GNSS I2C

Application Note

GNSS Module Series

Rev. GNSS_I2C_Application_Note_V1.0

Date: 2016-08-02

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

GENERAL NOTES

QUECTEL OFFERS THE INFORMATION AS A SERVICE TO ITS CUSTOMERS. THE INFORMATION PROVIDED IS BASED UPON CUSTOMERS' REQUIREMENTS. QUECTEL MAKES EVERY EFFORT TO ENSURE THE QUALITY OF THE INFORMATION IT MAKES AVAILABLE. QUECTEL DOES NOT MAKE ANY WARRANTY AS TO THE INFORMATION CONTAINED HEREIN, AND DOES NOT ACCEPT ANY LIABILITY FOR ANY INJURY, LOSS OR DAMAGE OF ANY KIND INCURRED BY USE OF OR RELIANCE UPON THE INFORMATION. ALL INFORMATION SUPPLIED HEREIN IS SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.

COPYRIGHT

About the Document

History

Revision	Date	Author	Description
1.0	2016-08-02	Simon HU	Initial

Contents

About the Document 2
Contents 3
Table Index 4
Figure Index 5

1 Introduction 6

2 GNSS Outputs NMEA Through I2C Bus 7
- 2.1 GNSS I2C Specification 7
- 2.2 Host Processor (I2C-master) Receives NMEA Flow 7
  - 2.2.1 I2C Data Packet Format in Slave Buffer 8
  - 2.2.2 Three Types of I2C Packet That Master Read from Slave 9
  - 2.2.3 How to Extract Valid NMEA Data from Many I2C Packets 12

3 Input SDK Command Through I2C Bus (GNSS) 13

4 Sequence Chart and Sample Code for Reading and Writing I2C Buffer 14
- 4.1 Sequence Charts for Reading and Writing I2C Buffer 14
- 4.2 Sample Code for Reading and Writing I2C Buffer 15

5 Receive and Parse NMEA Sentence 17
- 5.1 Flow of Receiving and Parsing NMEA Sentence 17
- 5.2 Sample Code for Receiving and Parsing NMEA Sentence 18

Table Index

TABLE 1: FUNCTION DESCRIPTION 18

Figure Index

FIGURE 1: POLLING MODE MASTER READING FLOW 8
FIGURE 2: PACKET FORMAT IN SLAVE I2C BUFFER 8
FIGURE 3: 254 VALID NMEA BYTES AND 1 END CHAR <LF> IN SLAVE I2C BUFFER 9
FIGURE 4: I2C PACKET FORMAT OF FIRST TYPE 9
FIGURE 5: EXAMPLE OF I2C PACKET FORMAT OF FIRST TYPE 10
FIGURE 6: I2C PACKET FORMAT OF SECOND TYPE 10
FIGURE 7: EXAMPLE OF I2C PACKET FORMAT OF SECOND TYPE 11
FIGURE 8: I2C PACKET FORMAT OF THIRD TYPE 11
FIGURE 9: EXAMPLE OF I2C PACKET FORMAT OF THIRD TYPE.. 12
FIGURE 10: SEQUENCE CHART FOR READING I2C BUFFER 14
FIGURE 11: SEQUENCE CHART FOR WRITING I2C BUFFER 14
FIGURE 12: RECEIVING AND PARSING FLOW OF NMEA SENTENCE 17

1 Introduction

This document describes how to receive NMEA data and input PQ command through I2C bus on GNSS module, and gives detailed reading flow. Quectel also gives sample code for reference.

If you are interested in I2C reading flow and writing flow, please refer to ***Chapter 2 & 3***. If you are only interested in how to port sample code and get NMEA data from GNSS module, please refer to ***Chapter 4 & 5***.

2 GNSS Outputs NMEA Through I2C Bus

2.1 GNSS I2C Specification

- GNSS supports fast mode, bit rate up to 400kbit/s.
- GNSS supports 7-bit address.
- GNSS works on slave mode.
- GNSS default slave address is 0x10.
- GNSS I2C pins: I2C_SDA and I2C_SCL.

2.2 Host Processor (I2C-master) Receives NMEA Flow

(1) The capacity of buffer for GNSS I2C TX is 255 bytes, and the master can read one I2C data packet for maximum 255 bytes at a time. The master needs to read several I2C data packets and extract valid NMEA data from them in order to read entire NMEA packet of one second.

(2) After reading one I2C data packet, the master will then sleep for 2ms before it starts to receive next I2C data packet because GNSS needs 2ms to upload new I2C data into slave I2C buffer. If entire NMEA packet of one second is read, the master can sleep more time to wait entire NMEA packet of next second being ready.

Quectel only supports polling mode for reading NMEA through I2C now.

In polling mode, master can read entire NMEA packet of one second in each polling time interval. The time interval can be configured according to GPS fix interval, and it should be less than GPS fix interval.

The following figure shows the pollingmode master reading flow.

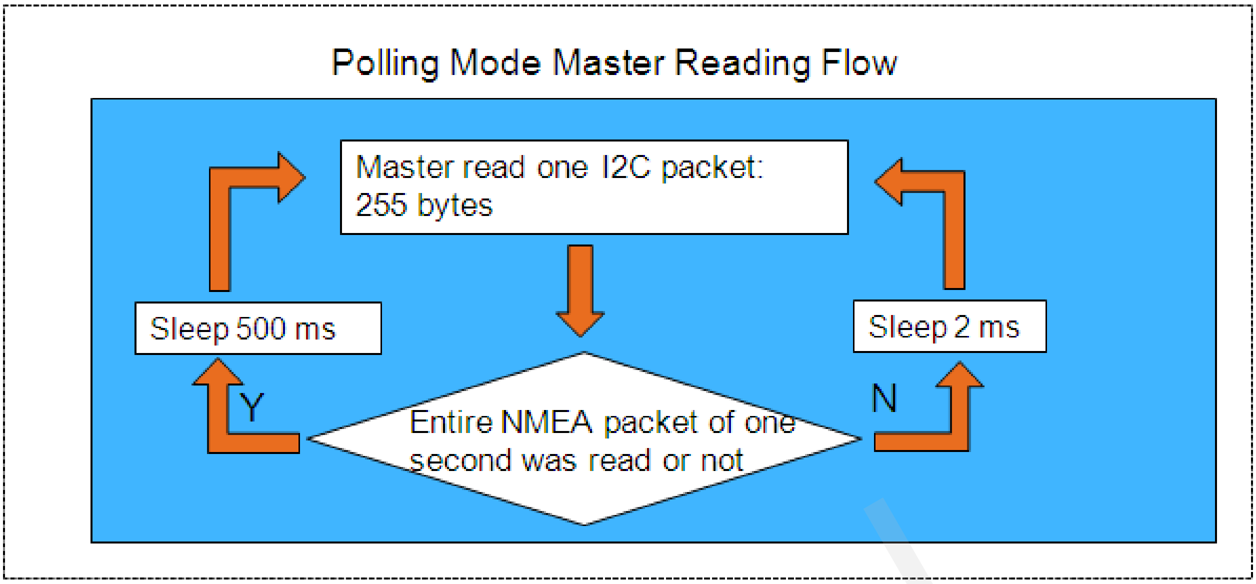

Polling Mode Master Reading Flow
Master read one I2C packet: 255 bytes
Sleep 500 ms
Sleep 2 ms
Y
N
Entire NMEA packet of one second was read or not

Figure 1: Polling Mode Master Reading Flow

NOTE

The figure above assumes that GPS fix interval is 1 second; therefore the polling time interval is set to 500ms.

2.2.1 I2C Data Packet Format in Slave Buffer

I2C data packet in slave buffer has 254 valid NMEA bytes at most and one end char <LF>, so master must can read maximum 255-byte I2C data packet at one time. When the slave buffer is empty, the slave will keep providing I2C data packet (255 bytes) for the master to read; however the content in the packet could be garbage bytes because information in the data is nothing new. The garbage bytes will be explained in the following sub-chapter.

Packet format in slave I2C buffer is as following figure:

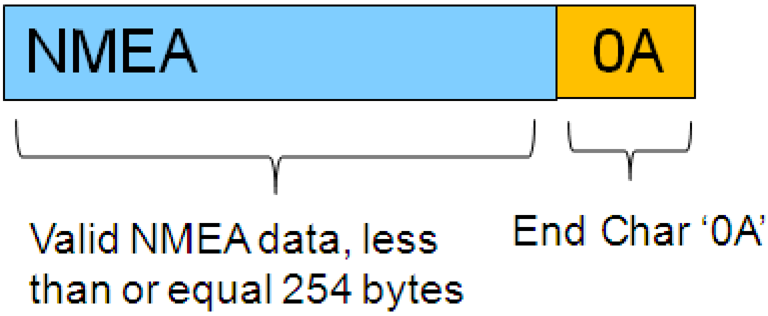
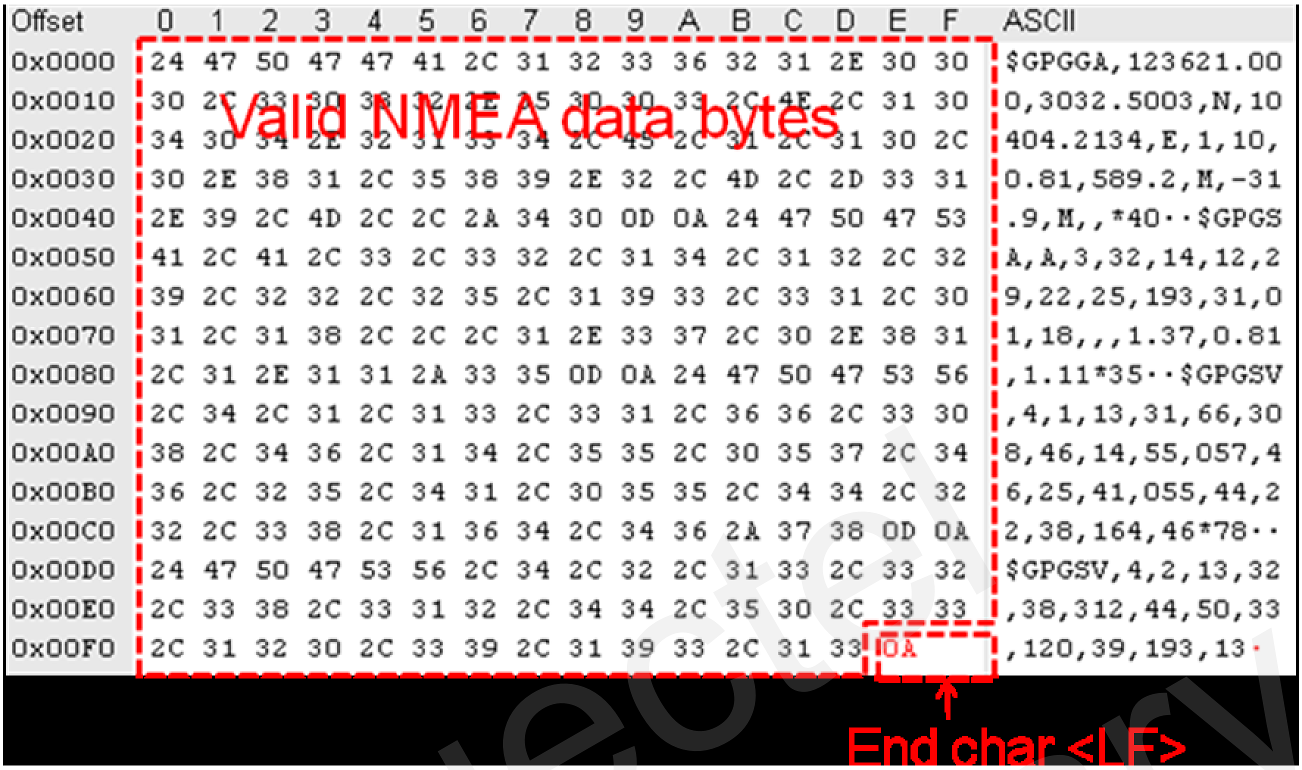


Figure 2: Packet Format in Slave I2C Buffer

There are 254 valid NMEA bytes and 1 end char <LF> in slave I2C buffer as following figure.



```
Offset  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   ASCII
0x0000 24 47 50 47 47 41 2C 31 32 33 36 32 31 2E 30 30  $GPGGA,123621.00
0x0010 30 2C 33 30 33 32 2E 35 30 30 33 2C 4E 2C 31 30  0,3032.5003,N,10
0x0020 34 30 34 2E 32 31 33 34 2C 45 2C 31 2C 31 30 2C  404.2134,E,1,10,
0x0030 30 2E 38 31 2C 35 38 39 2E 32 2C 4D 2C 2D 33 31  0.81,589.2,M,-31
0x0040 2E 39 2C 4D 2C 2C 2A 34 30 0D 0A 24 47 50 47 53  .9,M,,*40··$GPGS
0x0050 41 2C 41 2C 33 2C 33 32 2C 31 34 2C 31 32 2C 32  A,A,3,32,14,12,2
0x0060 39 2C 32 32 2C 32 35 2C 31 39 33 2C 33 31 2C 30  9,22,25,193,31,0
0x0070 31 2C 31 38 2C 2C 2C 31 2E 33 37 2C 30 2E 38 31  1,18,,,1.37,0.81
0x0080 2C 31 2E 31 31 2A 33 35 0D 0A 24 47 50 47 53 56  ,1.11*35··$GPGSV
0x0090 2C 34 2C 31 2C 31 33 2C 33 31 2C 36 36 2C 33 30  ,4,1,13,31,66,30
0x00A0 38 2C 34 36 2C 31 34 2C 35 35 2C 30 35 37 2C 34  8,46,14,55,057,4
0x00B0 36 2C 32 35 2C 34 31 2C 30 35 35 2C 34 34 2C 32  6,25,41,055,44,2
0x00C0 32 2C 33 38 2C 31 36 34 2C 34 36 2A 37 38 0D 0A  2,38,164,46*78··
0x00D0 24 47 50 47 53 56 2C 34 2C 32 2C 31 33 2C 33 32  $GPGSV,4,2,13,32
0x00E0 2C 33 38 2C 33 31 32 2C 34 34 2C 35 30 2C 33 33  ,38,312,44,50,33
0x00F0 2C 31 32 30 2C 33 39 2C 31 39 33 2C 31 33 0A     ,120,39,193,13·
```

Figure 3: 254 Valid NMEA Bytes and 1 End Char <LF> in Slave I2C Buffer

2.2.2 Three Types of I2C Packet That Master Read from Slave

1. When the slave buffer has already had some data stored, the master will read one I2C packet (255 bytes) from the slave, including some valid data in the header and some garbage bytes in the end of a packet.

I2C packet format of first type is as following figure:

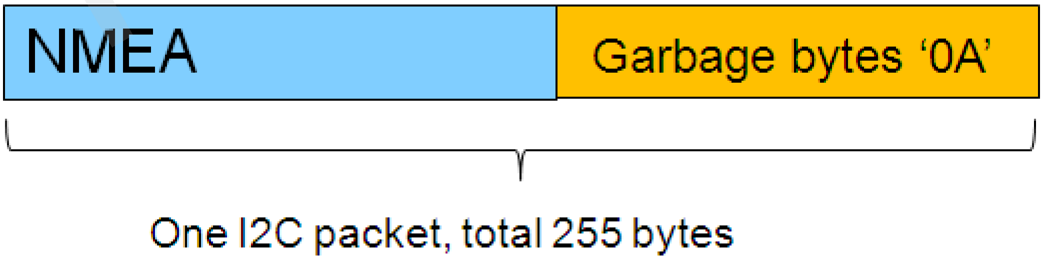
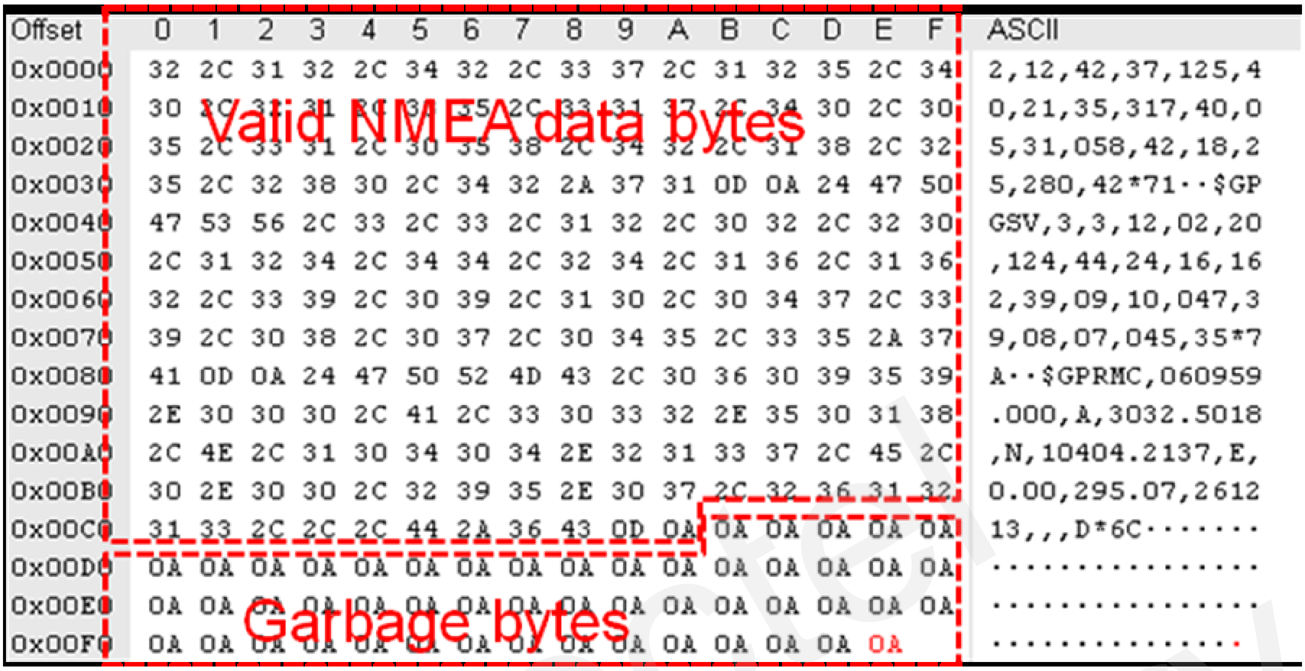


Figure 4: I2C Packet Format of First Type

If slave I2C buffer has 202 bytes NMEA data, the master will read one I2C packet (255 bytes); the packet format that master read from slave is as following figure.



```
Offset   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   ASCII
0x0000  32 2C 31 32 2C 34 32 2C 33 37 2C 31 32 35 2C 34   2,12,42,37,125,4
0x0010  30 2C 32 31 2C 33 35 2C 33 31 37 2C 34 30 2C 30   0,21,35,317,40,0
0x0020  35 2C 33 31 2C 30 35 38 2C 34 32 2C 31 38 2C 32   5,31,058,42,18,2
0x0030  35 2C 32 38 30 2C 34 32 2A 37 31 0D 0A 24 47 50   5,280,42*71··$GP
0x0040  47 53 56 2C 33 2C 33 2C 31 32 2C 30 32 2C 32 30   GSV,3,3,12,02,20
0x0050  2C 31 32 34 2C 34 34 2C 32 34 2C 31 36 2C 31 36   ,124,44,24,16,16
0x0060  32 2C 33 39 2C 30 39 2C 31 30 2C 30 34 37 2C 33   2,39,09,10,047,3
0x0070  39 2C 30 38 2C 30 37 2C 30 34 35 2C 33 35 2A 37   9,08,07,045,35*7
0x0080  41 0D 0A 24 47 50 52 4D 43 2C 30 36 30 39 35 39   A··$GPRMC,060959
0x0090  2E 30 30 30 2C 41 2C 33 30 33 32 2E 35 30 31 38   .000,A,3032.5018
0x00A0  2C 4E 2C 31 30 34 30 34 2E 32 31 33 37 2C 45 2C   ,N,10404.2137,E,
0x00B0  30 2E 30 30 2C 32 39 35 2E 30 37 2C 32 36 31 32   0.00,295.07,2612
0x00C0  31 33 2C 2C 2C 44 2A 36 43 0D 0A 0A 0A 0A 0A 0A   13,,,D*6C·······
0x00D0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A   ················
0x00E0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A   ················
0x00F0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A      ···············
```

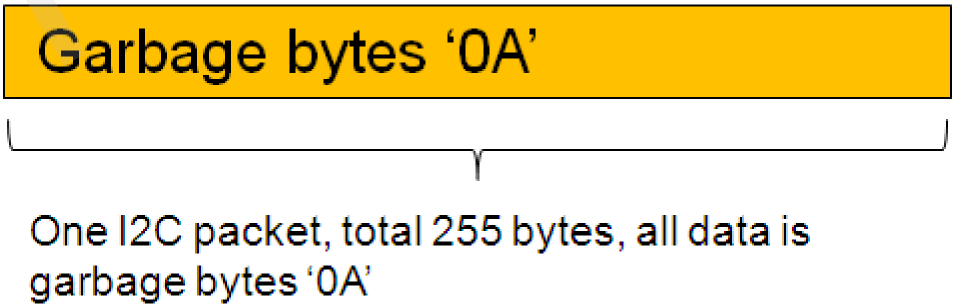
Figure 5: Example of I2C Packet Format of First Type

NOTE

Why Garbage byte is ‘0A’: Because if GNSS I2C buffer is empty, GNSS will output the last valid byte repeatedly until new data is uploaded into I2C buffer, “0A” is the last valid byte in the last NMEA packet.

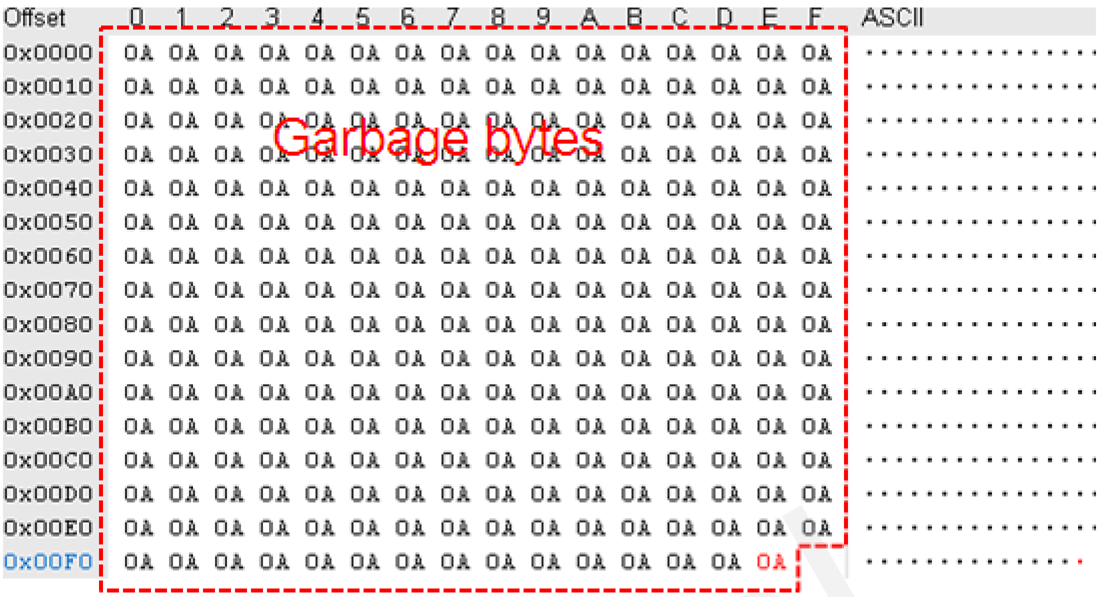
2. When the slave buffer is empty, the master will read one I2C packet (255 bytes) from slave. All data in packet are garbage bytes.

I2C packet format of second type is as following figure:



Garbage bytes ‘0A’
One I2C packet, total 255 bytes, all data is garbage bytes ‘0A’

Figure 6: I2C Packet Format of Second Type



```
Offset   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   ASCII
0x0000  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0010  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0020  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0030  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0040  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0050  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0060  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0070  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0080  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x0090  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x00A0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x00B0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x00C0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x00D0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x00E0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A  ................
0x00F0  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A     ...............
```

Figure 7: Example of I2C Packet Format of Second Type

3. If GNSS I2C buffer is empty, the master starts to read one I2C packet (will read garbage bytes in the beginning). When this reading procedure is not over, GNSS uploads new data into I2C buffer, and master will read valid NMEA data bytes at this time.

I2C packet format of third type is as following figure:

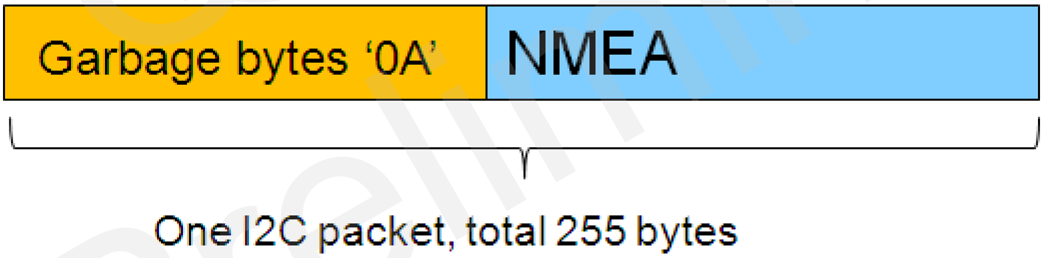
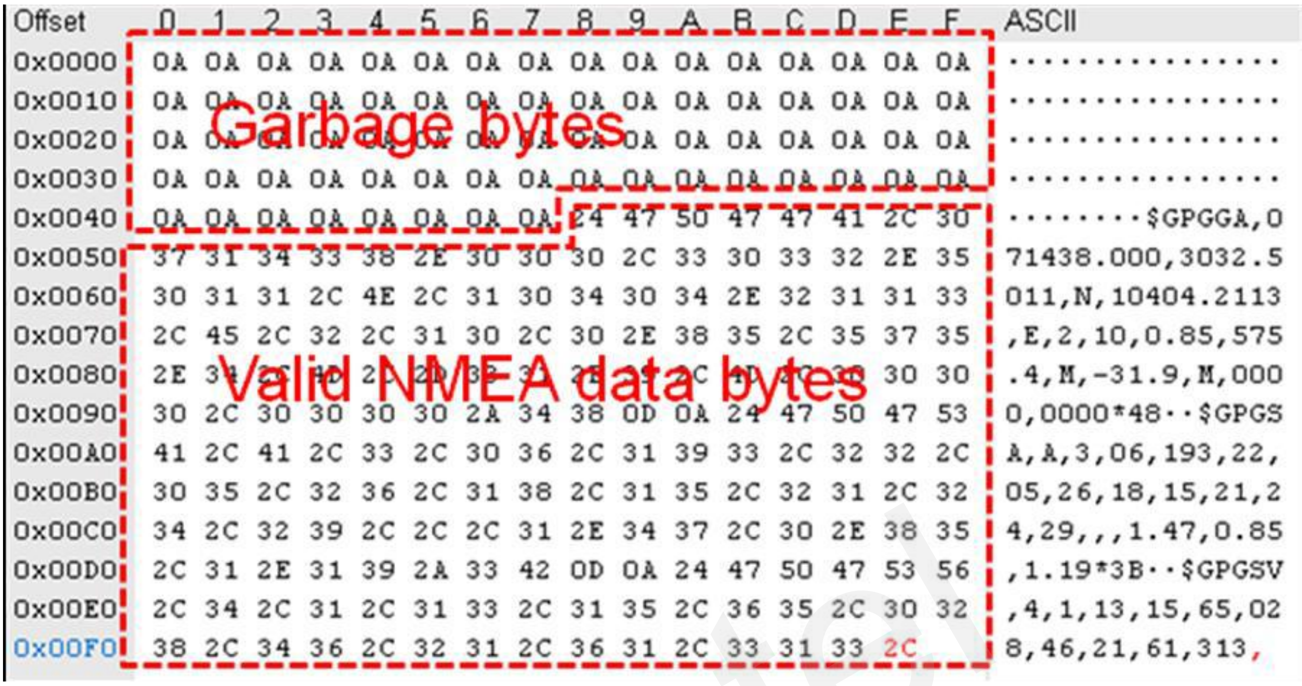


Figure 8: I2C Packet Format of Third Type



```
Offset  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  ASCII
0x0000  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A ................
0x0010  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A ................
0x0020  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A ................
0x0030  0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A ................
0x0040  0A 0A 0A 0A 0A 0A 0A 0A 24 47 50 47 47 41 2C 30 ........$GPGGA,0
0x0050  37 31 34 33 38 2E 30 30 30 2C 33 30 33 32 2E 35 71438.000,3032.5
0x0060  30 31 31 2C 4E 2C 31 30 34 30 34 2E 32 31 31 33 011,N,10404.2113
0x0070  2C 45 2C 32 2C 31 30 2C 30 2E 38 35 2C 35 37 35 ,E,2,10,0.85,575
0x0080  2E 34 2C 4D 2C 2D 33 31 2E 39 2C 4D 2C 30 30 30 .4,M,-31.9,M,000
0x0090  30 2C 30 30 30 30 2A 34 38 0D 0A 24 47 50 47 53 0,0000*48··$GPGS
0x00A0  41 2C 41 2C 33 2C 30 36 2C 31 39 33 2C 32 32 2C A,A,3,06,193,22,
0x00B0  30 35 2C 32 36 2C 31 38 2C 31 35 2C 32 31 2C 32 05,26,18,15,21,2
0x00C0  34 2C 32 39 2C 2C 2C 31 2E 34 37 2C 30 2E 38 35 4,29,,,1.47,0.85
0x00D0  2C 31 2E 31 39 2A 33 42 0D 0A 24 47 50 47 53 56 ,1.19*3B··$GPGSV
0x00E0  2C 34 2C 31 2C 31 33 2C 31 35 2C 36 35 2C 30 32 ,4,1,13,15,65,02
0x00F0  38 2C 34 36 2C 32 31 2C 36 31 2C 33 31 33 2C    8,46,21,61,313,
```

Figure 9: Example of I2C Packet Format of Third Type

2.2.3 How to Extract Valid NMEA Data from Many I2C Packets

As described in chapter above, valid NMEA data need to be extracted from many I2C packets, and sample code will be provided for valid NMEA data. It will be introduced in next chapter.

Note: When extracting NMEA data from I2C packets, all ‘0A’ Characters should be discarded. An I2C packet comes in 3 formats: (1) ‘0A’ is allocated in the end char in an I2C packet; (2) Garbage bytes (‘0A’ under normal circumstances); (3) The <LF> char ‘0A’ in NMEA sentence. Discarding ‘0A’ doesn’t affect parsing NMEA sentence.

3 Input SDK Command Through I2C Bus (GNSS)

Customer can input SDK command through I2C bus since the capacity of GNSS I2C RX buffer is 255 bytes. For one I2C packet that the slave & master transmits, the size should be less than 255 bytes, and the time interval of two input I2C packet inputs cannot be less than 10 milliseconds because the slave needs 10 milliseconds to process input data.

4 Sequence Chart and Sample Code for Reading and Writing I2C Buffer

4.1 Sequence Charts for Reading and Writing I2C Buffer

The sequence charts for reading and writing I2C buffer are shown below.

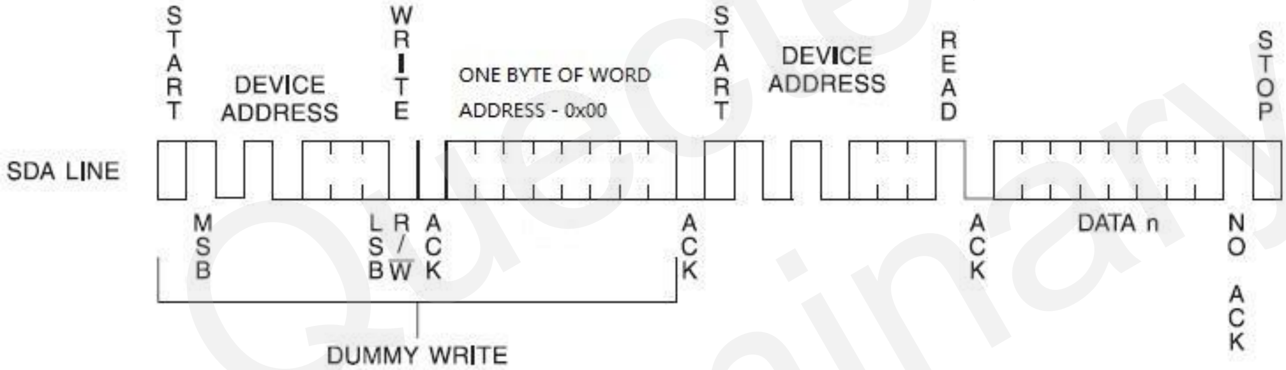


Figure 10: Sequence Chart for Reading I2C Buffer

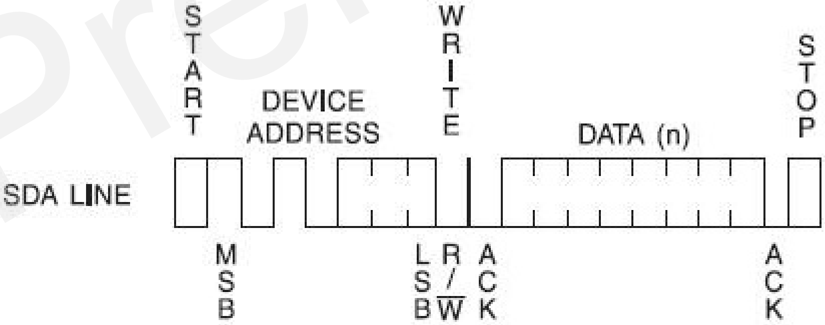


Figure 11: Sequence Chart for Writing I2C Buffer

4.2 Sample Code for Reading and Writing I2C Buffer

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 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 Receive and Parse NMEA Sentence

This chapter describes the flow and sample code for receiving and parsing NMEA sentence.

5.1 Flow of Receiving and Parsing NMEA Sentence

The receiving and parsing flow of NMEA sentence is shown below.

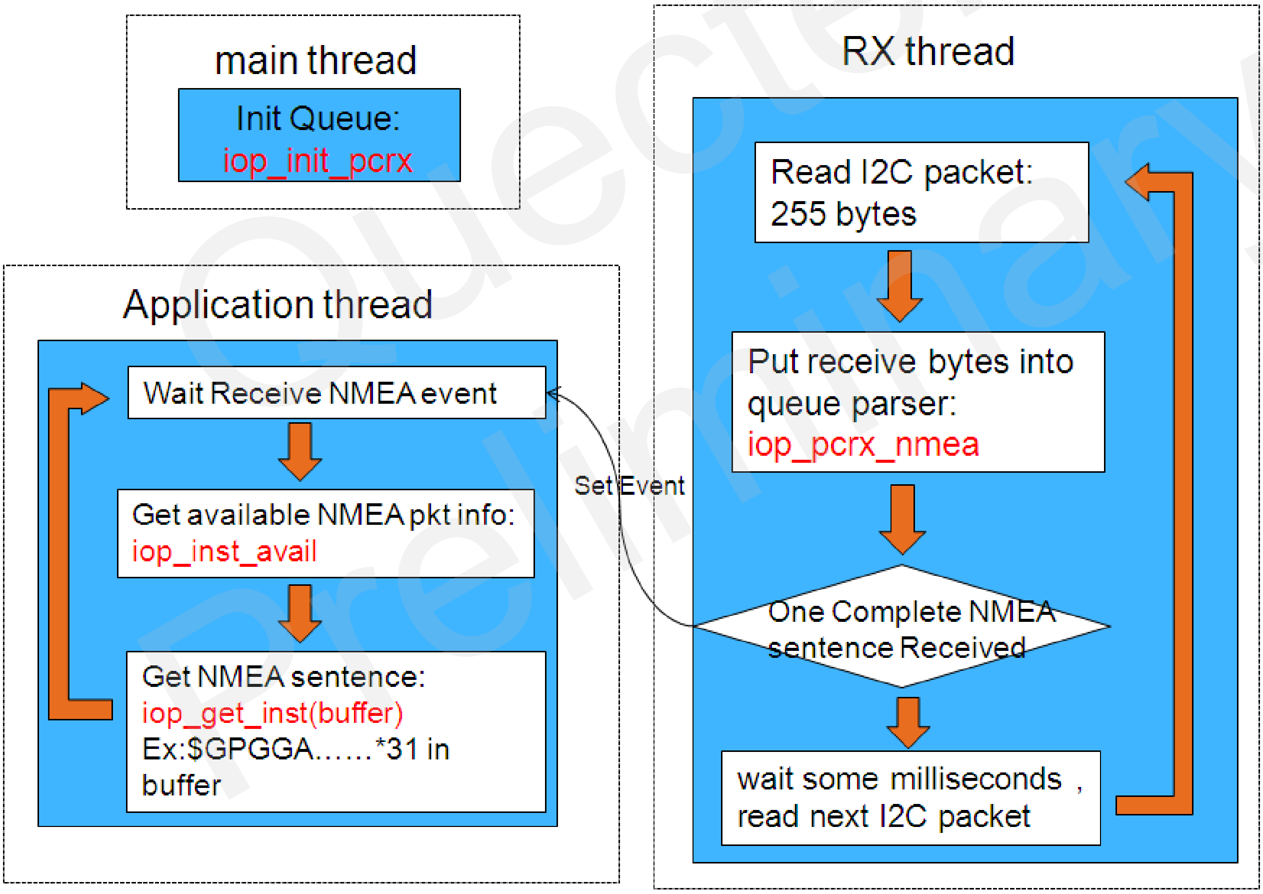


Figure 12: Receiving and Parsing Flow of NMEA Sentence

5.2 Sample Code for Receiving and Parsing NMEA Sentence

After NMEA sentence is received, it will extract NMEA and debug data from many I2C packets. It will also discard garbage bytes and 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 and debug log code, 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 receive 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 fetch the characters between/includes '$' and <CR>.
* That is, character <CR><LF> is 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 receive 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/includes '$' and <CR>.
* That is, character <CR><LF> is 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 receive 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 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_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 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++;
            }
            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 */
```