L76-L I2C Application Note

GNSS Module Series

Rev. L76-L_I2C_Application_Note_V1.0

Date: 2016-09-14

Our aim is to provide customers with timely and comprehensive service. For any assistance, please contact our company headquarters:

Quectel Wireless Solutions Co., Ltd.

Office 501, Building 13, No.99, Tianzhou Road, Shanghai, China, 200233

Tel: +86 21 5108 6236

Email: info@quectel.com

Or our local office. For more information, please visit:

http://www.quectel.com/support/salesupport.aspx

For technical support, or to report documentation errors, please visit:

http://www.quectel.com/support/techsupport.aspx

Or email to: Support@quectel.com

About the Document

History

Revision	Date	Author	Description
1.0	2016-09-14	Simon HU	Initial

Contents

Table Index

Figure Index

1 Introduction

This document introduces the I2C function of Quectel L76-L module. It provides detailed introduction on how the master receives/parses NMEA sentence and sends SDK command via I2C bus. To assist in I2C buffer reading/writing as well as NMEA sentence receiving/parsing, the document provides detailed flow charts and sample codes for your reference.

2 NMEA Output Through I2C Bus

Quectel L76-L provides a set of I2C interface which outputs NMEA data by default while reading. This chapter provides detailed introductions on how the master reads and parses NMEA data packet through the I2C bus.

In this document, client-side MCU works as the host processor (master), and Quectel L76-L module works as the slave.

2.1. I2C Features

● Supports fast mode, with bit rate up to 400kbit/s
● Supports 7-bit address
● Works in slave mode
● Default slave address values are: Write: 0x20, Read: 0x21
● I2C pins: I2C_SDA and I2C_SCL

2.2. Flow of Host Processor (Master) I2C Receiving NMEA

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

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

Presently, in L76-L module, the NMEA packet can be read via I2C in polling mode only. In the mode, the master can read entire NMEA packet of one second in each polling time interval. The time interval can be configured according to the GNSS fix interval, and it should be less than the GNSS fix interval.

The following figure illustrates how the master reads NMEA packet via I2C in polling mode.
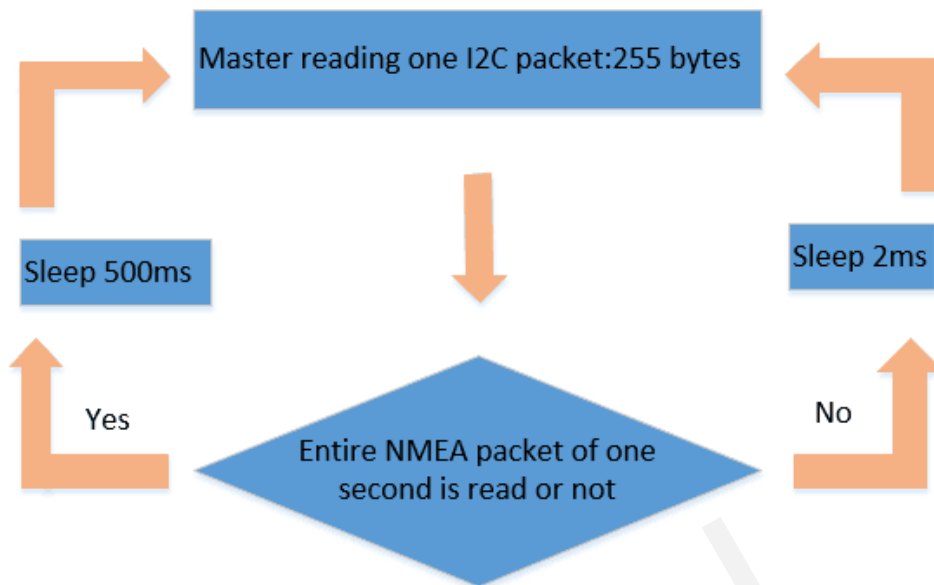
Figure 1: Flow Chart of How the Master Reads NMEA via I2C in Polling Mode

---

NOTE

The figure above assumes that the GNSS fix interval is 1 second, and therefore the polling time interval is set to 500ms.

2.2.1. I2C Data Packet Formats in Slave I2C Buffer

The data packet in slave I2C buffer (I2C data packet) has 254 valid NMEA bytes at most and one end character <LF>, so the master can read maximally 255-byte I2C data packet at a time. The following figure illustrates the construction of I2C data packet.
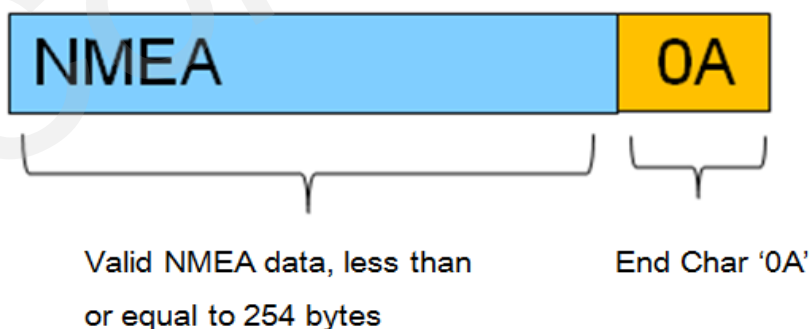
Figure 2: I2C Data Packet Construction

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

Figure 3: Example Illustration for I2C Data Packet Construction

Whether there are NMEA data saved in the slave buffer, the master can read one I2C data packet (255 bytes) from the slave, and there are three types of I2C data packet that the master may read from the slave.

2.2.1.1. The First Type of I2C Data Packet: Valid Data Bytes + Garbage Bytes

When the slave 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 saved in the buffer, then the last byte will be the end character <LF>.
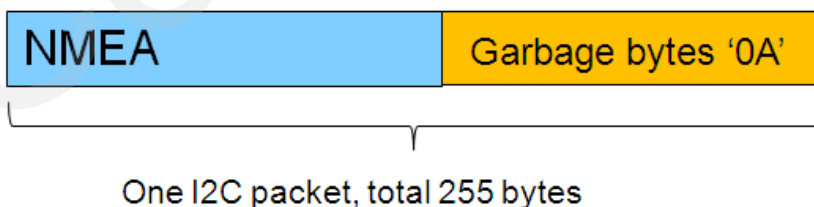
Figure 4: I2C Packet Format (Valid Data Bytes + Garbage Bytes)

For example, if the slave I2C buffer has saved NMEA data of 202 bytes, then 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: I2C Packet Example (Valid Data Bytes + Garbage Bytes)

> **NOTE**
>
> Why garbage bytes are '0A'?
> Because if the slave I2C buffer is empty, the slave (L76-L) will output the last valid byte repeatedly until new data is uploaded into I2C buffer, and "0A" is the last valid byte in the NMEA packet.

2.2.1.2. The Second Type of I2C Packet Format: All Garbage Bytes

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

Figure 6: I2C Packet Format (All Garbage Bytes)

Figure 7: I2C Packet Example (All Garbage Bytes)

2.2.1.3. The Third Type of I2C Packet Format: Garbage Bytes + Valid Data Bytes

If the slave I2C buffer is empty when the master starts reading, but new data is uploaded into the I2C buffer before reading is over, the master will read garbage bytes first and then valid NMEA data bytes.
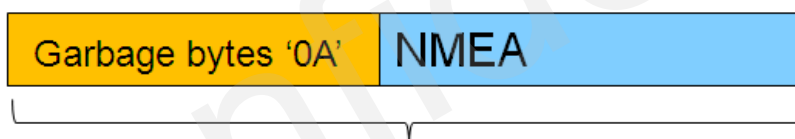
Figure 8: I2C Packet Format (Garbage Bytes + Valid Data Bytes)

Figure 9: I2C Packet Example (Garbage Bytes + Valid Data Bytes)

2.2.2. How to Extract Valid NMEA Data from Many I2C Packets

After the master reads sufficient I2C data packets, it needs to parse and extract valid NMEA data from these packets. Quectel provides sample code for customers to extract the valid data. Please refer to *Chapter 5.2* for details.

NOTE

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

(1) The end character of an I2C packet

(2) Garbage bytes

(3) The end character <LF> of NMEA sentence. When it is discarded, there will be no effect on NMEA sentence parsing.

---

3 SDK Command Sending Through I2C Bus

The master can send SDK commands to the slave via I2C bus. As the slave's I2C RX buffer has maximum capacity of 255 bytes, each SDK command that the master inputs should be less than 255 bytes. The time interval of two input SDK commands cannot be less than 10 milliseconds as the slave needs 10 milliseconds to process the input data.

4 Procedures for Reading and Writing I2C Buffer

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

4.1. Sequence Charts

The sequence charts for reading and writing I2C buffer are shown below.
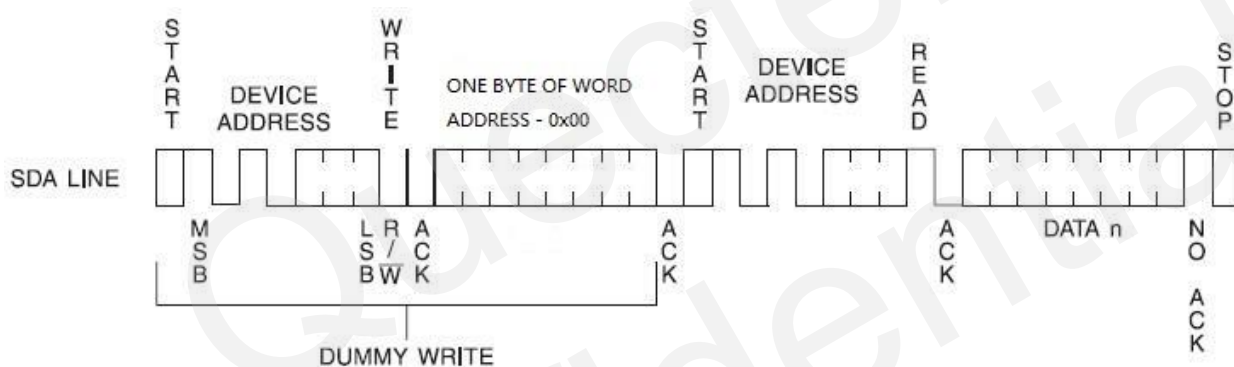
Figure 10: Sequence Chart for Reading I2C Buffer
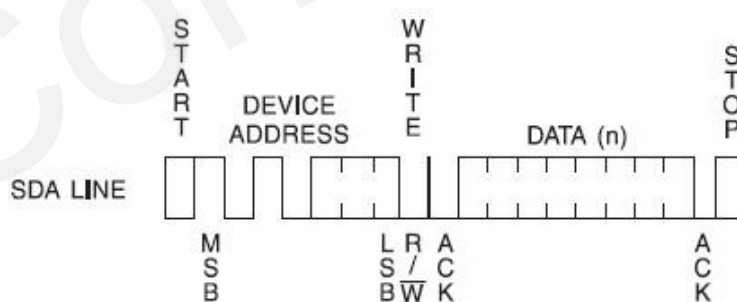
Figure 11: Sequence Chart for Writing I2C Buffer

4.2. Sample Code

The sample code for reading and writing I2C buffer is shown below.

```
#define MAX_I2C_BUF_SIZE   255
char rd_buf[MAX_I2C_BUF_SIZE+1];
#define EE_DEV_ADDR        0x20     //Shift the 7-bit slave address (0x10) 1 bit to the left.
#define I2C_WR  0
#define I2C_RD   1


BOOL I2C_read_bytes(char *buf, uint length)
{
    uint16_t i;
    i2c_Start();
    i2c_SendByte(EE_DEV_ADDR | I2C_WR);
    if (i2c_WaitAck() != 0)
    {
        i2c_Stop();
        return FALSE;
    }

    i2c_SendByte((uint8_t)0x00);
    if (i2c_WaitAck() != 0)
    {
        i2c_Stop();
        return FALSE;
    }

    i2c_Start();
    i2c_SendByte(EE_DEV_ADDR | I2C_RD);

    if (i2c_WaitAck() != 0)
    {
        i2c_Stop();
        return FALSE;
    }

    for (i = 0; i < MAX_I2C_BUF_SIZE; i++)
    {
        buf[i] = i2c_ReadByte();

        if (i != MAX_I2C_BUF_SIZE - 1)
        {
            i2c_Ack();
```

```
        }
        else
        {
            i2c_NAck();
        }
    }

    i2c_Stop();
    return TRUE;
}


BOOL I2C_write_bytes(char *buf, uin16_t length)
{
    uin16_t i=0;
    i2c_Stop();
    i2c_Start();
    i2c_SendByte(EE_DEV_ADDR | I2C_WR);
    if (i2c_WaitAck() != 0)
    {
        //dbg_printf("send I2C dev addr fail!\r\n");
        goto cmd_fail;
    }

    for(i = 0; i < length; i++)
    {
        i2c_SendByte(buf[i]);
        if (i2c_WaitAck() != 0)
        {
            //dbg_printf("send fail at buf[%d]\r\n",i);
            goto cmd_fail;
        }
    }
    i2c_Stop();
    return TRUE;

    cmd_fail:
    i2c_Stop();
    return FALSE;
}
```

5 Procedures for Receiving and Parsing NMEA Sentence

This chapter provides the flow chart and sample code for the master I2C to receive and parse NMEA sentence.

5.1. Flow Chart

The flow chart for receiving and parsing NMEA sentence is shown below.
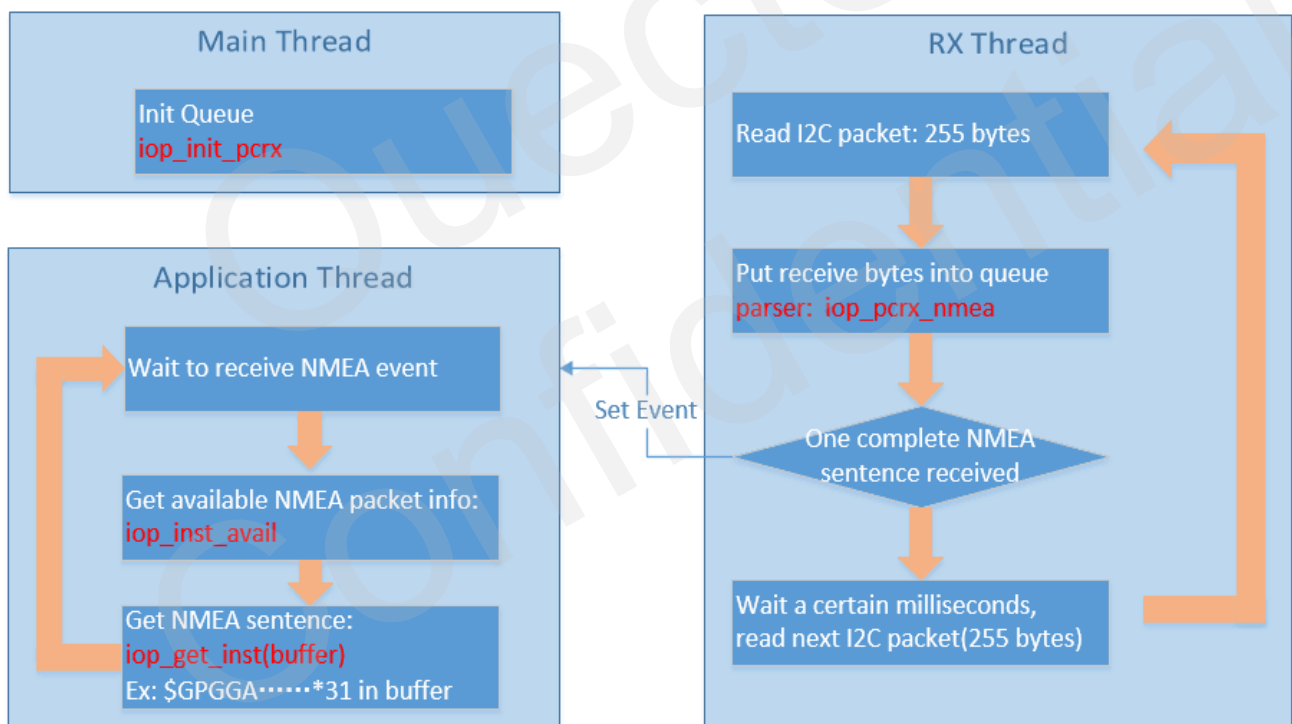
Figure 12: Flow Chart for Receiving and Parsing NMEA Sentence

5.2. Sample Code

After NMEA sentence is received from the slave, the master will extract NMEA and debug data from many I2C packets. It will also discard garbage bytes and extract valid data automatically.

Table 1: Function Description

Function Name	Description
iop_init_pcrx	Initialize receive queue
iop_inst_avail	Get available NMEA sentence information.
iop_get_inst	Get NMEA sentence data from queue buffer.
iop_pcrx_nmea	Process I2C packets, get valid NMEA data and discard garbage bytes.
iop_pcrx_nmea_dbg_hbd_bytes	Process I2C packets, get valid NMEA data&debug log code, and discard garbage bytes.

```
#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>).
* That is, characters <CR> and <LF> are skipped.
* And the maximum size of the sentence fetched by this procedure is 256.
* $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, 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 fetch the characters between '$' and <CR> (including '$' and <CR>).
* That is, characters <CR> and <LF> are skipped.
* And the maximum size of the sentence fetched by this procedure is 256.
* $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, 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 A Reference

Table 2: Terms and Abbreviations

Abbreviation	Description
GNSS	Global Navigation Satellite System
I2C	Inter-Integrated Circuit
MCU	Main Computational Unit
NMEA	National Marine Electronics Association
SDK	Software Development Kit