t cfg_cmd[9] = {0};
    uint8_t recv_cmd[9] = {0};
    uint32_t receive_reg_value = 0;
    uint32_t temp_offset = Offset;
    Ql_SPI_Send_Receive_TypeDef spi_send_and_receive_config = {0};
    uint32_t temp_length = 0;
    uint32_t total_length = 0;
    uint8_t fail_counter = 0;
    static  uint8_t temp_host_mux_send_buf[4+1] = {0};
    static  uint8_t temp_host_mux_receive_buf[4+1] = {0};

_restart:
    temp_length = 4;
    cfg_cmd[0] = CFG_RD_CMD;
    cfg_cmd[1] = Offset & 0xff;
```

```
    cfg_cmd[2] = (Offset >> 8) & 0xff;
    cfg_cmd[3] = (Offset >> 16) & 0xff;
    cfg_cmd[4] = (Offset >> 24) & 0xff;
    cfg_cmd[5] = (temp_length - 1) & 0xff;
    cfg_cmd[6] = ((temp_length - 1) >> 8) & 0xff;
    cfg_cmd[7] = ((temp_length - 1) >> 16) & 0xff;
    cfg_cmd[8] = ((temp_length - 1) >> 24) & 0xff;
    HAL_SPI_WriteRead(cfg_cmd,recv_cmd,9);
    if(Ql_SPI_Query_Slave_Status(SPIS_STA_TXRX_FIFO_RDY, \
                                 SPIS_STA_TXRX_FIFO_RDY, \
                                 MAX_ERROR_NUMBER) == STATUS_ERROR)
    {
        if(fail_counter >= FAIL_MAX_TIME)
        {
            fail_counter = 0;
            return STATUS_ERROR;
        }
        fail_counter ++;
        goto _restart;
    }
    else
    {
        fail_counter = 0;
    }
    switch(temp_offset)
    {
        case SLAVE_TX_LEN_REG://Host read
        {
            temp_host_mux_send_buf[0] = RD_CMD;
            spi_send_and_receive_config.ReceiveLength = 4+1;
            spi_send_and_receive_config.SendLength = 1;
            spi_send_and_receive_config.SendData = temp_host_mux_send_buf;
            spi_send_and_receive_config.ReceiveBuffer = temp_host_mux_receive_buf;
            HAL_SPI_WriteRead(spi_send_and_receive_config.SendData,     \
                              spi_send_and_receive_config.ReceiveBuffer,\
                              spi_send_and_receive_config.ReceiveLength);
            if(Ql_SPI_Query_Slave_Status(SPIS_STA_RDWR_FINISH, \
                                         SPIS_STA_RDWR_FINISH, \
                                         MAX_ERROR_NUMBER) == STATUS_ERROR)
            {
                if(fail_counter >= FAIL_MAX_TIME)
                {
                    fail_counter = 0;
                    return STATUS_ERROR;
```

```
            }
            fail_counter ++;
            goto _restart;
        }
        else
        {
            fail_counter = 0;
        }
        receive_reg_value = temp_host_mux_receive_buf[1]      | \
                            (temp_host_mux_receive_buf[2]<<8) | \
                            (temp_host_mux_receive_buf[3]<<16)| \
                            (temp_host_mux_receive_buf[4]<<24);
        if(receive_reg_value > SPI_MAX_BUFF_LENGTH)
        {
            total_length = receive_reg_value;
            receive_reg_value = SPI_MAX_BUFF_LENGTH;
        }
        if(receive_reg_value==0)
        {
            *Length = receive_reg_value;
            return STATUS_OK;
        }
        temp_length = receive_reg_value;
        Offset = SLAVE_TX_BUF_REG;
        cfg_cmd[0] = CFG_RD_CMD;
        cfg_cmd[1] = Offset & 0xff;
        cfg_cmd[2] = (Offset >> 8) & 0xff;
        cfg_cmd[3] = (Offset >> 16) & 0xff;
        cfg_cmd[4] = (Offset >> 24) & 0xff;
        cfg_cmd[5] = (temp_length - 1) & 0xff;
        cfg_cmd[6] = ((temp_length - 1) >> 8) & 0xff;
        cfg_cmd[7] = ((temp_length - 1) >> 16) & 0xff;
        cfg_cmd[8] = ((temp_length - 1) >> 24) & 0xff;
        HAL_SPI_WriteRead(cfg_cmd, recv_cmd, 9);
        if(Ql_SPI_Query_Slave_Status(SPIS_STA_TXRX_FIFO_RDY, \
                                     SPIS_STA_TXRX_FIFO_RDY, \
                                     MAX_ERROR_NUMBER) == STATUS_ERROR)
        {
            if(fail_counter >= FAIL_MAX_TIME)
            {
                fail_counter = 0;
                return STATUS_ERROR;
            }
            fail_counter ++;
```

```
            goto _restart;
        }
        else
        {
            fail_counter = 0;
        }
        Ql_SPI_SendBuffer[0] = RD_CMD;
        spi_send_and_receive_config.ReceiveLength = temp_length+1;
        spi_send_and_receive_config.SendLength = 1;
        spi_send_and_receive_config.SendData = Ql_SPI_SendBuffer;
        spi_send_and_receive_config.ReceiveBuffer = Buffer;
        HAL_SPI_WriteRead(spi_send_and_receive_config.SendData,      \
                          spi_send_and_receive_config.ReceiveBuffer, \
                          spi_send_and_receive_config.ReceiveLength);
        if(Ql_SPI_Query_Slave_Status(SPIS_STA_RDWR_FINISH, \
                                     SPIS_STA_RDWR_FINISH, \
                                     MAX_ERROR_NUMBER) == STATUS_ERROR)
        {
            if(fail_counter >= FAIL_MAX_TIME)
            {
                fail_counter = 0;
                return STATUS_ERROR;
            }
            fail_counter ++;
            goto _restart;
        }
        else
        {
            fail_counter = 0;
        }
        *Length = receive_reg_value;
        return STATUS_OK;
    }
    break;

    case SLAVE_RX_LEN_REG://Host send
    {
        temp_host_mux_send_buf[0] = RD_CMD;
        spi_send_and_receive_config.ReceiveLength = 4+1;
        spi_send_and_receive_config.SendLength = 1;
        spi_send_and_receive_config.SendData = temp_host_mux_send_buf;
        spi_send_and_receive_config.ReceiveBuffer = temp_host_mux_receive_buf;
        HAL_SPI_WriteRead(spi_send_and_receive_config.SendData,      \
                          spi_send_and_receive_config.ReceiveBuffer, \
```

```
                    spi_send_and_receive_config.ReceiveLength);
        if(Ql_SPI_Query_Slave_Status(SPIS_STA_RDWR_FINISH, \
                              SPIS_STA_RDWR_FINISH, \
                              MAX_ERROR_NUMBER) == STATUS_ERROR)
        {
            if(fail_counter >= FAIL_MAX_TIME)
            {
                fail_counter = 0;
                return STATUS_ERROR;
            }
            fail_counter ++;
            goto _restart;
        }
        else
        {
            fail_counter = 0;
        }

        receive_reg_value = temp_host_mux_receive_buf[1]      | \
                      (temp_host_mux_receive_buf[2]<<8) | \
                      (temp_host_mux_receive_buf[3]<<16)| \
                      (temp_host_mux_receive_buf[4]<<24);

        if(receive_reg_value > *Length)
        {
            receive_reg_value = *Length;
        }
        if(receive_reg_value == 0)
        {
            return STATUS_OK;
        }
        temp_length = receive_reg_value;
        Offset = SLAVE_RX_BUF_REG;
        cfg_cmd[0] = CFG_WR_CMD;
        cfg_cmd[1] = Offset & 0xff;
        cfg_cmd[2] = (Offset >> 8) & 0xff;
        cfg_cmd[3] = (Offset >> 16) & 0xff;
        cfg_cmd[4] = (Offset >> 24) & 0xff;
        cfg_cmd[5] = (temp_length - 1) & 0xff;
        cfg_cmd[6] = ((temp_length - 1) >> 8) & 0xff;
        cfg_cmd[7] = ((temp_length - 1) >> 16) & 0xff;
        cfg_cmd[8] = ((temp_length - 1) >> 24) & 0xff;
        HAL_SPI_WriteRead(cfg_cmd, recv_cmd, 9);
        if(Ql_SPI_Query_Slave_Status(SPIS_STA_TXRX_FIFO_RDY, \
```

```
                                SPIS_STA_TXRX_FIFO_RDY, \
                                MAX_ERROR_NUMBER) == STATUS_ERROR)
            {
                if(fail_counter >= FAIL_MAX_TIME)
                {
                    fail_counter = 0;
                    return STATUS_ERROR;
                }
                fail_counter++;
                goto _restart;
            }
            else
            {
                fail_counter = 0;
            }
            Ql_SPI_SendBuffer[0] = WR_CMD;
            memcpy(&Ql_SPI_SendBuffer[1], Buffer, receive_reg_value);
            HAL_SPI_WriteRead(Ql_SPI_SendBuffer, Ql_SPI_RecvBuffer, 
receive_reg_value+1);
            memset(Ql_SPI_RecvBuffer, 0, receive_reg_value+1);
            if(Ql_SPI_Query_Slave_Status(SPIS_STA_RDWR_FINISH, \
                                SPIS_STA_RDWR_FINISH, \
                                MAX_ERROR_NUMBER) == STATUS_ERROR)
            {
                if(fail_counter >= FAIL_MAX_TIME)
                {
                    fail_counter = 0;
                    return STATUS_ERROR;
                }
                fail_counter ++;
                goto _restart;
            }
            else
            {
                fail_counter = 0;
            }
            *Length = receive_reg_value;
            return STATUS_OK;
        }
        break;

        default:
        {
            // No Code
```

```
        }
        break;
    }
    return STATUS_ERROR;
}

uint8_t Ql_SPI_Write_Data(uint8_t* DataBuffer,uint32_t Length)
{
    uint8_t result = 0;
    Ql_SPI_SlavePowerOn();
    HAL_Delay(2);
    result = Ql_SPI_WriteAndRead(SLAVE_RX_LEN_REG,DataBuffer,&Length);
    return result;
}

uint8_t Ql_SPI_Read_Data(uint8_t* DataBuffer,uint32_t* Length)
{
    uint8_t result = 0;
    Ql_SPI_SlavePowerOn();
    HAL_Delay(2);
    result = Ql_SPI_WriteAndRead(SLAVE_TX_LEN_REG,DataBuffer,Length);
    return result;
}
```

7 Appendix References

Table 5: Related Documents

Document Name
[1] Quectel_LC76G_Series_Hardware_Design
[2] Quectel_LC26G&LC26G-T&LC76G&LC86G_Series_GNSS_Protocol_Specification

Table 6: Terms and Abbreviations

Abbreviation	Description
CLK	Serial Clock
CPHA	Clock Phase
CPOL	Clock Polarity
CS	Chip Select
FIFO	First In First Out
GNSS	Global Navigation Satellite System
LSB	Least Significant Bit
MCU	Microcontroller Unit
MISO	Master In Slave Out
MOSI	Master Out Slave In
MSB	Most Significant Bit
NMEA	NMEA (National Marine Electronics Association) 0183 Interface Standard
SPI	Serial Peripheral Interface