L76-L&L96 I2C

Application Note

GNSS Module Series

Rev. L76-L&L96_I2C_Application_Note_V1.1

Date: 2018-10-16

Status: Released

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

Quectel Wireless Solutions Co., Ltd.
7th Floor, Hongye Building, No.1801 Hongmei Road, Xuhui District, Shanghai 200233, China
Tel: +86 21 5108 6236
Email: info@quectel.com

Or our local office. 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 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	2016-09-14	Simon HU	Initial
1.1	2018-10-16	Jenn XIANG	Added L96 as an applicable module of this document.

Contents

Table Index

Figure Index

1 Introduction

This document introduces the I2C function of Quectel L76-L and L96 modules. These modules working as a slave provide an I2C interface which outputs NMEA data only when the data is read out by a master (client-side MCU).

In this document, customers will find a detailed introduction on how the master receives/parses NMEA sentences and sends SDK commands via I2C bus. And flow charts and sample codes are provided to assist in I2C buffer reading/writing as well as NMEA sentence receiving/parsing.

2 NMEA Data Reading through I2C Bus

This chapter provides a detailed introduction on how the master reads and parses NMEA data packets through the I2C bus.

2.1. Features of I2C Interface

The features of L76-L and L96 modules' I2C interface include:

- 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. 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 with a 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.

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 buffer. When the entire NMEA packet of one second is read, the master can sleep for a longer time (e.g. 500ms) to wait for the entire NMEA packet of next second to be ready.

For L76-L and L96 modules, the NMEA data packet can be read via I2C only in polling mode. To avoid data loss, the master should read the entire NMEA packet of one second in a 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 data packets via I2C in polling mode.

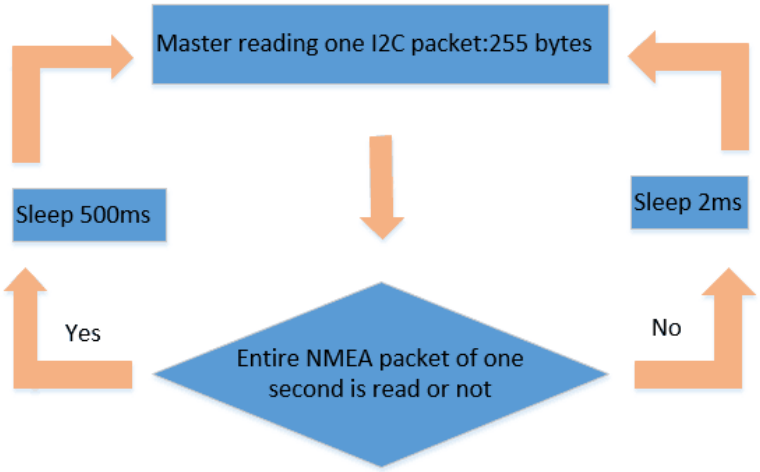


Figure 1: NMEA Data Reading Flow of the Master in Polling Mode

NOTE

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

2.3. I2C Data Packets

2.3.1. Format of I2C Data Packet

The data packet in the slave's 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 format of I2C data packet.

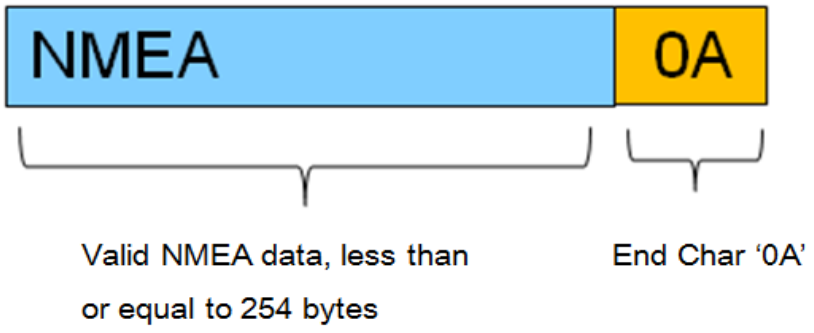


Figure 2: Format of I2C Data Packet

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

shown below:

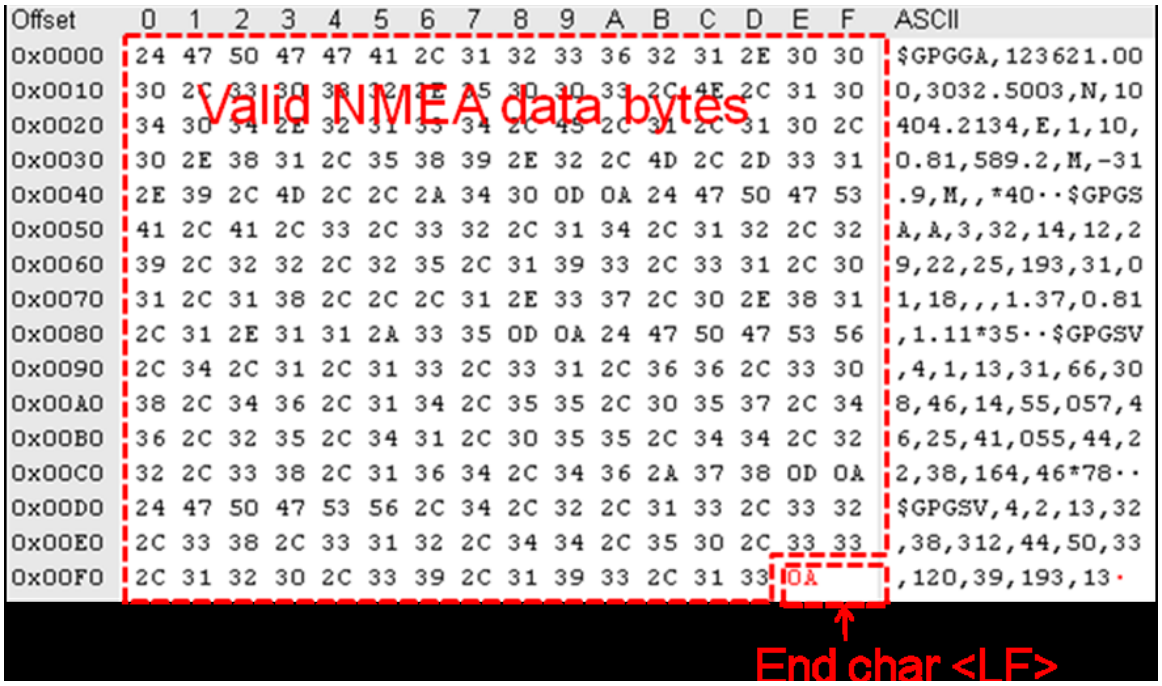


Figure 3: Example of I2C Data Packet Format

2.3.2. Three Types of I2C Data Packets

No matter whether there are NMEA data saved in the I2C buffer, the master can read one I2C data packet (255 bytes) from the slave. There are three types of I2C data packets that the master can read from the slave.

2.3.2.1. The First I2C Packet Type: 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 saved in the buffer, then the last byte will be the end character <LF>.

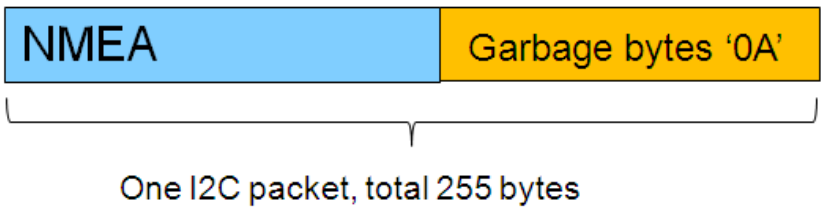
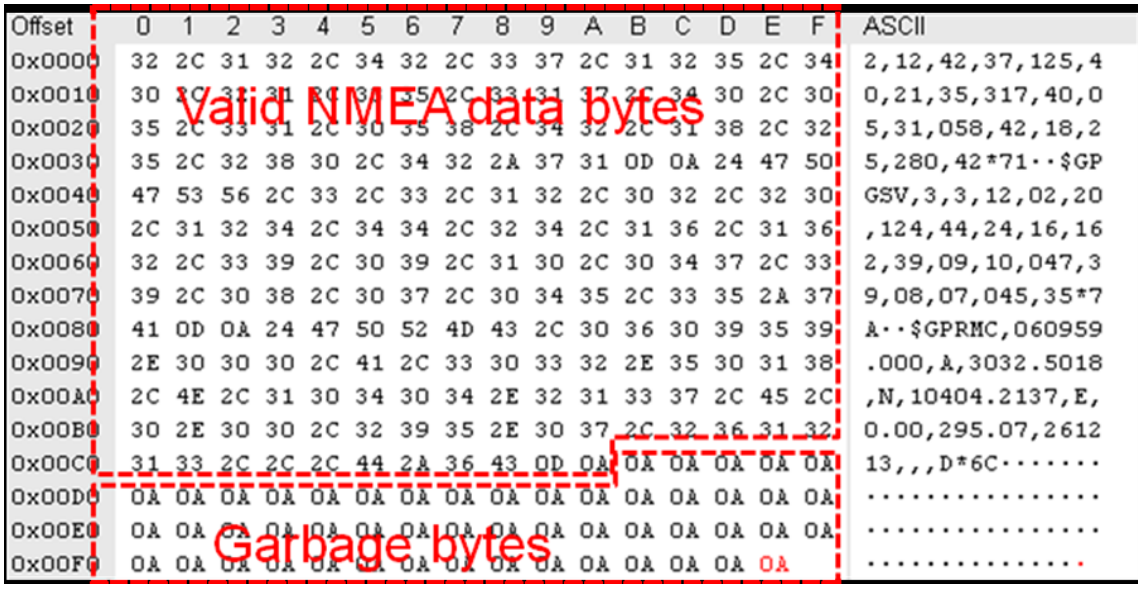


Figure 4: The First I2C Packet Type (Valid Data Bytes + Garbage Bytes)

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 as below:



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	0A	·····

Figure 5: Example of the First I2C Packet Type (Valid Data Bytes + Garbage Bytes)

NOTE

Why garbage bytes are '0A'?
If the slave's I2C buffer is empty, the slave (L76-L or L96) 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.

2.3.2.2. The Second I2C Packet Type: All Garbage Bytes

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

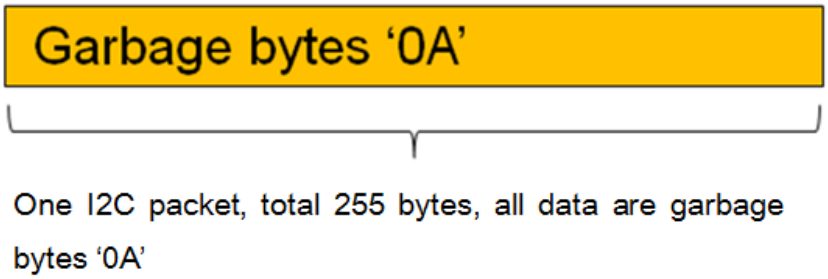


Figure 6: The Second I2C Packet Type (All Garbage Bytes)

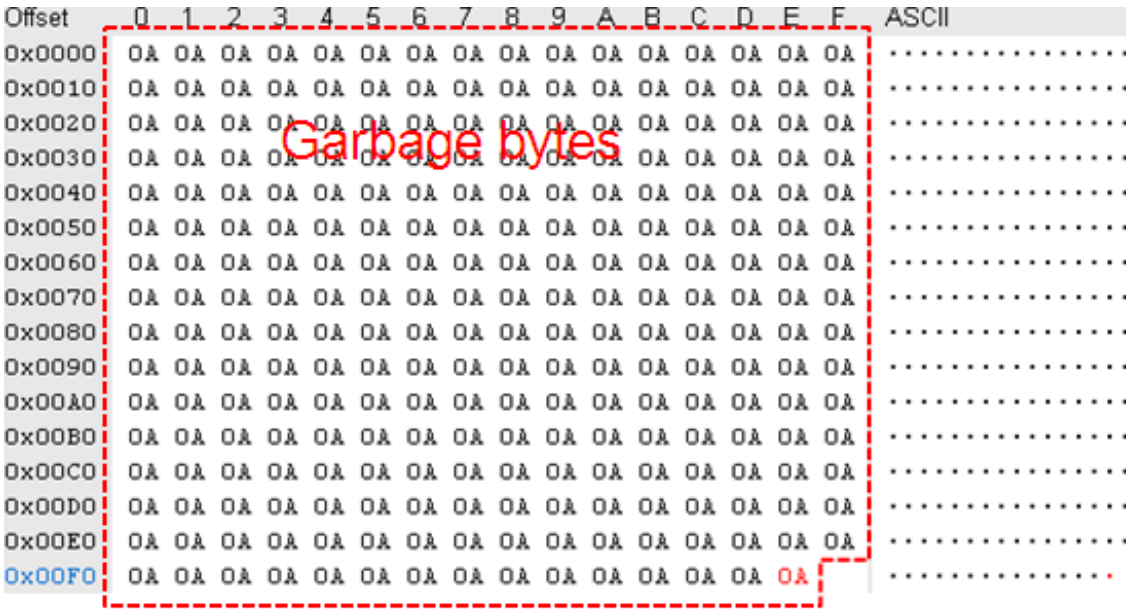


Figure 7: Example of the Second I2C Packet Type (All Garbage Bytes)

2.3.2.3. The Third I2C Packet Type: 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.

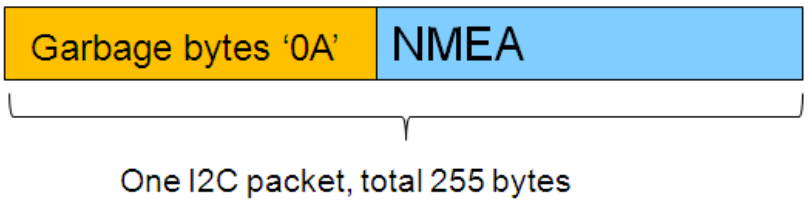


Figure 8: The Third I2C Packet Type (Garbage Bytes + Valid Data Bytes)

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 the Third I2C Packet Type (Garbage Bytes + Valid Data Bytes)

2.3.3. 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. Quectel provides the 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 forms 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

L76-L and L96 modules support SDK commands which are defined and developed by Quectel. The master can send SDK commands to the slave via I2C bus. Please check *Quectel_GNSS_SDK_Commands_Manual* for detailed information about SDK commands.

As the slave's I2C buffer has a 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 as below.

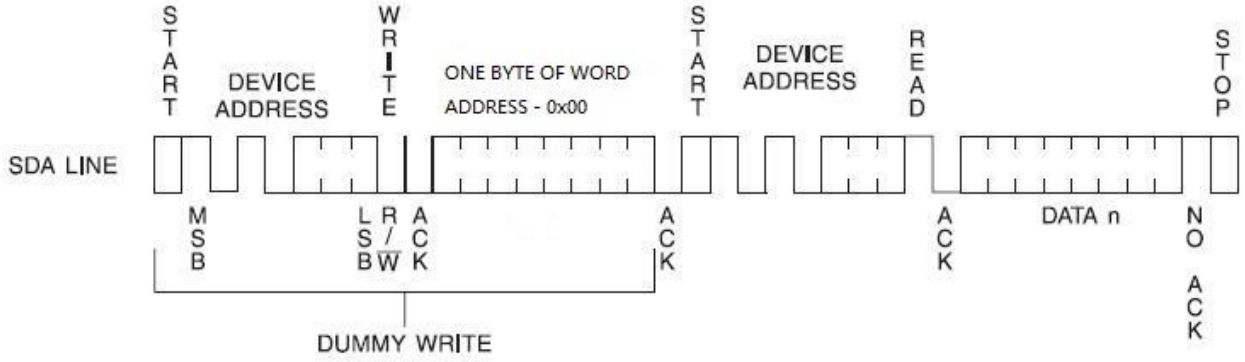


Figure 10: Sequence Chart for Reading I2C Buffer

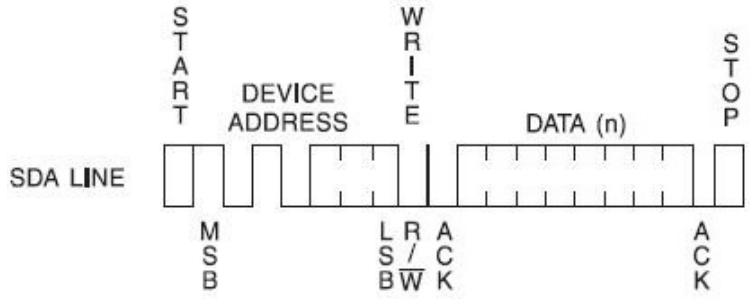


Figure 11: Sequence Chart for Writing I2C Buffer

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 Sentences

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

5.1. Flow Chart

The flow chart for receiving and parsing NMEA sentences is shown below.

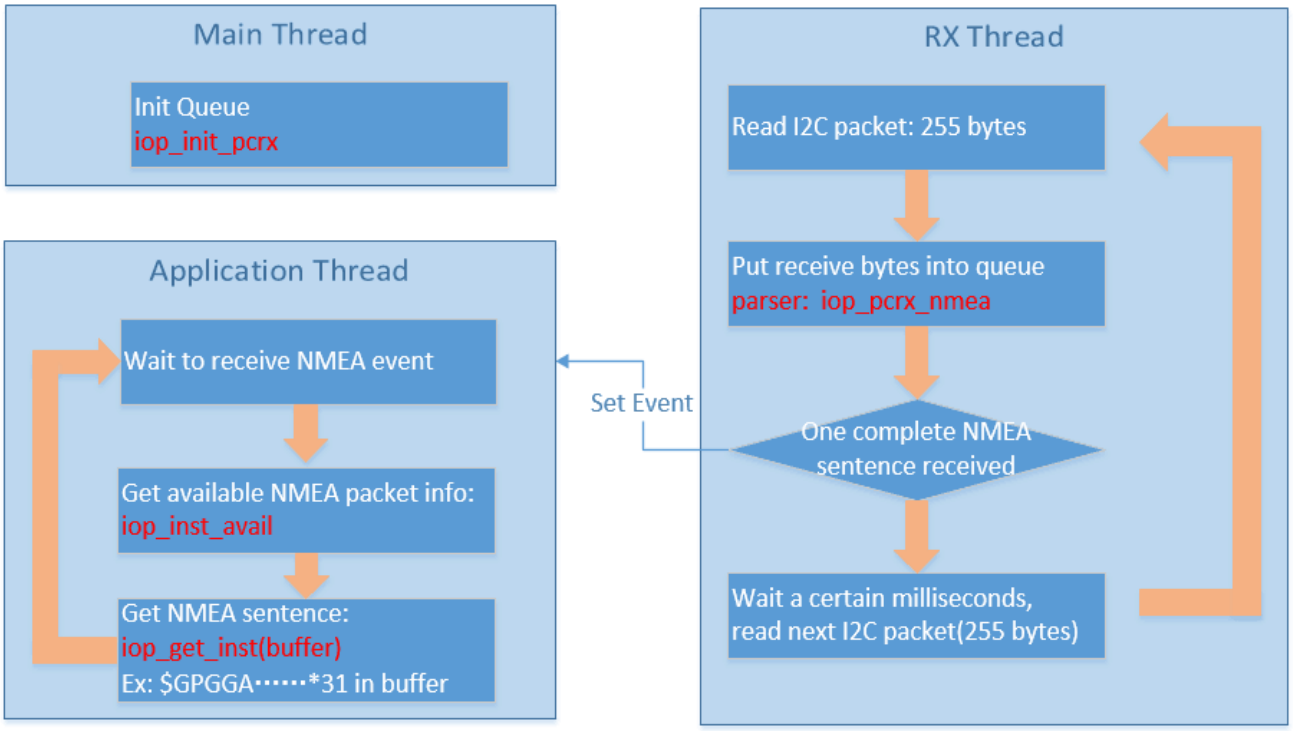


Figure 12: Flow Chart for Receiving and Parsing NMEA Sentence

5.2. Sample Code

After receiving the NMEA data packets, the master will parse 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.

The sample code for receiving and parsing I2C NMEA sentences is shown below.

```
#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
ACK	Acknowledge
GNSS	Global Navigation Satellite System
I2C	Inter-Integrated Circuit
LSB	Least Significant Bit
MCU	Main Computational Unit
MSB	Most Significant Bit
NMEA	National Marine Electronics Association
SDK	Software Development Kit