摘要

本应用笔记，旨在帮助用户使用芯海通用 MCU CS32F03x 系列芯片软件实现 LIN 总线协议解析和通信。

本文以 CS32F036 为平台，使用 USART 来实现 LIN 从机功能，处理 LIN 协议解析，执行数据响应和命令执行，从而与 LIN 总线进行通信。详细介绍其实现原理，移植开发说明，并提供 demo 作为代码范例，以供用户参考；用户可根据实际使用的芯海 MCU 型号，参考或调用本应用笔记中相关函数，即可快速简单地进行移植开发。

版本

历史版本	修改内容	日期
V1.0	初版生成	2022-07-07

目录

1 应用说明

本应用笔记介绍 CS32F03x 芯片使用 USART 接口来实现 LIN 从机，从而与 LIN 主机进行通信的功能。详细介绍其实现原理，和移植开发说明，结尾提供 demo 以供参考。

2 LIN 协议

LIN(Local Interconnect Network)是一种串行网络协议，它主要用于汽车上各个模块之间的通信。LIN 是 CAN 总线的一种互补协议，但是 LIN 协议代价更低，足以满足一些不太重要的汽车上的模块的需求。

LIN 网络一般由一个主节点和至多 15 或 16 个从节点组成，如果从节点再多的话，通信的稳定性将得不到保证，整个网络只需要一根信号线和另外的一根地线就可以了。网络中的所有通信都由主节点发起，主节点也管理着整个网络的通信。

LIN 总线的一帧主要由两部分组成，即报文头（Header）和报文响应（Response）。其中，报文头是由一个主机节点的主机任务发出的，而报文响应（以下简称响应）是由一个主机节点或从机节点的从机任务发出的。其中报文头由同步间隔场（最小 13 个显性位）、同步场（1 个字节，数据不变，0x55）、和 PID 场（1 个字节）三部分组成；报文响应由 2/4/8 个字节的数据场、校验和场（1 个字节）所组成。

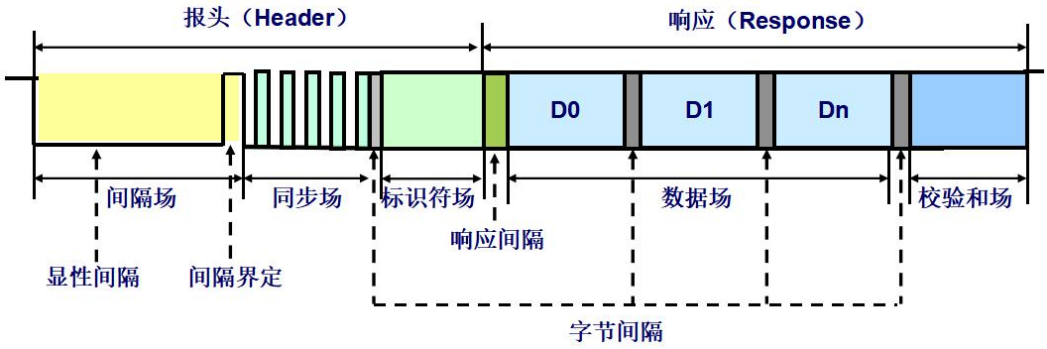


图 1 LIN 协议帧结构

LIN 网络中的节点任务分为主机任务和从机任务两类。对于 LIN 从机任务，它主要完成以下功能：

1.等待主机任务发送的同步间隔，使从机与主机于同步场中获得同步。
2.分析标识符场，若与自己相关，则接收或发送数据，若与自己无关则什么都不做。
3.检查和发送校验和。

综上，主机报文的标识符能触发与之对应的不同从机之间的通信。

3 USART 软件模拟 LIN

相关 USART 外设初始化和接收非空中断、断开帧检测中断（035，036 芯片不支持此中断）、发送中断、帧错误中断使能后，此时程序开始等待检测 LIN 协议最开始的断开帧，一旦检测到断开帧，此时立即更新 LIN 从机的状态机，进入到检测同步帧状态。同样的，接收到正确的同步帧后，状态机更新为进入接收 PID 状态，此后，程序执行 PID 判断，如果接收到的 PID 满足从机响应数据或者从机接收并执行命令的条件，则分别执行相应的响应数据任务或命令执行任务。在此过程中，任何接收错误或者 PID 不满足执行条件的情况下，状态机均重置为等待断开帧状态，重新开始下一次的 LIN 断开帧检测。基本流程如图 2 所示。

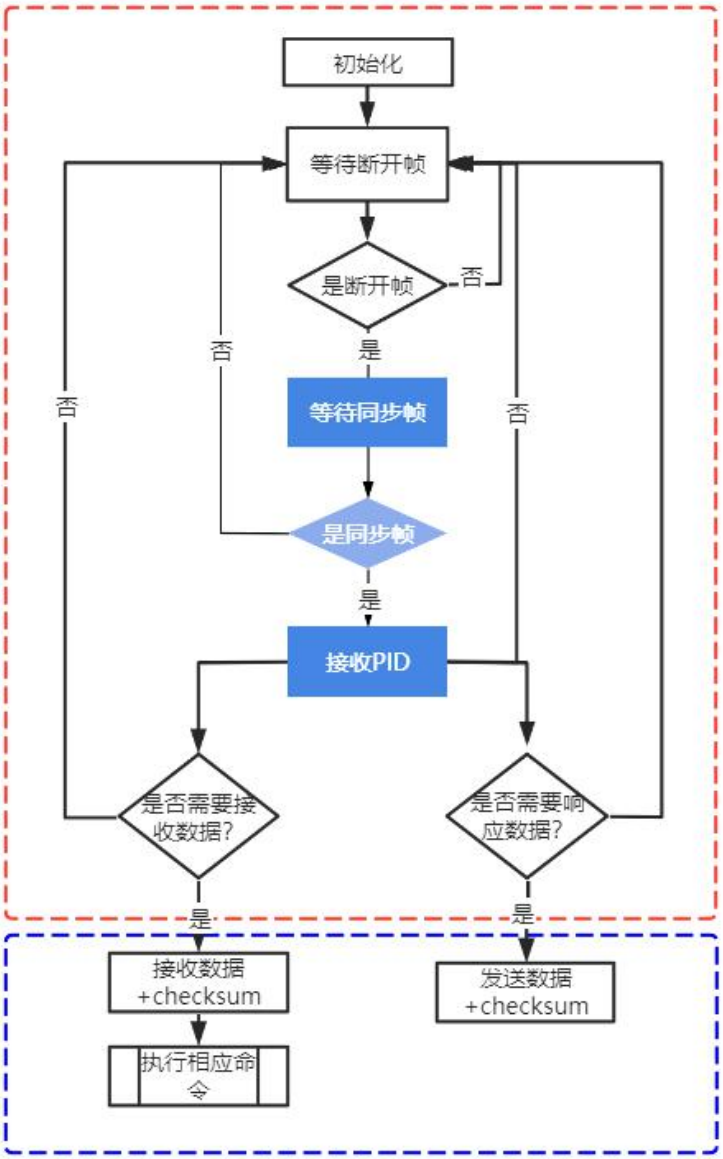


图 2 LIN 从机通信流程

在上述流程中，红色虚线框中的实现，属于协议解析层，相关功能提供模块化的接口。蓝色虚线框中具体任务的实现，由用户来实现，另外，本应用笔记第五节会提供用户任务的 demo 范例以供参考。

对于蓝色虚线框中的具体任务，需要在 LIN 从机初始化前事先注册。从而使得 LIN 从机在接收到指定的 PID 时，进入到指定的已事先注册的函数里去。这对用户而言，只需要调用 lin_response_task_register() 和 lin_execute_task_register() 这两个接口即可。例如，lin_response_task_register(0xEC,lin_slave_response_task0)，调用此接口即表明，从机在接收到 LIN 帧 PID 为 0xEC 时，会进入 lin_slave_response_task0 这个函数里，用户可以填充此函数，例如在该函数内调用 lin_slave_response()实现数据的发送。lin_response_task_register()和 lin_execute_task_register()代码如下：

```
/**
  * @brief  注册从机命令执行任务
  * @param
  * @return
  * @note   用户必须在调用 lin_slave_init()前注册并填充自己的命令执行任务函数
pfunc
  */
void lin_execute_task_register(uint8_t pid, uint8_t * cmd, void (*pfunc)(void))
{

   lin_execute_task[execute_register_num].pid = pid;
   lin_execute_task[execute_register_num].cmd = cmd;
   lin_execute_task[execute_register_num].pfunc = pfunc;

   execute_register_num++;
}


/**
  * @brief  注册从机响应任务(向主机反馈数据)
  * @param
  * @return
  * @note   用户必须在调用 lin_slave_init()前注册并填充自己的响应任务函数 pfunc
  */
void lin_response_task_register(uint8_t pid,void (*pfunc)(void))
{

   lin_response_task[response_register_num].pid = pid;
   lin_response_task[response_register_num].pfunc = pfunc;

   response_register_num++;
}
```

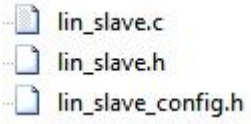
注册用户自己的 LIN 从机任务之后，调用 lin_slave_init()执行 LIN 初始化，程序即进入等待接收 LIN 帧的过程中。接收 LIN 帧的处理是在 USART 中断中进行的，以保证不影响芯片其他工作的进行。在中断中，数据准确接收与否是与当前时刻的 LIN 状态机密切相关的，例如，只有当前的状态机为 LINS_WAIT_FOR_RX_DATA 时，此后所接收的数据才会被认为是 LIN 主机发来的数据和校验值(checksum)。

对状态机的处理和更新主要是在初始化中和中断中。它们被封装成 lin_slave_process()函数，它对应用层不可见。用户进行移植开发时，只需要在注册从机任务后调用 lin_slave_init()执行 LIN 初始化，并在自己的 USART 中断中调用 lin_irq_handler()即可。

4 移植说明

只需进行简单的配置和个别接口的调用，即可将自己指定的 USART 作为 LIN 从机与 LIN 总线通信。

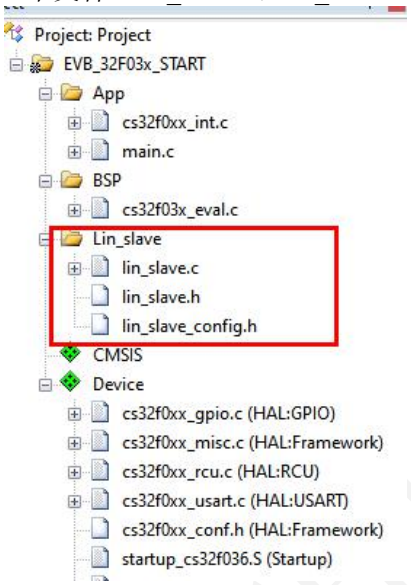
相关的函数或声明分别整理到 3 个文件。分别为 lin_slave.c，lin_slave.h，lin_slave_config.h. 文件里的代码或内容见本应用笔记下文。



其中，lin_slave.c 主要包含 USART 软件实现 LIN 协议解析和收发的接口函数；lin_slave.h 主要包含了一些状态枚举、类型定义和函数声明，lin_slave_config.h 包含了一些配置项，是在使用 lin_slave.c 中函数时预先定义的宏参数，例如 LIN 通讯速率。具体源代码见本应用笔记下文。

只需简单几步，即可实现配置 USART 为 LIN 从机。

1. 在自己的工程中，加入以下三个文件：lin_slave.c，lin_slave.h，lin_slave_config.h。



2. 配置 lin_slave_config.h 中的参数，例如 LIN_SLAVE_RESPONSE_TASK_NUM_MAX 参数，该参数只需比用户需要注册的从机响应任务个数大即可。

3. 在用户自己的文件，例如在 main.c 中，包含 lin_slave.h 头文件，即在文件起始处添加 #include "lin_slave.h"，调用 lin_response_task_register()和 lin_execute_task_register()来注册自己的任务处理函数（类似于注册自己的 callback 函数）并调用 lin_slave_init()执行初始化，例如下框中所示：

```
/**
  * @file    main.c
  */


/*收到 0xEC 的 PID,执行 lin_slave_response_task0 函数 */
lin_response_task_register(0xEC,lin_slave_response_task0);

/*收到 0xA3 的 PID,命令为 cmd_data[8]时，执行 lin_slave_execute_task0 函数， */
uint8_t cmd_data[8] = {0x11,0x22, 0x33, 0x01, 0x55, 0x66, 0x77, 0x88};
lin_execute_task_register(0xA3,cmd_data,lin_slave_execute_task0);

/* LIN 从机初始化 */
lin_slave_init();
```

lin_slave_init()的代码如下：

```
/**
  * @file    lin_slave.c
  */

void lin_slave_init(void)
{
```

```
    nvic_config_t  nvic_config_struct;
    usart_config_t usart_config_struct;

#if LIN_PORT == 0
    __RCU_AHB_CLK_ENABLE(RCU_AHB_PERI_GPIOA);
#elif LIN_PORT == 1
    __RCU_AHB_CLK_ENABLE(RCU_AHB_PERI_GPIOB);
#endif

#if LIN_SLAVE_USART_INSTANCE == 1
    __RCU_APB2_CLK_ENABLE(RCU_APB2_PERI_USART1);
#elif LIN_SLAVE_USART_INSTANCE == 2
    __RCU_APB1_CLK_ENABLE(RCU_APB1_PERI_USART2);
#endif

#if LIN_PORT == 0
    gpio_mf_config(GPIOA, LIN_SLAVE_USART_PIN_TX,
LIN_SLAVE_GPIO_MF_SEL);   //Tx
    gpio_mf_config(GPIOA, LIN_SLAVE_USART_PIN_RX,
LIN_SLAVE_GPIO_MF_SEL);  //Rx

    gpio_mode_set(GPIOA, LIN_SLAVE_USART_PIN_TX,
GPIO_MODE_MF_PP(GPIO_SPEED_MEDIUM));
    gpio_mode_set(GPIOA, LIN_SLAVE_USART_PIN_RX,
GPIO_MODE_MF_PP(GPIO_SPEED_MEDIUM));
#elif LIN_PORT == 1
    gpio_mf_config(GPIOB, LIN_SLAVE_USART_PIN_TX,
LIN_SLAVE_GPIO_MF_SEL);   //Tx
    gpio_mf_config(GPIOB, LIN_SLAVE_USART_PIN_RX,
LIN_SLAVE_GPIO_MF_SEL);  //Rx

    gpio_mode_set(GPIOB, LIN_SLAVE_USART_PIN_TX,
GPIO_MODE_MF_PP(GPIO_SPEED_MEDIUM));
    gpio_mode_set(GPIOB, LIN_SLAVE_USART_PIN_RX,
GPIO_MODE_MF_PP(GPIO_SPEED_MEDIUM));
#endif

#if LIN_SLAVE_USART_INSTANCE == 1
    __USART_DEF_INIT(USART1);
#elif LIN_SLAVE_USART_INSTANCE == 2
    __USART_DEF_INIT(USART2);
#endif

    usart_config_struct.baud_rate = LIN_SLAVE_BAUD_RATE;
    usart_config_struct.data_width = USART_DATA_WIDTH_8;
    usart_config_struct.stop_bits = USART_STOP_BIT_1;
    usart_config_struct.parity = USART_PARITY_NO;
    usart_config_struct.flow_control = USART_FLOW_CONTROL_NONE;
    usart_config_struct.usart_mode = USART_MODE_RX | USART_MODE_TX;

#if LIN_SLAVE_USART_INSTANCE == 1
    usart_init(USART1, &usart_config_struct);
#elif LIN_SLAVE_USART_INSTANCE == 2
    usart_init(USART2, &usart_config_struct);
#endif

    /* 此部分代码适用于 03x 系列增强型 USART,不适用于 035,036 芯片 */
```

```
    //usart_lin_break_detect_set(USART1,USART_LIN_BREAK_11BIT);
    //__USART_FUNC_ENABLE(USART1,LIN_MODE);  //使能 LIN 模式，检测 13 位断开帧，只有 USART1 有此功能

#if LIN_SLAVE_USART_INSTANCE == 1
    nvic_config_struct.IRQn = IRQn_USART1;
#elif LIN_SLAVE_USART_INSTANCE == 2
    nvic_config_struct.IRQn = IRQn_USART2;
#endif


    nvic_config_struct.priority = 0;
    nvic_config_struct.enable_flag = ENABLE;
    nvic_init(&nvic_config_struct);

#if LIN_SLAVE_USART_INSTANCE == 1
    __USART_ENABLE(USART1);
    //__USART_INTR_ENABLE(USART1, LINBK);  //LIN 断开帧检测开启，not available for CS32F036 devices
    __USART_INTR_ENABLE(USART1, RXNE);
    __USART_INTR_DISABLE(USART1,  TXE);
    __USART_INTR_ENABLE(USART1,  FERR);
#elif LIN_SLAVE_USART_INSTANCE == 2
    __USART_ENABLE(USART2);
    //__USART_INTR_ENABLE(USART2, LINBK);  //LIN 断开帧检测开启，not available for CS32F036 devices
    __USART_INTR_ENABLE(USART2, RXNE);
    __USART_INTR_DISABLE(USART2,  TXE);
    __USART_INTR_ENABLE(USART2,  FERR);
#endif

    lin_slave_process();
}
```

状态机的处理 lin_slave_process()的代码如下:

```
/**
  * @file   lin_slave.c
  */


void lin_slave_process(void)
{
    uint8_t rec_checksum = 0;
    uint8_t res_num = 0;
    uint8_t exe_num = 0;

    switch (current_state)
    {
        case LINS_IDLE:

            break;

        case LINS_BREAK_RECEIVED:

            current_state = LINS_WAIT_FOR_HEADER;
```

```
        break;

    case LINS_WAIT_FOR_HEADER:

        if(1 == header_received_flag)
        {

            header_received_flag = 0;

            current_state = LINS_HEADER_RECEIVED;

        }
        else
        {

        }
        break;

    case LINS_HEADER_RECEIVED:

        if(0x55 == lin_header[0])
        {

        /* 循环判断是否有需要响应的 PID*/
        for(res_num = 0; res_num < response_register_num; res_num++)

        {

            if(lin_response_task[res_num].pid == lin_header[1])
            {
                /* 从机响应任务函数，由用户来注册*/
                lin_response_task[res_num].pfunc();

                current_state = LINS_IDLE;

                __USART_INTR_ENABLE(LIN_SLAVE_USART, RXNE);

            }
        }

        for(exe_num = 0; exe_num < execute_register_num; exe_num++)
        {

        if(lin_execute_task[exe_num].pid == lin_header[1])
            {

                exe_id = exe_num;

                current_state = LINS_WAIT_FOR_RX_DATA;

                break;

            }

            else
            {
                //__USART_INTR_ENABLE(LIN_SLAVE_USART, LINBK);
```

```
                current_state = LINS_IDLE;
                __USART_INTR_ENABLE(LIN_SLAVE_USART, RXNE);

            }
        }

        }
        else
        {
            current_error_status = LINS_SYNC_ERROR;
            current_state = LINS_IDLE;

            __USART_INTR_ENABLE(LIN_SLAVE_USART, RXNE);

        }
        break;

    case LINS_WAIT_FOR_RX_DATA:

      if(1 == response_received_flag)
      {

        __USART_INTR_DISABLE(LIN_SLAVE_USART, RXNE);

        response_received_flag = 0;

        current_state = LINS_RX_DATA_RECEIVED;

      }
      else
      {

      }
      break;

    case LINS_RX_DATA_RECEIVED:


      rec_checksum = lin_checksum_get((lin_execute_task[exe_id].pid &
0x3f),lin_execute_task[exe_id].cmd,8);

      if(rec_checksum == lin_data[8])  //检查 checksum 8byte 数据 + 1 字节 checksum

      {
        /* 从机命令执行任务函数，由用户来注册*/
        lin_execute_task[exe_id].pfunc();

        __USART_INTR_ENABLE(LIN_SLAVE_USART, RXNE);
        current_state = LINS_IDLE;
        current_error_status = LINS_NO_ERROR;

      }
      else
      {
        __USART_INTR_ENABLE(LIN_SLAVE_USART, RXNE);
        current_state = LINS_IDLE;
```

```
                current_error_status = LINS_CHECKSUM_INVALID_ERROR;

            }
            break;

          default:

            break;

        }
    }
```

4. 在指定的 USART 中断函数里，加入 lin_irq_handler()，即如下框中所示：

```
void USART1_IRQHandler(void)
{
  lin_irq_handler();
}
```

lin_irq_handler()函数也是定义在 lin_slave.c 中，代码如下：

```
/**
  *  @file   lin_slave.c
  */


/**
 * @brief  lin 从机中断服务
 * @param
 * @return
 * @note   需要在指定 USART 的中断服务函数中调用此函数
 */
void lin_irq_handler(void)
{
  uint8_t receive_data;

  if(__USART_FLAG_STATUS_GET(LIN_SLAVE_USART, OVRERR) == SET) // 溢出
  {

    __USART_FLAG_CLEAR(LIN_SLAVE_USART, USART_FLAG_OVRERR);
    __USART_DATA_RECV(LIN_SLAVE_USART);

  }

  if (__USART_FLAG_STATUS_GET(LIN_SLAVE_USART, TXE) == SET)
  {

  }

  if (__USART_FLAG_STATUS_GET(LIN_SLAVE_USART, RXNE) == SET)
  {
    receive_data = __USART_DATA_RECV(LIN_SLAVE_USART);

    if( current_state == LINS_WAIT_FOR_RX_DATA)
    {
      if(response_rec_counter < LIN_CMD_LENGTH + 1)
```

```
            {

              lin_data[response_rec_counter] = receive_data ;   // 8byte 数据 + 1 字节 checksum

              response_rec_counter += 1 ;

              if(response_rec_counter == LIN_CMD_LENGTH + 1)
              {
                response_rec_counter = 0 ;
                response_received_flag = 1 ;

                lin_slave_process();

                lin_slave_process();

              }
            }
          }

          if(current_state == LINS_WAIT_FOR_HEADER)
          {
            if(header_rec_counter < 2)
            {
              lin_header[header_rec_counter] = receive_data ;   // PID(6 bits ID +  2 bit 校验位)

              header_rec_counter += 1 ;

              if(header_rec_counter == 2)
              {
                header_rec_counter = 0 ;
                header_received_flag = 1 ;

                lin_slave_process();
                lin_slave_process();

              }

            }
          }
        }

        /* 此部分代码适用于 03x 系列增强型 USART,不适用于 035,036 芯片 */
    //    if(__USART_FLAG_STATUS_GET(LIN_SLAVE_USART, LINBK) == SET)
    //    {
    //
    //      __USART_FLAG_CLEAR(LIN_SLAVE_USART,USART_FLAG_LINBK);
    //      //__USART_INTR_DISABLE(LIN_SLAVE_USART, LINBK);
    //
    //      //lin_rate_synchronize();
    //
    //      current_state = LINS_BREAK_RECEIVED;
    //
    //      __USART_FLAG_CLEAR(LIN_SLAVE_USART,USART_FLAG_RXNE);
    //      __USART_INTR_ENABLE(LIN_SLAVE_USART,RXNE);

    //        lin_slave_process();
    //
```

```
    //  }

      if((__USART_FLAG_STATUS_GET(LIN_SLAVE_USART, FERR) == SET) &&
    (__USART_DATA_RECV(LIN_SLAVE_USART) == 0))
      {
        __USART_FLAG_CLEAR(LIN_SLAVE_USART,USART_FLAG_FERR);

        //lin_rate_synchronize();

        current_state = LINS_BREAK_RECEIVED;

        __USART_FLAG_CLEAR(LIN_SLAVE_USART,USART_FLAG_RXNE);
        __USART_INTR_ENABLE(LIN_SLAVE_USART,RXNE);

        lin_slave_process();

      }
    }
```

涉及到的几个结构体定义在 lin_slave.h 中，如下：

```
    /**
      *  @file   lin_slave.h
      */

      typedef struct
      {
        uint8_t pid;                /* 需要从机反馈数据的 PID */
        void   (*pfunc)(void);      /* 对应的反馈处理函数 */
      } response_task_t;


      typedef struct
      {
        uint8_t   pid;              /* 需要从机执行命令的 PID */
        uint8_t * cmd;              /* 主机的命令（2,4 或 8 字节数据）*/
        void    (*pfunc)(void);     /* 对应的命令执行函数 */
      } execute_task_t;


      typedef enum
      {
        LINS_IDLE,
        LINS_BREAK_RECEIVED,        /* 接收到至少 13bit 低电平同步断开帧 */
        LINS_WAIT_FOR_HEADER,       /* 等待接收完 header */
        LINS_HEADER_RECEIVED,       /* 接收完 header */
        LINS_RX_DATA,               /* 主机发 header 和 response,从机开始接收
                                       response */
        LINS_WAIT_FOR_RX_DATA,      /* 等待从机接收完 response */
        LINS_RX_DATA_RECEIVED,      /* 从机全部接收完 response */
        LINS_TX_DATA                /* 主机发 header,从机开始发送 response */
      } lin_slave_state_t;
```

5 Demo 参考范例

本应用笔记给出了一个 demo，如本节所展示。demo 注册了 2 个从机响应任务（反馈数据）和 2 个从机命令执行任务（接收数据并执行）。

2 个从机响应任务。 即收到 0xEC 的 PID 时,向 LIN 总线发送 2 字节数据，即[0x1a,0x1b]，校验和为 0xdd; 收到 0x99 的 PID 时，向 LIN 总线发送 8 字节数据，即[0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08]校验和为 0x42。

2 个从机命令执行任务。第一个任务为：接收到 PID 为 0xA3 的 LIN 帧，并且所接收的数据的第 4 字节和第 5 字节分别为 0x01 和 0x05 时,执行蓝色 LED 电平翻转任务；第二个任务为：接收到 PID 为 0xFB 的 LIN 帧，并且所接收的数据的第 5 字节和第 6 字节分别为 0x12 和 0x13 时,也执行一次蓝色 LED 电平翻转任务。

任务注册的代码如下：

```
/**
  *  @file    main.c
  */

  /* 收到 0xEC,响应 0x1a,0x1b，0xdd*/
  /* 收到 0x99,响应 0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x42 */
  lin_response_task_register(0xEC,lin_slave_response_task0);
  lin_response_task_register(0x99,lin_slave_response_task1);

  /*接收到 PID 为 0xA3 或 0xFB 的帧，如果接收的数据符合条件，则翻转电平 */
  uint8_t cmd_data[8]  = {0x11,0x22, 0x33, 0x01, 0x55, 0x66, 0x77, 0x88};
  uint8_t cmd_data1[8] = {0x01, 0x11, 0x12, 0x12, 0x12, 0x13, 0x14, 0x15};
  lin_execute_task_register(0xA3,cmd_data,lin_slave_execute_task0);
  lin_execute_task_register(0xFB,cmd_data1,lin_slave_execute_task1);

  lin_slave_init(); //初始化

  while(1)
  {

  }
```

任务执行的代码如下：

```
/**
  * @file    main.c
  */

/* lin 从机响应任务 0 */
void lin_slave_response_task0(void)
{
   uint8_t rec_pid = lin_pid_received();
   uint8_t response_data[2] = {0x1a,0x1b};
      lin_slave_response(rec_pid,response_data,2);
}

/* lin 从机响应任务 1*/
void lin_slave_response_task1(void)
```

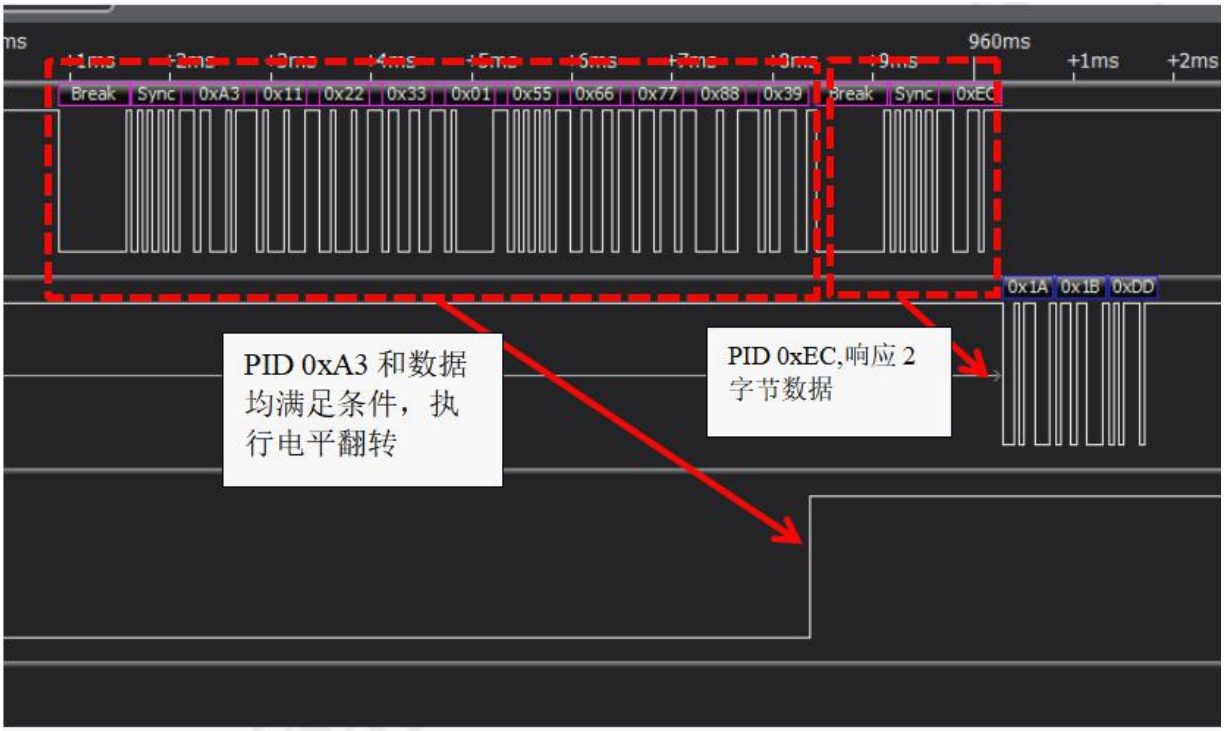
```
{
    uint8_t rec_pid = lin_pid_received();
    uint8_t response_data[8] = {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
      lin_slave_response(rec_pid,response_data,8);
}


/* lin 从机命令执行任务 0  */
void lin_slave_execute_task0(void)
{
    uint8_t * rec_data = lin_execute_data_received();
    if((0x01 == rec_data[3]) && (0x55 == rec_data[4]))   //满足命令条件
    {
      cs_eval_led_toggle();  //执行命令，翻转蓝色 led 灯电平状态
    }

}


/* lin 从机命令执行任务 1 */
void lin_slave_execute_task1(void)
{
    uint8_t * rec_data = lin_execute_data_received();
    if((0x12 == rec_data[4]) && (0x13 == rec_data[5]))   //满足命令条件
    {
      cs_eval_led_toggle();  //执行命令，翻转蓝色 led 灯电平状态
    }
}
```

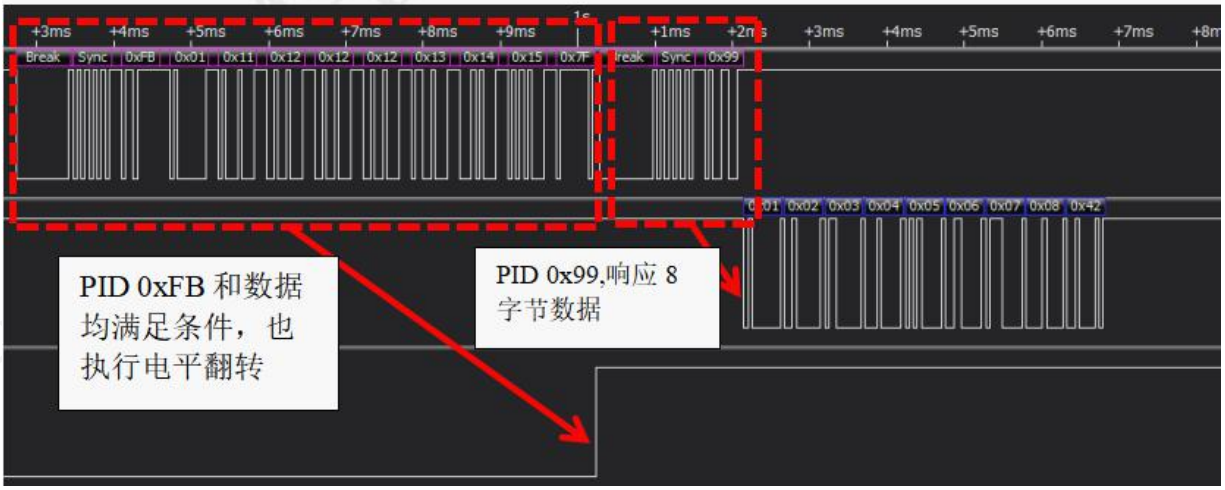
同时，有一些宏定义需要预先定义，例如 UART 复用的 GPIO 引脚等，这些宏定义在 lin_slave_config.h 里，如下所示：

```
/**
  *  @file   lin_slave_config.h
  */

  /* 从机响应任务的最大个数，该值需要比用户注册的任务个数大 */
  #define   LIN_SLAVE_RESPONSE_TASK_NUM_MAX    5

  /* 从机命令执行任务的最大个数，该值需要比用户注册的任务个数大 */
  #define   LIN_SLAVE_EXECUTE_TASK_NUM_MAX     5


  /* 0 为 GPIOA 或 1 为 GPIOB */
  #define   LIN_PORT                                    0

  /* 1 为 USART1 或 2 为 USART2 */
  #define   LIN_SLAVE_USART_INSTANCE                    1

  /* 定义 Tx 引脚  */
  #define   LIN_SLAVE_USART_PIN_TX                      GPIO_PIN_9
  /* 配置 Rx 引脚  */
  #define   LIN_SLAVE_USART_PIN_RX                      GPIO_PIN_10

  /* GPIO 配置复用选择 */
  #define   LIN_SLAVE_GPIO_MF_SEL                      GPIO_MF_SEL1
```

```
/* 定义 lin 波特率 */
#define    LIN_SLAVE_BAUD_RATE                    19200
```

编译并下载，利用逻辑分析仪抓取波形，通道 1 为主机发来的 header 或 response，通道 2 为从机响应的 response，通道 3 为蓝色 LED 灯电平翻转信号。如下图所示：





免责声明和版权公告

本文档中的信息，包括供参考的 URL 地址，如有变更，恕不另行通知。

本文档可能引用了第三方的信息，所有引用的信息均为“按现状”提供，芯海科技不对信息的准确性、真实性做任何保证。

芯海科技不对本文档的内容做任何保证，包括内容的适销性、是否适用于特定用途，也不提供任何其他芯海科技提案、规格书或样品在他处提到的任何保证。

芯海科技不对本文档是否侵犯第三方权利做任何保证，也不对使用本文档内信息导致的任何侵犯知识产权的行为负责。本文档在此未以禁止反言或其他方式授予任何知识产权许可，不管是明示许可还是暗示许可。

Wi-Fi 联盟成员标志归 Wi-Fi 联盟所有。蓝牙标志是 Bluetooth SIG 的注册商标。

文档中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。

芯海科技
CHIPSEA

股票代码:688595