Lx6&LG77L I2C Application Note

GNSS Module Series

Version: 1.2

Date: 2021-12-15

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	**Lx6&LG77L I2C Application Note**
Subtitle	GNSS Module Series
Document Type	Application Note
Document Status	Released

Revision History

Revision	Date	Description
1.0	2016-09-14	Initial
1.1	2018-10-16	Added L96 as an applicable module of this document.
1.2	2021-12-15	Added L26-LB, L76-LB and LG77L as applicable modules of this document.

Contents

Table Index

Figure Index

1 Introduction

This document introduces the I2C function and use. The modules working as slaves provide an I2C interface which outputs NMEA data read by a master (MCU). The modules' I2C interface includes the following features:

- Supports fast mode, with a bit rate up to 400 kbps.
- Supports 7-bit address.
- Works in slave mode.
- Default slave address values: Write: 0x20, Read: 0x21.
- I2C pins: I2C_SDA and I2C_SCL.

This document also provides a detailed introduction as well as a flow chart and sample code to illustrate how the master reads/parses NMEA sentences and sends messages via the I2C bus.

This document is applicable to the following Quectel GNSS modules:

- L26-LB
- L76-L
- L76-LB
- L96
- LG77L

NOTE

The I2C interface is only supported on firmware versions ending with "SC".

2 NMEA Data Reading via I2C Bus

This chapter provides a detailed introduction on how the master reads and parses NMEA data packets via I2C bus. The master can read a 255-byte data packet via I2C bus at a time and the data need to be processed because some of them are garbage bytes which are not useful.

2.1. NMEA Data Reading Flow of the Master

The slave's I2C buffer has a capacity of 255 bytes, which means that the master can read one I2C data packet of a maximum size of 255 bytes at a time. In order to get a complete NMEA packet of one second, the master needs to read several I2C data packets and then extract valid NMEA data from them.

After reading one I2C data packet, the master should be set to sleep for 2 ms before it starts to receive the next I2C data packet, as the slave needs 2 ms to upload the new I2C data into the I2C buffer. When the entire NMEA packet of one second is read, the master can sleep for a longer time (e.g., 500 ms) to wait for the entire NMEA packet of the next second to be ready.

The NMEA data packet can be read via the I2C bus only in polling mode. To avoid data loss, the master should read the entire NMEA packet of one second in a polling interval. The interval can be configured with **PMTK314** according to the GNSS fix interval and it should be shorter than the GNSS fix interval. See ***document [1]*** for details on **PMTK314**.

The following figure illustrates how the master reads NMEA data packets via I2C in polling mode.

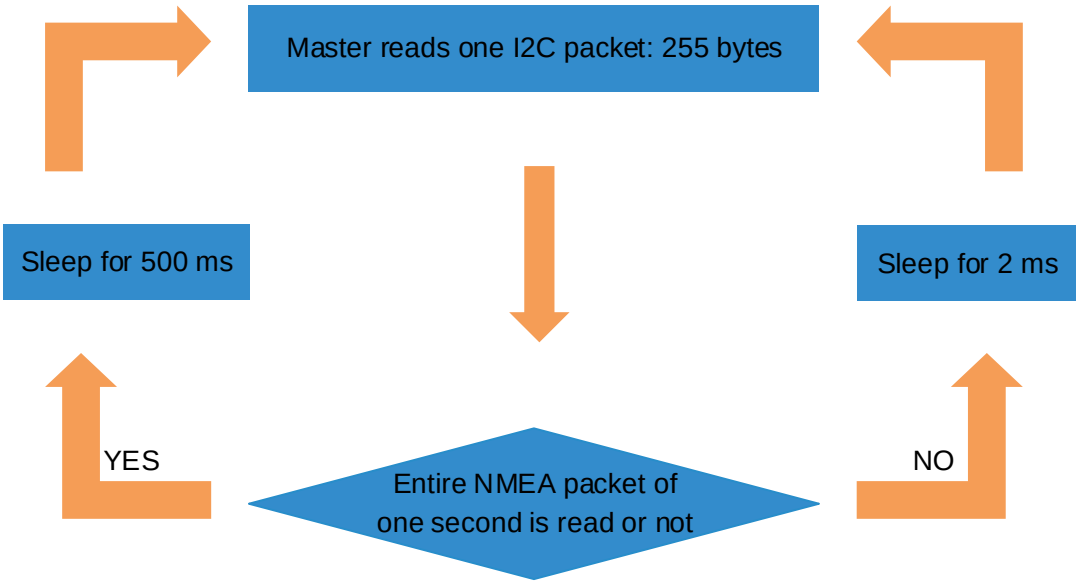


Figure 1: NMEA Data Reading Flow of the Master in Polling Mode

NOTE

The figure above assumes that the GNSS fix interval is 1 s, and the recommended polling interval is 500 ms.

2.2. I2C Data Packets

2.2.1. Format of I2C Data Packet

The data packet in the slave's I2C buffer (I2C data packet) includes 254 valid NMEA bytes at most and one end character **<LF>**, so the master can read maximally a 255-byte I2C data packet at a time. The following figure illustrates I2C data packet format.

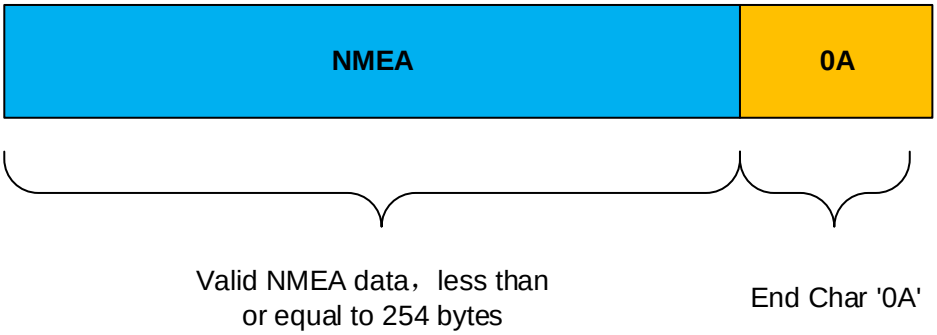


Figure 2: Format of I2C Data Packet

There are maximally 254 valid NMEA data bytes and one end character **<LF>** in one I2C data packet, as shown below:

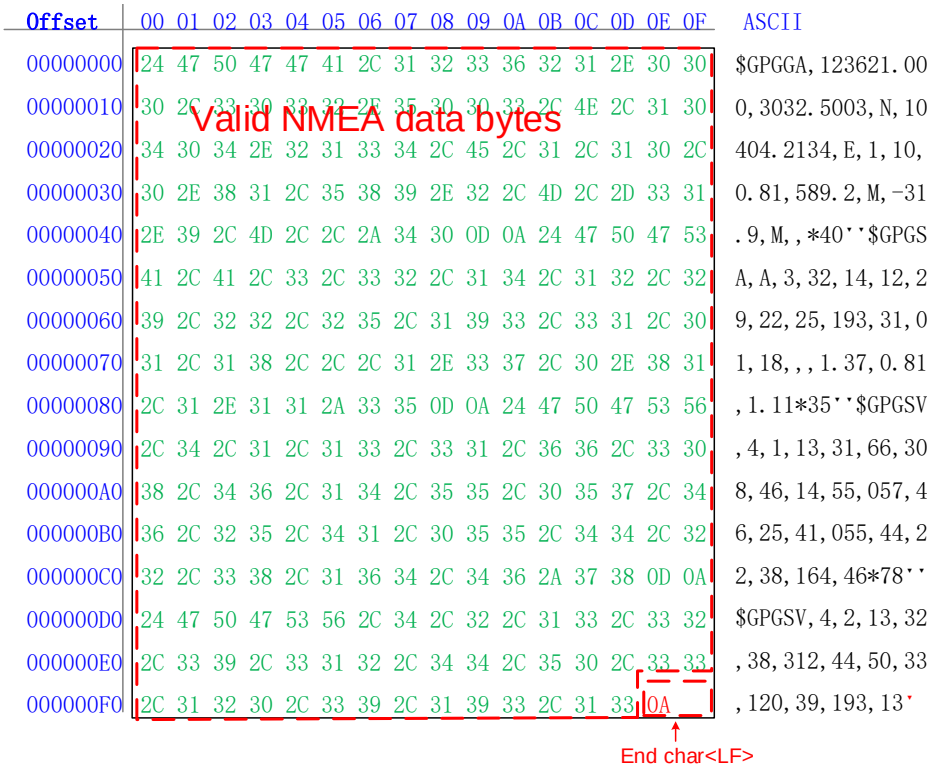


Figure 3: Example of I2C Data Packet Format

2.2.2. Three Types of I2C Data Packets

Regardless of whether NMEA data are stored in the I2C buffer, the master can read one I2C data packet (255 bytes) from the slave at a time. There are three types of I2C data packets that the master can read from the slave.

2.2.2.1. Type 1: Valid Data Bytes + Garbage Bytes

When the I2C buffer has already stored some data, the master will read the stored data first, and then garbage bytes. If 254 valid NMEA bytes are all stored in the buffer, the last byte will be the end character **<LF>**.

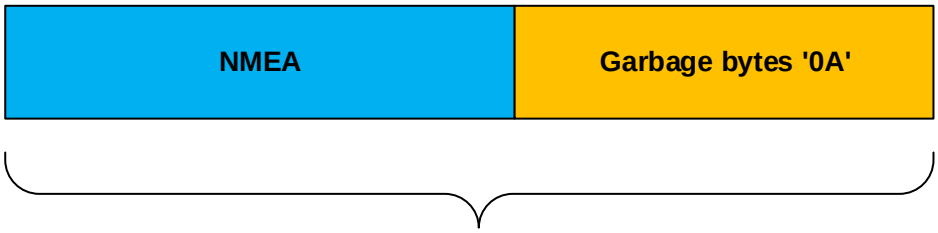


Figure 4: Type 1 (Valid Data Bytes + Garbage Bytes)

For example, if the slave I2C buffer has stored 202-byte NMEA data, the 255-byte I2C data packet read by the master includes 202 valid data bytes and 53 garbage bytes. An example is shown below:

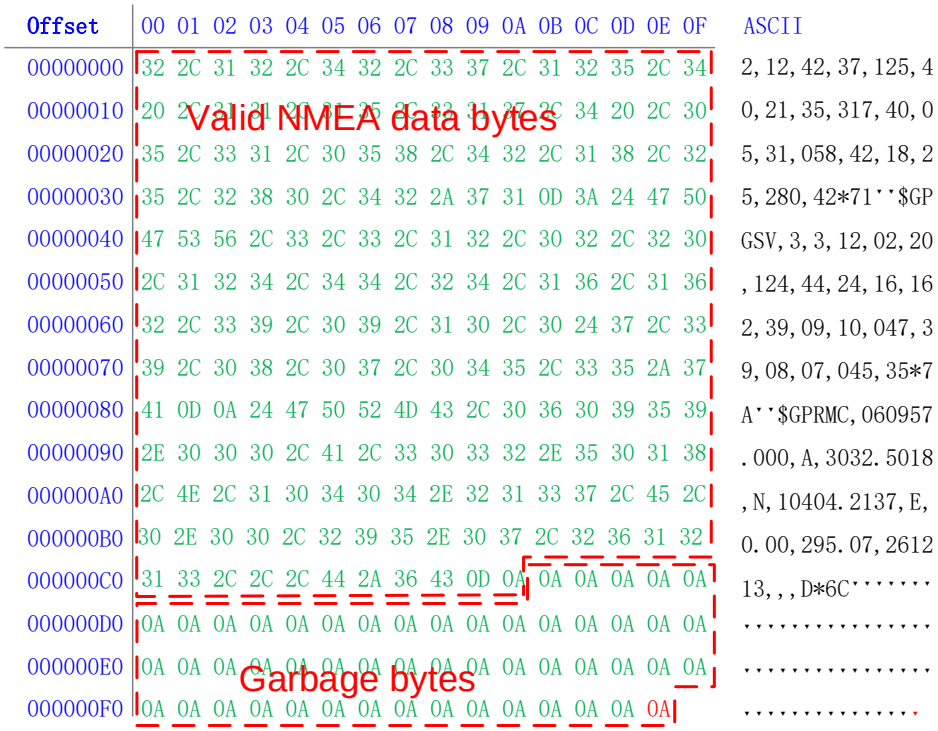


Figure 5: Example of Type 1 (Valid Data Bytes + Garbage Bytes)

NOTE

Why are garbage bytes '0A'?

If the slave's I2C buffer is empty, the slave will repeatedly output the last valid byte until new data are uploaded into the I2C buffer, and '0A' is the last valid byte in the NMEA packet.

2.2.2.2. Type 2: All Garbage Bytes

When the slave I2C buffer is empty, the master will read only garbage bytes.

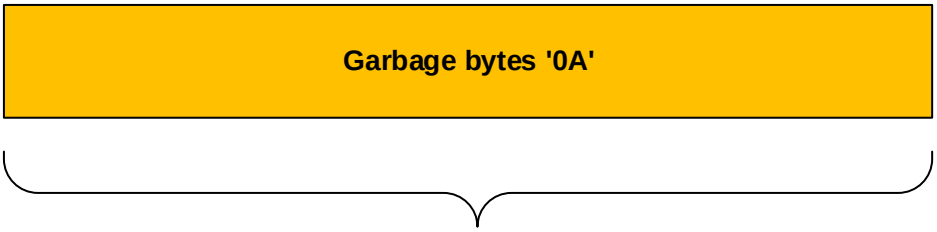


Figure 6: Type 2 (All Garbage Bytes)

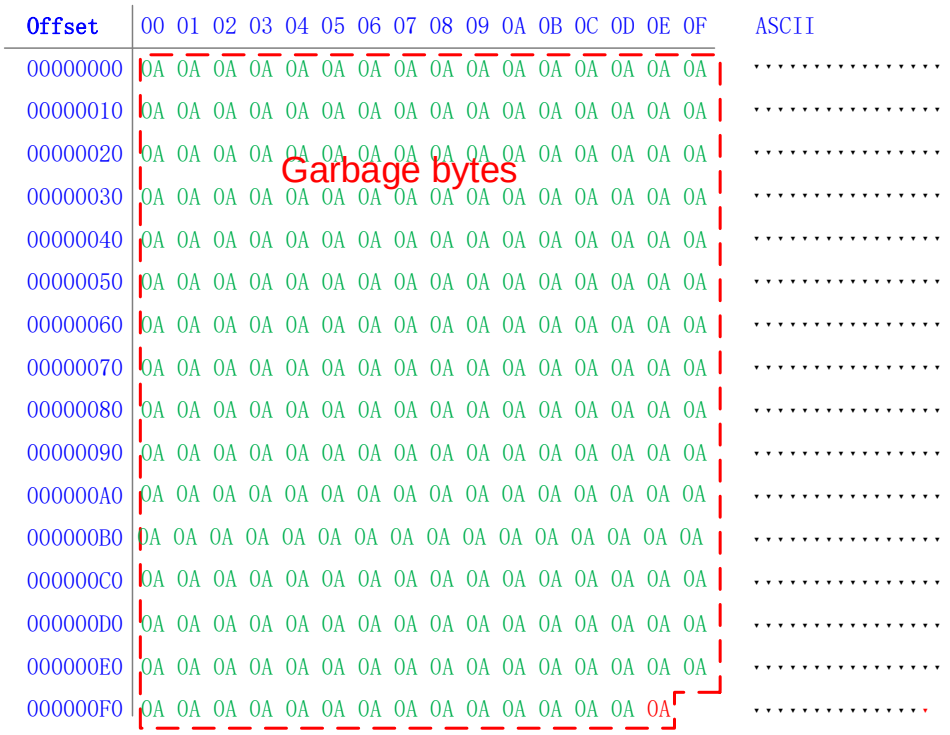


Figure 7: Example of Type 2 (All Garbage Bytes)

2.2.2.3. Type 3: Garbage Bytes + Valid Data Bytes

If the slave I2C buffer is empty when the master starts reading, but the slave starts uploading new data into the I2C buffer before the reading is over, the master will read garbage bytes first and then valid NMEA data bytes.

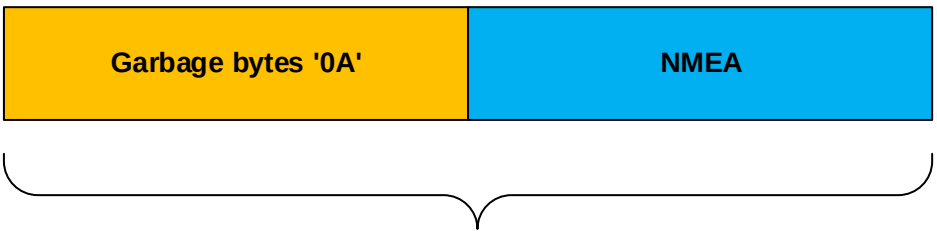
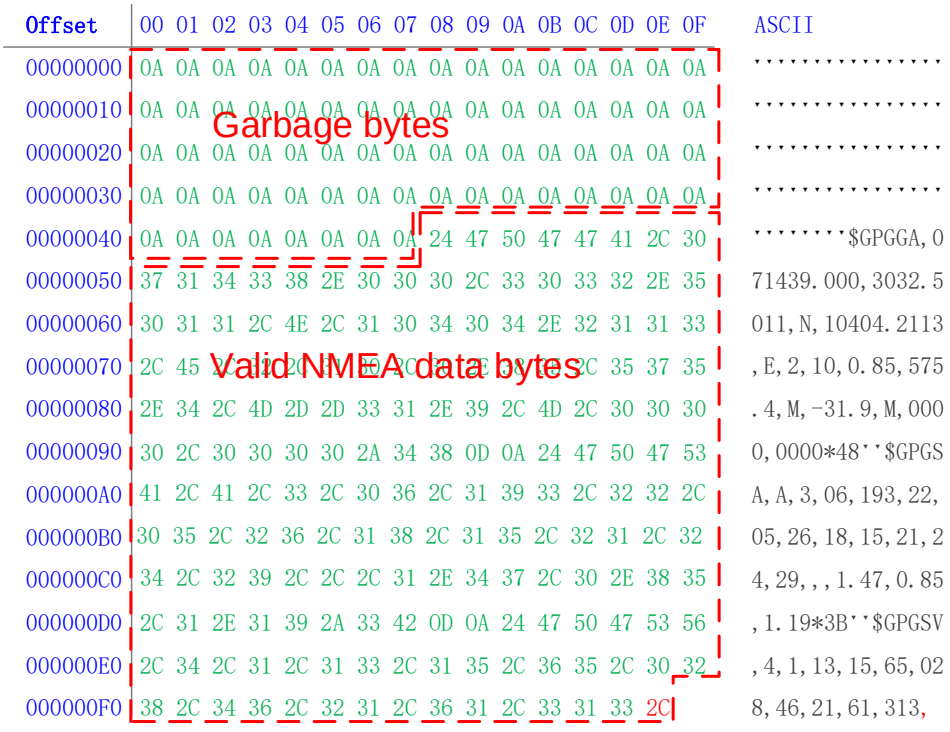


Figure 8: Type 3 (Garbage Bytes + Valid Data Bytes)



Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00000000	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	····
00000010	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	····
00000020	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	····
00000030	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	0A	····
00000040	0A	0A	0A	0A	0A	0A	0A	0A	24	47	50	47	47	41	2C	30	····$GPGGA,0
00000050	37	31	34	33	39	2E	30	30	30	2C	33	30	33	32	2E	35	71439.000,3032.5
00000060	30	31	31	2C	4E	2C	31	30	34	30	34	2E	32	31	31	33	011,N,10404.2113
00000070	2C	45	2C	32	2C	31	30	2C	30	2E	38	35	2C	35	37	35	,E,2,10,0.85,575
00000080	2E	34	2C	4D	2D	2D	33	31	2E	39	2C	4D	2C	30	30	30	.4,M,-31.9,M,000
00000090	30	2C	30	30	30	30	2A	34	38	0D	0A	24	47	50	47	53	0,0000*48··$GPGS
000000A0	41	2C	41	2C	33	2C	30	36	2C	31	39	33	2C	32	32	2C	A,A,3,06,193,22,
000000B0	30	35	2C	32	36	2C	31	38	2C	31	35	2C	32	31	2C	32	05,26,18,15,21,2
000000C0	34	2C	32	39	2C	2C	2C	31	2E	34	37	2C	30	2E	38	35	4,29,,,1.47,0.85
000000D0	2C	31	2E	31	39	2A	33	42	0D	0A	24	47	50	47	53	56	,1.19*3B··$GPGSV
000000E0	2C	34	2C	31	2C	31	33	2C	31	35	2C	36	35	2C	30	32	,4,1,13,15,65,02
000000F0	38	2C	34	36	2C	32	31	2C	36	31	2C	33	31	33	2C		8,46,21,61,313,

Figure 9: Example of Type 3 (Garbage Bytes + Valid Data Bytes)

2.2.3. How to Extract Valid NMEA Data from Several I2C Packets

After the master reads sufficient I2C data packets, it needs to parse and extract valid NMEA data from these packets. See ***Chapter 5.2*** for the sample code provided by Quectel to extract the valid data.

NOTE

When extracting NMEA data from I2C packets, all '0A' characters should be discarded. The '0A' character may come in the form of:

1. End character of an I2C packet.
2. Garbage bytes.
3. End character **<LF>** of an NMEA sentence. If it is discarded, there is no effect on NMEA sentence parsing.

3 Sending Messages via I2C Bus

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

As the slave's I2C buffer has a maximum capacity of 255 bytes, each message inputted by the master should be 255 bytes at most. The interval between two input messages cannot be shorter than 10 ms as the slave needs 10 ms to process the inputted data.

4 Procedures for I2C Data Reading and Writing

The chapter provides the sequence charts and sample code for I2C data reading and writing.

4.1. Sequence Charts

The sequence charts for reading data from and writing data to the I2C buffer are provided below.

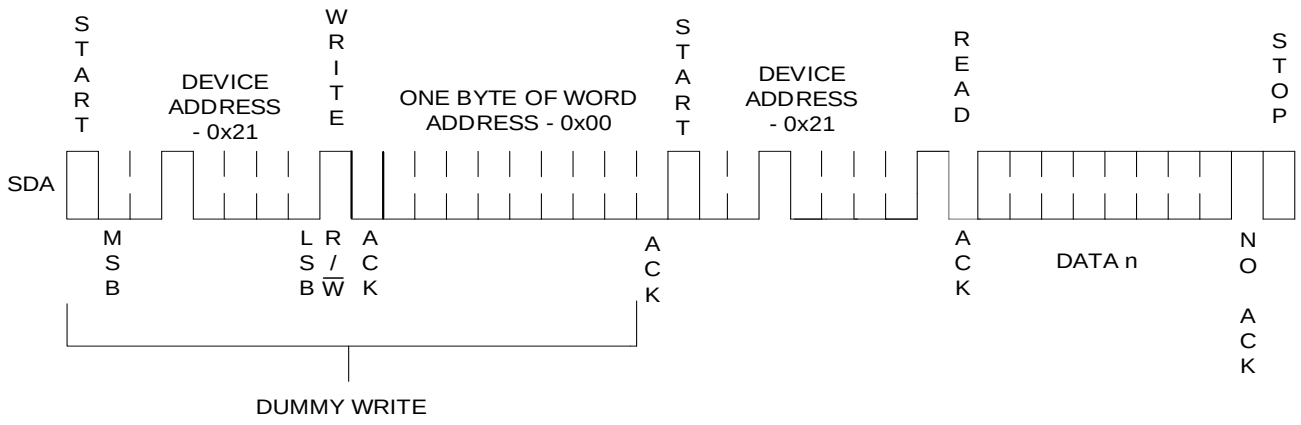


Figure 10: Sequence Chart for Reading Data from I2C Buffer

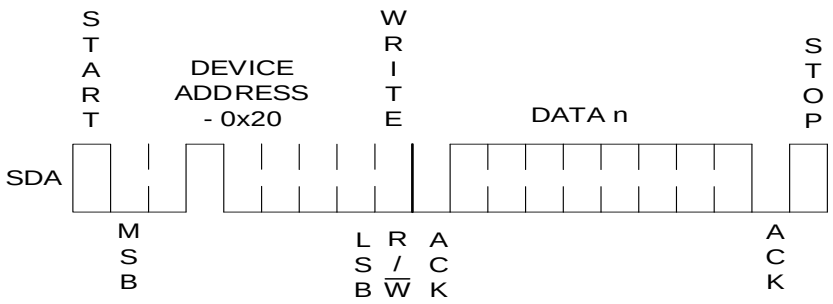


Figure 11: Sequence Chart for Writing Data to I2C Buffer

4.2. Sample Code

The sample code for reading data from and writing data to the I2C buffer:

```
#define WRITE_I2C_DATA_ADDRESS  0x20
#define READ_I2C_DATA_ADDRESS   0x21
#define READ_I2C_DATA_LENGTH    255

uint8_t i2c_recive_data[READ_I2C_DATA_LENGTH + 1];

void read_i2c_bytes(uint8_t *buf, uint16_t length)
{
  uint16_t i;
  IIC_Start();
  IIC_Send_Byte(WRITE_I2C_DATA_ADDRESS);
  if (IIC_Wait_Ack() != 0)
  {
    IIC_Stop();
    return;
  }

  IIC_Send_Byte((uint8_t)0x00);
  if (IIC_Wait_Ack() != 0)
  {
    IIC_Stop();
    return ;
  }
  IIC_Start();
  IIC_Send_Byte(READ_I2C_DATA_ADDRESS);

  if(IIC_Wait_Ack() != 0)
  {
    IIC_Stop();
    return;
  }

 for (i = 0; i < READ_I2C_DATA_LENGTH; i++)
 {
   buf[i] = IIC_Read_Byte();

   if (i != READ_I2C_DATA_LENGTH - 1)
   {
     IIC_Ack();
```

```
    }
    else
    {
      IIC_NAck();
    }
  }
  IIC_Stop();
}

void write_i2c_bytes (uint8_t *buf, uint16_t length)
{
  uint16_t i=0;
  IIC_Stop();
  IIC_Start();
  IIC_Send_Byte(WRITE_I2C_DATA_ADDRESS);
  if (IIC_Wait_Ack() != 0)
  {
    return;
  }

  for(i = 0; i < length; i++)
  {
    IIC_Send_Byte(buf[i]);
    if (IIC_Wait_Ack() != 0)
    {
      return;
    }
  }
  IIC_Stop();
  return;
}
```

5 Procedures for Receiving and Parsing NMEA Sentences

This chapter provides the flow chart and sample code on how the master receives and parses NMEA sentences via I2C.

5.1. Flow Chart

The flow chart on how the master receives and parses NMEA sentences is shown below.

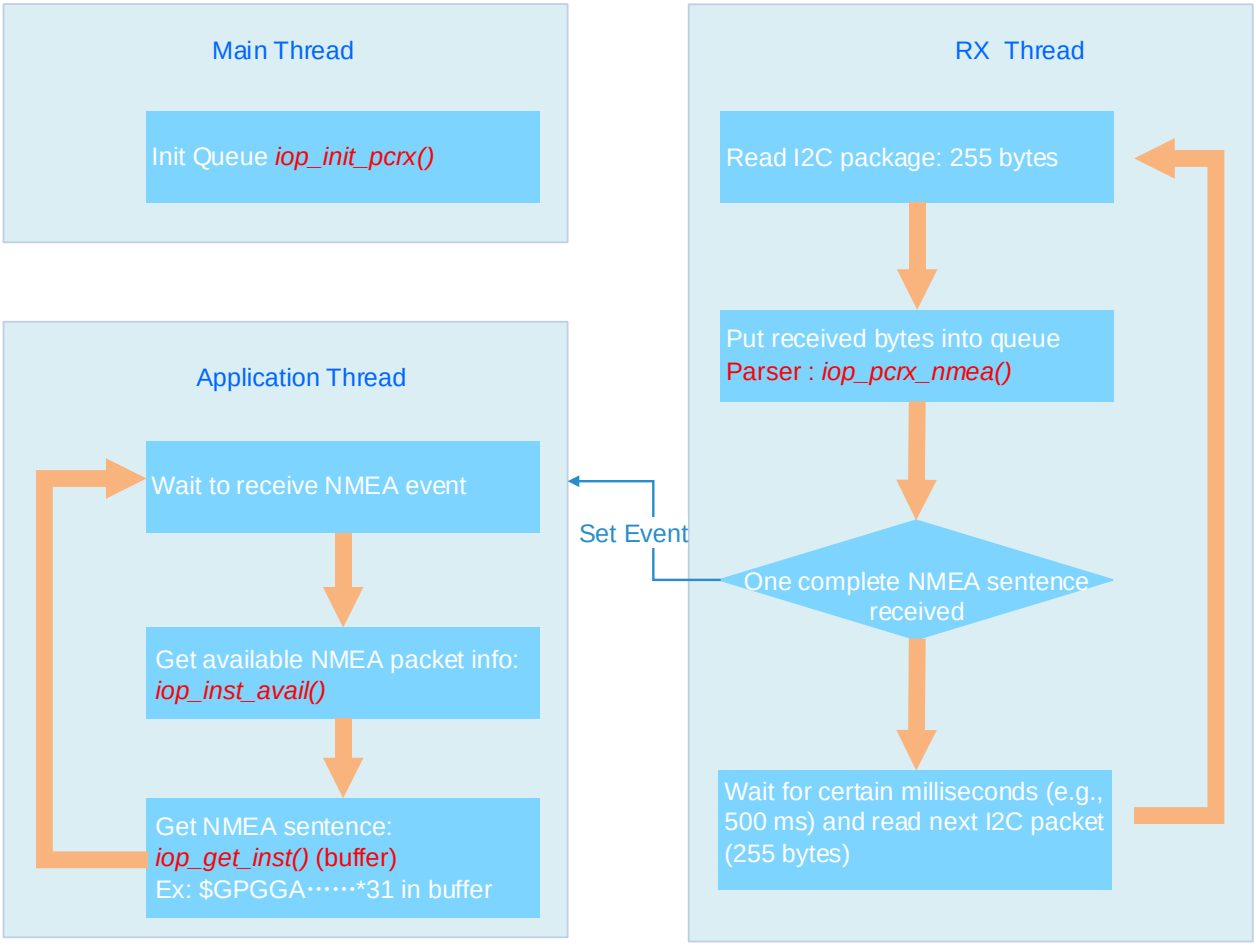


Figure 12: Flow Chart for Receiving and Parsing NMEA Sentences

5.2. Sample Code

This chapter provides the sample code for receiving and parsing NMEA sentences. The master will parse the received NMEA and debug data so as to discard garbage bytes and extract valid data from multiple data packets.

The functions used in the sample code are presented in the following table.

Table 1: Function Description

Function	Description
iop_init_pcrx()	Initializes reception queue.
iop_inst_avail()	Gets available NMEA sentence information.
iop_get_inst()	Gets NMEA sentence data from queue buffer.
iop_pcrx_nmea()	Processes I2C packets, gets valid NMEA data and discards garbage bytes.
iop_pcrx_nmea_dbg_hbd_bytes()	Processes I2C packets, gets valid NMEA data, debug log code and discards garbage bytes.

The sample code for receiving and parsing I2C NMEA sentences:

```
#define IOP_LF_DATA      0x0A    //<LF>
#define IOP_CR_DATA      0x0D    //<CR>
#define IOP_START_DBG    0x23    //Debug log start char ‘#’
#define IOP_START_NMEA   0x24   //NMEA start char ‘$’
#define IOP_START_HBD1  'H'     //HBD debug log start char ‘H’
#define IOP_START_HBD2  'B'
#define IOP_START_HBD3  'D'
#define NMEA_ID_QUE_SIZE 0x0100
#define NMEA_RX_QUE_SIZE 0x8000
typedef enum
{
  RXS_DAT_HBD, //Receive HBD data
  RXS_PRM_HBD2, //Receive HBD preamble 2
  RXS_PRM_HBD3, //Receive HBD preamble 3
  RXS_DAT, //Receive NMEA data
  RXS_DAT_DBG, //Receive DBG data
  RXS_ETX, //End-of-packet
  } RX_SYNC_STATE_T;
struct
```

```
{
  short inst_id; //1 - NMEA, 2 - DBG, 3 - HBD
  short dat_idx;
  short dat_siz;
} id_que[NMEA_ID_QUE_SIZE];
char rx_que[NMEA_RX_QUE_SIZE];
unsigned short id_que_head;
unsigned short id_que_tail;
unsigned short rx_que_head;
RX_SYNC_STATE_T rx_state;
unsigned int u4SyncPkt;
unsigned int u4OverflowPkt;
unsigned int u4PktInQueue;
//Queue Functions
BOOL iop_init_pcrx( void )
{
  /*-------------------------------------------------------------
  variables
  -------------------------------------------------------------*/
  short i;
  /*-------------------------------------------------------------
  initialize queue indexes
  -------------------------------------------------------------*/
  id_que_head = 0;
  id_que_tail = 0;
  rx_que_head = 0;
  /*-------------------------------------------------------------
  initialize identification queue
  -------------------------------------------------------------*/
  for( i=0; i< NMEA_ID_QUE_SIZE; i++)
  {
    id_que[i].inst_id = -1;
    id_que[i].dat_idx = 0;
  }
  /*-------------------------------------------------------------
  initialize receiving state
  -------------------------------------------------------------*/
  rx_state = RXS_ETX;
  /*-------------------------------------------------------------
  initialize statistic information
  -------------------------------------------------------------*/
  u4SyncPkt = 0;
  u4OverflowPkt = 0;
  u4PktInQueue = 0;
```

```
  return TRUE;
}
/*****************************************************************
* PROCEDURE NAME:
* iop_inst_avail - Get available NMEA sentence information
*
* DESCRIPTION:
* inst_id - NMEA sentence type
* dat_idx - Start data index in queue
* dat_siz - NMEA sentence size
*****************************************************************/
BOOL iop_inst_avail(short *inst_id, short *dat_idx,
short *dat_siz)
{
  /*-------------------------------------------------------
  variables
  -------------------------------------------------------*/
  BOOL inst_avail;
  /*-------------------------------------------------------
  if packet is available then return id and index
  -------------------------------------------------------*/
  if ( id_que_tail != id_que_head )
  {
    *inst_id = id_que[ id_que_tail ].inst_id;
    *dat_idx = id_que[ id_que_tail ].dat_idx;
    *dat_siz = id_que[ id_que_tail ].dat_siz;
    id_que[ id_que_tail ].inst_id = -1;
    id_que_tail = ++id_que_tail & (unsigned short)(NMEA_ID_QUE_SIZE - 1);
    inst_avail = TRUE;
    if (u4PktInQueue > 0)
    {
      u4PktInQueue--;
    }
  }
  else
  {
    inst_avail = FALSE;
  }
  return ( inst_avail );
} /* iop_inst_avail() end */
/*****************************************************************
* PROCEDURE NAME:
* iop_get_inst - Get available NMEA sentence from queue
*
```

```
* DESCRIPTION:
* idx - Start data index in queue
* size - NMEA sentence size
* data - Data buffer used to save NMEA sentence
*****************************************************************/
void iop_get_inst(short idx, short size, void *data)
{
  /*-----------------------------------------------------------
  variables
  -----------------------------------------------------------*/
  short i;
  unsigned char *ptr;
  /*-----------------------------------------------------------
  copy data from the receive queue to the data buffer
  -----------------------------------------------------------*/
  ptr = (unsigned char *)data;
  for (i = 0; i < size; i++)
  {
    *ptr = rx_que[idx];
    ptr++;
    idx = ++idx & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
  }
} /* iop_get_inst() end */
/*****************************************************************
* PROCEDURE NAME:
* iop_pcrx_nmea - Receive NMEA code
*
* DESCRIPTION:
* The procedure fetches the characters between '$' and <CR> (including '$' and <CR>).
* I.e., <CR> and <LF> characters are skipped.
* And the maximum size of the sentence fetched by this procedure is 256 bytes.
* $xxxxxx*AA
*
*****************************************************************/
void iop_pcrx_nmea( unsigned char data )
{
  /*-----------------------------------------------------------
  determine the receiving state
  -----------------------------------------------------------*/
  if (data == IOP_LF_DATA){
  return;
  }
  switch (rx_state)
  {
```

```
case RXS_DAT:
  switch (data)
  {
  case IOP_CR_DATA:
    //Count total number of sync packets
    u4SyncPkt += 1;
    id_que_head = ++id_que_head & (unsigned short)(NMEA_ID_QUE_SIZE - 1);
    if (id_que_tail == id_que_head)
    {
      //Count total number of overflow packets
      u4OverflowPkt += 1;
      id_que_tail = ++id_que_tail & (unsigned short)(NMEA_ID_QUE_SIZE - 1);
    }
    else
    {
      u4PktInQueue++;
    }
    rx_state = RXS_ETX;
    /*-------------------------------------------------------------
    set RxEvent signaled
    -------------------------------------------------------------*/
    SetEvent(hRxEvent);
    break;
  case IOP_START_NMEA:
  {
    //Restart NMEA sentence collection
    rx_state = RXS_DAT;
    id_que[id_que_head].inst_id = 1;
    id_que[id_que_head].dat_idx = rx_que_head;
    id_que[id_que_head].dat_siz = 0;
    rx_que[rx_que_head] = data;
    rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
    id_que[id_que_head].dat_siz++;
    break;
  }
  default:
    rx_que[rx_que_head] = data;
    rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
    id_que[id_que_head].dat_siz++;
    //If NMEA sentence length > 256 bytes, stop NMEA sentence collection.
    if (id_que[id_que_head].dat_siz == MAX_NMEA_STN_LEN)
    {
      id_que[id_que_head].inst_id = -1;
      rx_state = RXS_ETX;
```

```
      }
      break;
    }
    break;
  case RXS_ETX:
    if (data == IOP_START_NMEA)
    {
      rx_state = RXS_DAT;
      id_que[id_que_head].inst_id = 1;
      id_que[id_que_head].dat_idx = rx_que_head;
      id_que[id_que_head].dat_siz = 0;
      rx_que[rx_que_head] = data;
      rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
      id_que[id_que_head].dat_siz++;
    }
    break;
  default:
    rx_state = RXS_ETX;
    break;
  }
} /* iop_pcrx_nmea() end */
/*************************************************************************
* PROCEDURE NAME:
* void iop_pcrx_nmea_dbg_hbd_bytes(unsigned char aData[], int i4NumByte)
* Receive NMEA and debug log code
*
* DESCRIPTION:
* The procedure fetches the characters between '$' and <CR> (including '$' and <CR>).
* I.e., characters <CR> and <LF> are skipped.
* And the maximum size of the sentence fetched by this procedure is 256 bytes.
* $xxxxxx*AA
*
*************************************************************************/
void iop_pcrx_nmea_dbg_hbd_bytes(unsigned char aData[], int i4NumByte)
{
    int i;
    unsigned char data;
  for (i = 0; i < i4NumByte; i++)
  {
    data = aData[i];
    if (data == IOP_LF_DATA){
    continue;
    }
    /*-----------------------------------------------------------
```

```
  determine the receiving state
  ------------------------------------------------------------*/
  switch (rx_state)
  {
    case RXS_DAT:
      switch (data)
      {
      case IOP_CR_DATA:
        //Count total number of sync packets
        u4SyncPkt += 1;
        id_que_head = ++id_que_head & (unsigned short)(NMEA_ID_QUE_SIZE - 1);
        if (id_que_tail == id_que_head)
        {
          //Count total number of overflow packets
          u4OverflowPkt += 1;
          id_que_tail = ++id_que_tail & (unsigned short)(NMEA_ID_QUE_SIZE - 1);
        }
        else
        {
          u4PktInQueue++;
        }
        rx_state = RXS_ETX;
        /*-------------------------------------------------------------
        set RxEvent signaled
        -------------------------------------------------------------*/
        SetEvent(hRxEvent);
        break;
      case IOP_START_NMEA:
      {

        //Restart NMEA sentence collection
        rx_state = RXS_DAT;
        id_que[id_que_head].inst_id = 1;
        id_que[id_que_head].dat_idx = rx_que_head;
        id_que[id_que_head].dat_siz = 0;
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        break;
      }

      case IOP_START_DBG:
      {
        //Restart DBG sentence collection
```

```
        rx_state = RXS_DAT_DBG;
        id_que[id_que_head].inst_id = 2;
        id_que[id_que_head].dat_idx = rx_que_head;
        id_que[id_que_head].dat_siz = 0;
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        break;
      }
      default:
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        //If NMEA sentence bytes > 256, stop NMEA sentence collection.
        if (id_que[id_que_head].dat_siz == MAX_NMEA_STN_LEN)
        {
        id_que[id_que_head].inst_id = -1;
        rx_state = RXS_ETX;
        }
        break;
      }
      break;
    case RXS_DAT_DBG:
      switch (data)
      {
        case IOP_CR_DATA:
          //Count total number of sync packets
          u4SyncPkt += 1;
          id_que_head = ++id_que_head & (unsigned short)(NMEA_ID_QUE_SIZE - 1)
          if (id_que_tail == id_que_head)
          {
          //Count total number of overflow packets
          u4OverflowPkt += 1;
          id_que_tail = ++id_que_tail & (unsigned short)(NMEA_ID_QUE_SIZE - 1);
          }
          else
          {
          u4PktInQueue++;
          }
          rx_state = RXS_ETX;
          /*-----------------------------------------------------------
          set RxEvent signaled
          -----------------------------------------------------------*/
          SetEvent(hRxEvent);
```

```
        break;
      case IOP_START_NMEA:
      {

        //Restart NMEA sentence collection
        rx_state = RXS_DAT;
        id_que[id_que_head].inst_id = 1;
        id_que[id_que_head].dat_idx = rx_que_head;
        id_que[id_que_head].dat_siz = 0;
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        break;
      }

      case IOP_START_DBG:
      {
        //Restart DBG sentence collection
        rx_state = RXS_DAT_DBG;
        id_que[id_que_head].inst_id = 2;
        id_que[id_que_head].dat_idx = rx_que_head;
        id_que[id_que_head].dat_siz = 0;
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        break;
      }
      default:
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        //If NMEA sentence length > 256 bytes, stop NMEA sentence collection.
        if (id_que[id_que_head].dat_siz == MAX_NMEA_STN_LEN)
        {
        id_que[id_que_head].inst_id = -1;
        rx_state = RXS_ETX;
        }
        break;
    }
    break;
  case RXS_DAT_HBD:
    switch (data)
    {
      case IOP_CR_DATA:
```

```
        //Count total number of sync packets
        u4SyncPkt += 1;
        id_que_head = ++id_que_head & (unsigned short)(NMEA_ID_QUE_SIZE - 1);
        if (id_que_tail == id_que_head)
        {
        //Count total number of overflow packets
        u4OverflowPkt += 1;
        id_que_tail = ++id_que_tail & (unsigned short)(NMEA_ID_QUE_SIZE - 1);
        }
        else
        {
            u4PktInQueue++;
        }
        rx_state = RXS_ETX;
        /*-------------------------------------------------------------
        set RxEvent signaled
        -------------------------------------------------------------*/
        SetEvent(hRxEvent);
        break;
      case IOP_START_NMEA:
      {
        //Restart NMEA sentence collection
        rx_state = RXS_DAT;
        id_que[id_que_head].inst_id = 1;
        id_que[id_que_head].dat_idx = rx_que_head;
        id_que[id_que_head].dat_siz = 0;
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        break;
      }
      case IOP_START_DBG:
      {
        //Restart DBG sentence collection
        rx_state = RXS_DAT_DBG;
        id_que[id_que_head].inst_id = 2;

        id_que[id_que_head].dat_idx = rx_que_head;
        id_que[id_que_head].dat_siz = 0;
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        break;
      }
```

```
      default:
        rx_que[rx_que_head] = data;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        //If NMEA sentence bytes > 256, stop NMEA sentence collection.
        if (id_que[id_que_head].dat_siz == MAX_NMEA_STN_LEN)
        {
        id_que[id_que_head].inst_id = -1;
        rx_state = RXS_ETX;
        }
        break;
    }
    break;
  case RXS_ETX:
    if (data == IOP_START_NMEA)
    {
      rx_state = RXS_DAT;
      id_que[id_que_head].inst_id = 1;
      id_que[id_que_head].dat_idx = rx_que_head;
      id_que[id_que_head].dat_siz = 0;
      rx_que[rx_que_head] = data;
      rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
      id_que[id_que_head].dat_siz++;
    }
    else if (data == IOP_START_DBG)
    {
      rx_state = RXS_DAT_DBG;
      id_que[id_que_head].inst_id = 2;
      id_que[id_que_head].dat_idx = rx_que_head;
      id_que[id_que_head].dat_siz = 0;
      rx_que[rx_que_head] = data;
      rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
      id_que[id_que_head].dat_siz++;
    }
    else if (data == IOP_START_HBD1)
    {
      rx_state = RXS_PRM_HBD2;
    }
    break;
  case RXS_PRM_HBD2:
    if (data == IOP_START_HBD2)
    {
      rx_state = RXS_PRM_HBD3;
```

```
      }
      else
      {
      rx_state = RXS_ETX;
      }
      break;
    case RXS_PRM_HBD3:
      if (data == IOP_START_HBD3)
      {
        rx_state = RXS_DAT_HBD;
        //Start to collect the packet
        id_que[id_que_head].inst_id = 3;
        id_que[id_que_head].dat_idx = rx_que_head;
        id_que[id_que_head].dat_siz = 0;
        rx_que[rx_que_head] = IOP_START_HBD1;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        rx_que[rx_que_head] = IOP_START_HBD2;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
        rx_que[rx_que_head] = IOP_START_HBD3;
        rx_que_head = ++rx_que_head & (unsigned short)(NMEA_RX_QUE_SIZE - 1);
        id_que[id_que_head].dat_siz++;
      }
      else
      {
        rx_state = RXS_ETX;
      }
      break;
    default:
      rx_state = RXS_ETX;
      break;
   }
  }
} /* iop_pcrx_nmea_dbg_hbd_bytes() end */
```

6 Appendix References

Table 2: Related Document

Document Name
[1] Quectel_Lx0&Lx6L&LC86L&LG77L_GNSS_Protocol_Specification

Table 3: Terms and Abbreviations

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