Interop.TSMasterAPI 编程指导 V0.8

1. 目录

1. 什么情况下需要此文档?

用户基于 DotNet 平台的编程语言，对目前市面上主流的如 Vector，TOSUN，Peak，英特佩斯等工具进行二次开发的时候，需要参考本文档，调用 API 函数来实现对设备的程序控制。

2. 驱动特点

本驱动提供了硬件抽象层，通过映射机制，同样一套 API 函数，可以应用于多款市面上主流的 CAN 工具硬件而不用修改代码。其作用原理图如下所示：

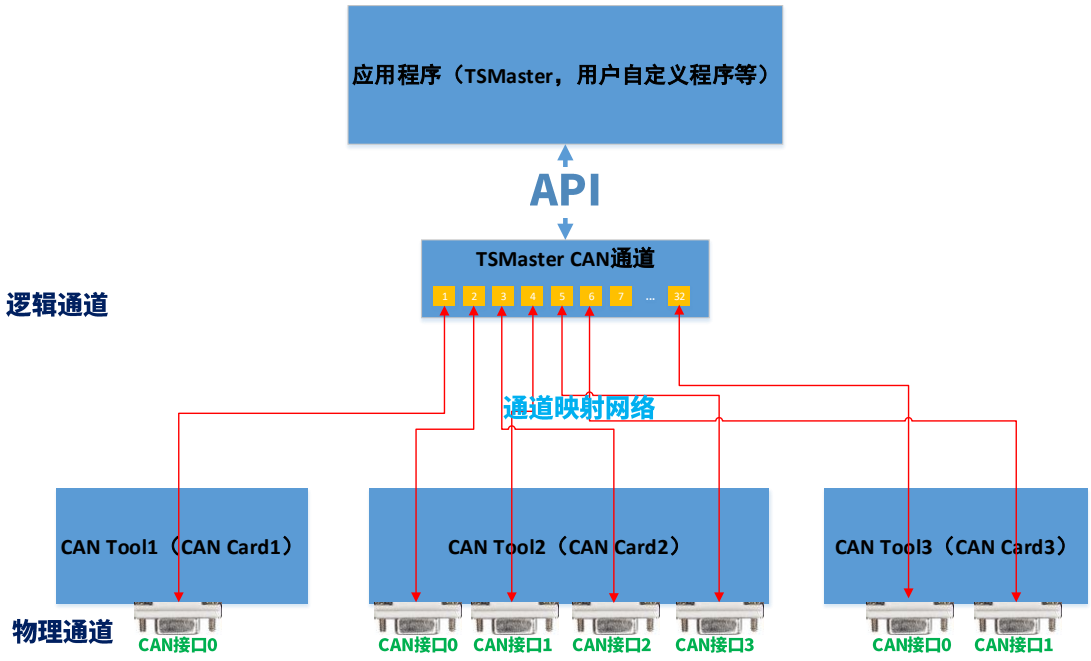


图1. TSMasterAPI 通过逻辑通道给上层提供统一的 API 接口

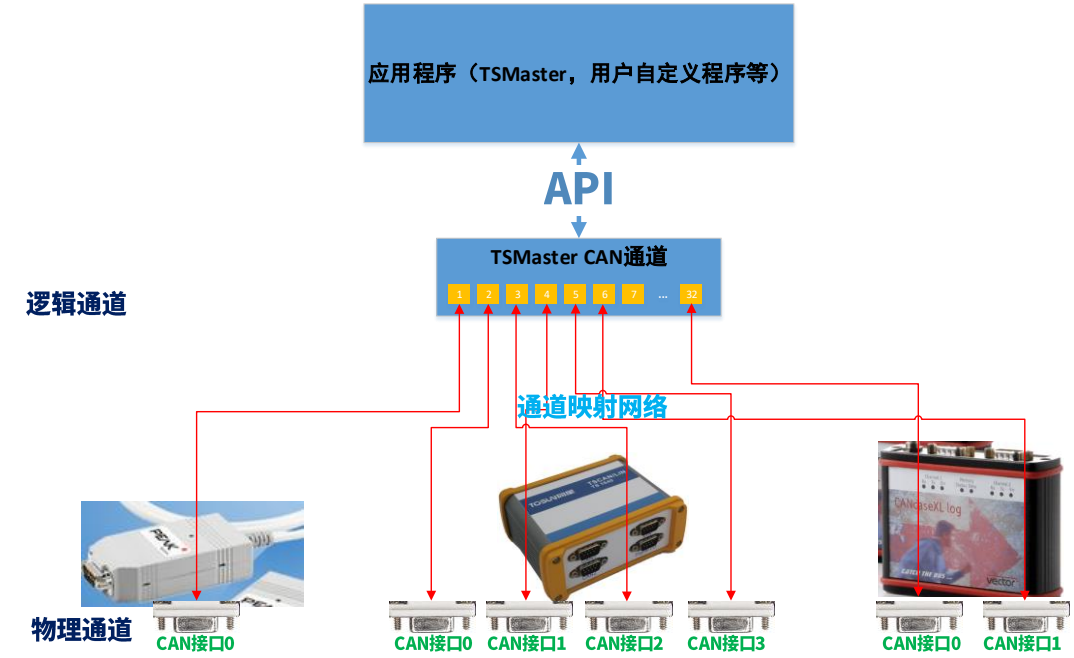


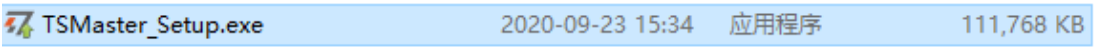
图2. 基于 TSMasterAPI 混合连接不同硬件板卡示意图

如上图所示，TSMaster API 内部实现了对目前市面上主流硬件 CAN 卡的兼容，提供了统一的 API 接口给上层应用层。用户在开发上层的时候，只需要开发一套上层代码，底层硬件可以任意切换，为用户硬件选择提供了最大的灵活性。

3. 添加 API 库文件

1. 安装驱动运行环境 TSMaster

TSMaster API 提供了对多种工具平台的支持，因此要使用 TSMaster 驱动，首先要安装 TSMaster 运行环境。该运行环境大小 100 多兆，安装文件如下：

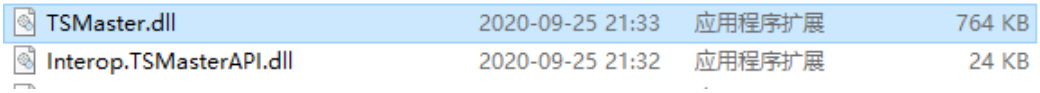


TSMaster 运行时环境为免费软件，可以直接到上海同星公司的官网上下载，下来链接为：http://www.tosun.tech/TOSUNSoftware/TSMaster_Setup.exe

下载该文件，并完成安装，然后进入下一步。

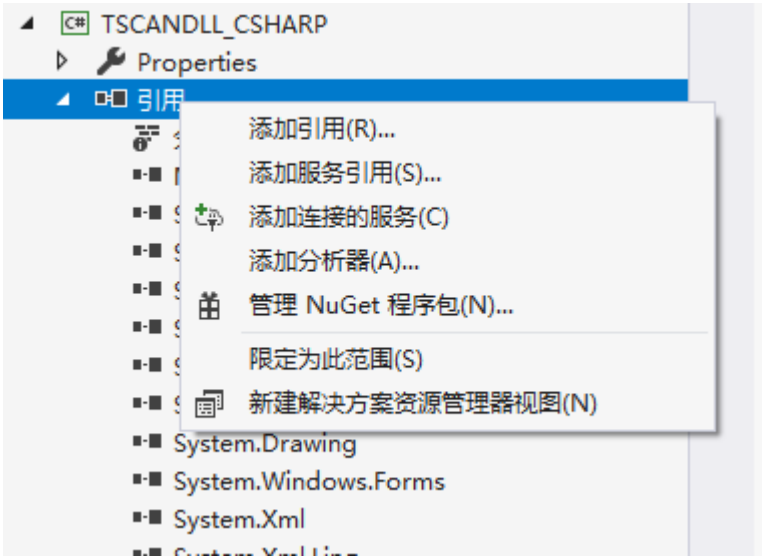
2. 添加库文件引用

安装完运行环境后，用户只需要把 TSMaster.dll 和 Interop.TSMaster.dll 添加到工程中，即完成了开发库文件的添加。因为有运行时环境的支撑，这两个文件均较小，这样尽可能保持了用户工程文件的简洁，如下图所示：

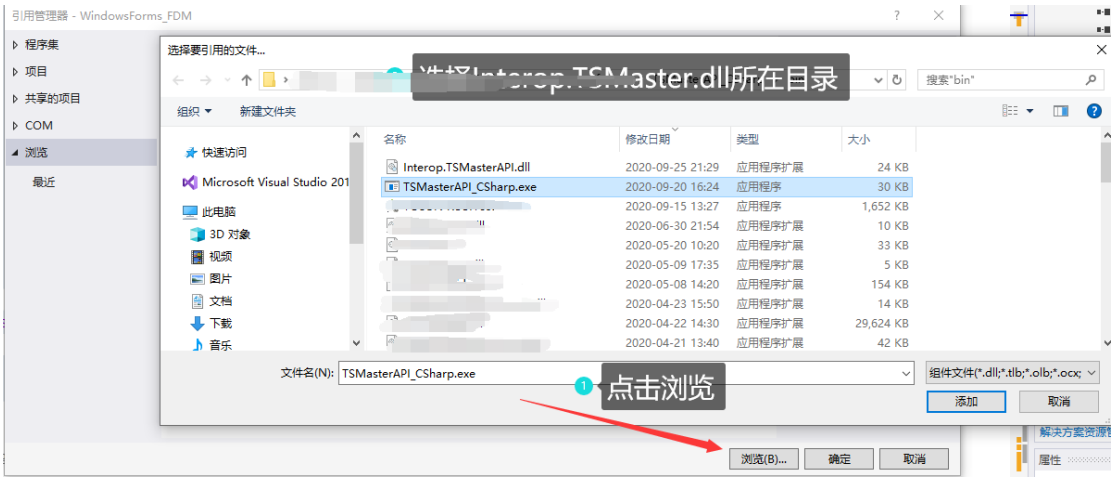


对于这两个文件，具体的添加方法如下：

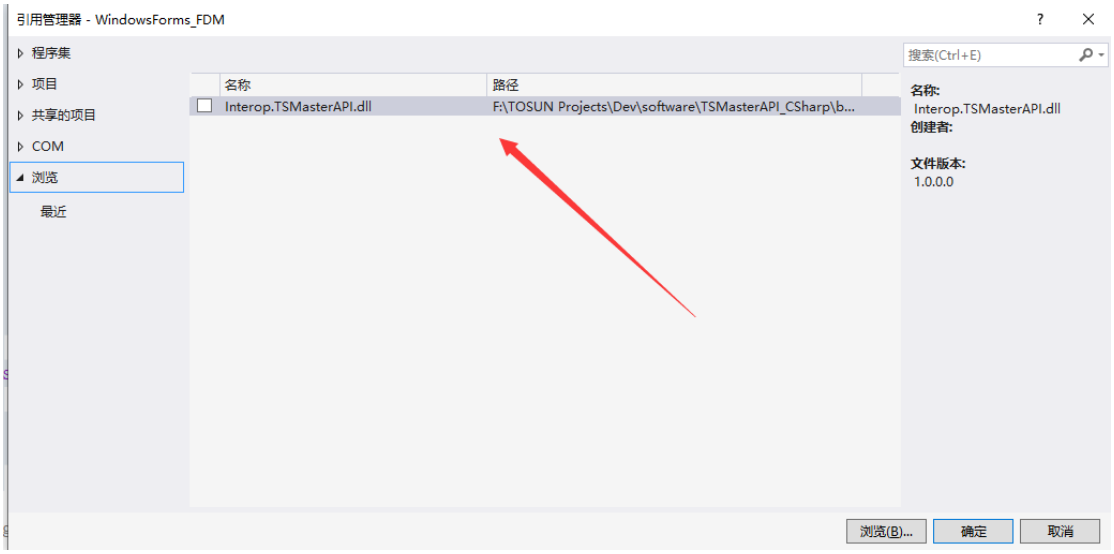
1. 把 TSMaster.dll 拷贝到用户的可执行目录下面。比如，常用 dotnet 工程的话就是拷贝到 Debug 文件下面。
2. 打开 VisualStudio 工程，在引用中添加 Interop.TSMasterAPI.dll。如下所示：
   - 引用->右键->添加引用



   - 在引用管理器中点击浏览，弹出文件选择框：



➢ 文件目录中查找到：TsCANApi.Interop.dll，点击添加。



3. 添加完成后

添加完成后，用户工作目录中只需要留下三个文件（用户自己开发的 exe 文件，TSMaster.dll, Interop.TSMasterAPI.dll），即可实现对 CAN 总线工具的访问。如果出现缺失库文件的情况，请检查是否正确安装了 TSMaster 运行时环境。

4. 注意事项（必看）：

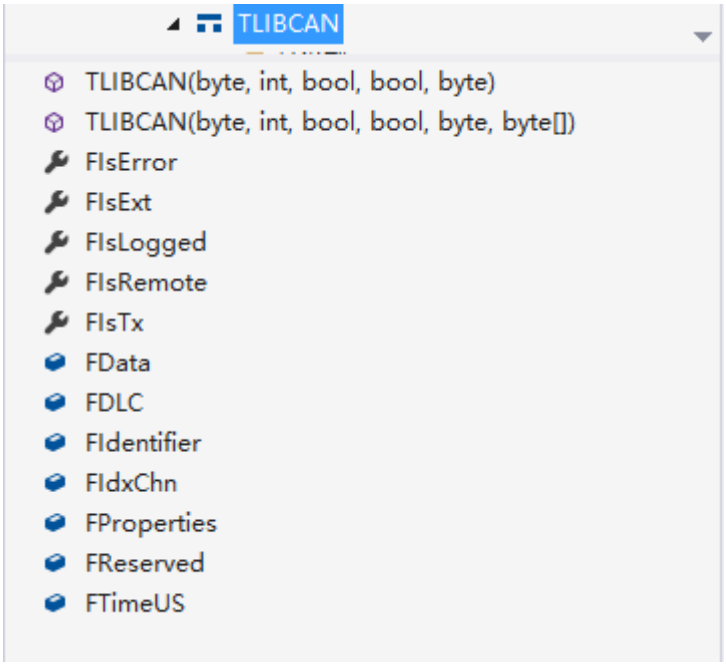
用户目录下面只需要保留 TSMaster.dll 文件即可，千万不要把 libTSMaster.dll 等库文件也放在用户目录下面，这样会造成程序异常。对于.net 程序，用于只需要操作

Interop.TSMasterAPI.dll，TSMaster.dll 两个库文件即可，尽可能的保持文件目录的整洁，同时也避免带来不必要的错误。

4. 数据类型定义

1. TLIBCAN：Classic CAN 数据类型

Struct 视图



构造函数：

本结构体创建了两个构造函数，传入的初始化参数不一样。用户可灵活调用。在构造函数中没有传入的参数，可以通过访问结构体对象直接修改。比如构造函数 1 没有传输数据组，但是可以在创建结构体对象过后，直接给 FData 成员赋值。

构造函数 1：

```
/// <summary>
```

```
        /// TCAN帧构造函数：通过此函数构造一个TCAN帧数据类型，调用此函数创建的数据帧默认数据位0，
        /// 用户还需要通过TCANObj.FData[x]填入要发送的数据
        /// </summary>
        /// <param name="AIdxChn">发送的数据通道</param>
        /// <param name="AID">帧ID</param>
        /// <param name="AIsStd">是否标准帧</param>
        /// <param name="AIsData">是否数据帧</param>
        /// <param name="ADLC">数据长度</param>
        public TCAN(byte AIdxChn, UInt32 AID, Boolean AIsExt, Boolean AIsRemote, byte ADLC)
```

构造函数 2：

```
/// <summary>
        /// TCAN帧构造函数：通过此函数构造一个TCAN帧数据类型
        /// </summary>
        /// <param name="AIdxChn">发送数据通道</param>
        /// <param name="AID">帧ID</param>
        /// <param name="AIsStd">是否标准帧</param>
        /// <param name="AIsData">是否数据帧</param>
        /// <param name="ADLC">数据长度</param>
        /// <param name="ADataArray">要发送的数组</param>
        public TCAN(byte AIdxChn, UInt32 AID, Boolean AIsExt, Boolean AIsRemote, byte ADLC, byte[] ADataArray)
```

成员：

FData：帧数据。最大程度为 8Bytes

FDLC：帧长度。

FIdentifier：帧 ID，如果为 0xFFFFFFFF，表示当前帧为错误帧

FIdxChn：帧通道，注意 CHANNEL_INDEX. CHN1 = 0,实际上是从 0 开始计算的。

FTImeUS：帧时间戳，64 位 us 级时间戳。

FProperties：存储 CAN 相关的属性，比如是否远程帧，是否扩展帧。

属性：

FIsError:是否错误帧

FIsExt：是否扩展帧

FIsRemote：是否远程帧

FIsTx：是否发送出去的报文。True:该报文是发送出去的报文；false：该报文是接收的报文

调用示例：

TLIBCAN msg = new TCAN(0,0xFD,false,false,3); //创建一个 TLIBCAN 报文，该报文 ID 为 0xFD，不是远程帧（也就是为数据帧），不是扩展帧（为标准帧），数据长度 DLC 为 3 个字节。

msg.FData[0] = 0x01;

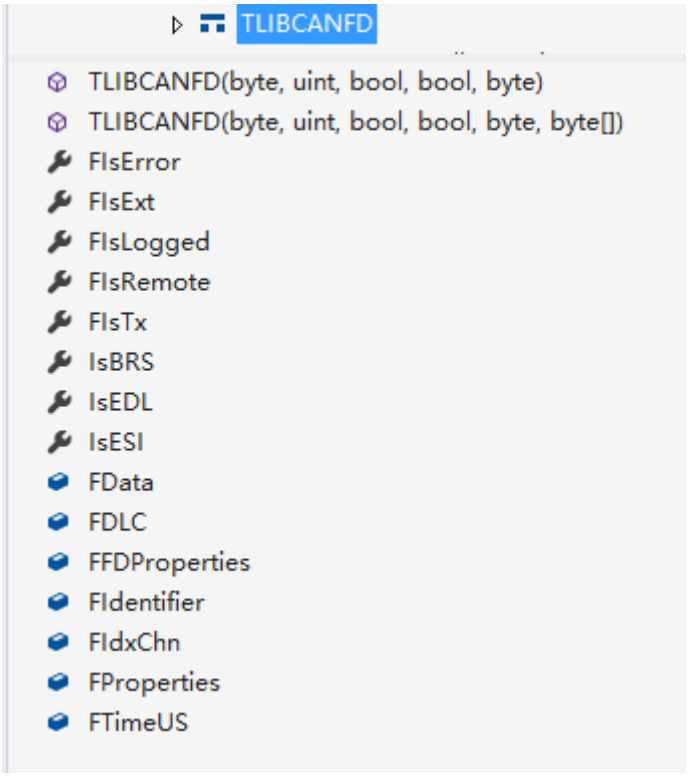
msg.FData[1] = 0x02;

msg.FData[2] = 0x03;

表示创建一个 TCAN 类型的帧 msg，发送通道为 0，帧 ID 为 0xFD，标准帧，数据帧，数据长度为 3 个字节。然后三个字节分别为 0x01,0x02,0x03.

2. TLIBCANFD：CANFD 报文类型

Struct 视图



构造函数：

本结构体创建了两个构造函数，传入的初始化参数不一样。用户可灵活调用。在构造函数中没有传入的参数，可以通过访问结构体对象直接修改。比如构造函数 1 没有传输数据组，但是可以在创建结构体对象过后，直接给 FData 成员赋值。

构造函数 1：

```
/// <summary>
        /// TCAN帧构造函数：通过此函数构造一个TCAN帧数据类型，调用此函数创建的数据帧默认数据位0，
        /// 用户还需要通过TCANObj.FData[x]填入要发送的数据
        /// </summary>
        /// <param name="AIdxChn">发送的数据通道</param>
        /// <param name="AID">帧ID</param>
        /// <param name="AIsStd">是否标准帧</param>
        /// <param name="AIsData">是否数据帧</param>
```

```
        /// <param name="ADLC">数据长度</param>
        public TCANFD (byte AIdxChn, UInt32 AID, Boolean AIsExt, Boolean AIsRemote, byte ADLC, Boolean AIsFD, Boolean AIsBRS,)
```

构造函数 2：

```
/// <summary>
        /// TCAN帧构造函数：通过此函数构造一个TCAN帧数据类型
        /// </summary>
        /// <param name="AIdxChn">发送数据通道</param>
        /// <param name="AID">帧ID</param>
        /// <param name="AIsStd">是否标准帧</param>
        /// <param name="AIsData">是否数据帧</param>
        /// <param name="ADLC">数据长度</param>
        /// <param name="ADataArray">要发送的数组</param>
        public TCANFD (byte AIdxChn, UInt32 AID, Boolean AIsExt, Boolean AIsRemote, byte ADLC, Boolean AIsFD, Boolean AIsBRS, byte[] ADataArray)
```

成员：

FData：帧数据。最大程度为 8Bytes

FDLC：帧长度。

FIdentifier：帧 ID，如果为 0xFFFFFFFF，表示当前帧为错误帧

FIdxChn：帧通道，注意 CHANNEL_INDEX. CHN1 = 0,实际上是从 0 开始计算的。

FTImeUS：帧时间戳，64 位 us 级时间戳。

FFDProperties：存储 FD 相关的属性，如是否 FD 报文，发送过程中是否波特率可变。

不同的字节位代表不同的属性值。

FProperties：存储 CAN 相关的属性，比如是否远程帧，是否扩展帧。

属性：

FIsError:是否错误帧

FIsExt：是否扩展帧

FIsRemote：是否远程帧

FIsTx：是否发送出去的报文。True:该报文是发送出去的报文；false：该报文是接收的报文

FIsFD：是否是 CANFD 帧，如果设置为 false，则依然作为 Classic 帧发送。

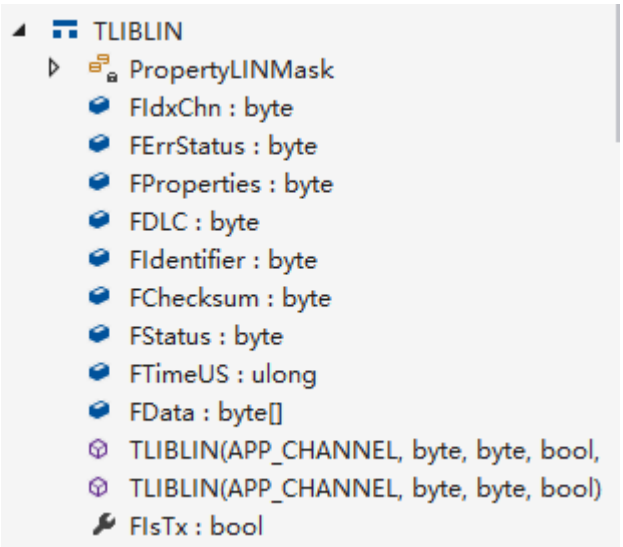
FIsBRS：如果是 CANFD 帧，发送过程中是否切换波特率。如果为 false，则整个报文按照仲裁场速率发送。

调用示例：

```
TCANFD msg = new TCANFD((byte)FCLCANChnIndex, 0xFD, false, false, 8);
msg.FIsFD = rBFD.Checked;
msg.FIsBRS = chkBRS.Checked;
//上述代码也可以采用下面的方式创建一个 CANFD 报文
//TCANFD msg = new TCANFD((byte)FCLCANChnIndex, 0xFD, false, false, 8, rBFD.Checked, chkBRS.Checked);
//
msg.FData[0] = 0x00;
uint ret = TsCANApi.SendCANFDASync(deviceHandle, ref msg);
```

3. TLIBLIN

Struct 结构



构造函数

本结构体创建了两个构造函数，传入的初始化参数不一样。用户可灵活调用。在构造函数中没有传入的参数，可以通过访问结构体对象直接修改。比如构造函数 1 没有传输数据组，但是可以在创建结构体对象过后，直接给 FData 成员赋值。

构造函数 1：

```
/// <summary>
        /// TCAN帧构造函数：通过此函数构造一个TCAN帧数据类型，调用此函数创建的数据帧默认数据位0，
        /// 用户还需要通过TCANObj.FData[x]填入要发送的数据
        /// </summary>
        /// <param name="AIdxChn">发送的数据通道</param>
        /// <param name="AID">帧ID</param>
        /// <param name="AIsStd">是否标准帧</param>
        /// <param name="AIsData">是否数据帧</param>
        /// <param name="ADLC">数据长度</param>
        public TLIBLIN(APP_CHANNEL AIdxChn, byte AID, byte ADLC, Boolean AIsTx);
```

构造函数 2：

```
/// <summary>
        /// TCAN帧构造函数：通过此函数构造一个TCAN帧数据类型
```

```
        /// </summary>
        /// <param name="AIdxChn">发送数据通道</param>
        /// <param name="AID">帧ID</param>
        /// <param name="AIsStd">是否标准帧</param>
        /// <param name="AIsData">是否数据帧</param>
        /// <param name="ADLC">数据长度</param>
        /// <param name="ADataArray">要发送的数组</param>
        public TLIBLIN(APP_CHANNEL AIdxChn, byte AID, byte ADLC, Boolean AIsTx, byte[]
ADataArray);
```

成员：

FData：帧数据。最大程度为 8Bytes

FDLC：帧长度。

FIdentifier：帧 ID，如果为 0xFFFFFFFF，表示当前帧为错误帧

FIdxChn：帧通道，注意 CHANNEL_INDEX. CHN1 = 0,实际上是从 0 开始计算的。

FTImeUS：帧时间戳，64 位 us 级时间戳。

FProperties：存储 LIN 相关的属性，比如报文方向，是接收报文还是发送报文。

FStatus：报文状态。

FErrStatus：如果是错误帧，对应的错误类型。

属性：

FIsTX: true：报文是发送出去的报文；false：报文是接收回来的报文。

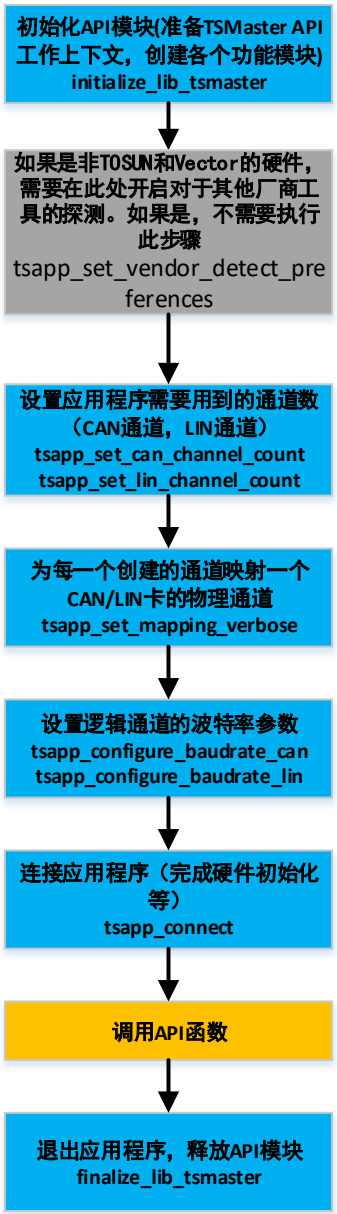
调用示例：

5. 设备初始化

1. CAN 初始化流

程图：

在实现 CAN 设备通讯之前，首先要进行设备初始化。TSMaster 设备初始化包括以下步骤：



注意：

- 在使用 API 模块之前，需要调用 initialize_lib_tsmaster 函数创建该模块；在使用过后，需要调用 finalize_lib_tsmaster 释放该模块。这两个函数一定是成对出现的。
- 第二步中，如果用户使用的是英特佩斯，Peak，Kavsar，ZLG 等硬件，需要在这一步开启对这些工具的探测，否则程序无法查询到这些硬件。

CAN 初始化示例代码：

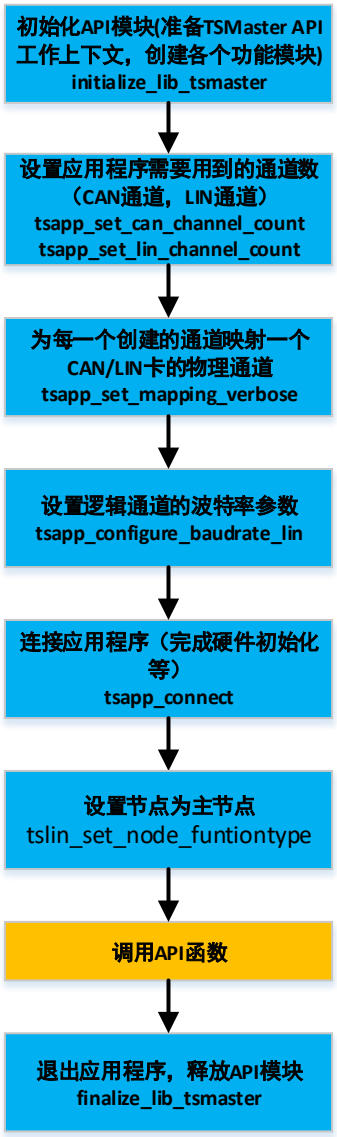
```
//第一步：初始化API模块
    TsMasterApi.initialize_lib_tsmaster(FProgramName);
//第二步：根据应用程序需要设置CAN，LIN通道数量，比如，这里需要2个CAN通道，0个LIN通道。
    if (TsMasterApi.tsapp_set_can_channel_count(2) == 0){
        Log("设置CAN通道数量成功!");
     }else{
          Log("设置CAN通道数量失败!");
    }
    if (TsMasterApi.tsapp_set_lin_channel_count(0) == 0){
        Log("设置LIN通道数量成功!");
    }else{
        Log("设置LIN通道数量失败!");
    }
    //第三步：按需创建通道映射:
    //把TC1005板卡的硬件通道1映射到驱动的逻辑通道1上面
    if (TsMasterApi. tsapp_set_mapping_verbose(FProgramName,
TLIBApplicationChannelType.APP_CAN,
                APP_CHANNEL.CHN1, "TC1005", TLIBBusToolDeviceType.TS_TC1005_DEVICE, 0,
0, HARDWARE_CHANNEL.CHN1) == 0){
       Log("映射通道1成功!");
     }
    if (TsMasterApi.tsapp_set_mapping_verbose(FProgramName,
TLIBApplicationChannelType.APP_CAN,
     APP_CHANNEL.CHN2, "TC1005", TLIBBusToolDeviceType.TS_TC1005_DEVICE, 0, 0,
HARDWARE_CHANNEL.CHN2) == 0){
        Log("映射通道2成功!");
            }
     //第四步：初始化通道参数
    if (TsMasterApi.tsapp_configure_baudrate_can((int)APP_CHANNEL.CHN1, 500, false,
true) == 0){
       Log("通道1波特率配置成功");
```

```
    }else{
       Log("通道1波特率配置失败");
    }
    if (TsMasterApi.tsapp_configure_baudrate_can((int)APP_CHANNEL.CHN2, 500, false,
true) == 0){
      Log("通道2波特率配置成功");
    }else{
      Log("通道2波特率配置失败");
    }
    //第五步：连接application：连接硬件通道并开启接收FIFO
    if (TsMasterApi.tsapp_connect()== 0){
        Log("连接应用程序成功！");
        TsMasterApi.tsapp_enable_receive_fifo(); //如果不使能内部FIFO，无法使用Receive
    函数读取内部报文
        Log("启动接收缓冲区Buffer!");
    }
```

2. LIN 初始化

流程图：

LIN 通道初始化示例流程图如下：



代码示例：

下述代码为初始化为主节点的代码示例：

```
private void CreateLINApplicationDemo()
{
            //FProgramName:唯一名称，后面的各种映射跟他绑定
           //第一步：初始化API模块,如果已经调用，这里则不需要调用
            TsMasterApi.initialize_lib_tsmaster(FProgramName);
            //第二步：按需设置需要的通道数，比如，这里需要0个CAN通道，1个LIN通道
            if (TsMasterApi.tsapp_set_can_channel_count(0) == 0)
            {
                Log("Set CAN Channel Count Success!");
            }
            else
```

```
                Log("Set CAN Channel Count Failed!");
            string retStr =
TsMasterApi.tsapp_get_error_description(TsMasterApi.tsapp_set_lin_channel_count(1));
            if (retStr == "OK")
            {
                Log("Set LIN Channel Count Success!");
            }
            else
                Log("Set LIN Channel Count Failed:" + retStr);
            //第三步：按需创建通道映射:
            if (TsMasterApi.tsapp_set_mapping_verbose(FProgramName,
TLIBApplicationChannelType.APP_LIN,
                  APP_CHANNEL.CHN1, "TL1001", TLIBBusToolDeviceType.TS_USB_DEVICE,
(int)TLIB_TS_Device_Sub_Type.TL1001, 0, HARDWARE_CHANNEL.CHN1) == 0)
            {
            }
            //第四步：初始化通道参数:设置LIN通道1的波特率为19.2k
            if (TsMasterApi.tsapp_configure_baudrate_lin((int)APP_CHANNEL.CHN1,
(Single)19.2) == 0)
            {
                Log("LIN Channel " + (1).ToString() + " baudrate has been configured");
            }
            else
            {
                Log("LIN Channel " +  (1).ToString() + " baudrate failed");
            }
            //第五步：连接application：连接硬件通道并开启接收FIFO
            string connectResult =
TsMasterApi.tsapp_get_error_description(TsMasterApi.tsapp_connect());
            if (connectResult == "OK")
            {
                Log("Connect Application Success!");
                TsMasterApi.tsapp_enable_receive_fifo();
                Log("Start Receive FIFO!");  //如果不使能内部FIFO，无法使用Receive函数
读取内部报文
            }
            else
            {
                Log(connectResult);
                Log("Connect Application Failed! Please check the mapping table and
whether the Hardware is Ready?!");
            }
            //设置LIN模块工作在主节点
            if (TsMasterApi.tslin_set_node_funtiontype(APP_CHANNEL.CHN1,
```

```
TLINNodeType.T_MasterNode) == 0x00)
            {
                Log("Set LIN Node Master Mode Success!");
            }
            else
            {
                Log("Set LIN Node Master Mode Failed!");
            }
}
```

6. CAN 报文收发

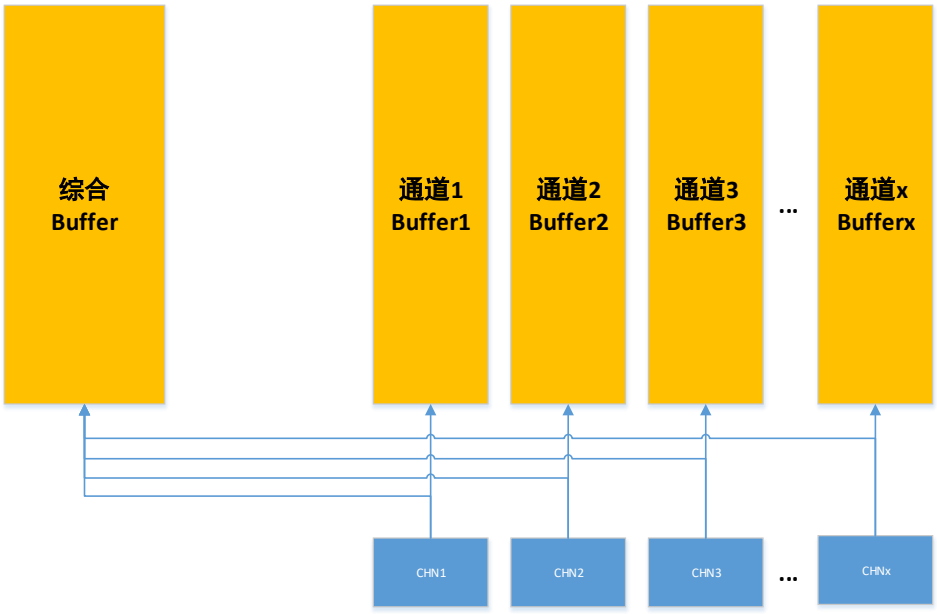
1. 报文发送

2. 报文接收

读取设备消息缓存的方式：

简介：

设备接收到报文过后，缓存在设备内部的 FIFO 中，外部程序调用函数接口从设备 FIFO 中把报文读取出来，FIFO 指针往后面移动；如果调用者一直不主动读取，会造成驱动内部 FIFO 溢出，最新的报文覆盖最旧的报文。TSMaster API 内部，报文缓存机制如下图所示：



综合FIFO：所有通道的报文根据接收顺序放在里面。
特点：
可以看到不同通道报文的相对接收顺序。报文在里面按照接收顺序存放，最新的报文覆盖最旧接收的报文。

通道FIFO：每一个通道有一个自己单独的报文FIFO
特点：
专用于存储跟本通道相关的报文,通道之间互不干扰。报文在里面按照接收顺序存放，最新的报文覆盖最旧接收的报文。

TSMaster 提供了报文读取驱动。可以选择读取指定通道的报文集合，也可以读取综合报文里面的报文集合。

ReceiveCANMsgList

- **功能：读取 Classic CAN 内部报文。**
- **参数详解：**

```
/// <summary>
        /// 通过读取设备内部缓存，读取接收数据
        /// </summary>
        /// <param name="ADeviceHandle">设备句柄</param>
        /// <param name="ACANMsgBuffer">数据Buffer，用于存储读取到的报文，该Buffer需要函数调用方创建</param>
        /// <param name="ACANBufferSize">消息Buffer的大小</param>
        /// <param name="AChn">目标通道：对于多通道设备，本函数选择读取哪一个通道的数据，该参数可以为空，默认为通道1</param>
        /// <param name="ATxRx">==0:仅仅接收Rx报文；>0: Tx Rx报文都读取回来，该参数可以为空，默认为只接收Rx报文</param>
        /// <returns>返回实际读取到的报文数量，如果没有任何报文，则返回值为0</returns>
        public static UInt32 ReceiveCANMsgList(IntPtr ADeviceHandle, ref TCAN[]
```

```
ACANMsgBuffer, UInt32 ACANBufferSize, CHANNEL_INDEX AChn = CHANNEL_INDEX.CHN1, byte ATxRx = 0)
```

ReceiveCANFDMsgList

- 功能：读取 CANFD 报文
- 参数详解：

```
/// <summary>
        /// 通过读取设备内部缓存，读取接收数据
        /// </summary>
        /// <param name="ADeviceHandle">设备句柄</param>
        /// <param name="ACANMsgBuffer">数据Buffer，用于存储读取到的报文，该Buffer需要函数调用方创建</param>
        /// <param name="ACANBufferSize">消息Buffer的大小</param>
        /// <param name="AChn">目标通道：对于多通道设备，本函数选择读取哪一个通道的数据，该参数可以为空，默认为通道1</param>
        /// <param name="ATxRx">==0:仅仅接收Rx报文；>0: Tx Rx报文都读取回来，该参数可以为空，默认为只接收Rx报文</param>
        /// <returns>返回实际读取到的报文数量，如果没有任何报文，则返回值为0</returns>
        public static UInt32 ReceiveCANFDMsgList(IntPtr ADeviceHandle, ref TCANFD[] ACANMsgBuffer, UInt32 ACANBufferSize, CHANNEL_INDEX AChn = CHANNEL_INDEX.CHN1, byte ARxTx = 0)
```

回调函数方式：

简介：

设备接收到报文过后，把报文整理成标准的 TCAN/TCANFD 数据结构。然后调用用户注册的接收回调函数，通过参数把接收到的报文传递给调用者。libTSCAN 内部维护一个独立的线程，每当消息达到后，就会通过回调函数主动把数据传递给调用者，用户不需要主动去调用读取函数。

注册回调函数：

采用代理机制（C#里面类 C 函数指针），注册回调函数。需要注意的是，一定要先申请

一个代理对象，然后把用户回调函数注册到该代理对象上，如下所示：

```
//
receiveCANCallBack += new TCANQueueEvent_Win32(ReceivedCANMsgCallBack);
receiveCANFDCallBack += new TCANFDQueueEvent_Win32(ReceivedCANFDMsgCallBack);
receiveLINCallBack += new TLINQueueEvent_Win32(ReceivedLINMsgCallBack);
connecttedCallBack += new TTSCANConnectedCallback_Win32(TSCANConnectedCallBack);
disconnectedCallBack += new TTSCANConnectedCallback_Win32(TSCANDisConnectedCallBack);
```

然后把代理对象调用回调函数注册到驱动中，如下所示：

```
uint ret = TsCANApi.RegisterCANRecvCallback_Win32(deviceHandle, receiveCANCallBack);
```

以上代码，有两点原因：

- 不采用代理对象，直接把函数比如 ReceivedCANMsgCallBack 注册到驱动中，程序也能够执行。但是因为.Net 是动态内存管理机制，运行一会儿过后，在 C#端的函数指针地址已经发生该表，注册在内部的函数指针无法跟着改变，会触发访问异常。
- 采用代理机制，意味着同一个代理代理对象，可以注册多个回调函数。比如 receiveCANCallBack，可以采用 += new 方式注册多个函数。用户可以把不同功能的处理函数注册到指针上。

回调函数使用

```
/// <summary>
        /// Classic CAN接收函数回调函数：需要注意的是，这个函数是在多线程中调用的。类似于串口控件的OnDataReceived事件机制，开发人员使用时候注意做好线程保护和同步。
        /// </summary>
        /// <param name="AData"></param>
        void ReceivedCANMsgCallBack(ref TCAN AData)
        {
            ///用户可以在此添加处理CAN报文AData的逻辑
            if (AData.FIsTx)
            {
              //表示这个报文是从本设备发出去的
            }
            if (AData.FIsExt)
            {
              //是否扩展帧
            }
            if (AData.FIsRemote)
            {
              //是否远程帧，否则就是数据帧
            }
            //依次类推
            DispMsg(AData.GetString());
```

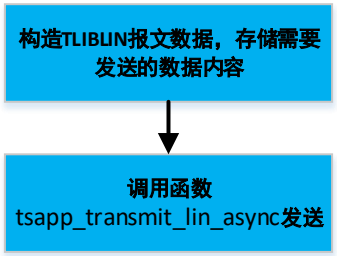
```
        }
```

/// 每当设备发送/收到一帧报文，就会调用此函数。参数：ref TCAN AData 就是对应的这一帧报文。用户可以根据报文的属性来决定如何处理这一帧报文。数据结构定义见章节：数据类型的定义。

7. LIN 报文收发

1. 报文发送

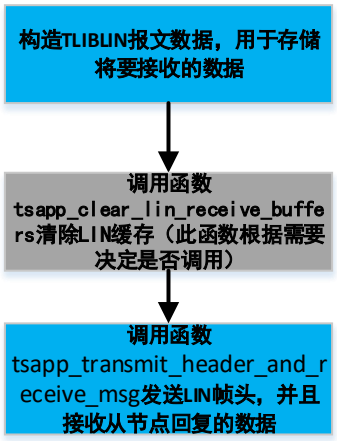
报文发送流程如下所示：



对应的代码如下：

```
TLIBLIN linMsg = new TLIBLIN(APP_CHANNEL.CHN1, 0x3C, 8, true);
            linMsg.FData[0] = 0x7F;
            linMsg.FData[1] = 0x03;
            linMsg.FData[2] = 0x22;
            linMsg.FData[3] = 0xFD;
            linMsg.FData[4] = 0x10;
            linMsg.FData[5] = 0xFF;
            linMsg.FData[6] = 0xFF;
            linMsg.FData[7] = 0xFF;
            if (TsMasterApi.tsapp_transmit_lin_async(ref linMsg) == 0x00)
            {
                Log("Async lin message success!");
            }
            else
            {
                Log("Async lin message failed!");
            }
```

2. 报文接收




```
TLIBLIN receivedMsg = new TLIBLIN();
TsMasterApi.tsapp_clear_lin_receive_buffers(APP_CHANNEL.CHN1);
if (TsMasterApi.tsapp_transmit_header_and_receive_msg(APP_CHANNEL.CHN1, 0x3D, 8, ref
receivedMsg, 200) == 0x00)
{
    Log(receivedMsg.FIdentifier.ToString("X2") + ":" + receivedMsg.FDLC.ToString() +
":" + receivedMsg.FData[0].ToString("X2") +
    receivedMsg.FData[1].ToString("X2") + receivedMsg.FData[2].ToString("X2") +
receivedMsg.FData[3].ToString("X2") +
    receivedMsg.FData[4].ToString("X2") + receivedMsg.FData[5].ToString("X2") +
receivedMsg.FData[6].ToString("X2") +
                    receivedMsg.FData[7].ToString("X2"));
}
else
{
    Log("Received LIN Message Failed");
}
```

8. 数据库（DBC）解析

TBD

9. 接口函数介绍

1. initialize_lib_tsmaster

函数名称	void initialize_lib_tsmaster(string AAppName);
功能介绍	初始化 TSMasterAPI 模块
调用位置	在使用 TSMasterAPI 之前，必须先执行模块初始化函数，初始化内部参数，为驱动的运行准备好环境。
输入参数	///<summary> /// 初始化API模块，为程序调用准备好运行环境 ///</summary> ///<param name="AAppName">应用程序名称，不能为空。硬件通道参数等都跟此名称绑定。 /// 用户可以输入设置的字符串，比如“TestDemo”，或者通过下列语句，直接获取当前应用程序的名称， /// 并作为参数传入初始化此 API 模块：</param> System.IO.Path.GetFileNameWithoutExtension(System.Windows.Forms.Application.ExecutablePath)
返回值	无
示例	TsMasterApi.initialize_lib_tsmaster(FProgramName);

2. initialize_lib_tsmaster

函数名称	void TsMasterApi.finalize_lib_tsmaster();
功能介绍	释放 TSMasterAPI 模块
调用位置	在退出程序之前，释放掉 TSMasterAPI 所使用的资源。

输入参数	无
返回值	无
示例	TsMasterApi.finalize_lib_tsmaster();

3. tsapp_set_can_channel_count

函数名称	public static extern int tsapp_set_can_channel_count(int ACount);
功能介绍	设置应用程序用到的 CAN 通道数量。可以通过 TSMaster 界面直接完成。
调用位置	映射 CAN 通道之前，用户可以根据应用程序的需求，分配需要用到的 CAN 通道数量。
输入参数	///<summary> /// 设置程序的CAN通道数量 ///</summary> ///<param name="ACount">需要设置的CAN通道数量</param> ///<returns></returns>
返回值	==0:设置成功 其他：失败，根据错误码查询错误类型。

示例	//设置当前应用程序用到4个CAN通道
	if (TsMasterApi.tsapp_set_can_channel_count(4) == 0)
	Log("Set CAN Channel Count Success!");
	else
	Log("Set CAN Channel Count Failed!");

4. tsapp_set_lin_channel_count

函数名称	public static extern int tsapp_set_lin_channel_count(int ACount);
功能介绍	设置应用程序用到的 CAN 通道数量。可以通过 TSMaster 界面直接完成。

调用位置	映射 LIN 通道之前，用户可以根据应用程序的需求，分配需要用到的 LIN 通道数量。
输入参数	///<summary> /// 设置LIN通道数量 ///</summary> ///<param name="ACount">需要设置的LIN通道数量</param> ///<returns></returns>
返回值	==0:设置成功 其他：失败，根据错误码查询错误类型

示例	//设置当前应该程序使用0个LIN通道
	if (TsMasterApi.tsapp_set_lin_channel_count(0) == 0)
	Log("Set LIN Channel Count Success!");
	else
	Log("Set LIN Channel Count Failed!");

5. tsapp_set_channel_mapping_verbose

函数名称	int tsapp_set_mapping_verbose(string AppName, TLIBApplicationChannelType AAppChannelType, APP_CHANNEL AAppChannel, string AHardwareName, TLIBBusToolDeviceType AHardwareType, int AHardwareSubType, int AHardwareIndex, HARDWARE_CHANNEL AHardwareChannel, Boolean AEnableMapping = true)
功能介绍	使用硬件设备之前，为应用程序的通道（CAN/CANFD/LIN）映射（也就是绑定）指定 CAN 设备的指定通道。这个操作也可以在 TSMaster 的设备管理界面完成，代码中可以不需要这部分操作。
调用位置	在应用程序连接 CAN 工具之前，调用此函数完成硬件的绑定。
输入参数	///<param name=" ">应用程序名称：该 Map 绑定到的应用程序名称，跟 initialize_lib_tsmaster 中名称要一致</param> ///<param name="AAppChannelType">通道类型：包括 APP_CAN,APP_LIN</param> ///<param name="AAppChannel">应用程序的通道编号：这是上层程序使用的通道编号</param>

	///<param name="AHardwareName">硬件名称：比如 TOSUN，Vector 等，也可以不填写，主要是方便用户查看</param>
	/// <param name="AHardwareType">硬件类型：根据 TLIBBusToolDeviceType 囊括了目前市面上所有主流的 CAN 卡硬件，</param>. 其中，工具枚举类型定义如下：
	public enum TLIBBusToolDeviceType : int {
	BUS_UNKNOWN_TYPE = 0,
	TS_TCP_DEVICE = 1,
	XL_USB_DEVICE = 2,
	TS_USB_DEVICE = 3,
	PEAK_USB_DEVICE = 4,
	KVASER_USB_DEVICE = 5,
	ZLG_USB_DEVICE = 6,
	ICS_USB_DEVICE = 7,
	TS_TC1005_DEVICE = 8
	};
	///<param name="AHardwareSubType">在该设备类型下面的子设备，比如同星的 TS_USB_DEVICE，下面就包含 TC1001，TL1002，TC1011 等等系列产品</param>
	///<param name="AHardwareIndex">支持同时插多个完全一样的硬件，比如同时插两个 TC1001，这里用于选择是 1 号设备的通道还是 2 号设备的通道</param>
	/// <param name="AHardwareChannel">硬件设备的通道，比如 TC1005 有 5 个硬件通道，这个参数就用于选择对应的硬件通道</param>
	///<param name="AEnableMapping">是否使能此映射，如果设置为 false，则该通道是不能使用的</param>

返回值	==0:执行成功 其他：查询错误码

示例	//如下代码：表示把TC1005设备的通道1分配给应用程序的通道1使用。这样应用程序使用通道1通讯的时候，实际上使用的就是TC1005的硬件通道1.
	if (TsMasterApi.tsapp_set_channel_mapping(
	FProgramName, //应用程序名称
	TLIBApplicationChannelType.APP_CAN, //映射CAN/CANFD通道
	APP_CHANNEL.CHN1, //映射（分配给）应用程序的通道1
	"TC1005", //设备名称为TC1005
	TLIBBusToolDeviceType.TS_TC1005_DEVICE, //设备类型为TS_TC1005_DEVICE
	0, //子类型为0，TC1005下面没有子类型，所以直接设置为0
	0, //设备编号为0，如果只接了一个设备，则设置硬件编号为0
	HARDWARE_CHANNEL.CHN1) == 0) //使用TC1005的0物理通道
	{
	}

6. tsapp_get_mapping

函数名称	int tsapp_get_mapping (TLIBApplicationChannelType AAppChannelType, APP_CHANNEL AAppChannel, ref TLIBTSMapping AMapping)
功能介绍	获取指定通道的映射信息值
调用位置	想查询应用程序指定通道当前绑定的硬件信息等内容的时候，调用此函数
输入参数	///<summary> /// 获取通道映射值 ///</summary> ///<param name="AAppName">应用程序名称</param> ///<param name="AAppChannelType">通道类型，类型定义如下: /// public enum TLIBApplicationChannelType : int ///{ /// APP_CAN = 0, /// APP_LIN = 1 ///};</param> ///<param name="AAppChannel">应用程序通道编号</param> ///<param name="AMapping">映射 TLIBTSMapping 类型变量，存储获取的映射值</param>
返回值	==0:执行成功 其他：查询错误码

示例	TLIBTSMapping m = new TLIBTSMapping();
	if(TsMasterApi.tsapp_get_channel_mapping(
	FProgramName,
	(TLIBApplicationChannelType)cbbChannelType1.SelectedIndex,
	(APP_CHANNEL)cbbAppChannelIndex.SelectedIndex,
	ref m) == 0)
	{
	Log("Get Mapping Success");
	MessageBox.Show("HardwareName:" + m.GetHWDeviceName() + "HardwareChannel:" + m.FHWChannelIndex.ToString());
	}

7. tsapp_del_mapping_verbose

函数名称	int tsapp_del_mapping_verbose(TLIBApplicationChannelType AAppChannelType, APP_CHANNEL AAppChannel)
功能介绍	删除映射信息，接触应用程序通道和硬件的绑定。
调用位置	如果想接触硬件和应用程序通道的绑定，调用此函数删除映射信息，解除配置信息即可。
输入参数	/// <summary> /// 删除通道映射 /// </summary> /// <param name="AAppChannelType">通道类型，类型定义如下： /// public enum TLIBApplicationChannelType : int /// { /// APP_CAN = 0, /// APP_LIN = 1 /// };</param> /// <param name="AAppChannel">应用程序通道编号</param>
返回值	==0:连接成功 其他：查询错误码

示例	if (TsMasterApi.tsapp_del_channel_mapping(
	FProgramName,
	(TLIBApplicationChannelType)cbbChannelType1.SelectedIndex,
	(APP_CHANNEL)cbbAppChannelIndex.SelectedIndex) == 0)
	{
	Log("Delete Mapping Success");
	}

8. tsapp_configure_baudrate_can

函数名称	int tsapp_configure_baudrate_can(int AIdxChn, Single ABaudrateKbps, Boolean AListenOnly, Boolean AInstallTermResistor120Ohm);
功能介绍	设置 CAN 通道的波特率等参数

调用位置	连接硬件设备之前，先设置通道的参数
输入参数	// hardware settings ///<summary> /// 设置通道波特率等参数 ///</summary> ///<param name="AIdxChn">应用程序通道编号</param> ///<param name="ABaudrateKbps">波特率（kbps）</param> ///<param name="AListenOnly">是否只听模式，如果开启，则只能接收数据</param> ///<param name="AInstallTermResistor120Ohm">是否只能设备内部终端电阻</param>
返回值	==0:连接成功 其他：检查错误码
示例	

9. tsapp_configure_baudrate_canfd

函数名称	int tsapp_configure_baudrate_canfd(int AIdxChn, Single AArbRateKbps, Single ADataRateKbps, TLIBCANFDControllerType AControllerType, TLIBCANFDControllerMode AControllerMode, Boolean AInstallTermResistor120Ohm);
功能介绍	配置 CANFD 通道的波特率等硬件参数
调用位置	连接 CAN 工具之前，先调用此函数配置硬件设备参数
输入参数	///<summary> /// 设置CANFD通道波特率等参数 ///</summary> ///<param name="AIdxChn">应用程序通道</param> ///<param name="AArbRateKbps">仲裁场波特率</param> ///<param name="ADataRateKbps">数据场波特率</param> ///<param name="AControllerType">控制器类型，包括： 经典CAN(lfdtCAN = 0),ISOCANFD(lfdtISOCAN = 1),NoISOCANFD(lfdtNonISOCAN = 2)</param> ///<param name="AControllerMode">控制器工作模式,包括: 正常工作模式(lfdmNormal = 0),关闭ACK模式(lfdmACKOff = 1),受限制模式(lfdmRestricted = 2)</param> ///<param name="AInstallTermResistor120Ohm">是否使能内部终端电阻</param>
返回值	==0:连接成功

	其他：检查错误码
示例	

10. tsapp_enable_receive_fifo

函数名称	void tsapp_enable_receive_fifo();
功能介绍	使能设备内部缓存接收模式
调用位置	应用程序如果想通过读取缓存的方式读取数据，在执行 tsapp_receive_can/canfd/lin_message_list 之前，必须要执行此函数。
输入参数	无
返回值	无
示例	

11. tsapp_disable_receive_fifo

函数名称	void tsapp_disable_receive_fifo();
功能介绍	关闭驱动内部的接收缓存机制
调用位置	用户不需要采用读取缓存的方式接收报文的时候，调用此函数关闭内部缓存。
输入参数	无
返回值	无

示例	

12. tsapp_connect

函数名称	`int tsapp_connect();`
功能介绍	连接应用程序。本函数执行的时候，会检查映射的硬件通道是否全部就绪，设置各个硬件参数，并完成设备和应用程序的连接。
调用位置	完成各个硬件参数的设置过后，调用此函数完成设备的连接。后面就可以调用设备完成数据收发等功能了。
输入参数	无
返回值	==0:连接成功 其他，查询错误码
示例	

13. tsapp_get_error_description_message

函数名称	`string tsapp_get_error_description_message(int ACode)`
功能介绍	根据函数执行的返回值编码 ACode，查询该编码代表的意义
调用位置	TSCAN 的 API 函数执行过后，会返回一个 ErroCode 编码，通过调用此函数可以查询该编码代表的具体含义
输入参数	ACode：错误编码值
返回值	错误编码代表具体含义字符串

示例	

14. tsapp_disconnect

函数名称	`public int tsapp_disconnect();`
功能介绍	断开设备连接
调用位置	不需要使用硬件设备，调用此函数断开设备连接
输入参数	无
返回值	==0:断开设备成功 其他值，查询错误码
示例	

15. tsapp_transmit_can_async

函数名称	`int tsapp_transmit_can_async(ref TLIBCAN ACAN);`
功能介绍	以异步的方式发送 CAN 报文
调用位置	发送 CAN 报文
输入参数	ACAN：CAN 数据包。TLIBCAN 数据组成请查看章节：TLIBCAN：Classic CAN 数据类型。
返回值	==0：发送成功 其他值：发送失败，失败原因请查看错误码

示例	`//先准备好要发送的CAN报文`
	`TLIBCAN canMsg = new TLIBCAN(0, 0x456, false, false, 8);`
	`//然后调用发送函数把该报文数据发送出去`
	`if (TsMasterApi.tsapp_transmit_can_async(ref canMsg) == 0)`
	`{`
	`  Log("ASync Send CAN Message Success!");`
	`}`

	else
	{
	Log("ASync Send CAN Message Failed");
	}

16. tsapp_transmit_can_sync

函数名称	int tsapp_transmit_can_sync(ref TLIBCAN ACAN, int ATimeoutMS);
功能介绍	发送 CAN 报文，并检测到发送成功后，才退出此函数。此函数返回成功，代表 CAN 报文一定已经成功发送到了 CAN 总线上面。
局限性	因为此函数需要设备 API 深度支持，因此，目前支持此函数接口的只有 TS 和 Vector 系列设备。
调用位置	同步发送 CAN 报文的场合
输入参数	///<summary> /// 同步发送CAN报文 ///</summary> ///<param name="ACAN">报文数据</param> ///<param name="ATimeoutMS">同步等待的超时时间</param>
返回值	==0：发送成功 其他值：发送失败，查询错误码

示例	//构造CAN报文
	TLIBCAN canMsg = new TLIBCAN(0, 0x456, false, false, 8);
	//发送CAN报文，如果100ms内发送成功，则返回成功。否则，返回超时失
	//败
	if (TsMasterApi.tsapp_transmit_can_sync(ref canMsg, 100) == 0)
	{
	Log("Sync Send CAN Message Success!");
	}
	else
	{
	Log("Sync Send CAN Message Failed");
	}

17. tsapp_transmit_canfd_async

函数名称	int tsapp_transmit_canfd_async(ref TLIBCANFD ACANFD);

功能介绍	以异步的方式发送 CANFD 报文
调用位置	发送 CANFD 报文
输入参数	ACAN：CANFD 数据包。TLIBCAN 数据组成请查看章节：TLIBCANFD：CANFD 数据类型。
返回值	==0：发送成功 其他值：发送失败，失败原因请查看错误码

示例	//先准备好要发送的CAN报文
	TLIBCANFD canMsgFD = new TLIBCANFD(0, 0x456, false, false, 8);
	//然后调用发送函数把该报文数据发送出去
	if (TsMasterApi.tsapp_transmit_canfd_async (ref canMsgFD) == 0)
	{
	Log("ASync Send CANFD Message Success!");
	}
	else
	{
	Log("ASync Send CANFD Message Failed");
	}

18. tsapp_transmit_canfd_sync

函数名称	int tsapp_transmit_canfd_sync(ref TLIBCANFD ACANFD);
功能介绍	发送 CANFD 报文，并检测到发送成功后，才退出此函数。此函数返回成功，代表 CANFD 报文一定已经成功发送到了 CAN 总线上面。
调用位置	发送 CANFD 报文
局限性	因为此函数需要设备 API 深度支持，因此，目前支持此函数接口的只有 TS 和 Vector 系列设备。
输入参数	ACANFD：CANFD 数据包。TLIBCANFD 数据组成请查看章节：TLIBCANFD：CANFD 数据类型。
返回值	==0：发送成功 其他值：发送失败，失败原因请查看错误码

示例	//先准备好要发送的CANFD报文
	TLIBCANFD canMsg = new TLIBCANFD(0, 0x456, false, false, 8);
	if (TsMasterApi.tsapp_transmit_canfd_sync(ref canMsg, 200) == 0)
	{
	Log("Sync Send CANFD Message Success!");
	}
	else
	{
	Log("Sync Send CANFD Message Failed");

	}

19. tsapp_add_cyclic_msg_can

函数名称	int tsapp_add_cyclic_msg_can(ref TLIBCAN ACAN, Single APeriodMS);
功能介绍	增加周期发送的报文，增加完成后，TSMasterAPI 自动完成报文的周期发送。
调用位置	在需要周期性发送 CAN 报文的场合
输入参数	///<summary> 添加周期发送报文 ///</summary> ///<param name="ACAN">CAN报文数据</param> ///<param name="APeriodMS">周期值</param>
返回值	==0：添加成功 其他值：添加失败，查询错误码

示例	//添加周期发送的报文
	//第一步：创建CAN报文，并设置DLC，发送数据段等属性
	byte canid = Convert.ToByte(tbAddPeriodCANID.Text, 16);
	TLIBCAN canObj = new TLIBCAN(0, canid, false, false, 0);
	canObj.FDLC = 8;
	canObj.FData[0] = 0xAA;
	canObj.FData[1] = 0xBB;
	Single intervalTime = 10; //10ms周期值为10ms
	//把此报文添加到周期发送列表中
	if (TsMasterApi.tsapp_add_cyclic_msg_can(ref canObj, intervalTime) == 0)
	{
	Log("Add Period CAN Message Success!");
	}

20. tsapp_delete_cyclic_msg_can

函数名称	int tsapp_delete_cyclic_msg_can(ref TLIBCAN ACAN);
功能介绍	停止周期性发送 CAN 报文

调用位置	在需要停止周期性发送报文的场合
输入参数	///<summary> /// 删除周期发送报文 ///</summary> ///<param name="ACAN">需要被停止的周期报文</param> ///<returns></returns>
返回值	==0：删除成功 其他值：删除失败，查询错误码

示例	//首先，创建CAN报文：主要是ID，是否标准帧，是否数据帧等属性
	byte canid = Convert.ToByte(tbDelPeriodCANID.Text, 16);
	TLIBCAN canObj = new TLIBCAN(0, canid, false, false, 0);
	//从周期发送列表中删除该CAN报文
	if (TsMasterApi.tsapp_delete_cyclic_msg_can(ref canObj) == 0)
	{
	Log("Delete Period CAN Message Success!");
	}

21. tsapp_receive_can_message_list

函数名称	int tsapp_receive_can_message_list(ref TLIBCAN[] ACANMsgBuffer, int ACANBufferSize, APP_CHANNEL AChn = APP_CHANNEL.CHN1, READ_TX_RX_DEF ATxRx = READ_TX_RX_DEF.ONLY_RX_MESSAGES);
功能介绍	读取报文序列
调用位置	在需要读取报文数据包的场合
输入参数	///<summary> /// 通过读取设备内部缓存，读取接收数据 ///</summary> ///<param name="ACANMsgBuffer">数据Buffer，用于存储读取到的报文，该Buffer需要函数调用方创建</param> ///<param name="ACANBufferSize">消息Buffer的大小</param> ///<param name="AChn">目标通道：对于多通道设备，本函数选择读取哪一个通道的数据，该参数可以为空，默认为通道1</param> ///<param name="ATxRx">==0:仅仅接收Rx报文；>0: Tx Rx报文都读取回来，该参数可以为空，默认为只接收Rx报文</param> ///<returns>返回实际读取到的报文数量，如果没有任何报文，则返回值为 0</returns>

返回值	实际读取到的报文数量，如果没有任何报文，则返回值为 0。 // 如果一直无法读取到报文值，请检查是否通过 TsMasterApi.tsapp_enable_receive_fifo()开启了内部 Buffer

示例	//申请存储100个报文的数组
	TLIBCAN[] canBuffer = new TLIBCAN[100];
	int revCnt = 0;
	//读取报文缓存，如果Rev>0，表示读取到了报文
	revCnt = TsMasterApi.tsapp_receive_can_message_list(
	ref canBuffer,
	100,
	APP_CHANNEL.CHN1, //读取通道1的报文数据
	READ_TX_RX_DEF.TX_RX_MESSAGES); //接收TX/RX所有报文，如果只读取接收端的报文，则修改为READ_TX_RX_DEF.ONLY_RX_MESSAGES
	if (revCnt == 0)
	{
	//Log("No Message Received! ");
	return;
	}
	msgcount += revCnt;
	lblCount.Text = msgcount.ToString();
	for (int i = 0; i < revCnt; i++)
	{
	string msg = "CAN Msg: ";
	if (canBuffer[i].FIsTx)
	msg += "Tx ";
	else
	msg += "Rx ";
	msg += canBuffer[i].FIdentifier.ToString("X8");
	Log(msg);
	}

22. tsapp_receive_canfd_message_list

函数名称	int tsapp_receive_canfd_message_list(ref TLIBCANFD[] ACANMsgBuffer, int ACANBufferSize, APP_CHANNEL AChn = APP_CHANNEL.CHN1, READ_TX_RX_DEF ARxTx = READ_TX_RX_DEF.ONLY_RX_MESSAGES)
功能介绍	读取 CANFD 报文缓存

调用位置	在需要读取 CANFD 报文的场合
输入参数	///<summary> /// 通过读取设备内部缓存，读取接收数据 ///</summary> ///<param name="ADeviceHandle">设备句柄</param> ///<param name="ACANMsgBuffer">数据Buffer，用于存储读取到的报文，该Buffer需要函数调用方创建</param> ///<param name="ACANBufferSize">消息Buffer的大小</param> ///<param name="AChn">目标通道：对于多通道设备，本函数选择读取哪一个通道的数据，该参数可以为空，默认为通道1</param> ///<param name="ATxRx">==0:仅仅接收Rx报文；>0: Tx Rx报文都读取回来，该参数可以为空，默认为只接收Rx报文</param> ///<returns>返回实际读取到的报文数量，如果没有任何报文，则返回值为 0</returns>
返回值	返回实际读取的报文数量，如果没有任何报文，则返回值为 0。如果一直无法读取报文，请检查是否通过函数 TsMasterApi.tsapp_enable_receive_fifo();开启了驱动内部 Buffer

示例	TLIBCANFD[] canBuffer = new TLIBCANFD[10];
	int revCnt = 0;
	revCnt = TsMasterApi.tsapp_receive_canfd_message_list(ref canBuffer, 10, APP_CHANNEL.CHN1, READ_TX_RX_DEF.TX_RX_MESSAGES);
	if (revCnt == 0)
	{
	Log("No Message Received! ");
	return;
	}
	for (int i = 0; i < revCnt; i++)
	{
	string msg = "CANFD Msg: ";
	if (canBuffer[i].FIsTx)
	msg += "Tx ";
	else
	msg += "Rx ";
	msg += canBuffer[i].FIdentifier.ToString("X8");
	Log(msg);
	}

23. tsapp_configure_baudrate_lin

函数名称	int tsapp_configure_baudrate_lin(int AIdxChn, Single ABaudrateKbps);

功能介绍	设置 LIN 通道的波特率等参数
调用位置	连接硬件设备之前，先设置通道的参数
输入参数	// hardware settings ///<summary> /// 设置通道波特率等参数 ///</summary> ///<param name="AIdxChn">应用程序通道编号</param> ///<param name="ABaudrateKbps">波特率（kbps）</param>
返回值	==0:连接成功 其他：检查错误码

示例	if(TsMasterApi.tsapp_configure_baudrate_lin((int)APP_CHANNEL.CHN1, (Single)19.2) == 0)
	{
	Log("LIN Channel " + (1).ToString() + " baudrate has been configured");
	}
	else
	{
	Log("LIN Channel " + (1).ToString() + " baudrate failed");
	}

24. tsapp_register_event_can

函数名称	int tsapp_register_event_can(IntPtr AObj, TCANQueueEvent AEvent);
功能介绍	注册 CAN 数据包接收回调函数
调用位置	如果用户想基于接收回调机制来处理接收的报文，在连接工具成功 过后，可以调用此函数注册回调函数。
输入参数	///<summary> /// 注册CAN报文接收回调函数 ///</summary> ///<param name="AObj">唯一句柄，可以用作标识符用</param> ///<param name="AEvent">处理接收报文的回调函数</param>

	///<returns></returns>
返回值	==0：注册成功 其他值：注册失败
示例	TsMasterApi.tsapp_register_event_can((IntPtr)0, vCANQueueEventObj);

25. tslin_set_node_funtiontype

函数名称	int tslin_set_node_funtiontype(APP_CHANNEL AIdxChn, TLINNodeType AFunctionType);
功能介绍	设置 LIN 节点工作模式：主节点，从节点，监听节点
调用位置	初始化 LIN 通道的函数中
输入参数	/// <summary> /// 设置LIN节点工作模式 /// </summary> /// <param name="AIdxChn">通道编号</param> /// <param name="AFunctionType">0：主节点；1：从节点；2：监听节点，节点类型定义如下： public enum TLINNodeType { T_MasterNode = 0, T_SlaveNode = 1, T_MonitorNode = 2 };</param> /// <returns></returns>
返回值	==0：设置成功 其他值：设置失败
示例	TsMasterApi.tslin_set_node_funtiontype(APP_CHANNEL.CHN1, TLINNodeType.T_MasterNode)

26. tsapp_transmit_lin_async

函数名称	int tsapp_transmit_lin_async(ref TLIBLIN ALIN);
功能介绍	异步发送 LIN 报文
调用位置	在需要发送 LIN 报文的场合

输入参数	/// <summary> /// 异步发送LIN报文 /// </summary> /// <param name="ALIN">LIN报文数据</param> /// <returns></returns>
返回值	==0：发送成功 其他值：发送失败

示例	TLIBLIN linMsg = new TLIBLIN(APP_CHANNEL.CHN1, 0x3C, 8, true);
	linMsg.FData[0] = 0x7F;
	linMsg.FData[1] = 0x03;
	linMsg.FData[2] = 0x22;
	linMsg.FData[3] = 0xFD;
	linMsg.FData[4] = 0x10;
	linMsg.FData[5] = 0xFF;
	linMsg.FData[6] = 0xFF;
	linMsg.FData[7] = 0xFF;
	if (TsMasterApi.tsapp_transmit_lin_async(ref linMsg) == 0x00)
	{
	Log("Async lin message success!");
	}
	else
	{
	Log("Async lin message failed!");
	}

27. tsapp_clear_lin_receive_buffers

函数名称	Int tsapp_clear_lin_receive_buffers(APP_CHANNEL AChn);
功能介绍	清除 LIN 模块的接收 Buffer 缓存
调用位置	在准备开始一次发送和接收之前，调用此函数清除驱动中的 LIN 缓存中的数据，确保接收到的数据是最新的
输入参数	/// <summary> /// 清除LIN通道的缓存数据 /// </summary> /// <param name="AChn">需要清除缓存的LIN通道编号</param> /// <returns></returns>
返回值	==0：清除成功 其他值：清除失败

示例	TsMasterApi.tsapp_clear_lin_receive_buffers(APP_CHANNEL.CHN1);

28. tsapp_transmit_header_and_receive_msg

函数名称	int tsapp_transmit_header_and_receive_msg(APP_CHANNEL AChn, byte AIdentifier,byte ADLC, ref TLIBLIN ALINData, int ATimeoutMs);
功能介绍	LIN 主节点：发送 LIN 报文的 Header，并接收从节点回复的 LIN 报文数据。
调用位置	此函数用于主节点，发送 LIN 报文头，并且接收从节点回复的数据。
输入参数	/// <summary> /// 发送LIN报文Header，并接受从节点回复的报文数据 /// </summary> /// <param name="AChn">通道</param> /// <param name="AIdentifier">报文ID</param> /// <param name="ADLC">要接收的数据直接长度</param> /// <param name="ALINData">用来存储接收数据的LIN报文</param> /// <param name="ATimeoutMs">接收等待超时时间，超过此时间，则接收失败</param> /// <returns></returns>
返回值	==0：接收成功 其他值：接收失败

示例	TLIBLIN receivedMsg = new TLIBLIN();
	
	TsMasterApi.tsapp_clear_lin_receive_buffers(APP_CHANNEL.CHN1);
	if (TsMasterApi.tsapp_transmit_header_and_receive_msg(APP_CHANNEL.CHN1, 0x3D, 8, ref receivedMsg, 200) == 0x00)
	{
	Log(receivedMsg.FIdentifier.ToString("X2") + ":" + receivedMsg.FDLC.ToString() + ":" + receivedMsg.FData[0].ToString("X2") + receivedMsg.FData[1].ToString("X2") + receivedMsg.FData[2].ToString("X2") + receivedMsg.FData[3].ToString("X2") + receivedMsg.FData[4].ToString("X2") + receivedMsg.FData[5].ToString("X2") + receivedMsg.FData[6].ToString("X2") + receivedMsg.FData[7].ToString("X2"));
	}
	else
	{

	Log("Received LIN Message Failed"); }

29. tsapp_unregister_event_can

函数名称	int tsapp_unregister_event_can(IntPtr AObj, TCANQueueEvent AEvent);
功能介绍	反注册 CAN 数据接收函数
调用位置	在不需要采用回调机制接收 CAN 报文的时候，调用此函数
输入参数	///<summary> /// 反注册CAN报文接收回调函数 ///</summary> ///<param name="AObj">唯一句柄，可以用作标识符用</param> ///<param name="AEvent">处理接收报文的回调函数</param> ///<returns></returns>
返回值	==0：注册成功 其他值：注册失败
示例	TsMasterApi.tsapp_unregister_event_can((IntPtr)0, vCANQueueEventObj);

30. tsapp_enumerate_hw_devices

函数名称	int tsapp_enumerate_hw_devices(out int ACount);
功能介绍	枚举当前插在电脑上的能够使用的 CAN 设备数量
调用位置	这个函数必须在 Application 没有连接（Connect）之前使用。因为在连接状态下调用此函数的话，会影响一些设备的 USB 通讯。
输入参数	/// <summary> /// 获取当前可以使用的设备数量，这个函数必须在APP处于未连接状态的时候使用 /// </summary> /// <param name="ACount">传入out参数，枚举到的当前设备数量</param> /// <returns></returns>
返回值	==0：枚举设备成功 其他值：枚举设备失败
示例	TsMasterApi.tsapp_enumerate_hw_devices(out hardwareNum); 获取当前 PC 端可以使用的设备数量，获取值存放在 hardwareNum 变量中。

31. tsapp_get_hw_info_by_index

函数名称	int tsapp_get_hw_info_by_index(int AIndex,ref TLIBHWInfo AHWInfo);
功能介绍	根据索引值获取硬件设备的信息
调用位置	在需要查询硬件设备信息的场合。同星，Vector 等设备都有自己的唯一信息，用户应用程序可以读取此唯一信息，用于自己应用程序进行权限管理，加密等场合。
输入参数	/// <summary> /// 根据设备索引值获取该设备的详细信息 /// </summary> /// <param name="AIndex">该设备的索引值</param> /// <param name="AHWInfo">设备的详细信息存放在此结构体中</param> /// <returns></returns>
返回值	==0：注册成功 其他值：注册失败

示例	//首先调用API函数获取设备信息值
	if (TsMasterApi.tsapp_get_hw_info_by_index(i, ref tmpDeviceInfo) == 0)
	{
	//把获取的信息值打印到界面上
	LogDeviceInformation(tmpDeviceInfo.FDeviceInformation);
	}

32. tsapp_get_hw_info_by_index_verbose

函数名称	int tsapp_get_hw_info_by_index_verbose(int AIndex, TLIBBusToolDeviceType* ADeviceType, char* AVendorNameBuffer, int AVendorNameBufferSize, char* ADeviceNameBuffer, int ADeviceNameBufferSize, char* ASerialStringBuffer, int ASerialStringBufferSize);
功能介绍	根据设备索引值获取设备信息。跟 tsapp_get_hw_info_by_index 函数相比，获取的信息直接存储在输入的数组中，用户根据自己习惯调用。

调用位置	在需要查询硬件设备信息的场合。同星，Vector 等设备都有自己的唯一信息，用户应用程序可以读取此唯一信息，用于自己应用程序进行权限管理，加密等场合。
输入参数	/// <summary> /// 根据设备索引值获取当前设备的信息 /// </summary> /// <param name="AIndex">设备索引值</param> /// <param name="ADeviceType">设备类型：CAN/LIN</param> /// <param name="AVendorNameBuffer">存放供应商名称的数组</param> /// <param name="AVendorNameBufferSize">存放供应商名称的数组大小</param> /// <param name="ADeviceNameBuffer">存放设备名称的数组</param> /// <param name="ADeviceNameBufferSize">存放设备名称的数组大小</param> /// <param name="ASerialStringBuffer">存放设备串行编号数组</param> /// <param name="ASerialStringBufferSize">存放设备串行编号数组大小</param> /// <returns></returns>
返回值	==0：注册成功 其他值：注册失败
示例	

33. tsapp_start_logging

函数名称	int tsapp_start_logging(string AFileName)
功能介绍	启动报文记录，记录文件格式未 blf 格式
调用位置	需要记录整个运行过程中总线上的报文通讯
输入参数	/// <summary> /// <summary> /// 启动报文记录，记录文件格式未blf格式 /// </summary> /// <param name="AFileName">记录文件名（后缀为.blf），可以为空字符串“”</param> /// <returns>==0:函数执行成功；其他：错误码</returns> 注意:如果 AFileName 传输为空字符串“”,则数据文件默认存储在 TSMaster 的安装路径下面。AFile 数据格式路径不能出错，如果路径是无效的，创建该数据文件无效，会触发模块内部异常。
返回值	==0：执行成功 其他值：查询错误码

示例	TsMasterApi.tsapp_start_logging(@".\TestData" + DateTime.Now.ToString("dd-HH-mm-ss") + ".blf");

34. tsapp_stop_logging

函数名称	int tsapp_stop_logging()
功能介绍	停止报文记录
调用位置	该函数需要和 tsapp_start_logging 配合使用，停止报文记录，报文会保存为相应的 blf 文件。
输入参数	无
返回值	==0：执行成功 其他值：查询错误码
示例	

10.附件

1. 示例程序

2. 错误码编码信息：

```
ERR_CODE_LIST: array [0..ERR_CODE_COUNT-1] of string = (
    'OK',                                   // IDX_ERR_OK                          0
    'Index out of range',                   // IDX_ERR_IDX_OUT_OF_RANGE            1
    'Connect failed',                       // IDX_ERR_CONNECT_FAILED              2
    'Device not found',                     // IDX_ERR_DEV_NOT_FOUND               3
    'Read status timed out',                // IDX_ERR_READ_STATUS_TIMEDOUT        44
    'Callback already exists',              // IDX_ERR_CALLBACK_ALREADY_EXISTS     45
    'Callback not exists',                  // IDX_ERR_CALLBACK_NOT_EXISTS         46
    'File corrupted or not recognized',     // IDX_ERR_FILE_INVALID                = 47; // database file corrupted or not recognized
    'Database unique id not found',         // IDX_ERR_DB_ID_NOT_FOUND             = 48; // database unique id not found
```

'Software API parameter invalid', // IDX_ERR_SW_API_PAEAMETER_INVALID = 49; // software api parameter invalid
'Software API generic timed out', // IDX_ERR_SW_API_GENERIC_TIMEOUT = 50; // software api generic timed out
'Software API set hw config. failed', // IDX_ERR_SW_API_SET_CONF_FAILED = 51; // software api set hw conf failed
'Index out of bounds', // IDX_ERR_SW_API_INDEX_OUT_OF_BOUNDS = 52; // index out of bounds
'RX wait timed out', // IDX_ERR_SW_API_WAIT_TIMEOUT = 53; // rx wait timed out
'Get I/O failed', // IDX_ERR_SW_API_GET_IO_FAILED = 54; // get io failed
'Set I/O failed', // IDX_ERR_SW_API_SET_IO_FAILED = 55; // set io failed
'An active replay is already running', // IDX_ERR_SW_API_REPLAY_ON_GOING = 56; // a replay is already on goning
'Instance not exists', // IDX_ERR_SW_API_INSTANCE_NOT_EXISTS = 57; // instance not exists
'CAN message transmit failed', // IDX_ERR_HW_CAN_TRANSMIT_FAILED = 58; // can transmit frame failed
'No response from hardware', // IDX_ERR_HW_NO_RESPONSE = 59; // hardware no response to pc
'CAN message not found', // IDX_ERR_SW_CAN_MSG_NOT_FOUND = 60; // can cyclic message id not found
'User CAN receive buffer empty', // IDX_ERR_SW_CAN_RECV_BUFFER_EMPTY = 61; // user can recv message buffer empty
'CAN total receive count <> desired count',// IDX_ERR_SW_CAN_RECV_PARTIAL_READ = 62; // total read count <> desired read count
'LIN config failed', // IDX_ERR_SW_API_LINCONFIG_FAILED = 63;
'LIN frame number out of range', // IDX_ERR_SW_API_FRAMENUM_OUTOFRANGE = 64;
'LDF config failed', // IDX_ERR_SW_API_LDFCONFIG_FAILED = 65;
'LDF config cmd error', // IDX_ERR_SW_API_LDFCONFIG_CMDERR = 66;
'TSMaster envrionment not ready', // IDX_ERR_SW_ENV_NOT_READY = 67; // encryption random number not retrieved
'reserved failed', // IDX_ERR_RESERVED_FAILED = 68;
'XL driver error', // IDX_ERR_XL_ERROR = 69;
'index out of range', // IDX_ERR_SEC_INDEX_OUTOFRANGE = 70;
'string length out of range', // IDX_SEC_ERR_STRINGLENGTH_OUTFOF_RANGE = 71;
'key is not initialized', // IDX_SEC_ERR_KEY_IS_NOT_INITIALIZATION = 72;
'key is wrong', // IDX_SEC_ERR_KEY_IS_WRONG = 73;
'write not permitted', // IDX_SEC_ERR_NOT_PERMIT_WRITE = 74;
'16 bytes multiple', // IDX_SEC_ERR_16BYTES_MULTIPLE = 75;
'LIN channel out of range', // IDX_ERR_LIN_CHN_OUTOF_RANGE = 76;
'DLL not ready', // IDX_ERR_DLL_NOT_READY = 77;
'Feature not supported', // IDX_ERR_FATURE_NOT_SUPPORTED = 78;
'common service error', // IDX_ERR_COMMON_SERV_ERROR = 79;
'read parameter overflow', // IDX_ERR_READ_PARA_OVERFLOW = 80;
'Invalid application channel mapping', // IDX_ERR_INVALID_CHANNEL_MAPPING
'libTSMaster generic operation failed', // IDX_ERR_TSLIB_GENERIC_OPERATION_FAILED = 82;

```
    'item already exists',                          // IDX_ERR_TSLIB_ITEM_ALREADY_EXISTS          = 83;
    'item not found',                               // IDX_ERR_TSLIB_ITEM_NOT_FOUND               = 84;
    'logical channel invalid',                      // IDX_ERR_TSLIB_LOGICAL_CHANNEL_INVALID      = 85;
    'file not exists',                              // IDX_ERR_FILE_NOT_EXISTS                    = 86;
    'no init access, cannot set baudrate',          // IDX_ERR_NO_INIT_ACCESS                     = 87;
    'the channel is inactive',                      // IDX_ERR_CHN_NOT_ACTIVE                     = 88;
    'the channel is not created',                   // IDX_ERR_CHN_Not_Created                    = 89;
    'length of the appname is out of range',        // IDX_ERR_APPNAME_LENGTH_OUT_OF_RANGE        = 90;
    'project is modified',                          // IDX_ERR_PROJECT_IS_MODIFIED                = 91;
    'signal not found in database',                 // IDX_ERR_SIGNAL_NOT_FOUND_IN_DB             = 92;
    'message not found in database',                // IDX_ERR_MESSAGE_NOT_FOUND_IN_DB            = 93;
    'TSMaster is not installed',                    // IDX_ERR_TSMASTER_IS_NOT_INSTALLED          = 94;
    'Library load failed',                          // IDX_ERR_LIB_LOAD_FAILED                    = 95;
    'Library function not found',                   // IDX_ERR_LIB_FUNCTION_NOT_FOUND             = 96;
    'Library not initialized',                      // IDX_ERR_LIB_NOT_INITIALIZED                = 97;
    'PCAN generic operation error',                 // IDX_ERR_PCAN_GENRIC_ERROR                  = 98;
    'Kvaser generic operation error',               // IDX_ERR_KVASER_GENERIC_ERROR               = 99;
    'ZLG generic operation error',                  // IDX_ERR_ZLG_GENERIC_ERROR                  = 100;
    'ICS generic operation error',                  // IDX_ERR_ICS_GENERIC_ERROR                  = 101;
    'TC1005 generic operation error',               // IDX_ERR_TC1005_GENERIC_ERROR               = 102;
    'System variable not found',                    // IDX_ERR_SYSTEM_VAR_NOT_FOUND               = 103;
    'Incorrect system variable type',               // IDX_ERR_INCORRECT_SYSTEM_VAR_TYPE          = 104;
    'Message not existing, update failed',          // IDX_ERR_CYCLIC_MSG_NOT_EXIST               = 105;
    'Specified baudrate not available'              // IDX_ERR_BAUD_NOT_AVAIL                     = 106;
);
```

11. 常见异常解答

1. 调用 API 开发的程序无法正常打开 CAN 驱动

- **症状描述：**

TSMaster 已经安装（不代表完全正常），可以打开正常使用。但是基于 TSMaster API 开发的程序无法正常工作，总是提示打开 CAN 总线设备失败。如果打印 API 的返回值，发现都是 97，也就是代表驱动没有正常初始化。

- **原因分析：**

TSMaster 在安装的时候需要把一些配置信息和路径信息等写入到注册表中，但是很多电脑里面设置了权限管理，TSMaster 在安装的时候无法正常完成所有配置信息的写入。造成用户程序在调用 API 的时候，查找不到库文件的路径，从而无法完成 API 驱动的初始化。

- **解决办法：**

建议：TSMaster 里面增加一个模块来管理这些注册表信息是否就绪，如果未就绪，看看能否直接补齐，或者至少给与用户提示信息。