RT1170 LIN demo Code Flow Introduction

ConstYu

LIN Stack关键函数列表

```
/*!This timer returns the count of nanoseconds between two consequencing bits.
 * @param ns number of nanoseconds between two bits, return parameter
 返回两个连续bit的ns值，按照最大20KHZ，每个bit最小宽度是50ns，最大宽度是416ns@2400Hz*/
void timerGetTimeIntervalCallback0(uint32_t *ns);
//计算寄存器值for支持的波特率
void CalculateBaudrates(uint32_t instance, uint32_t clocksSource,
lin_baudrate_values_t *baudRatesVals);
//处理frame id，如果节点是publisher，响应被发出。如果节点是subscriber，等待响应， id表示帧
ID
void SlaveProcessId(uint8_t instance, uint8_t id);
//设置总线activity超时
void SlaveTimeoutService(uint8_t instance);
//处理底层错误
void SlaveHandleError(uint8_t instance, lin_event_id_t event_id, uint8_t id);
//更新信号，状态，以及 response发出后的Flag
void SlaveUpdateTx(uint8_t instance, uint8_t id);
//更新 response发出后的Flag数组
void SlaveUpdateTxFlags(uint8_t instance, uint8_t id);
//更新response接收到后信号，状态，以及Flag
void SlaveUpdateRx(uint8_t instance, uint8_t id);
//处理无条件帧
void SlaveProcesUncdFrame(uint8_t instance, uint8_t id, uint8_t type);
//从RAM Table中返回数组ID的index
static inline uint8_t SlaveGetFrameIndex(uint8_t instance, uint8_t id);
//从给定的波特率列表中返回波特率的index
static inline uint32_t CheckIndex(uint32_t baudrate);

 *******************************************************************************/
/* 定时器的counter值存储， storage for timer counter */
uint16_t timerCounterValue[2] = {0u};
/*定时器溢出次数 number of timer overflow */
volatile uint32_t timerOverflowInterruptCount = 0u;
/* buffer handling messages between lower and higher level communication */
static uint8_t g_linResponseBuffer[LIN_NUM_OF_IFCS][10];
/* 支持的自动波特率，最低2400 supported baudrate for autobaudrate feature */
uint32_t baudRateArray[LIN_NUM_OF_SUPP_BAUDRATES] =
{2400,4800,9600,14400,19200};
```

```
lin_baudrate_values_t g_baudRatesValues[LIN_NUM_OF_SUPP_BAUDRATES];
/* 最大的header时间--- maximal header duration time */
static volatile  uint16_t linMaxHeaderTimeoutVal[LIN_NUM_OF_IFCS];
/* 最大的响应responce 时间--- maximal response duration time */
static uint16_t linMaxResTimeoutVal[LIN_NUM_OF_IFCS];
```

LIN 数据传输调度表

```
//slave端
/*@brief Informations of frame */
typedef struct
{
    lin_frame_type frm_type;/*!< Frame information (unconditional or event
triggered) */
    uint8_t frm_len;                 /*!< 帧长度 */
    lin_frame_response frm_response; /*!< 收到PID后如何响应，作为发布节点还是收听节点*/
    uint8_t frm_offset;              /*!< frame buffer偏移地址 */
    uint8_t flag_offset;             /*!< Flag byte offset in flag buffer */
    uint8_t flag_size;               /*!< Flag size in flag buffer */
    uint32_t delay;                  /*!< Frame delay */
    const uint8_t *frame_data_ptr;   /*!< List of Signal to which the frame is
associated and its offset */
} lin_frame_struct;

/*@brief Protocol configuration structure */
typedef struct
{
    lin_protocol_handle protocol_version;     /*!< Protocol version */
    lin_protocol_handle language_version;     /*!< Language version */
    uint8_t number_of_configurable_frames;    /*!< 诊断帧之外的帧的个数*/
    uint8_t frame_start;                      /*!< 报文帧列表的起始index */
    const lin_frame_struct *frame_tbl_ptr;    /*!< 报文帧列表，见下面的结构体 */
    const uint16_t *list_identifiers_ROM_ptr; /*!< ROM中的报文帧列表的ID,是个数组 */
    uint8_t *list_identifiers_RAM_ptr;        /*!< RAM中的报文帧列表的ID,是个数组 */
    uint16_t max_idle_timeout_cnt;            /*!< Max Idle timeout counter */
    uint16_t max_message_length;              /*!< 最大的message长度*/
} lin_protocol_user_config_t;                 /* defined in lin_cfg.c */

/***********************************Slave Frame table
**********************************/
static const lin_frame_struct lin_frame_tbl[LIN_NUM_OF_FRMS] ={
//LIN_NUM_OF_FRMS=6
    { LIN_FRM_UNCD, 8, LIN_RES_SUB, 0, 0, 1, 10, 0 }
    ,{ LIN_FRM_UNCD, 5, LIN_RES_PUB, 8, 1, 1, 10, 0 }
    ,{ LIN_FRM_UNCD, 8, LIN_RES_PUB, 0, 2, 1 ,10,   }
   ,{ LIN_FRM_UNCD, 6, LIN_RES_SUB, 13, 6, 1 ,10, 0 }
   ,{ LIN_FRM_DIAG, 8, LIN_RES_SUB, 0, 0, 0 ,2, 0 }
   ,{ LIN_FRM_DIAG, 8, LIN_RES_PUB, 0, 0, 0 ,2, 0 }
}; //5=长度，8=framebuffer偏移，1=flagbuffer偏移，1=Flag size的大小，10=帧延迟

/* definition and initialization of signal array */
uint8_t g_lin_flag_handle_tbl[LIN_FLAG_BUF_SIZE] = { 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,0x00 };                                          //
LIN_FLAG_BUF_SIZE=7
uint8_t  g_lin_frame_data_buffer[LIN_FRAME_BUF_SIZE] =  //LIN_FRAME_BUF_SIZE=19
{'L','I','N',' ','D','E','M','O','*','*','*','*','*','M','A','S','T','E','R'};
/*ID 头*/
```

```
static uint8_t LIO_lin_configuration_RAM[LIO_LIN_SIZE_OF_CFG]= {0x00, 0x30,
0x33, 0x36, 0x2D, 0x3C, 0x3D ,0xFF};        //LIO_LIN_SIZE_OF_CFG=8
const uint16_t  LIO_lin_configuration_ROM[LIO_LIN_SIZE_OF_CFG]= {0x0000, 0x30,
0x33, 0x36, 0x2D, 0x3C, 0x3D ,0xFFFF};
/* Size of configuration in ROM and RAM used for interface: LIO */
#define LIO_LIN_SIZE_OF_CFG  8U
```

LIN_LPUART_IRQHandler中断调度流程

LIN_LPUART_IRQHandler函数中，先去检测是否是Break中断

- 如果检测到Break(一般是指Slave)，调用LIN_LPUART_ProcessBreakDetect函数，该函数中判断Node状态是不是LIN_NODE_STATE_SLEEP_MODE；(以下是LIN_LPUART_ProcessBreakDetect函数的处理)
    - 如果是Sleep模式，调用LIN_LPUART_GotoIdleState，将Node状态状态**修改为Idle状态**，并**关闭RX边沿中断**，保留RX接收，帧错误以及Break中断；
    - 如果不是Sleep模式，**关闭Break中断**，置位LIN bus为busy状态。如果当前节点是Master且节点状态是LIN_NODE_STATE_SEND_BREAK_FIELD，把状态更新到LIN_NODE_STATE_SEND_PID，并发送0x55。如果当前是Slave节点，触发LIN_RECV_BREAK_FIELD_OK事件的callback，并把状态**修改为LIN_NODE_STATE_RECV_SYNC**；
    - 总结：LIN_LPUART_ProcessBreakDetect函数判断如果是Sleep状态修改为Idle状态，如果是非sleep状态变更到LIN_NODE_STATE_RECV_SYNC状态；
- 如果不是Break进中断(接收到数据RX Full或者Frame Error错误)，
    - 如果是RX边沿中断，并且节点是Sleep状态，去判断唤醒信号LIN_LPUART_CheckWakeupSignal。
    - 如果是错误中断，清理错误，并根据当前状态做进一步处理，如果是LIN_NODE_STATE_SEND_DATA也就是数据发送过程中，调用Callback处理LIN_FRAME_ERROR。如果是非阻塞，且当前状态是接收数据过程中，同样调用Callback处理LIN_FRAME_ERROR。然后将节点调用LIN_LPUART_GotoIdleState，将Node状态状态**修改为Idle状态**，并**关闭RX边沿中断**，保留RX接收，帧错误以及Break中断；
    - 如果是RX data中断，读取数据，然后调用LIN_LPUART_ProcessFrame(instance, tmpByte);进行进一步处理发送或者接收数据。

  LIN_LPUART_GoToSleepMode: 函数会关闭Break中断，RX接收中断，帧错误中断，保留RX边沿中断;

  LIN_LPUART_GotoIdleState: 函数会打开Break中断，RX接收中断，帧错误中断，关闭RX边沿中断;

```
#define DEMO_LIN_IRQHandler LPUART0_LPUART1_IRQHandler

/*! This interrupt routine handles LIN bus low level communication */
void DEMO_LIN_IRQHandler(void)
{
    LIN_IRQHandler(MASTER_INSTANCE);
}

void LIN_IRQHandler(uint8_t instance)
{
    if(instance < FSL_FEATURE_SOC_LPUART_COUNT)
    {
        LPUART_Type *base = g_linLpuartBase[instance];
        /* Call lower level function */
```

```
        LIN_LPUART_IRQHandler(base);
    }
}

void LIN_LPUART_IRQHandler(LPUART_Type *base)
{
    uint8_t tmpByte = 0xFFU;

    uint32_t instance = LIN_LPUART_GetInstance(base);

    /* Get the current LIN state of this LPUART instance. */
    lin_state_t *linCurrentState = g_linStatePtr[instance];

    /* Check RX Input Active Edge interrupt enable */
    bool activeEdgeIntState = (bool)(base->BAUD & (1 <<
LPUART_BAUD_RXEDGIE_SHIFT));

    /* If LIN break character has been detected. */
    if (LIN_LPUART_GetStatusFlags(base) & kLPUART_LinBreakFlag)
    {
        LIN_LPUART_ProcessBreakDetect(instance);
    }
    else
    {
        /* If LPUART_RX Pin Active Edge has been detected. */
        if ((LIN_LPUART_GetStatusFlags(base) & kLPUART_RxActiveEdgeFlag) &&
activeEdgeIntState)
        {
            /* Clear LPUART_RX Pin Active Edge Interrupt Flag. Write 1 to clear
*/
            LIN_LPUART_ClearStatusFlags(base, kLPUART_RxActiveEdgeFlag);

            /* Check if the node is in SLEEP MODE */
            /* If yes, then check if a wakeup signal has been received */
            if (linCurrentState->currentNodeState == LIN_NODE_STATE_SLEEP_MODE)
            {
                LIN_LPUART_CheckWakeupSignal(instance);
            }
        }
        else
        {
            /* If Framing Error has been detected Write 1 to clear */
            if (LIN_LPUART_GetStatusFlags(base) & kLPUART_FramingErrorFlag)
            {
                /* Clear Framing Error Interrupt Flag */
                LIN_LPUART_ClearStatusFlags(base, kLPUART_FramingErrorFlag);

                /* Read dummy to clear LPUART_RX_DATA_REG_FULL flag */
                LIN_LPUART_ReadByte(base, &tmpByte);
                /* Set current event id to LIN_FRAME_ERROR */
                linCurrentState->currentEventId = LIN_FRAME_ERROR;

                if (linCurrentState->currentNodeState ==
LIN_NODE_STATE_SEND_DATA)
                {
                    /* Callback function to handle Framing Error Event */
                    if (linCurrentState->Callback != NULL)
                    {
```

```
                        linCurrentState->Callback(instance, linCurrentState);
                    }
                }

                if (linCurrentState->isRxBlocking == false)
                {
                    if (linCurrentState->currentNodeState ==
LIN_NODE_STATE_RECV_DATA)
                    {
                        /* Callback function to handle Framing Error Event */
                        if (linCurrentState->Callback != NULL)
                        {
                            linCurrentState->Callback(instance,
linCurrentState);
                        }
                    }
                }

                /* Clear Bus busy Flag */
                linCurrentState->isBusBusy = false;
                /* Change node's state to IDLE */
                (void)LIN_LPUART_GotoIdleState(base);
            }
            else
            {
                if (LIN_LPUART_GetStatusFlags(base) & kLPUART_RxDataRegFullFlag)
                {
                    LIN_LPUART_ClearStatusFlags(base,
kLPUART_RxDataRegFullFlag);
                    /* Get data from Data Register & Clear
LPUART_RX_DATA_REG_FULL flag */
                    LIN_LPUART_ReadByte(base, &tmpByte);
                    /* Process data in Data Register while receive, send data */
                    LIN_LPUART_ProcessFrame(instance, tmpByte);
                }
            } /* End else: if (LIN_LPUART_GetStatusFlags(base) &
kLPUART_FramingErrorFlag) */
        }     /* End else: if ((LIN_LPUART_GetStatusFlags(base) &
kLPUART_RxActiveEdgeFlag) && activeEdgeIntState) */
    }         /* End else: if (LIN_LPUART_GetStatusFlags(base) &
kLPUART_LinBreakFlag) */

    /* Get status RX overrun flag */
    if (LIN_LPUART_GetStatusFlags(base) & kLPUART_RxOverrunFlag)
    {
        /* Clear overrun flag */
        LIN_LPUART_ClearStatusFlags(base, kLPUART_RxOverrunFlag);
    }
} /* End void LIN_LPUART_DRV_IRQHandler(uint32_t instance) */
```

Master节点数据帧发送流程：

初始化后Node处于Sleep状态，TPM中断中循环调用MasterScheduleTick函数，首先判断Frame delay延迟，如果调度时间到，接着判断节点是否是LIN_NODE_STATE_SLEEP_MODE，如果是不做处理，如果不是【Sleep模式，而通过**自身发送0X80的RX边沿触发中断**进入Idle模式(idle状态会关闭RX边沿中断，打开其他的所有中断)】，该调度函数先去获取数组中的Frame ID，然后调用LIN_MasterSendHeader函数发送Break帧头，而且增加当前的Frame ID 指针，并将当前的节点状态修改为LIN_NODE_STATE_SEND_BREAK_FIELD状态。由于**自身Break帧的发出，会触发Break的中断**，对应进行LIN_LPUART_ProcessBreakDetect的处理，该函数会关闭Break的中断，如果是master会发送将当前的节点状态修改为LIN_NODE_STATE_SEND_PID。

发送的中断触发

Break发送：MasterScheduleTick函数的LIN_MasterSendHeader(MASTER_INSTANCE, current_id);发送一个break，该函数把**当前状态修改成LIN_NODE_STATE_SEND_BREAK_FIELD**，并调用LIN_LPUART_QueueBreakChar(base); 实际**发送Break**, 计算好PID先不发送。

SYNC发送：Break发送后自己也会收到，触发LIN_LPUART_IRQHandler中的Break中断，调用LIN_LPUART_ProcessBreakDetect(instance); 该函数会把**当前状态修改成LIN_NODE_STATE_SEND_PID**;并**发送0x55同步段。**

PID发送：**同步段0x55发送后自己也会收到**，触发IN_LPUART_IRQHandler中的RX DATA接收中断，其中再调用LIN_LPUART_ProcessFrame(instance, tmpByte); 的**case LIN_NODE_STATE_SEND_PID**中的LIN_LPUART_ProcessFrameHeader(instance, tmpByte);函数中的**LIN_NODE_STATE_SEND_PID**，把**当前状态修改成LIN_NODE_STATE_RECV_PID**，并**把PID发送出去。**

数据段发送：PID段发送后自己也会收到，触发触发IN_LPUART_IRQHandler中的RX DATA接收中断，其中再调用LIN_LPUART_ProcessFrame(instance, tmpByte); 的**case LIN_NODE_STATE_RECV_PID**中的LIN_LPUART_ProcessFrameHeader(instance, tmpByte);函数中的**LIN_NODE_STATE_RECV_PID**，判断接收到的PID是不是发送出去的PID？如果是，把**事件ID linCurrentState->currentEventId 修改为LIN_PID_OK;**，然后**调用LIN_PID_OK的处理callback,即CallbackHandler。**LIN_PID_OK事件对应会调用MasterProcessId(instance, linCurrentState->currentId); 该事件首先会从RAM table中获取一个ID的index(第一次发送的PID也是从RAM table获取的，所以拿这个ID再反推回去得到它在整个RAM table中的index)。拿到ID的index后就能从Frame buffer中获取数据的内容了，(帧ID 和帧数据是两个独立的数组，类似以下截图)。

有了数据后，根据下表的帧类型，长度，发布还是收听，调用MasterProcesUncdFrame(instance, id, LIN_MAKE_UNCONDITIONAL_FRAME);去组成帧，然后调用LIN_SetResponse(instance, &(prot_state_ptr->response_buffer_ptr[0]), response_length, lin_max_frame_res_timeout_val); 指定发送的长度(根据下表得到)，超时时间，buffer内容发送数据段。

```
/*********************************  Frame table **********************************/
static const lin_frame_struct lin_frame_tbl[LIN_NUM_OF_FRMS] = {
    { LIN_FRM_UNCD, 8U, LIN_RES_PUB, 0U, 0U, 1U, 15U, 0U }
   ,{ LIN_FRM_UNCD, 5U, LIN_RES_SUB, 8U, 1U, 1U, 15U, 0U }
   ,{ LIN_FRM_UNCD, 8U, LIN_RES_SUB, 0U, 2U, 1U, 15U, 0U }
  ,{ LIN_FRM_UNCD, 6U, LIN_RES_PUB, 13U, 6U, 1U ,15U, 0U }
  ,{ LIN_FRM_DIAG, 8U, LIN_RES_PUB, 0U, 0U, 0U ,5U, g_lin_go_to_sleep_buffer }
  ,{ LIN_FRM_DIAG, 8, LIN_RES_SUB, 0U, 0U, 0U ,5U, 0U }
};

static uint8_t LI0_lin_configuration_RAM[LI0_LIN_SIZE_OF_CFG]= {0x00, 0x30, 0x33, 0x36, 0x2D, 0x3C, 0x3D ,0xFF};
const uint16_t  LI0_lin_configuration_ROM[LI0_LIN_SIZE_OF_CFG]= {0x0000, 0x30, 0x33, 0x36, 0x2D, 0x3C, 0x3D ,0xFFFF};
```

LIN_SetResponse内部调用的是LIN_SendFrameData(instance, response_buff, response_length); 其**内部还是每次只发送一个字节** LIN_LPUART_WriteByte(base, *linCurrentState->txBuff); 此处把**节点状态修改为LIN_NODE_STATE_SEND_DATA。**第一个**数据发送后自己也会收到**，所以会再次进入LIN_LPUART_IRQHandler中断函数，读出发出的数据，然后调用

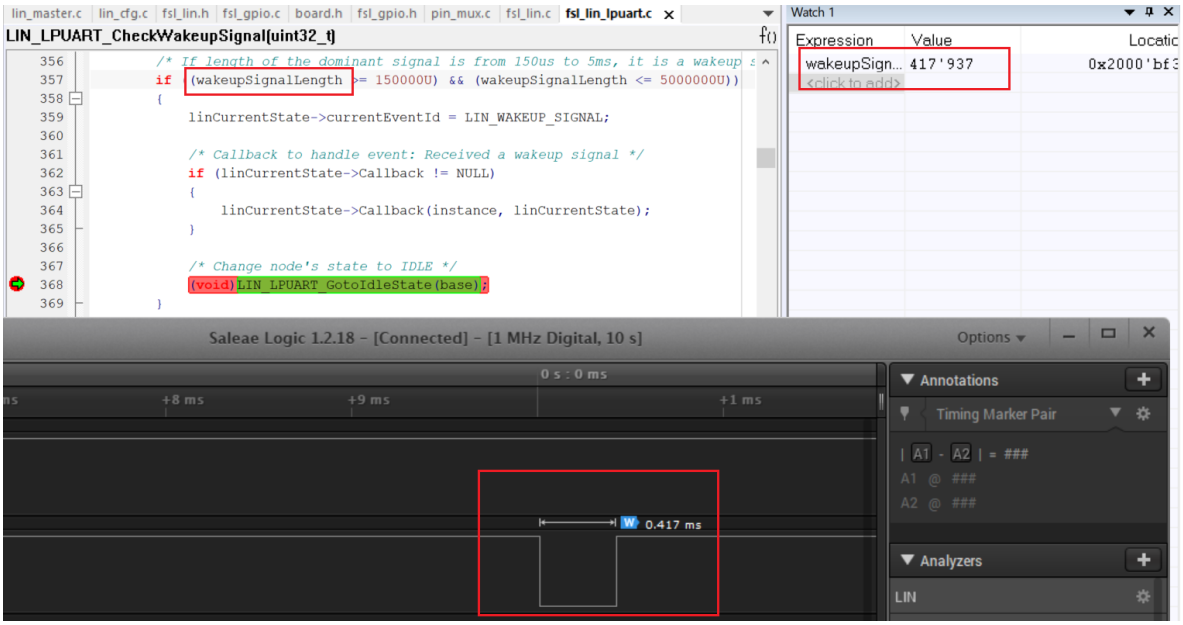
LIN_LPUART_ProcessFrame(instance, tmpByte); 里面调用LIN_NODE_STATE_SEND_DATA事件的Case， 即LIN_LPUART_ProcessSendFrameData(instance, tmpByte);在该函数中，读取TX data empty标志判断发送完，并根据回读到的字节判断当前的checksum是否正确，如果OK，继续发送下一字节数据。

判断到数据长度=指定长度后，把当前节点状态修改为LIN_NODE_STATE_SEND_DATA_COMPLETED，回调事件事件修改为LIN_TX_COMPLETED，并触发发送回调。之后关闭接收数据中断，清除总线的busy状态，让总线进入idle状态。至此结束整个过程。

Master节点上电后代码执行流程

Master在按键后会发送wakeup信号0X80，7bit 0，按照19200波特率，理论值是1/19200x7=365us。

```
/* Send a wakeup signal */
LIN_SendWakeupSignal(MASTER_INSTANCE);
```



如果满足条件LIN wakup信号在150us-5ms之间，LIN就会进入Idle状态。在LIN_LPUART_GotoIdleState函数中，会打开Break中断，RX接收中断，帧错误中断，关闭RX边沿中断，并且 把节点状态修改为linCurrentState->currentNodeState = LIN_NODE_STATE_IDLE;

> **LIN_LPUART_GoToSleepMode**: 函数会关闭Break中断，RX接收中断，帧错误中断，保留RX边沿中断;
>
> **LIN_LPUART_GotoIdleState**:　函数会打开Break中断，RX接收中断，帧错误中断，关闭RX边沿中断;

然后在DEMO_TPM_IRQHandler函数中每隔5ms执行一次 MasterScheduleTick(); 在该函数中会去判断是否已经退出LIN_NODE_STATE_SLEEP_MODE，如果已经退出，会先从list_identifiers_RAM_ptr数组获取frame ID，然后根据这个frame ID去获取帧延迟 frame_delay_count，帧超时时间 frame_timeout_cnt。然后调用LIN_MasterSendHeader(MASTER_INSTANCE, current_id);函数发送帧头，特别强调此处只会发送Break，参数ID只是在该函数中参与运算即将发送数据的CRC，填充到linCurrentState->currentId 结构体中。如果LIN_MasterSendHeader发送出错，MCU会重新进入会进入sleep状态。

```
void MasterScheduleTick(void)
{
    lin_status_t status = LIN_SUCCESS;
    uint8_t current_id;
```

```
    lin_protocol_state_t *prot_state_ptr =
&g_lin_protocol_state_array[HARDWARE_INSTANCE];
    const lin_protocol_user_config_t *prot_user_config_ptr =
&g_lin_protocol_user_cfg_array[HARDWARE_INSTANCE];
    lin_state_t * linCurrentState = g_linStatePtr[MASTER_INSTANCE];
    static uint32_t prev_error_in_transfer = 0U;
    static int curr_id_index = 1;
    static int32_t frame_delay_count = 0;

    /* check if frame delay runs out */
    if(frame_delay_count<=0)
    {

        /* check for go to sleep flag */
        if(prot_state_ptr->go_to_sleep_flg == (bool)1U)
        {
            /* go to sleep mode */
            LIN_GoToSleepMode(MASTER_INSTANCE);
            /* clear sleep mode flag */
            prot_state_ptr->go_to_sleep_flg = (bool)0U;
        }
        else if(linCurrentState->currentNodeState != LIN_NODE_STATE_SLEEP_MODE)
        {
            /* check for error occured during last tranfer */
            if( (prot_state_ptr->error_in_response != prev_error_in_transfer) )
            {
                /* if previous transfer was not succesfull, try again */
                if(curr_id_index > 1U)
                {
                    curr_id_index--;
                }
            }
            /* get frame id */
            current_id = prot_user_config_ptr-
>list_identifiers_RAM_ptr[curr_id_index];

            if (0xFFU != current_id)
            {
                /* get delay of frame to be sent */
                frame_delay_count = prot_user_config_ptr-
>frame_tbl_ptr[curr_id_index-1].delay;
                prot_state_ptr->frame_timeout_cnt =
linMaxResTimeoutVal[HARDWARE_INSTANCE] +
linMaxHeaderTimeoutVal[HARDWARE_INSTANCE];
                /* send a frame header */
                status = LIN_MasterSendHeader(MASTER_INSTANCE, current_id);
                /* check if was succesfull */
                if(status != LIN_SUCCESS)
                {
                    /* go to sleep mode */
                    LIN_GoToSleepMode(MASTER_INSTANCE);
                }
                prev_error_in_transfer = prot_state_ptr->error_in_response;
            }
            curr_id_index++;
            /* first and last index of RAM_configuration list are not valid */
            if(curr_id_index >=  prot_user_config_ptr-
>number_of_configurable_frames)
```

```
            {
                curr_id_index = 1;
            }
        }
        else
        {
            ;/* do nothing */
        }
    }
    else
    {
        /* decrease frame delay count */
        frame_delay_count--;
    }

}
```

在发送完Break后，自己会收到进入LIN_IRQHandler中断，在该函数中调用LIN_LPUART_ProcessBreakDetect发送sync场0x55，并更改LIN状态到linCurrentState->currentNodeState = **LIN_NODE_STATE_SEND_PID**。

Sync 0x55 自己又会收到，中断中会去调用LIN_LPUART_ProcessFrame(instance, tmpByte); 在该函数中调用LIN_LPUART_WriteByte(base, linCurrentState->currentPid);发送 PID，并把自己的状态修改为linCurrentState->**currentNodeState** = **LIN_NODE_STATE_RECV_PID**;

PID 自己又会收到，会先判断下刚才发出的PID是否正确，linCurrentState->**currentEventId** = LIN_PID_OK； 然后调用linCurrentState->Callback(instance, linCurrentState); 也就是CallbackHandler，处理LIN_PID_OK，即MasterProcessId，其中会去判断帧的类型，如果是LIN_RES_PUB，会调用LIN_SetResponse，该函数会再调用LIN_SendFrameData(instance, response_buff, response_length); 会继续把状态做个更改，linCurrentState->**currentNodeState** = LIN_NODE_STATE_SEND_DATA; linCurrentState->**currentEventId** = LIN_NO_EVENT；发送第一个字节。

之后MCU会再次进入中断，自己又会收到自己发送的数据段数据，进入LIN_LPUART_ProcessFrame，进一步是LIN_LPUART_ProcessSendFrameData，在该函数中调用一次 LIN_LPUART_WriteByte(base, *linCurrentState->txBuff);发送数据。然后依次类推按照上面的流程发送指定长度的数据，发送完成后更改节点状态和event状态，linCurrentState->currentEventId = **LIN_TX_COMPLETED** ;linCurrentState->currentNodeState = **LIN_NODE_STATE_SEND_DATA_COMPLETED**;并且关闭UART的接收中断，调用linCurrentState->Callback去处理发送完成的callback，最后进入idle状态。

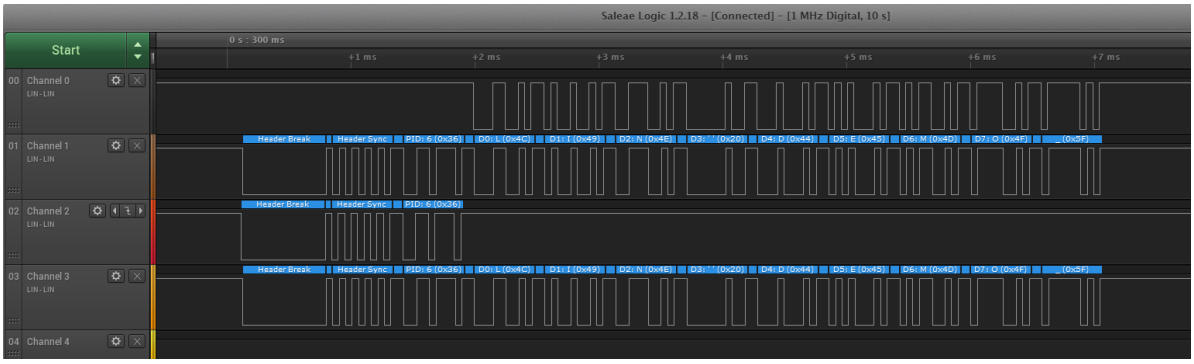
上面提到的linCurrentState->Callback会去做什么呢？答案是更新protocol state，MasterUpdateTx，会增加prot_state_ptr->num_of_processed_frame++;增加成功传输的帧个数prot_state_ptr->num_of_successfull_frame++; 打印接收消息PrintBuffer(instance, id)；从RAM table中获取当前frame的index，frame_index = MasterGetFrameIndex(instance, id); 并更新TX flags，

```
void MasterUpdateTx(uint8_t instance, uint8_t id)
{
    uint8_t frame_index;
    lin_protocol_state_t *prot_state_ptr =
&g_lin_protocol_state_array[HARDWARE_INSTANCE];

    /* Set successful transfer */
    prot_state_ptr->successful_transfer = 1U;

    if (prot_state_ptr->num_of_processed_frame < 0xFFU)
    {
```

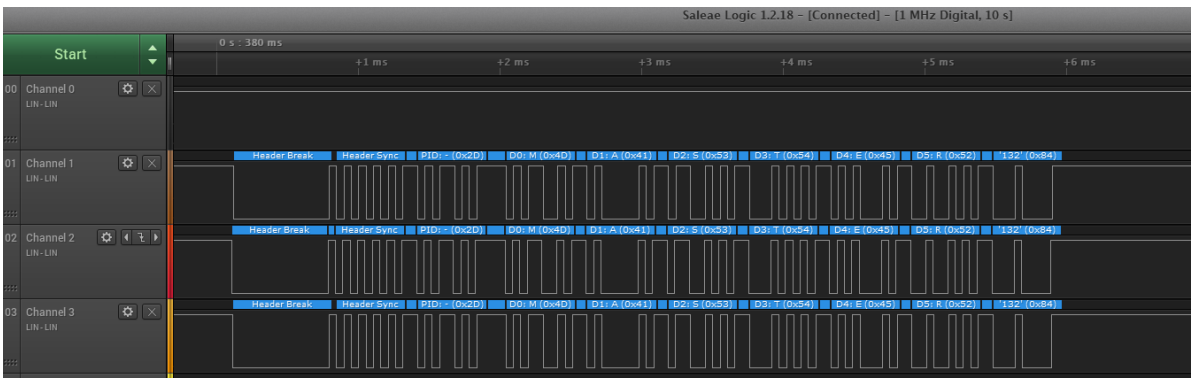
```
        /* increase a number of processed frames */
        prot_state_ptr->num_of_processed_frame++;
    }
    /* Check if frame contains error signal */
    if ((bool)1U == prot_state_ptr->transmit_error_resp_sig_flg)
    {
        /* Set error in response */
        prot_state_ptr->error_in_response = 0U;
        /* clear error flag */
        prot_state_ptr->transmit_error_resp_sig_flg = (bool)0U;
    }
    else
    {
        /* increase a number of succesfull frames */
        prot_state_ptr->num_of_successfull_frame++;
    }
    /* print received response */
    PrintBuffer(instance, id);
    /* get index of current frame from RAM table */
    frame_index = MasterGetFrameIndex(instance, id);
    /* update Tx flags */
    MasterUpdateTxFlags(instance, frame_index);
}
```

实际通讯波形

Master作为Subscribe角色时，发送Header，由Slave发送Respone



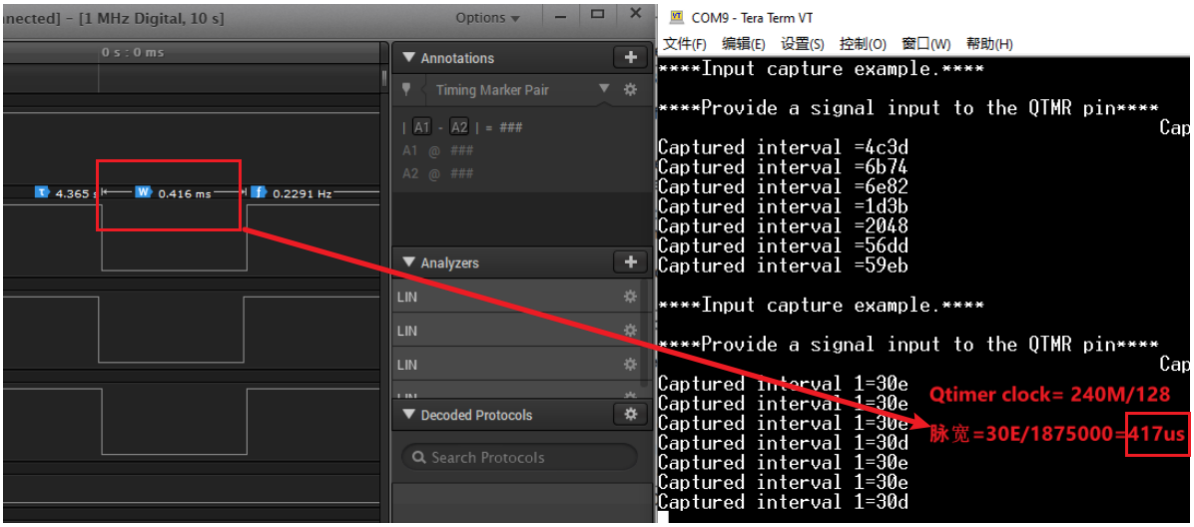
Master作为PUBLIC角色时，同时发送Header，以及Respone



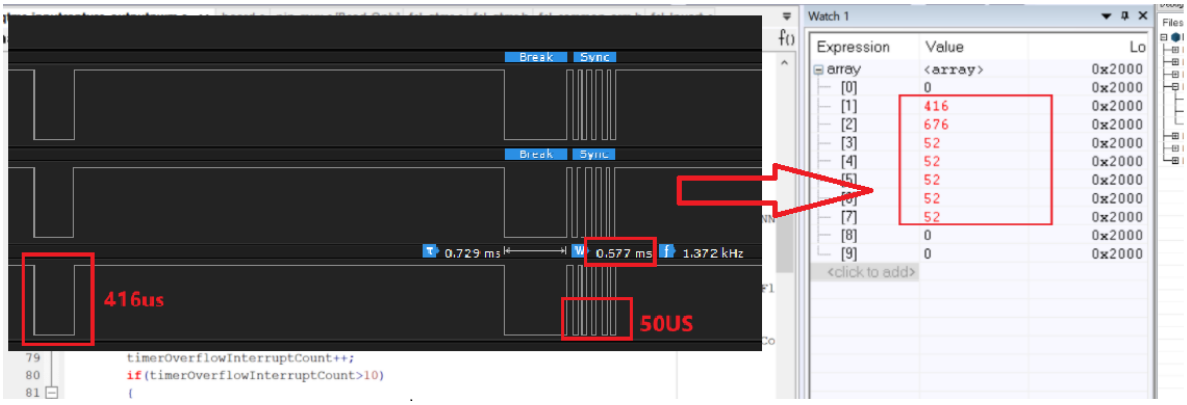
按照调度表依次发送LI0_lin_configuration_RAM数组定义的PID数据

> static uint8_t LI0_lin_configuration_RAM[LI0_LIN_SIZE_OF_CFG]= {0x00, 0x30, 0x33, 0x36, 0x2D, 0x3C, 0x3D ,0xFF};

Qtimer准确读取wake up信号的脉冲宽度



Slave使能Auto baud rate后读取到的每个脉冲宽度数据



免责声明: