



上海航芯

SHANGHAI AISINOCHIP

基于RT-Thread在ACM32F403上实现OBD

主要内容

CONTENTS



OBD简介

ACM32F403简介

- CAN模块简介

RT-Thread的CAN驱动适配

- CAN驱动框架简介
- CAN驱动适配

RT-Thread的ISO15765适配

- ISO15765-2规范介绍
- RT-Thread的ISO15765代码适配



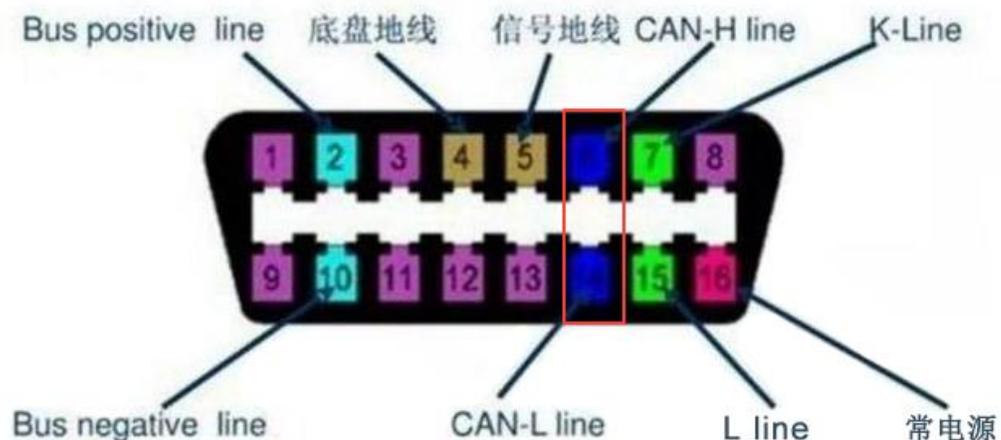
OBD简介

OBD是英文On-Board Diagnostics的缩写，中文翻译为车载自动诊断系统。这个系统将从发动机的运行状况随时监控汽车是否尾气超标，一旦超标，会马上发出警示。当系统出现故障时，故障(MIL)灯或检查发动机(Check Engine)警告灯亮，同时动力总成控制模块(PCM)将故障信息存入存储器，通过一定的程序可以将故障码从PCM中读出。



功能简介

- 🔗 读取数据流
- 🔗 查询故障码
- 🔗 清除故障码
- 🔗 ECU控制



OBD协议栈

本次我们主要关注物理层、数据链路层、网络层及传输层

物理层和数据链路层中涉及的规范:

ISO11898-1:

- [ISO - ISO 11898-1:2015 - Road vehicles — Controller area network \(CAN\) — Part 1: Data link layer and physical signalling](#)

ISO11898-2:

- [ISO - ISO 11898-2:2016 - Road vehicles — Controller area network \(CAN\) — Part 2: High-speed medium access unit](#)

ISO11898部分的内容由MCU的CAN模块实现，我们只需要使用其驱动接口即可。

网络层和传输层涉及的协议有:

ISO15765-2:

- [ISO - ISO 15765-2:2016 - Road vehicles — Diagnostic communication over Controller Area Network \(DoCAN\) — Part 2: Transport protocol and network layer services](#)

ISO15765-4:

- [ISO 15765-4:2021 — Road vehicles — Diagnostic communication over Controller Area Network \(DoCAN\) — Part 4: Requirements for emissions-related systems](#)

Table 1 — Enhanced and legislated on-board diagnostics specifications applicable to the OSI layers

OSI 7 layers ^a	Vehicle-manufacturer-enhanced diagnostics	Legislated OBD (on-board diagnostics)	Legislated WWH-OBD (on-board diagnostics)
Application (layer 7)	ISO 14229-1, ISO 14229-3	ISO 15031-5	ISO 27145-3, ISO 14229-1
Presentation (layer 6)	Vehicle manufacturer specific	ISO 15031-2, ISO 15031-5, ISO 15031-6, SAE J1930-DA, SAE J1979-DA, SAE J2012-DA	ISO 27145-2, SAE 1930-DA, SAE J1979-DA, SAE J2012-DA, SAE J1939-DA (SPNs), SAE J1939-73 Appendix A (FMIs)
Session (layer 5)	ISO 14229-2		
Transport protocol (layer 4)	ISO 15765-2	ISO 15765-2	ISO 15765-4, ISO 15765-2
Network (layer 3)			ISO 15765-4, ISO 11898-1
Data link (layer 2)	ISO 11898-1	ISO 11898-1	
Physical (layer 1)	ISO 11898-1, ISO 11898-2, ISO 11898-3, or vehicle manufacturer specific	ISO 11898-1, ISO 11898-2	ISO 11898-1, ISO 11898-2
			ISO 27145-4

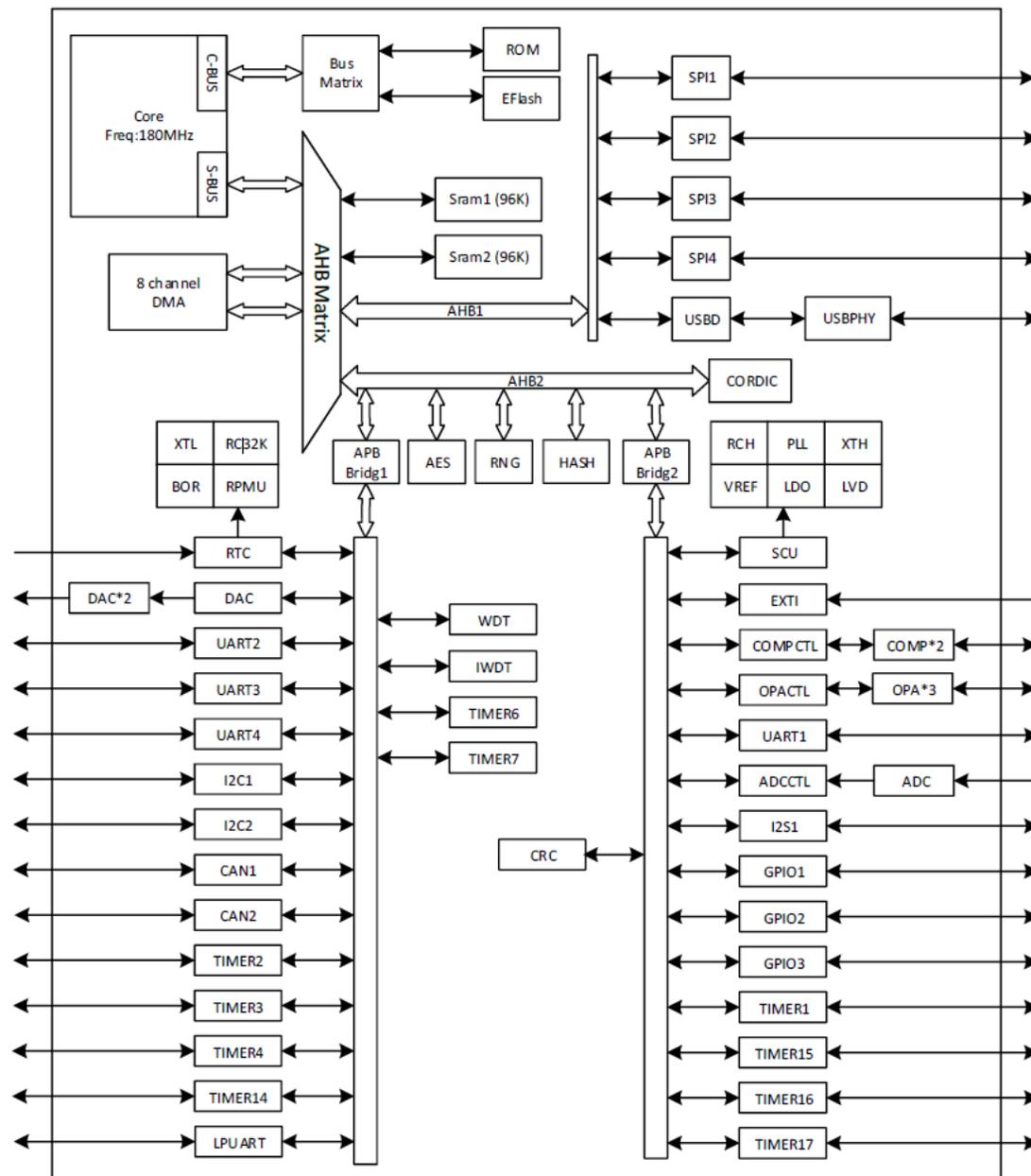
^a 7 layers according to ISO/IEC 7498-1 and ISO/IEC 10731



ACM32F403概览

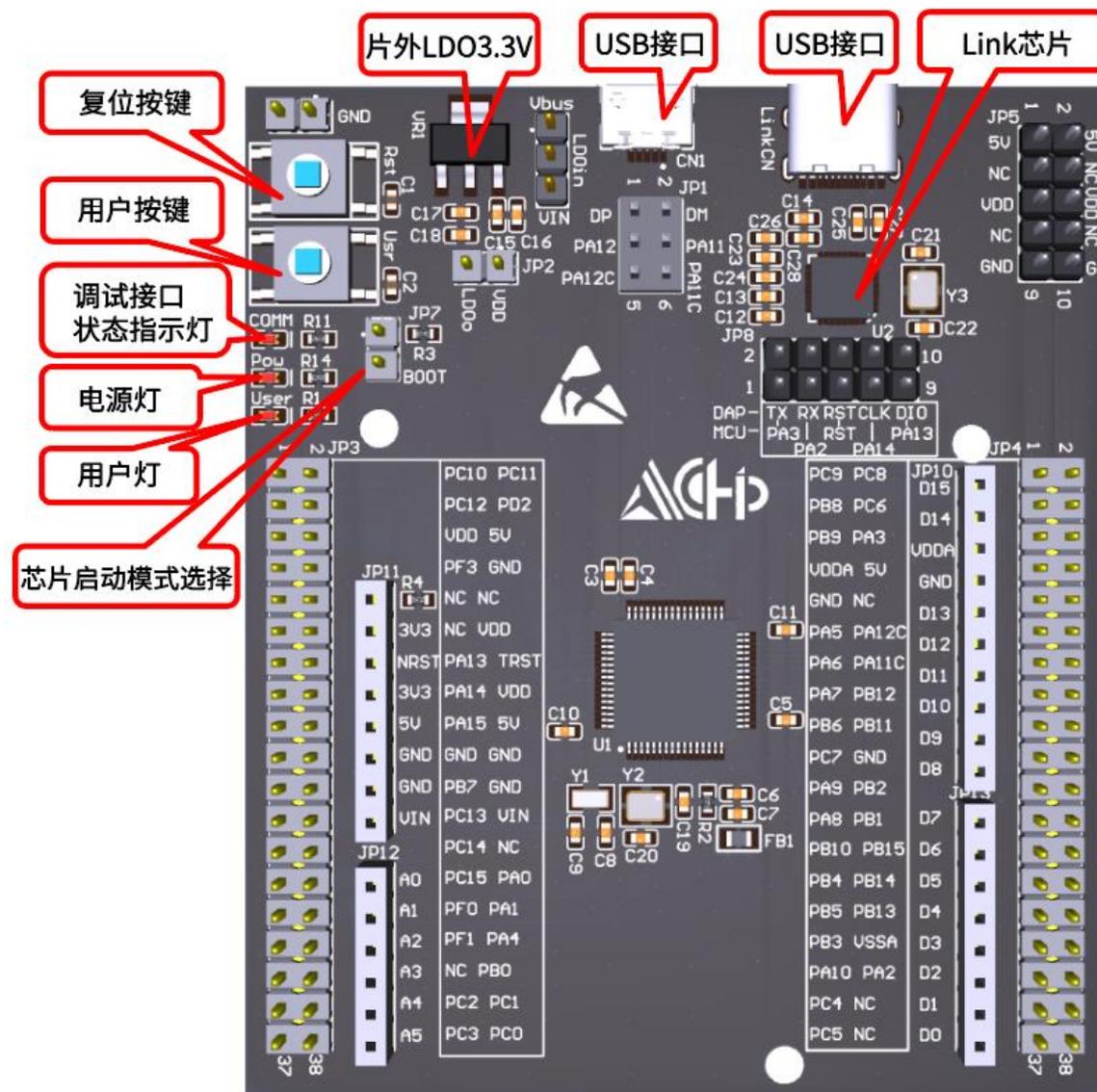
- ARMv8-M, Cortex-M33指令集, 180Mhz
- 支持DSP、FPU、MPU、CORDIC数学硬件加速
- AES、CRC、TRNG、HASH等算法模块
- 最大支持512KB的eFlash和最大192KB的SRAM, eFlash擦写次数10万次, 加密存储, 带加速器可实现0-wait程序执行
- 工业控制、智能家居、物联网、汽车电子

图 2-3 芯片系统架构图



ACM32F403 开发板

- ACM32F403RET7, LQFP64, 512KB eFlash, 192KB SRAM
- 提供了CMSIS-DAP方式下载、调试、USB虚拟串口打印功能
- 所有IO全部引出, 包含晶振占用的IO口

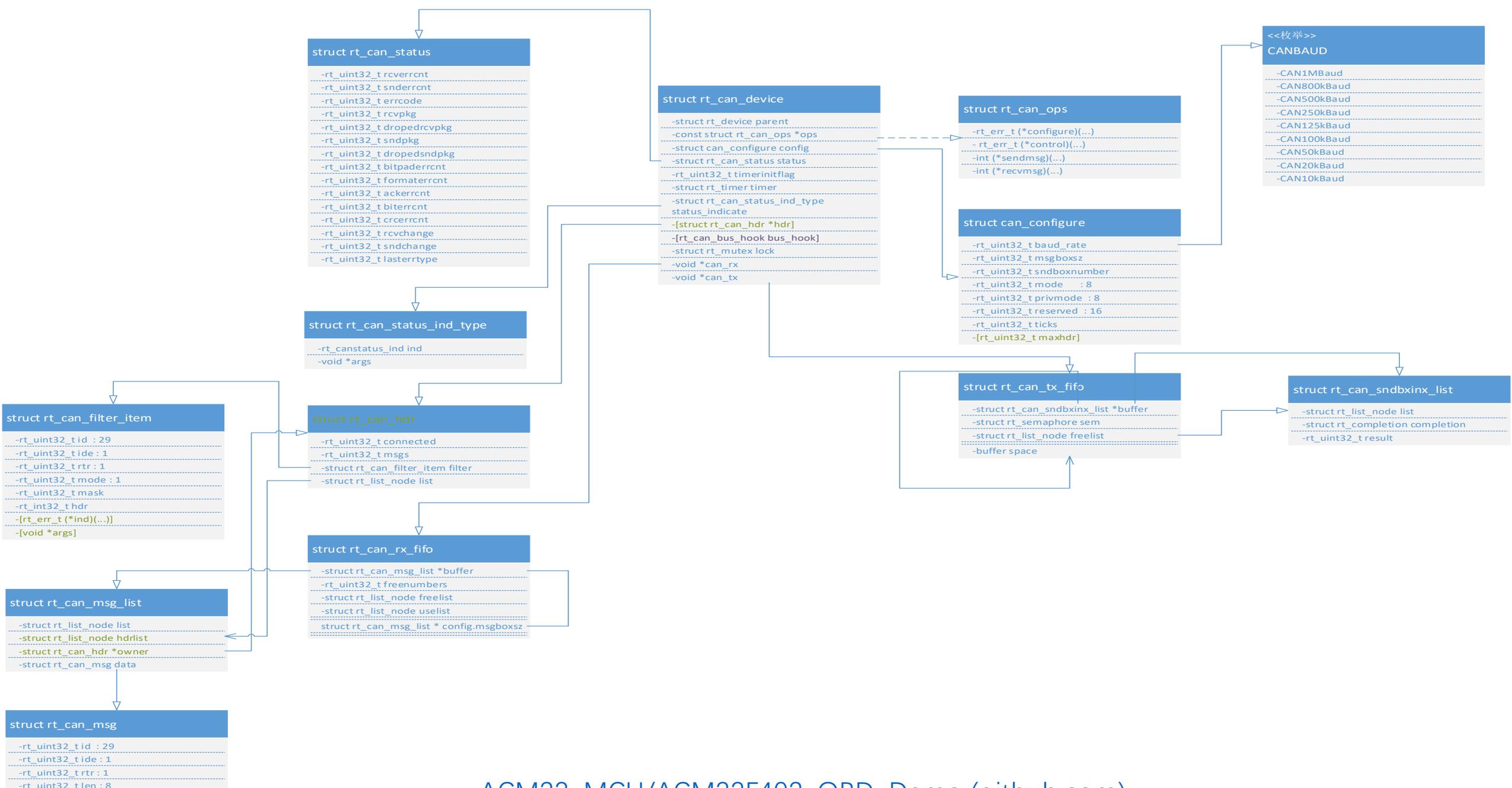


CAN介绍

- 支持CAN2.0A和CAN2.0B
- 支持125K到1M的波特率
- 64字节FIFO
- 支持热插拔
- 支持接收滤波
- 支持监听模式
- 记录仲裁失败后的bit位置
- 读写错误计数器
- 可编程的错误上限警告
- 支持CAN总线错误中断
- 支持自测模式
- 通过BOSCH CAN2.0测试

RT-Thread CAN驱动框架介绍

函数	描述
rt_device_find	查找设备
rt_device_open	打开设备
rt_device_read	读取数据
rt_device_write	写入数据
rt_device_control	控制设备
rt_device_set_rx_indicate	设置接收回调函数
rt_device_close	关闭设备



ACM32-MCU/ACM32F403-OBD-Demo (github.com)



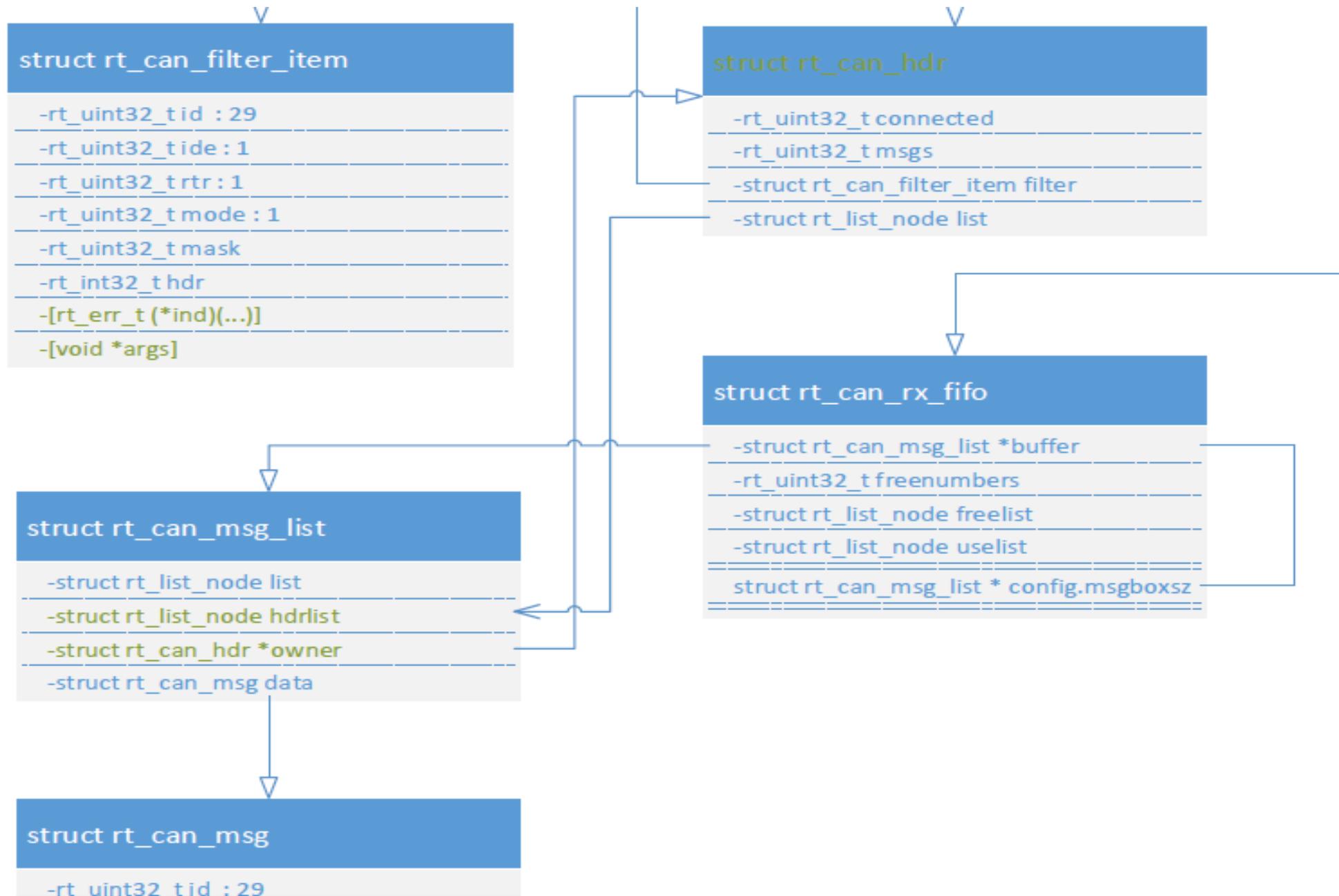
struct rt_can_ops

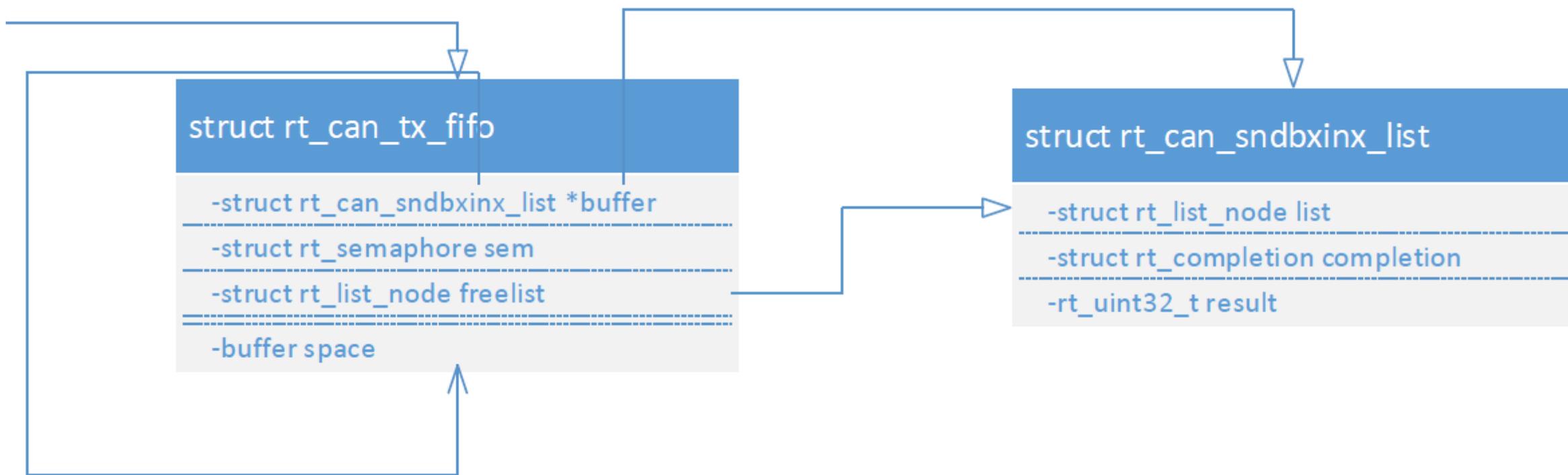
-rt_err_t (*configure)(...)
-rt_err_t (*control)(...)
-int (*sendmsg)(...)
-int (*recvmsg)(...)

struct can_configure

-rt_uint32_t baud_rate
-rt_uint32_t msgboxsz
-rt_uint32_t sndboxnumber
-rt_uint32_t mode : 8
-rt_uint32_t privmode : 8
-rt_uint32_t reserved : 16
-rt_uint32_t ticks
-[rt_uint32_t maxhdr]

-CAN500kBaud
-CAN250kBaud
-CAN125kBaud
-CAN100kBaud
-CAN50kBaud
-CAN20kBaud
-CAN10kBaud





驱动适配的工作内容

- board.h中为CAN模块增加配置宏定义，包含IO、中断的配置
- KConfig中为CAN模块增加配置开关，修改Sconscript增加HAL驱动支持
- **ACM32F403的bsp包中增加驱动文件并实现★**
- 重新生成工程编译测试，可以使用selftest模式进行

ACM32F403的bsp包中增加驱动并实现

- CAN驱动注册
- CAN中断实现
- CAN框架ops回调实现

```
#if defined(RT_USING_CAN)
#if defined(BSP_USING_CAN1) || defined(BSP_USING_CAN2)

static rt_err_t _configure(struct rt_can_device *can, struct can_configure *cfg)
+-- 3 lines: {-----

static rt_err_t _control(struct rt_can_device *can, int cmd, void *arg)
+-- 3 lines: {-----

static int _transmit(struct rt_can_device *can, const void *buf, rt_uint32_t boxno)
+-- 3 lines: {-----

static int _receive(struct rt_can_device *can, void *buf, rt_uint32_t boxno)
+-- 3 lines: {-----

static const struct rt_can_ops acm32_can_ops =
+-- 6 lines: {-----

int rt_hw_can_init(void)
+-- 3 lines: {-----
INIT_DEVICE_EXPORT(rt_hw_can_init);

void HAL_CAN_MspInit(CAN_HandleTypeDef *hcan)
+-- 2 lines: {-----

static void can_isr(rt_uint32_t idx)
+-- 2 lines: {-----

#ifdef BSP_USING_CAN1
void CAN1_IRQHandler(void)
+-- 10 lines: {-----
#endif

#ifdef BSP_USING_CAN2
void CAN2_IRQHandler(void)
+-- 10 lines: {-----
#endif

#endif /* defined(BSP_USING_CAN1) || defined(BSP_USING_CAN2) */
#endif /* RT_USING_CAN */
```

CAN驱动注册

- 定义CAN驱动对象结构体并为每个CAN模块定义一个对象
- 初始化好CAN驱动对象，包含设定好配置、设定过滤器等
- 注册对象到OS并把初始化函数导致到设备初始化表中

```
int rt_hw_can_init(void)
{
    struct can_configure cfg = CANDEFAULTCONFIG;

    cfg.ticks = 50;
#ifdef RT_CAN_USING_HDR
    cfg.maxhdr = 2;
#endif

    for(size_t i = 0; i < sizeof(can_config)/sizeof(can_config[0]); i++)
    {
        devObjs[i].config = can_config[i];
        devObjs[i].device.config = cfg;

        devObjs[i].filter.CAN_FilterMode = CAN_FilterMode_Single;
        devObjs[i].filter.CAN_FilterId1 = 0;
        devObjs[i].filter.CAN_FilterId2 = 0;
        devObjs[i].filter.CAN_FilterMaskId1 = 0xFFFFFFFF;
        devObjs[i].filter.CAN_FilterMaskId2 = 0xFFFFFFFF;

        rt_hw_can_register(&devObjs[i].device,
                          devObjs[i].config.name,
                          &acm32_can_ops,
                          RT_NULL);
    }

    return 0;
}
INIT_DEVICE_EXPORT(rt_hw_can_init);
```

CAN中断函数实现

- 为中断向量表中的每个CAN中断入口定义好函数
- 所有的中断操作都放入到了can_isr处理中，使用参数区分当前是哪个模块的中断
- 有相应的中断产生时调用CAN框架的中断服务函数

```
static void can_isr(rt_uint32_t idx)
-- 46 lines: {-----

#ifdef BSP_USING_CAN1
void CAN1_IRQHandler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    /* can interrupt service routine */
    can_isr(CAN1_INDEX);

    /* leave interrupt */
    rt_interrupt_leave();
}
#endif

#ifdef BSP_USING_CAN2
void CAN2_IRQHandler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    /* can interrupt service routine */
    can_isr(CAN2_INDEX);

    /* leave interrupt */
    rt_interrupt_leave();
}
#endif

#endif /* defined(BSP_USING_CAN1) || defined(BSP_USING_CAN2) */
#endif /* RT_USING_CAN */
```

CAN驱动对象定义及IO中断初始化

在board.h中有为CAN使用的IO和中断定义了一些宏定义，需要重写HAL_CAN_MspInit函数

```
void HAL_CAN_MspInit(CAN_HandleTypeDef *hcan)
{
    struct acm32_can_device *canObj;
    GPIO_InitTypeDef GPIO_InitStructure;

    RT_ASSERT(hcan != RT_NULL);

    /* get can object */
    canObj = rt_container_of(hcan, struct acm32_can_device, handle);

    /* Enable CAN clock */
    System_Module_Enable(canObj->config.enable_id);

    /* Initialization GPIO */
    GPIO_InitStructure.Pin = canObj->config.tx_pin;
    GPIO_InitStructure.Alternate=canObj->config.tx_alternate;
    GPIO_InitStructure.Pull=GPIO_PULLUP;
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    HAL_GPIO_Init(canObj->config.tx_port, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = canObj->config.rx_pin;
    GPIO_InitStructure.Alternate=canObj->config.rx_alternate;
    GPIO_InitStructure.Pull=GPIO_PULLUP;
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    HAL_GPIO_Init(canObj->config.rx_port, &GPIO_InitStructure);

    /* Initialization interrupt */
    NVIC_ClearPendingIRQ(canObj->config.irq_type);
    NVIC_SetPriority(canObj->config.irq_type, canObj->config.irq_priority);
    NVIC_EnableIRQ(canObj->config.irq_type);
}
```

CAN驱动框架ops实现

RT-Thread的CAN框架是通过调用CAN对象的ops中回调来操作实际的CAN驱动，在can.h中可以找到ops的定义，分别指向驱动模块相应的操作

```
struct rt_can_ops
{
    rt_err_t (*configure)(struct rt_can_device *can, struct can_configure *cfg);
    rt_err_t (*control)(struct rt_can_device *can, int cmd, void *arg);
    int (*sendmsg)(struct rt_can_device *can, const void *buf, rt_uint32_t boxno);
    int (*recvmsg)(struct rt_can_device *can, void *buf, rt_uint32_t boxno);
};
```

回调函数configure实现

- 调用这个函数把CAN框架的配置转换成驱动HAL库

识别的格式，调用驱动的初始化函数

- 主要配置是模式与波特率，由于这两个配置在control中也会使用，所以就定义成了内联函数
- 波特率的计算见代码中的公式

```
static rt_err_t _configure(struct rt_can_device *can, struct can_configure *cfg)
{
    RT_ASSERT(can != RT_NULL);
    RT_ASSERT(cfg != RT_NULL);

    struct acm32_can_device *canObj;
    canObj = rt_container_of(can, struct acm32_can_device, device);

    canObj->handle.Instance=canObj->config.instance;
    canObj->handle.Init.CAN_ABOM=CAN_ABOM_DISABLE;          /* enable bus off recover */
    canObj->handle.Init.CAN_Mode = CAN_Mode_Normal;

    if(RT_TRUE != _set_baudrate(&canObj->handle, cfg->baud_rate))
+--- 3 lines: {-----
}

    if(RT_TRUE != _set_mode(&canObj->handle, cfg->mode))
+--- 3 lines: {-----
}

    HAL_CAN_Init(&canObj->handle);

    HAL_CAN_ConfigFilter(&canObj->handle, &canObj->filter);

    return RT_EOK;
}
```

```
rt_inline rt_bool_t _set_baudrate(CAN_HandleTypeDef *handle, rt_uint32_t baudrate)
{
    /* baudrate calculate, pclk=90MHz
    * baudrate = 1 / (((SJW+1)+(TSEG1+1)+(TSEG2+1)) * (((BRP+1)*2)/pclk))
    * example 500kbps = 1 / ((1+4+5)*2*9*(1/90M))
    */
    switch(baudrate)
+--- 34 lines: {-----
}
    return RT_TRUE;
}

rt_inline rt_bool_t _set_mode(CAN_HandleTypeDef *handle, rt_uint32_t mode)
{
    switch(mode)
+--- 16 lines: {-----
}
    return RT_TRUE;
}
```

回调函数control实现

- 把CAN框架中的使用的命令码实现

- RT_CAN_CMD_SET_BAUD

- RT_CAN_CMD_SET_FILTER

- RT_CAN_CMD_SET_MODE

- RT_CAN_CMD_SET_PRIV

- RT_CAN_CMD_SET_INT

- RT_CAN_CMD_CLR_INT

- RT_CAN_CMD_GET_STATUS

```
static rt_err_t _control(struct rt_can_device *can, int cmd, void *arg)
{
    RT_ASSERT(can != RT_NULL);

    struct acm32_can_device *canObj;
    canObj = rt_container_of(can, struct acm32_can_device, device);

    switch(cmd)
    {
        case RT_DEVICE_CTRL_SET_INT:
            RT_ASSERT(arg != RT_NULL);
            switch((rt_uint32_t)arg)
            +---- 16 lines: {-----
                break;
            case RT_DEVICE_CTRL_CLR_INT:
                RT_ASSERT(arg != RT_NULL);
                switch((rt_uint32_t)arg)
            +---- 22 lines: {-----
                break;
            #ifdef RT_CAN_USING_HDR
            case RT_CAN_CMD_SET_FILTER:
            +---- 65 lines: {-----
            #endif
            case RT_CAN_CMD_SET_BAUD:
                RT_ASSERT(arg != RT_NULL);
                if(RT_TRUE != _set_baudrate(&canObj->handle, (rt_uint32_t)arg))
            +---- 3 lines: {-----
                break;

            case RT_CAN_CMD_SET_MODE:
                if(RT_TRUE != _set_mode(&canObj->handle, (rt_uint32_t)arg))
            +---- 3 lines: {-----
                break;

            case RT_CAN_CMD_SET_PRIV:
                break;

            case RT_CAN_CMD_GET_STATUS:
                RT_ASSERT(arg != RT_NULL);
                canObj->device.status.rcverrncnt = HAL_CAN_GetReceiveErrorCounter(&canObj->handle);
                canObj->device.status.snderrcnt = HAL_CAN_GetTransmitErrorCounter(&canObj->handle);
                canObj->device.status.errcode = HAL_CAN_GetErrorCode(&canObj->handle, CAN_ErrorType_Err
                rt_memcpy(arg, &canObj->device.status, sizeof(canObj->device.status));
                break;

            default:
                return -RT_EINVAL;
            }
        }
    }
    return RT_EOK;
}
```

回调函数sendmsg实现

- 把CAN框架的命令包转换成HAL驱动的命令包
通过调用HAL的发送函数发送命令

```
static int _transmit(struct rt_can_device *can, const void *buf, rt_uint32_t boxno)
{
    RT_ASSERT(can != RT_NULL);
    RT_ASSERT(buf != RT_NULL);

    struct acm32_can_device *canObj;
    struct rt_can_msg *pmsg = (struct rt_can_msg *) buf;
    CanTxRxMsg txMsg = {0};

    canObj = rt_container_of(can, struct acm32_can_device, device);

    if(pmsg->ide == RT_CAN_STDID)
+--- 4 lines: {-----
    else
+--- 4 lines: {-----

    if(pmsg->rtr == RT_CAN_DTR)
+--- 3 lines: {-----
    else
+--- 3 lines: {-----
    txMsg.DLC = pmsg->len;

    rt_memcpy(txMsg.Data, pmsg->data, txMsg.DLC);

    HAL_CAN_Transmit(&canObj->handle, &txMsg);

    return RT_EOK;
}
```

回调函数recvmsg实现

- 获取HAL驱动接收的命令包输入到CAN框架的命令包中
- 目前这个函数只在中断中调用

```
static int _receive(struct rt_can_device *can, void *buf, rt_uint32_t boxno)
{
    RT_ASSERT(can != RT_NULL);
    RT_ASSERT(buf != RT_NULL);

    struct acm32_can_device *canObj;
    struct rt_can_msg *pmsg = (struct rt_can_msg *) buf;
    CanTxRxMsg rxMsg = {0};

    canObj = rt_container_of(can, struct acm32_can_device, device);

    HAL_CAN_GetRxMessage(&canObj->handle, &rxMsg);

    if(rxMsg.IDE == CAN_Id_Standard)
+--- 4 lines: {-----
    else
+--- 4 lines: {-----

    if(rxMsg.RTR == CAN_RTR_Data)
+--- 3 lines: {-----
    else
+--- 3 lines: {-----

    pmsg->len = rxMsg.DLC;

    rt_memcpy(pmsg->data, rxMsg.Data, rxMsg.DLC);

    return RT_EOK;
}
```

ISO15765-2协议介绍

网络层服务

- 给上层提供服务

传输层协议

- 提供收发大包数据的功能
- 上报传输完成/失败的功能

数据链路层用法

- 服务参数转CAN包

ISO15765-2 网络层

定义了给上层提供的服务项及各个服务项的参数格式

通讯服务

1. N_USData.request
2. N_USData.indication
3. N_USData_FF.indication
4. N_USData.confirm

协议参数设定服务

1. N_ChangeParameter.request
2. N_ChangeParameter.confirm

传输层功能

- 单帧传输
- 多帧传输
- PDU介绍
- 时序处理
- 错误处理

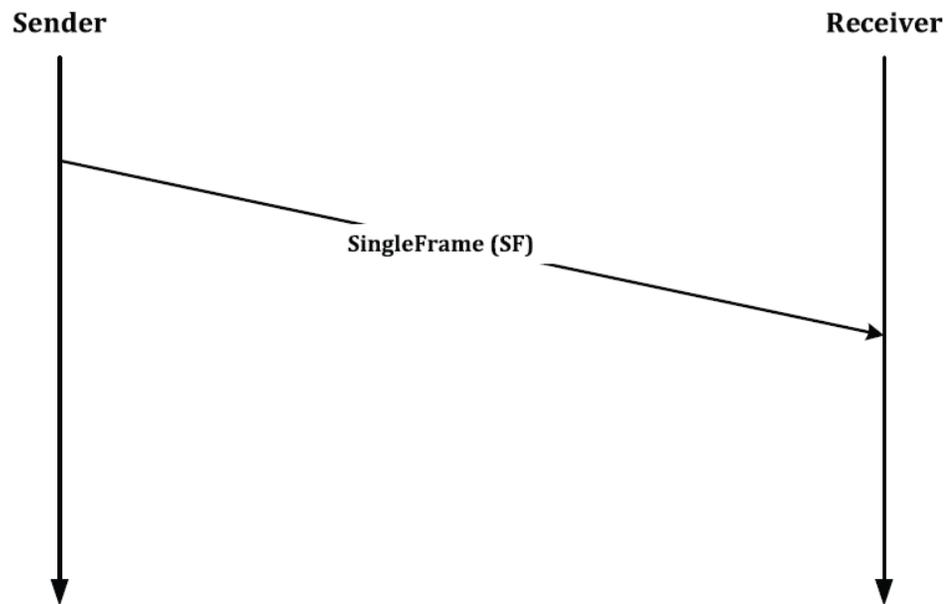


Figure 3 — Example of an unsegmented message

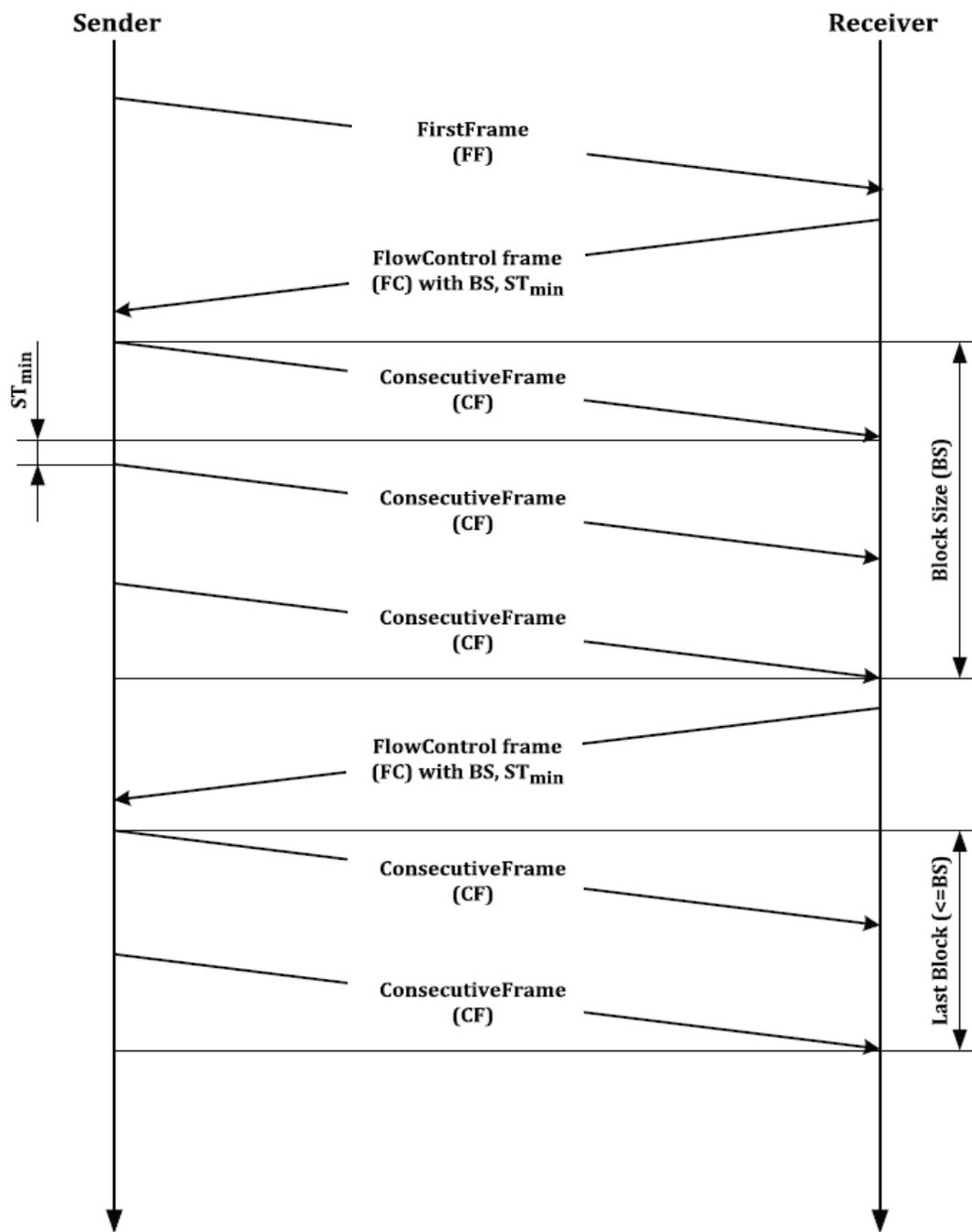


Table 5 — N_PDU format

Address information	Protocol control information	Data field
N_AI	N_PCI	N_Data

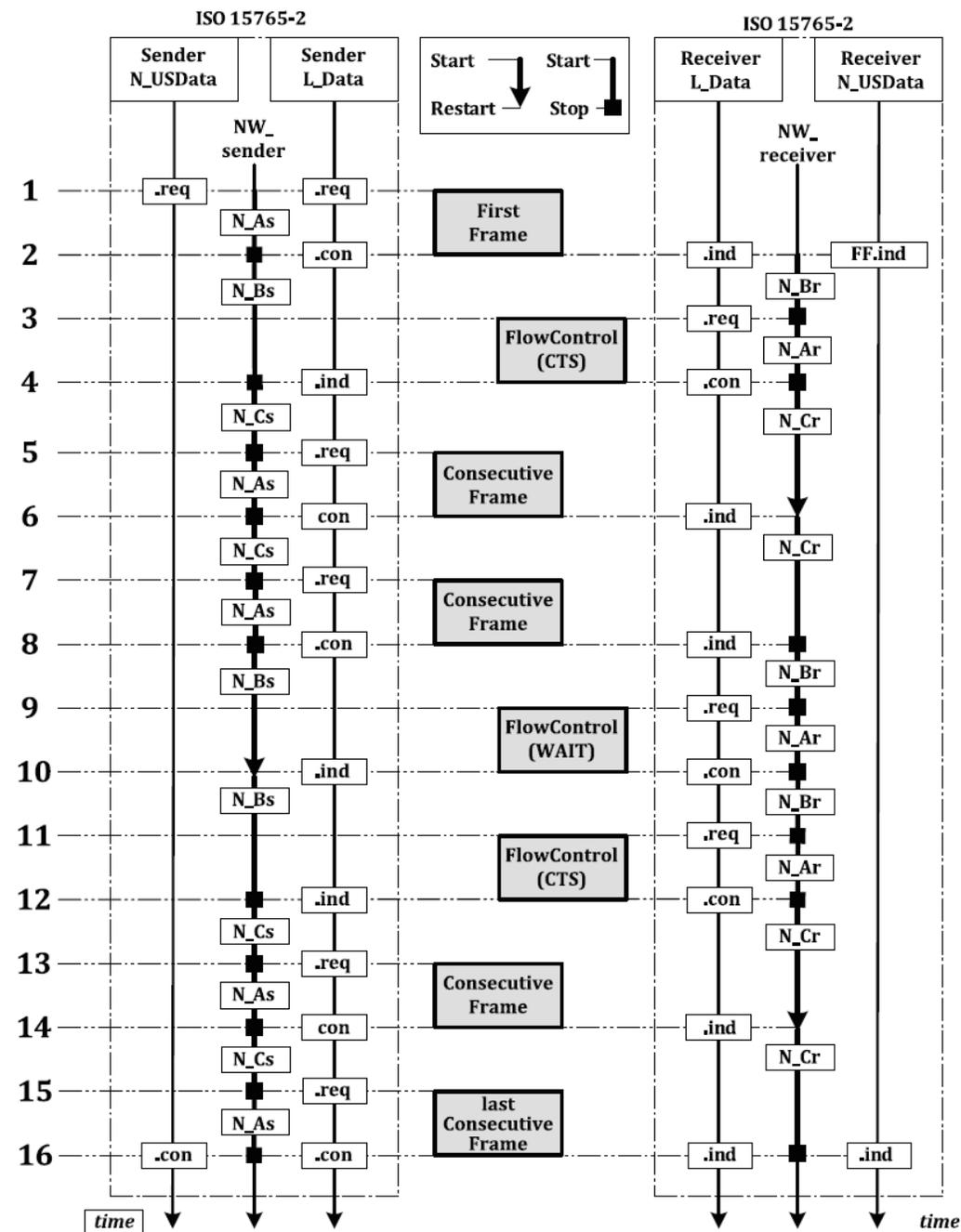
(N_SA, N_TA, N_TAtype [and N_AE])

Table 9 — Summary of N_PCI bytes

N_PDU name	N_PCI bytes						
	Byte #1 Bits 7 - 4	Byte #1 Bits 3 - 0	Byte #2	Byte #3	Byte #4	Byte #5	Byte #6
SingleFrame (SF) (CAN_DL ≤ 8)	0000 ₂	SF_DL	—	—	—	—	—
SingleFrame (SF) (CAN_DL > 8) ^a	0000 ₂	0000 ₂	SF_DL	—	—	—	—
FirstFrame (FF) (FF_DL ≤ 4 095)	0001 ₂	FF_DL		—	—	—	—
FirstFrame (FF) (FF_DL > 4 095) ^b	0001 ₂	0000 ₂	0000 0000 ₂	FF_DL			
ConsecutiveFrame (CF)	0010 ₂	SN	—	—	—	—	—
ConsecutiveFrame (CF)	0011 ₂	FS	BS	ST _{min}	N/A	N/A	N/A

相关时序参数的说明

N_As	发送端传输一个CAN帧(任何N_PDU)的时间
N_Ar	接收端传输一个CAN帧(任何N_PDU)的时间
N_Bs	到接收到下一个流控帧N_PDU的时间
N_Br	到传输下一个流控帧N_PDU的时间
N_Cs	到传输下一个连续帧N_PDU的时间
N_Cr	到接收下一个连续帧N_PDU的时间



时序参数	描述	数据链路层服务		超时时间 (ms)	性能要求 (ms)
		起始	结束		
N_As	发送端传输一个CAN帧(任何N_PDU)的时间	L_Data.request	L_Data.confirm	1000	-
N_Ar	接收端传输一个CAN帧(任何N_PDU)的时间	L_Data.request	L_Data.confirm	1000	-
N_Bs	到接收到下一个流控帧N_PDU的时间	L_Data.confirm(FF) L_Data.confirm(CF) L_Data.indication(FC)	L_Data.indication(FC)	1000	-
N_Br	到传输下一个流控帧N_PDU的时间	L_Data.indication(FF) L_Data.indication(CF) L_Data.confirm(FC)	L_Data.request(FC)	N/A	$(N_{Br} + N_{Ar}) < (0.9 \times N_{Bs})$
N_Cs	到传输下一个连续帧N_PDU的时间	L_Data.indication(FC) L_Data.confirm(CF)	L_Data.request(CF)	N/A	$(N_{Cs} + N_{As}) < (0.9 \times N_{Cr})$
N_Cr	到接收下一个连续帧N_PDU的时间	L_Data.confirm(CF) L_Data.indication(FC)	L_Data.indication(CF)	1000	-

数据链路层使用

N_PDU映射

- 地址格式包含标准地址、扩展地址、混合地址，不同的地址格式需要不同数据长度的CAN帧对包含数据的地址信息进行打包

CAN帧数据长度编码

- DLC固定为8，N_PDU比8短时，使用0xCC填充
- DLC不固定，SF、FC和最后一个CF按实际优化

RT-Thread ISO15765适配

- 使用了开源的实现<https://github.com/devcoons/iso15765-canbus>，暴露在外的接口如下：
 - ✓ 初始化接口iso15765_init
 - ✓ 请求发送接口iso15765_send
 - ✓ 接收到CAN数据包入队接口iso15765_enqueue
 - ✓ 消息处理接口iso15765_process
- 直接把源文件加入到RTT的工程中，使用方法参考readme.txt
 - ✓ 定义处理对象并初始化
 - ✓ 实现必备的回调
 - ✓ 新线程中调用消息处理接口处理消息
 - ✓ 接收到命令时放入到消息队列中

应用实例

- 实现一个函数导出的msh命令表，接收两种类型的参数，一种用于初始化；另一种用于获取外部输入的命令
- 按iso15765使用说明中实现相应逻辑，indication接口打印输出接收的数据
- 初始化时创建新线程，在新线程中调用处理接口，并把接收的数据包在线程中入队
- 接收到外部输入的命令时，调用iso15765_send接口发送命令



```

iso15765_sample.c
static iso15765_t handler =
{
    .addr_md = N_ADM_NORMAL,          /* Selected address mode of the TP
    .fr_id_type = CBUS_ID_T_STANDARD, /* CANBus frame type */
    .config.stmin = 0x05,             /* Default min. frame transmission
    .config.bs = 0x0F,               /* Maximun size of the block sequer
    .config.n_bs = 800,              /* Time until reception of the next
    .config.n_cr = 250,              /* Time until reception of the next
    .clbs.get_ms = getms,            /* Time-source for the library in r
    .clbs.on_error = on_error,       /* Callback which will be executed
    .clbs.send_frame = send_frame,   /* This callback will be fired wher
    .clbs.indn = indication          /* Indication Callback: Will be fir
                                     is available or an error occure
};

static rt_bool_t rcvpkg = RT_FALSE;

/* 接收数据回调函数 */
static rt_err_t can_rx_call(rt_device_t dev, rt_size_t size)
+--- 8 lines: {-----

static uint8_t send_frame(cbush_id_type id_type, uint32_t id,
                          cbush_fr_format fr_fmt, uint8_t dlc, uint8_t* dt)
+--- 21 lines: {-----

rt_inline void show_indication(n_indn_t* info)
+--- 24 lines: {-----

static void indication(n_indn_t* info)
+--- 4 lines: {-----

static void on_error(n_rslt err_type)
+--- 3 lines: {-----

static uint32_t getms()

```

```

int iso15765_sample(int argc, char *argv[])
{
    rt_thread_t thread;
    char can_cmd[16] = {0};
    char can_data[256] = {0};
    rt_size_t size;
    rt_uint8_t data[256] = {0};
    rt_err_t res;

    if (argc == 3)
+--- 4 lines: {-----
    else
+--- 4 lines: {-----

    if(0 == rt_strcmp(can_cmd, "cmd"))
+--- 12 lines: {-----
    else if(0 == rt_strcmp(can_cmd, "init"))
+--- 35 lines: {-----
    else
+--- 4 lines: {-----

    return 0;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(iso15765_sample, iso15765 transport sample);

```

```

static void iso15765_thread(void* arg)
{
    canbus_frame_t frame;
    struct rt_can_msg rxmsg = {0};

    iso15765_init(&handler);

    while(1)
    {
        iso15765_process(&handler);

        if(rcvpkg) /* 有收到CAN数据包 */
+--- 20 lines: {-----
            rt_thread_yield();
    }
}

```

THANKS FOR WATCHING