KL series LIN slave simply usage sharing

1 Abstract

LIN (Local Interconnect Network) is a concept for low cost automotive networks, which complements the existing portfolio of automotive multiplex networks. LIN is based on the UART/SCT protocol. It can be used in the area of automotive, home appliance, office equipment, etc. The UART module in NXP kinetis L series contains the LIN slave function, it can be used as the LIN slave device in the LIN bus. Because there is few LIN slave KL sample code for the customer's reference in our website, now this document mainly take KL43 as an example, explain how to use the FRDM-KL43 board as the LIN slave node to communicate with the LIN master device. LIN master use the specific LIN module: PCAN-USB Pro FD. Master send the publisher ID and subscriber ID, slave give the according LIN data response. This document will share the according code, hardware connection and the test result.

2 LIN bus basic knowledge review

For the convenient to understand the LIN bus, this chapter simply describe the basic knowledge for LIN bus. Mainly about the LIN topology and the LIN frame.

2.1 LIN bus topology structure

LIN bus just use the simple low cost single-wire, it uses single master to communicate with multiple slaves. The bus voltage is 12V, the speed can up to 20 kbit/s. LIN network can connect 16 nodes, but in the practical usage, normally use below 12 nodes.

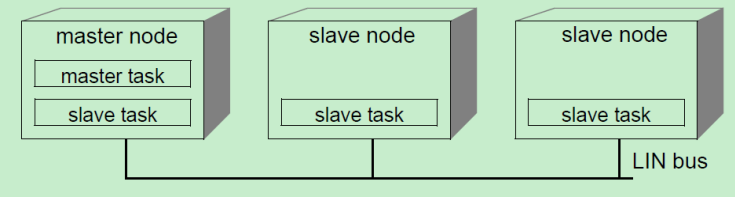


Figure 2-1. LIN bus topology

2.2 LIN bus frame structure

LIN Frame consists of a header (provided by the master task) and a response (provided by a slave task).

Master send publisher frame: Master send header+ data +checksum; slave just receive.

Master send subscriber frame: Master send header; slave receive send data +checksum.

The following figure is the structure of a LIN frame:

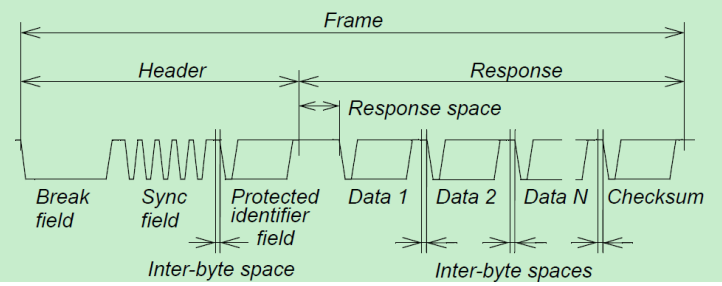


Figure 2-2. LIN frame structure

LIN frame is constructed of one Break field, sync byte field (0X55), PID, data and checksum.

2.2.1 Break filed and break delimiter

Break filed is consist of break and break delimiter. Break should at least 13 nominal bit times of dominant value (low voltage). The break delimiter shall be at least one nominal bit time long (high voltage).

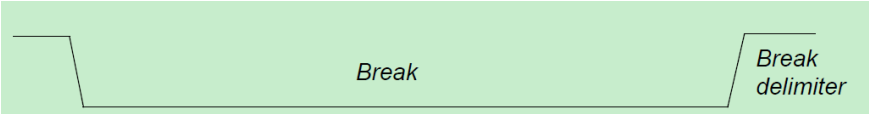


Figure 2-3. break field

2.2.2 Sync byte field

Sync is a byte field with the data value 0X55. The byte field is the standard UART protocol.

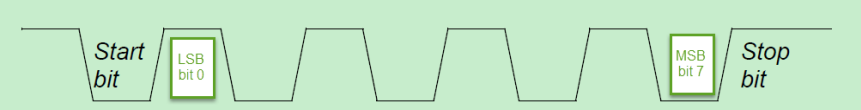


Figure 2-4. The sync byte field

2.2.3 Protected identifier field

A protected identifier field consists of two sub-fields: the frame identifier and the parity. Bits 0 to 5 are the frame identifier and bits 6 and 7 are the parity.

ID value range: 0x00-0x3f, 64 IDs in total. It determine the frame categories and direction.

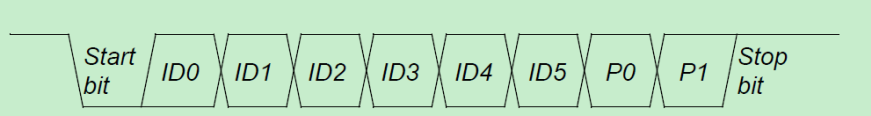


Figure 2-5. The sync byte field

$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4$

$P1 = -(ID1 \oplus ID3 \oplus ID4 \oplus ID5)$

$\oplus$ is the XOR，$-$is NOT。

ID can be split in three categories:

Frame categories		**Frame ID**
Signal carrying frame	Unconditional frame	0x00-0x3B
	Event triggered frame	
	Sporadic frame	
Diagnostic frame	Master request frame	0x3c
	Slave response frame	0x3d
Reserved frame		0x3e,0x3f

2.2.4 DATA

A frame carries between one and eight bytes of data. The number of data contained in a frame with a specific frame identifier shall be agreed by the publisher and all subscribers.

For data entities longer than one byte, the entity LSB is contained in the byte sent first and the entity MSB in the byte sent last (little-endian). The data fields are labeled data 1, data 2,... up to maximum data 8.

2.2.5 checksum

The checksum contains the inverted eight bits sum with carry over all data bytes **or** all data bytes and the protected identifier.

Classic checksum: Checksum calculation over the data bytes.

Enhanced checksum: Checksum calculation over the data bytes and the protected identifier byte.

Method: eight bits sum with carry is equivalent to sum all values and subtract 255 every time the sum is greater or equal to 256, at last, the sum data do bitwise invert. In the receive side, do the same sum, but at last, don't do invert, then add the received checksum data, if the result is 0XFF, it is correct, otherwise, it is wrong.

3 KL43 LIN slave example

This chapter use KL43 as the LIN slave, and communicate with the specific LIN master device, realize the LIN data sending and receiving.

3.1 Hardware prepare

Hardware: FRDM-KL43，TRK-KEA8，PCAN-USB Pro FD

LIN bus voltage is 12V, but the FRDM-KL43 don't have the LIN transceiver, so we need the external LIN transceiver connect the KL43 uart, to realize the LIN voltage switch. Here we use the TRK-KEA8 on board LIN transceiver MC33662LEF for the KL43. The MC33662LEF circuit is like this:

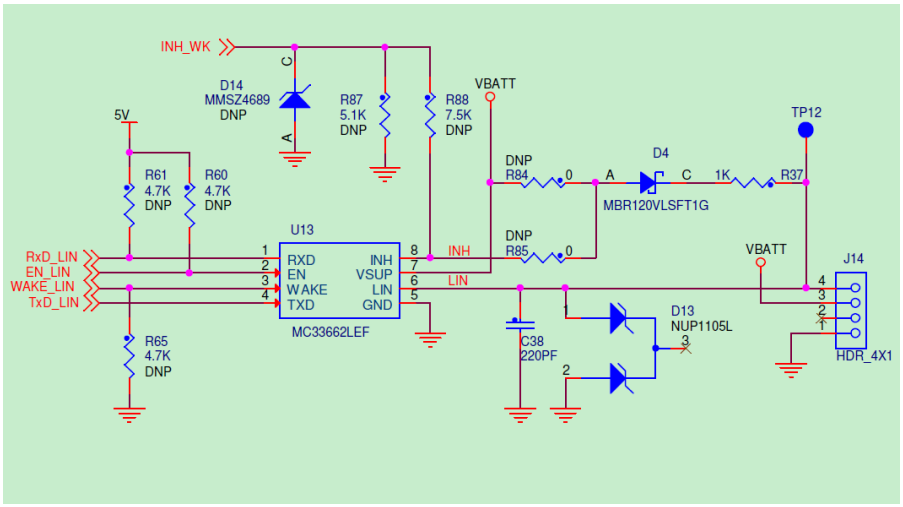


Figure 3-1. LIN transceiver schematic

3.1.1 FRDM-KL43 and TRK-KEA8 connections

FRDM-KL43 need to connect the UART port to the LIN transceiver. The connection shows in this table:

No.	FRDM-KL43	TRK-KEA8	note
1	J1-2	J10-5	UART0_RX
2	J1-4	J10-6	UART0_TX
3	J3-14	J14-1	GND

3.1.2 TRK-KEA8 and LIN master connections

LIN bus is using the signal wire. TRK-KEA8 J14_4 is the LIN wire, it should connect with the LIN wire in PCAN-USB Pro FD. GND also need to connect together.

TRK-KEA8 P1 need a 12V DC supplier. Master also need 12V DC supplier.

3.1.3 Object connection picture

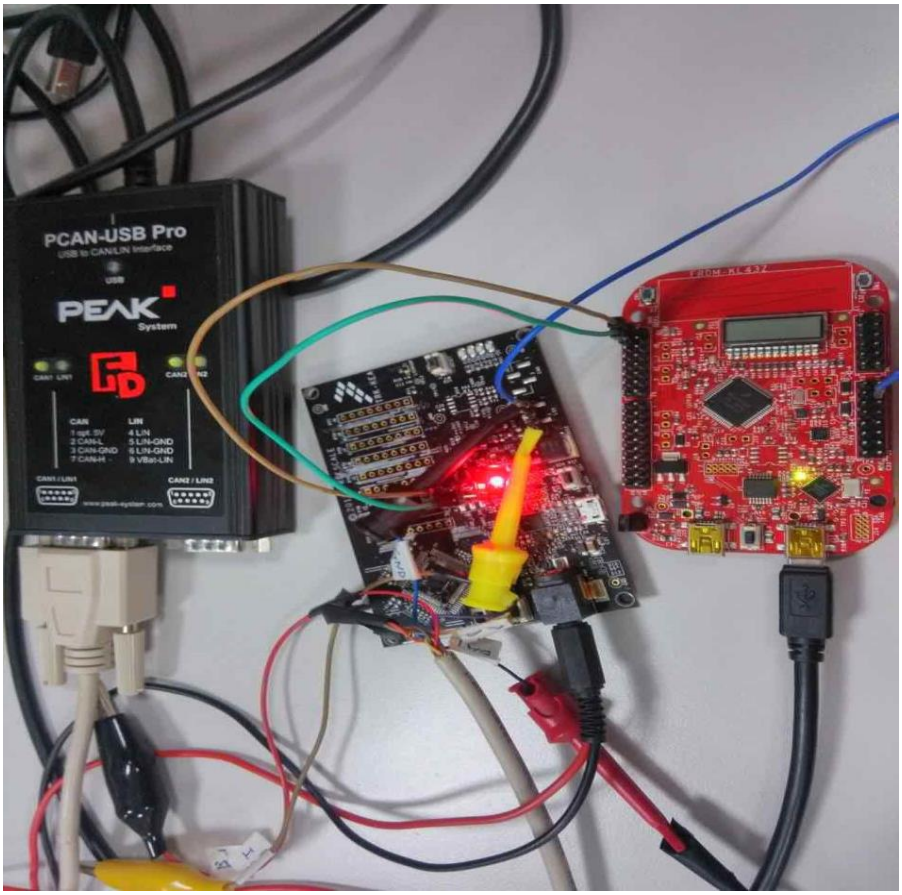


Figure 3-2. Object connections

3.2 Software flow chart and code

Now describe how to realize the LIN master and the LIN slave data transfer. LIN master send a publisher frame, the slave will receive the according data. LIN master send a subscriber frame, the slave will send the data to the master. The code is based on the KSDK2.2_FRDM-KL43 lpuart, add the LIN operation code.

3.2.1 Software flow chart

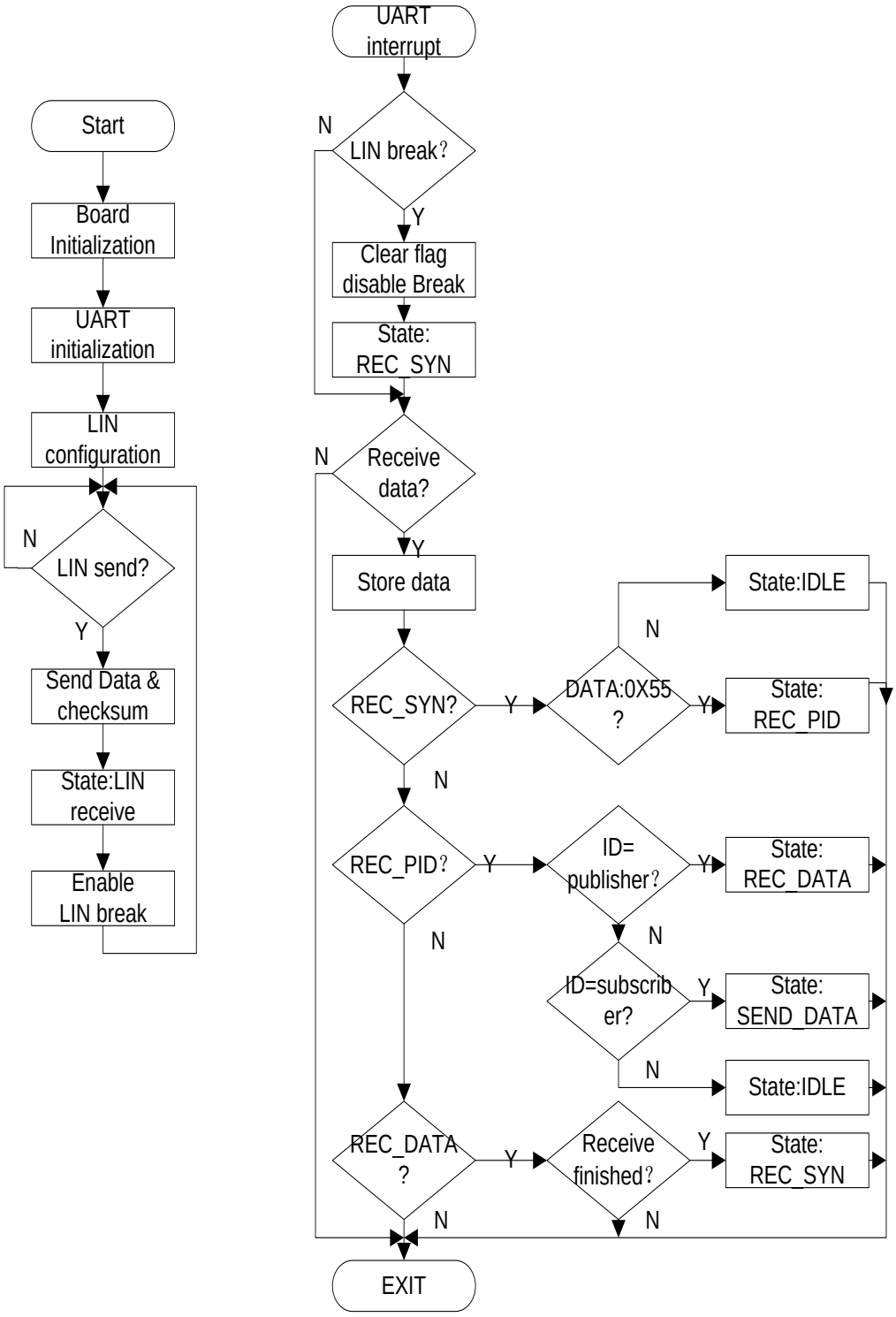


Figure 3-3. Software flow chart

3.2.2 software code

Code is based on KSDK2.2_FRDM-KL43 lpuart project, add the LIN operation code, the added code is list as follows:

```
void LPUART0_IRQHandler(void)
{
    if(LPUART0->STAT & LPUART_STAT_LBKDIF_MASK)
    {
     LPUART0->STAT |= LPUART_STAT_LBKDIF_MASK;// clear the bit
     Lin_BKflag = 1;
     cnt = 0;
     state = RECV_SYN;
     DisableLinBreak;
    }
   if(LPUART0->STAT & LPUART_STAT_RDRF_MASK)
    {
            rxbuff[cnt] = (uint8_t)((LPUART0->DATA) & 0xff);
       switch(state)
       {
         case RECV_SYN:
                  if(0x55 == rxbuff[cnt])
                  {
                    state = RECV_PID;
                  }
                  else
                  {
                    state = IDLE;
                    DisableLinBreak;
                  }
                  break;
         case RECV_PID:
                  if(0xAD == rxbuff[cnt])
                  {
                    state = RECV_DATA;
                  }
                  else if(0XEC == rxbuff[cnt])
                  {
                    state = SEND_DATA;
                  }
                  else
                  {
                    state = IDLE;
                    DisableLinBreak;
                  }
                  break;
         case RECV_DATA:
                  recdatacnt++;
                  if(recdatacnt >= 4) // 3 Bytes data + 1 Bytes checksum
                  {
                    recdatacnt=0;
                    state = RECV_SYN;
                    EnableLinBreak;
```

```
                }
                break;
      default:break;

      }
      cnt++;
   }
}
void uart_LIN_break(void)
{
   LPUART0->CTRL &= ~(LPUART_CTRL_TE_MASK | LPUART_CTRL_RE_MASK);   //Disable UART0 first
   LPUART0->STAT |= LPUART_STAT_BRK13_MASK; //13 bit times
   LPUART0->STAT |= LPUART_STAT_LBKDE_MASK;//LIN break detection enable
   LPUART0->BAUD |= LPUART_BAUD_LBKDIE_MASK;

   LPUART0->CTRL |= (LPUART_CTRL_TE_MASK | LPUART_CTRL_RE_MASK);
   LPUART0->CTRL |= LPUART_CTRL_RIE_MASK;
   EnableIRQ(LPUART0_IRQn);
}
int main(void)
{
   uint8_t ch;
   lpuart_config_t config;
   BOARD_InitPins();
   BOARD_BootClockRUN();
   CLOCK_SetLpuart0Clock(0x1U);
   LPUART_GetDefaultConfig(&config);
   config.baudRate_Bps = BOARD_DEBUG_UART_BAUDRATE;
   config.enableTx = true;
   config.enableRx = true;
   LPUART_Init(DEMO_LPUART, &config, DEMO_LPUART_CLK_FREQ);
   uart_LIN_break();
   while (1)
   {
     if(state == SEND_DATA)
     {
       while((LPUART0->STAT & LPUART_STAT_TDRE_MASK) == 0); // hex mode
            LPUART0->DATA = 0X01;
       while((LPUART0->STAT & LPUART_STAT_TDRE_MASK) == 0); // hex mode
            LPUART0->DATA = 0X02;
       while((LPUART0->STAT & LPUART_STAT_TDRE_MASK) == 0); // hex mode
            LPUART0->DATA = 0X10;//Checksum   0X10 correct, 0xaa is wrong
       recdatacnt=0;
       state = RECV_SYN;
       EnableLinBreak;
     }
   }
}
```

4 KL43 LIN slave test result

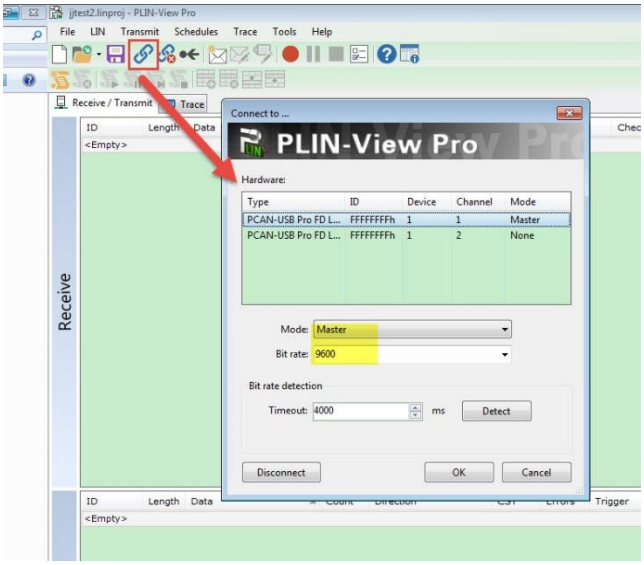
Master defines two frames:

Unconditional ID	Protected ID	Direction	Data	checksum
0X2C	0XEC	subscriber	0x01,0x02	0x10
0X2D	0XAD	Publisher	0x01,0x02,0x03	0x4c

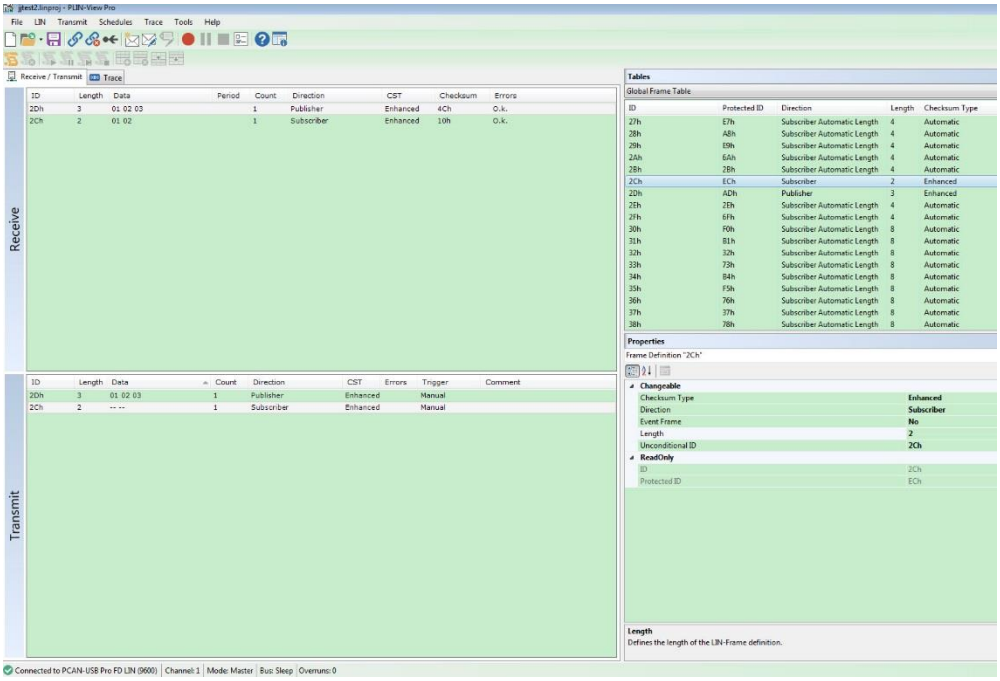
Now, master send 0X2C and 0X2D data, give the test result and the according waveform.

4.1 LIN master configuration

Uart baud rate is: 9600bps

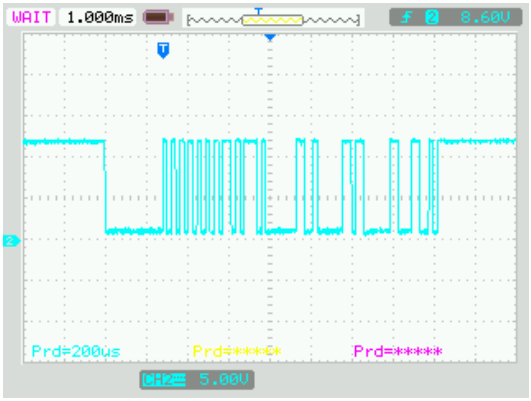


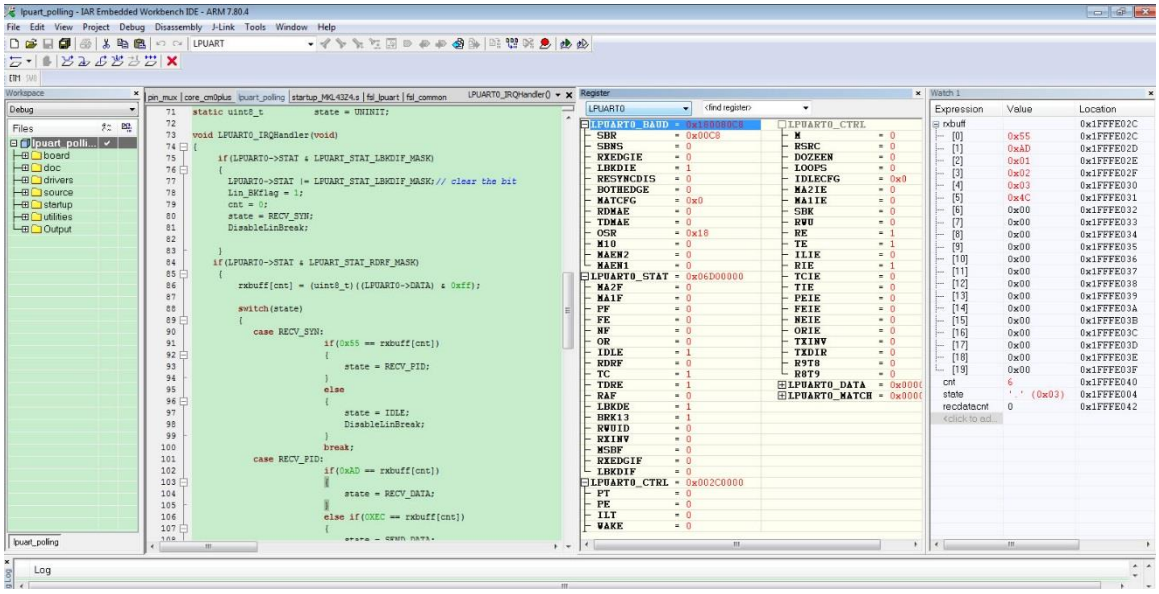
4.2 Send ID 0X2C and 0X2D frame



From the PC software of LIN master, we can find 0X2D ID can send the data successfully, and 0X2C ID can receive the correct data (0x01, 0x02) and checksum (0x10) from the KL43 LIN slave side.

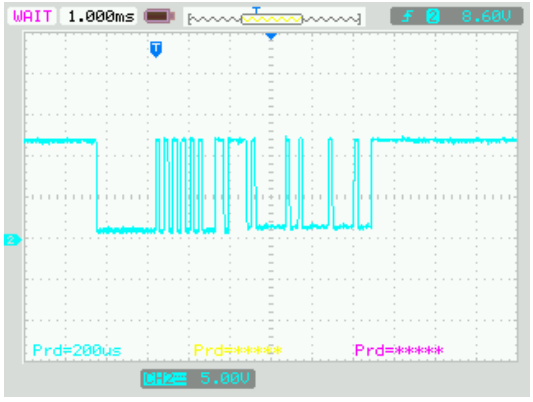
4.2.1 0X2D ID frame oscilloscope waveform and debug result



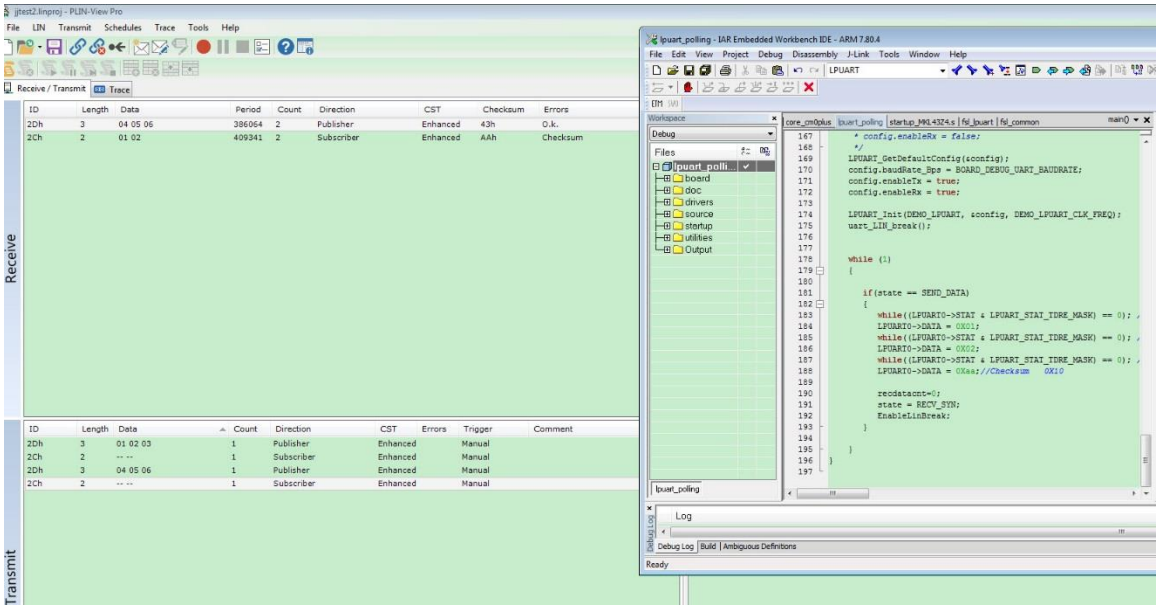


From the debug result, we can find the buff can receive the correct ID, data and checksum from the LIN master.

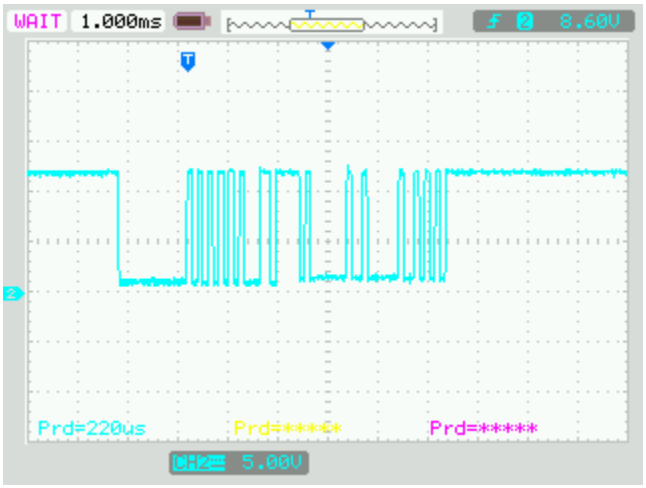
4.2.2 0X2C ID frame oscilloscope waveform



4.2.3 0X2C ID SLAVE send back the wrong checksum



From the PC software, we can find if the KL43 code modify the checksum to the wrong data 0XAA, then the PC software will display the checksum error.
This is the according oscilloscope waveform for the wrong checksum data.



From all the above test result. We can find, KL43 as the LIN slave, it can receive the correct data from the LIN master, and when LIN master send the subscriber ID, kl43 also can send back the correct LIN data to the master. More detail, please check the attached code project.

BTW, LIN spec can be downloaded from this link:

http://www.cs-group.de/wp-content/uploads/2016/11/LIN_Specification_Package_2.2A.pdf