

TsCAN API 编程指导

Linux C++版

V1.2



目录

1. 什么情况下需要此文档？	7
2. 添加库文件	7
1. C 语言：	7
2. CPP 语言	8
3. 测试验证	8
4. 总线数据类型定义	9
1. TLibCAN: CAN 总线数据类型	9
成员：	9
调用示例：	10
2. TLibCANFD: CANFD 总线数据类型	10
成员：	10
调用示例：	11
3. TLibLIN: LIN 总线数据类型	11
成员：	11
调用示例：	12
4. TLibFlexray: Flexray 总线数据类型	12
成员：	13
调用示例：	14
5. 报文发送	14
1. CAN 报文发送	14
单帧异步发送：	14
单帧同步发送：	14
周期发送：	14
删除周期发送：	14
2. CANFD 报文发送	14
can_frame.init_w_std_id(0x11, 0xf);	15
单帧异步发送：	15
单帧同步发送：	15
周期发送：	15

删除周期发送:	15
3. LIN 报文发送	15
单帧异步发送:	15
单帧同步发送:	15
4. Flexray 报文发送	16
单帧异步发送:	16
单帧同步发送:	16
6. 报文接收	16
1. 回调函数方式:	16
简介:	16
注册回调函数:	16
回调函数使用:	16
2. 读取设备消息缓存的方式:	18
简介:	18
CAN 报文获取	18
CANFD 报文获取	19
LIN 报文获取	21
Flexray 报文获取	22
7. 接口函数介绍	23
1. initialize_lib_tscan	23
2. finalize_lib_tscan	23
3. tscan_scan_devices	23
4. tscan_connect	24
5. tscan_disconnect_by_handle	24
6. tscan_config_can_by_baudrate	24
7. tscan_register_event_can	25
8. tscan_unregister_event_can	25
9. tsfifo_add_can_canfd_pass_filter	25
10. tsfifo_receive_can_msgs	26
11. tsfifo_clear_can_receive_buffers	27
12. tscan_transmit_can_async	27
13. tscan_config_canfd_by_baudrate	27

14. tscan_register_event_canfd	28
15. tscan_unregister_event_canfd	29
16. tsfifo_receive_canfd_msgs	29
17. tsfifo_clear_can_receive_buffers	30
18. tscan_transmit_canfd_async	30
19. tslin_set_node_funtiontype	31
20. tslin_config_baudrate	31
21. tslin_register_event_lin	31
22. tslin_unregister_event_lin	32
23. tsfifo_add_lin_pass_filter	32
24. tsfifo_receive_lin_msgs	33
25. tsfifo_clear_lin_receive_buffers	33
26. tslin_transmit_lin_async	34
27. tscan_register_event_canfd_whatle	34
28. tscan_unregister_event_canfd_whatle	34
29. tslin_register_event_lin_whatle	35
30. tslin_unregister_event_lin_whatle	35
31. tsflexray_unregister_event_flexray_whatle	35
32. tsflexray_register_event_flexray_whatle	36
33. tscan_register_event_fastlin	36
34. tscan_unregister_event_fastlin	36
35. tscan_get_device_info	37
36. tscan_get_can_channel_count	37
37. tscan_get_lin_channel_count	37
38. tscan_get_flexray_channel_count	38
39. tscan_disconnect_by_handle	38
40. tscan_disconnect_all_devices	38
41. initialize_lib_tscan	38
42. finalize_lib_tscan	39
43. tscan_transmit_can_sync	39
44. tscan_transmit_can_sequence	39
45. tscan_transmit_can_async	40

46. tscan_config_can_by_baudrate	40
47. tscan_add_cyclic_msg_can	40
48. tscan_delete_cyclic_msg_can	41
49. tscan_add_cyclic_msg_canfd	41
50. tscan_delete_cyclic_msg_canfd	41
51. tscan_transmit_canfd_sync	42
52. tscan_transmit_canfd_sequence	42
53. tscan_transmit_canfd_async	42
54. tscan_config_canfd_by_baudrate	43
55. tsfifo_receive_canfd_msgs	43
56. tsfifo_clear_canfd_receive_buffers	43
57. tsflexray_set_controller_frametrigger	44
58. tsflexray_set_controller	44
59. tsflexray_set_frametrigger	45
60. tsflexray_transmit_sync	45
61. tsflexray_transmit_async	45
62. tsfifo_receive_flexray_msgs	46
63. tsfifo_clear_flexray_receive_buffers	46
64. tsflexray_start_net	46
65. tsflexray_stop_net	47
66. tsfifo_read_flexray_buffer_frame_count	47
67. tsfifo_read_flexray_tx_buffer_frame_count	47
68. tsfifo_read_flexray_rx_buffer_frame_count	48
69. tslin_set_node_funtiontype	48
70. tslin_clear_schedule_tables	48
71. tslin_transmit_lin_sync	49
72. tslin_transmit_lin_async	49
73. tslin_transmit_fastlin_async	49
74. tslin_config_baudrate	50
75. tsfifo_receive_lin_msgs	50
76. tsfifo_receive_fastlin_msgs	50
77. tscan_get_error_description	51

78. tsreplay_add_channel_map	51
79. tsreplay_clear_channel_map	51
80. tsreplay_start_blf	52
81. tsreplay_stop	52
82. tsdiag_can_create	52
83. tsdiag_can_delete	53
84. tsdiag_can_delete_all	53
85. tsdiag_can_attach_to_tscan_tool	53
86. tstp_can_send_functional	54
87. tstp_can_send_request	54
88. tstp_can_request_and_get_response	55
89. tsdiag_can_session_control	55
90. tsdiag_can_routine_control	56
91. tsdiag_can_communication_control	56
92. tsdiag_can_security_access_request_seed	56
93. tsdiag_can_security_access_send_key	57
94. tsdiag_can_request_download	58
95. tsdiag_can_request_upload	58
96. tsdiag_can_transfer_data	59
97. tsdiag_can_request_transfer_exit	59
98. tsdiag_can_write_data_by_identifier	60
99. tsdiag_can_read_data_by_identifier	60
100. tsflexray_register_event_flexray	61
101. tsflexray_unregister_event_flexray	61
102. tsflexray_unregister_pretx_event_flexray	61
103. tsflexray_register_pretx_event_flexray	62
8. 示例 工程	62

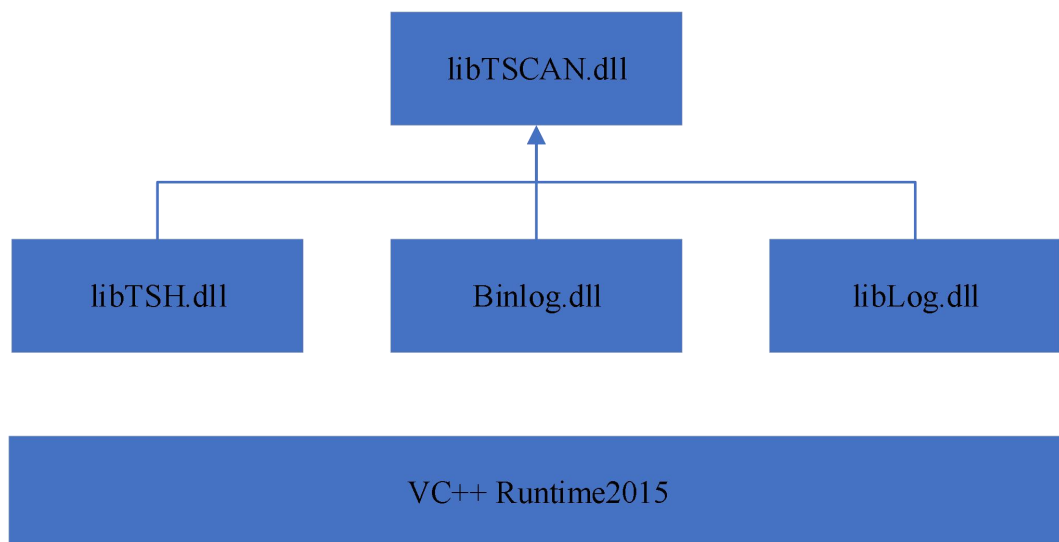
1. 什么情况下需要此文档？

用户基于 C++ 编程语言，对上海同星智能科技有限公司的 TSCAN 系列工具 (TC1001,TC1011,TC1012,TC1013,TC1014,TC1016)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. 添加库文件

1. C 语言：

要实现对 TOSUN 系列 CAN/CANFD, LIN 设备的操作，需要基于 libTSCAN.dll 动态链接库文件。该文件集成了上海同星公司对 TS 系列工具设备在 Win32 平台上的所有 API 接口。libTSCAN.dll 的运行，除了依赖常用的 C++运行库如 mfc140.dll, msvcrt140.dll 等，还需要依赖 libTSH.dll, binlog.dll 以及 liblog.dll。其库文件依赖关系如下图所示：



要调用 dll 内部的接口函数，需要在工程中添加 TSCANDef.h 头文件。该文件中主要定义了使用 API 所需要用到的数据结构类型以及函数指针类型，如下所示：



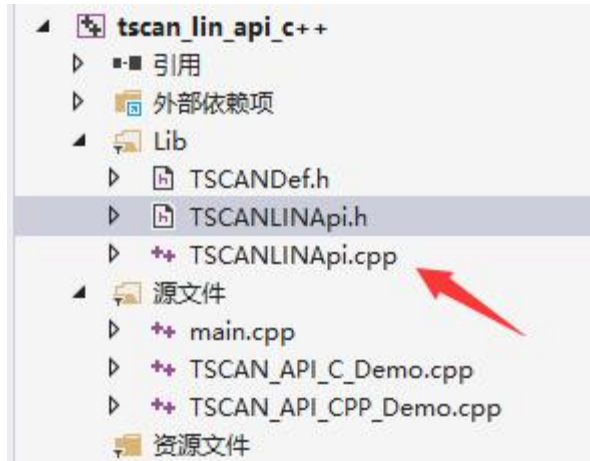
1. 引用头文件

开发人员可以根据头文件定义，直接引用 lib 库文件进行开发。也可以根据数据结构定

义，动态载入函数指针，详细情况见例程。

2. C++ 语言

为了方便 C++ 人员操作硬件设备，本工程提供了基于 C++ 类的封装 TSCANLINApi。要引用此定义，需要引用三个文件：TSCANLINApi.h, TSCANLINApi.c 以及 TSCANDef.h。如下图所示：

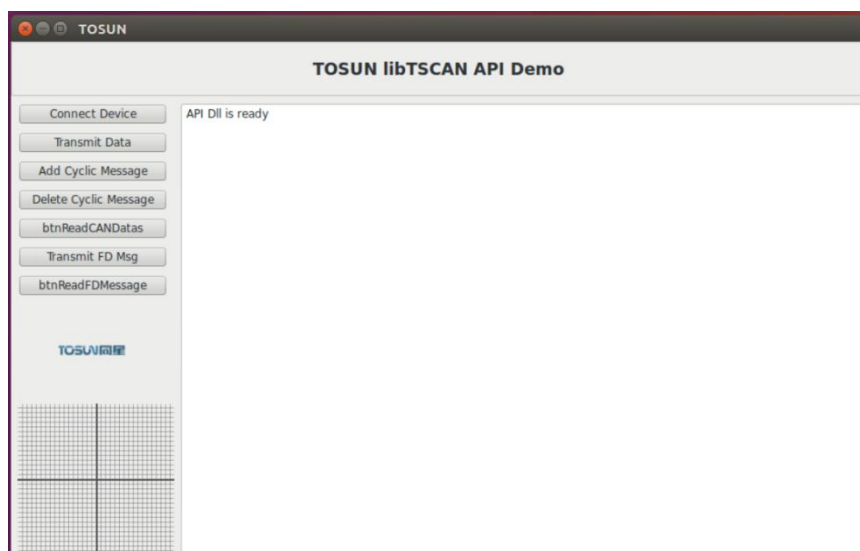


TSCANAPI C++ 库文件引用

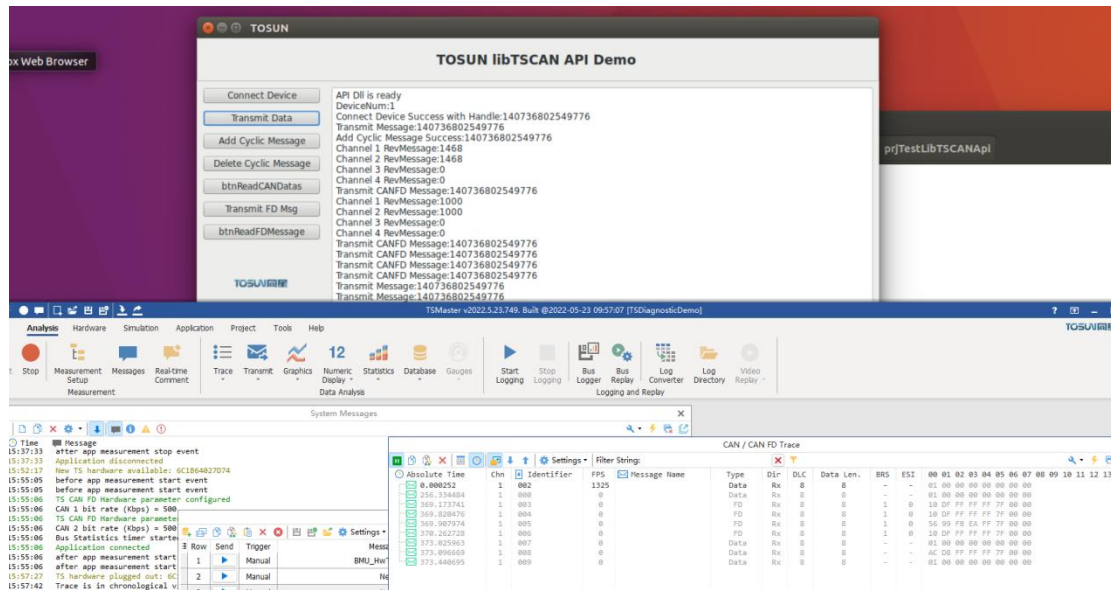
在 TSCANLINApi.c 中，封装了常用的操作函数，详细的调用方法见例程所示。

3. 测试验证

在驱动库的目录下面，附带了一个简单的测试 UI 程序 prjTestLibTSCANApi。用管理员权限打开此程序，界面如下图所示：



用户可以插上 TOSUN 的设备，并验证设备连接，周期发送报文，读取报文等功能。其中设备连接过程中，默认初始化各个通道的波特率参数为：仲裁场 500kBps，数据场 2000kBps，使能内部 120 欧终端电阻。联合调试效果如下图所示：



4. 总线数据类型定义

1. TLibCAN: CAN 总线数据类型

```
typedef struct _TLibCAN {
    u8 FIdxChn;                // channel index starting from 0
    TCANProperty FProperties;   //CAN Property
    u8 FDLC;                   // dlc from 0 to 8
    u8 FReserved;              // reserved to keep alignment
    s32 FIdentifier;           // CAN identifier
    u64 FTimeUS;               // timestamp in us
    u8 FData[8];               // 8 data bytes to send
} TLibCAN,*PLibCAN;
```

成员：

FData: 帧数据。最大长度为 8Bytes

FDLC: 帧长度。

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

FldxChn: 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。

其中, 属性字节定义如下:

【1】 FProperties: CAN 属性定义: 该参数默认为 0, 共八个 bits, 每一个位的定义如下:

Bit	意义
0	0: Rx 接收报文; 1: Tx 发送报文

1	0: data frame 数据帧; 1: remote frame 远程帧
2	0: std frame 标准帧; 1: extended frame 扩展帧
3-5	Reserved
6	0: 不记录; 1: 已经被记录
7	Reserved

调用示例:

```
TLibCAN CANMsg={0,0x1,8,0,0x123,0,{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08}};
```

2. TLibCANFD: CANFD 总线数据类型

```
typedef struct _TLibCANFD {  
    u8 FIdxChn;                // channel index starting from 0  
    TCANProperty FProperties;   //CAN Property  
    u8 FDLC;                   // dlc from 0 to 15  
    TCANFDProperty FFDProperties; //FD Property  
    s32 FIdentifier;           // CAN identifier  
    u64 FTimeUS;               // timestamp in us  
    u8 FData[64];              // 64 data bytes to send  
}TLibCANFD, * PLibCANFD;
```

成员:

FData: 帧数据。最大长度为 64Bytes

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。

其中, 两个属性字节定义如下:

【1】 FProperties: CAN 属性定义: 该参数默认为 0, 共八个 bits, 每一个位的定义如下:

Bit	意义
0	0: Rx 接收报文; 1: Tx 发送报文
1	0: data frame 数据帧; 1: remote frame 远程帧
2	0: std frame 标准帧; 1: extended frame 扩展帧
3-5	Reserved
6	0: 不记录; 1: 已经被记录
7	Reserved

【2】 FDProperty: FD 属性定义:

Bit	意义
0	0: 普通 CAN 报文; 1: FDCAN 报文
1	0: 关闭 BRS; 1: 开启 BRS
2	是否发生错误 (ESI Flag)
3-7	Reserved

// [7-3] tbd

// [2] ESI, The E RROR S TATE I NDICATOR (ESI) flag is transmitted dominant by error active nodes, recessive by error passive nodes. ESI does not exist in CAN format frames

// [1] BRS, If the bit is transmitted recessive, the bit rate is switched from the standard bit rate of the A RBITRATION P HASE to the preconfigured alternate bit rate of the D ATA P HASE . If it is transmitted dominant, the bit rate is not switched. BRS does not exist in CAN format frames.

// [0] EDL: 0-normal CAN frame, 1-FD frame, added 2020-02-12, The E XTENDED D ATA L ENGTH (EDL) bit is recessive. It only exists in CAN FD format frames

调用示例:

```
TLibCANFD CANFDMsg = {0, 0x100, true, false, false, 8, { 0, 1, 2, 3, 4, 5, 6, 7 }};
```

3. TLibLIN: LIN 总线数据类型

```
typedef struct _TLIN {  
    u8 FIdxChn;           // channel index starting from 0  
    u8 FErrCode;          // 0: normal  
    TLINProperty FProperties; // Property of LIN Message  
    u8 FDLC;              // dlc from 0 to 8  
    u8 FIdentifier;        // LIN identifier:0--64  
    u8 FChecksum;          // LIN checksum  
    u8 FStatus;            // place holder 1  
    u64 FTimeUS;           // timestamp in us //Modified by Eric 0321  
    u8x8 FData;            // 8 data bytes to send  
} TLibLIN, *PLibLIN;
```

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。

其中, 属性字节定义如下:

【1】 Properties: LIN 属性定义: 该参数默认为 0, 共八个 bits, 每一个位的定义如下:

Bit	意义
0	0: Rx 接收报文; 1: Tx 发送报文
1-3	Reserved
4-5	设备类型: 主节点, 从节点, 监听节点
6	0: 不记录; 1: 已经被记录
7	Reserved

调用示例:

```
TLIBLIN tLibLIN = {0,0x0,0x1,8,0x3F,0x0,0,0,{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08}};  
if (0 != tscan_connect()) return;  
if (0 == tslin_transmit_lin_async(ADeviceHandle, &tLibLIN)){  
}
```

4. TLibFlexray: Flexray 总线数据类型

```
typedef struct _TLIBFlexRay {  
    u8 FIdxChn;           // channel index starting from 0  
    u8 FChannelMask;      // 0: reserved, 1: A, 2: B, 3: AB  
    u8 FDir;              // 0: Rx, 1: Tx, 2: Tx Request  
    u8 FPayloadLength;    // payload length in bytes  
    u8 FActualPayloadLength; // actual data bytes  
    u8 FCycleNumber;      // cycle number: 0~63  
    u8 FCCType;           // 0 = Architecture independent, 1 = Invalid CC type, 2 =  
    Cyclone I, 3 = BUSDOCTOR, 4 = Cyclone II, 5 = Vector VN interface, 6 = VN - Sync - Pulse (only  
    in Status Event, for debugging purposes only)  
    u8 FFrameType;        // // 0 = raw flexray frame, 1 = error event, 2 = status, 3 =  
    start cycle  
    u16 FHeaderCRCA;      // header crc A  
    u16 FHeaderCRCB;      // header crc B  
    u16 FFrameStateInfo;   // bit 0~15, error flags  
    u16 FSlotId;           // static seg: 0~1023  
    u32 FFrameFlags;       // bit 0~22  
    // 0 1 = Null frame.  
    // 1 1 = Data segment contains valid data  
    // 2 1 = Sync bit  
    // 3 1 = Startup flag  
    // 4 1 = Payload preamble bit  
    // 5 1 = Reserved bit  
    // 6 1 = Error flag(error frame or invalid frame)
```

```

// 7 Reserved
// 8 Internally used in CANoe / CANalyzer
// 9 Internally used in CANoe / CANalyzer
// 10 Internally used in CANoe / CANalyzer
// 11 Internally used in CANoe / CANalyzer
// 12 Internally used in CANoe / CANalyzer
// 13 Internally used in CANoe / CANalyzer
// 14 Internally used in CANoe / CANalyzer
// 15 1 = Async.monitoring has generated this event
// 16 1 = Event is a PDU
// 17 Valid for PDUs only.The bit is set if the PDU is valid(either if the PDU has no
// update bit, or the update bit for the PDU was set in the received frame).
// 18 Reserved
// 19 1 = Raw frame(only valid if PDUs are used in the configuration).A raw frame may
// contain PDUs in its payload
// 20 1 = Dynamic segment 0 = Static segment
// 21 This flag is only valid for frames and not for PDUs. 1 = The PDUs in the
payload of // this frame are logged in separate logging entries. 0 = The PDUs in the payload
of this // frame must be extracted out of this frame.The logging file does not contain
separate // PDU - entries.
// 22 Valid for PDUs only.The bit is set if the PDU has an update bit
u32 FFrameCRC; // frame crc
u64 FReserved1; // 8 reserved bytes
u64 FReserved2; // 8 reserved bytes
u64 FTimeUs; // timestamp in us
u8 FData[254]; // 6254 data bytes
}TLIBFlexRay, * PLibFlexRay;

```

成员：

FIdxChn: 通道索引
 FChannelMask: 通道掩码
 FPayloadLength: 有效负载长度(字节)
 FActualPayloadLength: 实际数据字节
 FCycleNumber: 周期数
 FFrameType: 报文类型
 FHeaderCRCA: CRCA 标头
 FHeaderCRCB: CRCB 标头
 FFrameStateInfo: 帧状态信息
 FSlotId: 静态段
 FFrameFlags: 22 位帧标志
 FFrameCRC: 帧 CRC
 FReserved1: 8 个预留字节
 FReserved2: 8 个预留字节

FTimeUs:时间戳(微妙)

FData:254 数据字节

调用示例:

```
TLIBFlexray tLIBFlexray ;  
tsflexray_transmit_async(ADeviceHandle, &tLIBFlexray);
```

5. 报文发送

1. CAN 报文发送

```
TLIBCAN can_frame;//创建一个报文结构体  
can_frame.init_w_std_id(0x11, 8);//初始化ID为0x11,报文DLC为8  
can_frame.set_data(0, 1, 2, 3, 4, 5, 6, 7);//设置报文数据
```

单帧异步发送:

```
tscan_transmit_can_async(ADeviceHandle, &can_frame);
```

单帧同步发送:

```
# 100 表示超时参数  
tscan_transmit_can_sync(ADeviceHandle, &can_frame, 100);
```

周期发送:

```
# 100ms 周期发送 can_frame 报文  
tscan_add_cyclie_msg_can(ADeviceHandle, &can_frame, 100);
```

删除周期发送:

```
# 删除周期发送 can_frame 报文  
tscan_delete_cyclie_msg_can(ADeviceHandle, &can_frame);
```

2. CANFD 报文发送

```
TLIBCANFD can_frame;
```

```
can_frame.init_w_std_id(0x11, 0xf);
```

单帧异步发送:

```
tscan_transmit_canfd_async(ADeviceHandle, &can_frame);
```

单帧同步发送:

100 表示超时参数

```
tscan_transmit_canfd_sync(ADeviceHandle, &can_frame, 100)
```

周期发送:

100ms 周期发送 can_frame 报文

```
tscan_add_cyclie_msg_canfd(ADeviceHandle, &can_frame, 100)
```

删除周期发送:

100ms 周期发送 can_frame 报文

```
tscan_delete_cyclie_msg_canfd(ADeviceHandle, &can_frame)
```

3. LIN 报文发送

```
TLIN lin_frame;  
lin_frame.init_w_id(0x11, 8); //初始化PID为0x11 报文长度8  
lin_frame.property__set_is_tx(true); //设置为发送报文  
for (u32 i = 0; i < 8; i++) {  
    flexRay_frame.FData[i] = 0;  
}
```

单帧异步发送:

```
tslin_transmit_lin_async(ADeviceHandle, &lin_frame);
```

单帧同步发送:

100 表示超时参数

```
tslin_transmit_lin_sync(ADeviceHandle, &lin_frame, 100);
```

4. Flexray 报文发送

```
TFlexRay flexRay_frame;  
flexRay_frame.init_w_slot_id(0x11, 8); //初始化PID为0x11 报文长度8  
flexRay_frame.property__set_is_tx(true); //设置为发送报文  
for (u32 i = 0; i < 254; i++) {  
    flexRay_frame.FData[i] = 0;  
}
```

单帧异步发送:

```
tsflexray_transmit_async(ADeviceHandle, &flexRay_frame);
```

单帧同步发送:

```
# 100 表示超时参数  
tsflexray_transmit_sync(ADeviceHandle, &flexRay_frame, 100);
```

6. 报文接收

1. 回调函数方式:

简介:

回调函数方式相当于 MCU 中的中断机制。当驱动中收到一个完整的 CAN 数据包过后，就会触发此回调函数执行。

注册回调函数:

采用函数 `tscan_register_event_can` 注册报文接收回调函数。回调函数中报文处理结构体参考其它章节相关内容。

回调函数使用:

//注: 在回调事件中, 尽量只做数值变换操作, 避免耗时操作
CAN 回调:

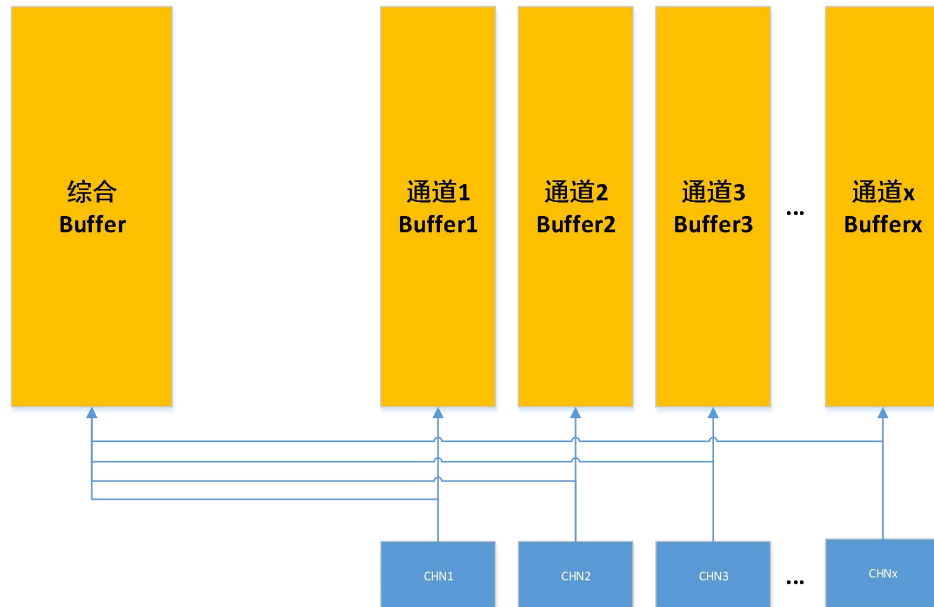

```
void ReceiveCANMessage(const TLibCAN* AData)
{
    //处理收到的报文数据 AData
}
retValue = tscan_register_event_can(ADeviceHandle, ReceiveCANMessage);
不使用 CAN 回调的时候:
retValue = tscan_unregister_event_can(ADeviceHandle, ReceiveCANMessage);
//ADeviceHandle 是设备句柄
CANFD 回调:
void ReceiveCANFDMessage(const TLibCANFD* AData)
{
    //处理收到的报文数据 AData
}
retValue = tscanfd_register_event_canfd(ADeviceHandle, ReceiveCANFDMessage);
不使用 CANFD 回调的时候:
retValue = tscanfd_unregister_event_canfd(ADeviceHandle, ReceiveCANFDMessage);
//ADeviceHandle 是设备句柄
LIN 回调:
void ReceiveLINMessage(const TLibLIN* AData)
{
    //处理收到的报文数据 AData
}
retValue = tslin_register_event_lin(ADeviceHandle, ReceiveLINMessage);
不使用 LIN 回调的时候:
retValue = tslin_unregister_event_lin(ADeviceHandle, ReceiveLINMessage);
//ADeviceHandle 是设备句柄
FlexRay 回调:
void ReceiveFlexRayMessage(const TLibFlexRay* AData)
{
    //处理收到的报文数据 AData
}
retValue = tsflexray_register_event_flexray(ADeviceHandle, ReceiveFlexRayMessage);
不使用 FlexRay 回调的时候:
retValue = tsflexray_unregister_event_flexray(ADeviceHandle, ReceiveFlexRayMessage);
//ADeviceHandle 是设备句柄
```

注：可以看到上述各总线回调的注册不同地方在 **ONXXEvent** 不同，使用注册函数不同

2. 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。参考章节 10 中 `tsfifo` 相关函数

CAN 报文获取

获取 Rx 报文：

```
TCAN canBuffer[100] ;
```

```
S32 revCnt = 100;
```

#revCnt 的大小为 TCANBuffer 的长度，可以小，当一定不能比 TCANBuffer 大

#注：每次传入 `tsfifo_receive_can_msgs` 的 `revCnt` 需要重新赋值，因为该变量为一个输入输出量，返回读到的报文数量，如果存在没有读到的情况，该变量会变为 0，此时再往里面传入，将一直读不到数据

```
tsfifo_receive_can_msgs(  
    canBuffer,  
    &revCnt,  
    CH1,                //读取通道1的报文数据  
    1); //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
```

获取 Rx Tx 报文：

```
TCAN canBuffer[100] ;  
S32 revCnt = 100;          //revCnt 的大小为 TCANBuffer 的长度，可以小，当一定不  
能比 TCANBuffer 大
```

#注：每次传入 `tsfifo_receive_can_msgs` 的 `revCnt` 需要重新赋值，因为该变量为一个输入输出量，返回读到的报文数量，如果存在没有读到的情况，该变量会变为 0，此时再往里面传入，将一直读不到数据

```
tsfifo_receive_can_msgs(  
    canBuffer,  
    &revCnt,  
    CH1,                //读取通道1的报文数据  
    1); //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
```

获取 fifo 报文数量：

```
#读取通道 0 fifo 的报文数量  
S32 ACount = 0;  
tsfifo_read_can_buffer_frame_count(0,ACount);
```

获取 fifo Tx 报文数量：

```
#读取通道 0 fifo Tx 的报文数量  
s32 ACount = 0;  
tsfifo_read_can_tx_buffer_frame_count(0,ACount);
```

获取 fifo Rx 报文数量：

```
#读取通道 0 fifo Rx 的报文数量  
s32 ACount = 0;  
tsfifo_read_can_rx_buffer_frame_count(0,ACount);
```

清空 fifo 报文：

```
#读取通道 0 fifo Rx 的报文数量  
tsfifo_clear_can_receive_buffers(0)
```

CANFD 报文获取

注：CANFD 向下包含 CAN，因此 CANFD 报文获取，是会包含

CAN 数据

获取 Rx 报文：

```
TCANFD canfdBuffer[100];  
s32 revCnt = 0;           //buffersize 的大小为 TCANFDBuffer 的长度，可以小，当一定不  
能比 TCANBuffer 大
```

//注：每次传入 tsfifo_receive_can_msgs 的 revCnt 需要重新赋值，因为该变量为一个输入输出量，返回读到的报文数量，如果存在没有读到的情况，该变量会变为 0，此时再往里面传入，将一直读不到数据

```
tsfifo_receive_canfd_msgs(  
    canfdBuffer,  
    &revCnt,  
    CH1,           //读取通道1的报文数据  
    READ_TX_RX_DEF. ONLY_RX_MESSAGES);
```

获取 Rx Tx 报文：

```
TCANFD canfdBuffer[100];  
s32 revCnt = 0;           #revCnt 的大小为 TCANFDBuffer 的长度，可以小，当一定不  
能比 TCANBuffer 大
```

#注：每次传入 tsfifo_receive_can_msgs 的 revCnt 需要重新赋值，因为该变量为一个输入输出量，返回读到的报文数量，如果存在没有读到的情况，该变量会变为 0，此时再往里面传入，将一直读不到数据

```
tsfifo_receive_canfd_msgs(  
    canfdBuffer,  
    &revCnt,  
    CH1,           //读取通道1的报文数据  
    READ_TX_RX_DEF. ONLY_RX_MESSAGES);
```

获取 fifo 报文数量：

```
#读取通道 0 fifo 的报文数量  
s32 ACount = 0;  
tsfifo_read_canfd_buffer_frame_count(0,ACount);
```

获取 fifo Tx 报文数量：

```
#读取通道 0 fifo Tx 的报文数量  
s32 ACount = 0;  
tsfifo_read_canfd_tx_buffer_frame_count(0,ACount);
```

获取 fifo Rx 报文数量：

```
#读取通道 0 fifo Rx 的报文数量  
s32 ACount = 0;  
tsfifo_read_canfd_rx_buffer_frame_count(0,ACount);
```

清空 fifo 报文：

#读取通道 0 fifo Rx 的报文数量

```
tsfifo_clear_canfd_receive_buffers(0);
```

LIN 报文获取

获取 Rx 报文：

```
TLIN linBuffer[100];
```

```
s32 revCnt = 100;      #revCnt 的大小为 TLINBuffer 的长度，可以小，当一定不能比  
TCANBuffer 大
```

#注：每次传入 `tsfifo_receive_can_msgs` 的 `revCnt` 需要重新赋值，因为该变量为一个输入输出量，返回读到的报文数量，如果存在没有读到的情况，该变量会变为 0，此时再往里面传入，将一直读不到数据

```
tsfifo_receive_lin_msgs(  
    linBuffer,  
    &revCnt,  
    CH1,          //读取通道1的报文数据  
    READ_TX_RX_DEF.ONLY_RX_MESSAGES);
```

获取 Rx Tx 报文：

```
TLIN linBuffer[100];
```

```
s32 revCnt = 100;      #revCnt 的大小为 TLINBuffer 的长度，可以小，当一定不能比  
TCANBuffer 大
```

#注：每次传入 `tsfifo_receive_can_msgs` 的 `revCnt` 需要重新赋值，因为该变量为一个输入输出量，返回读到的报文数量，如果存在没有读到的情况，该变量会变为 0，此时再往里面传入，将一直读不到数据

```
tsfifo_receive_lin_msgs(  
    linBuffer,  
    &revCnt,  
    CH1,          //读取通道1的报文数据  
    READ_TX_RX_DEF.ONLY_RX_MESSAGES);
```

获取 fifo 报文数量：

#读取通道 0 fifo 的报文数量

```
s32 ACount = 0;
```

```
tsfifo_read_lin_buffer_frame_count(0,ACount);
```

获取 fifo Tx 报文数量：

#读取通道 0 fifo Tx 的报文数量

```
s32c ACount = 0;
```

```
tsfifo_read_lin_tx_buffer_frame_count(0,ACount);
```

获取 fifo Rx 报文数量：

#读取通道 0 fifo Rx 的报文数量

```
s32 ACount = 0;
tsfifo_read_lin_rx_buffer_frame_count(0,ACount);
```

清空 fifo 报文:

```
#读取通道 0 fifo Rx 的报文数量
tsfifo_clear_lin_receive_buffers(0);
```

Flexray 报文获取

获取 Rx 报文:

```
TFlexray FlexrayBuffer[100];
int revCnt = sizeof(FlexrayBuffer)/sizeof(FlexrayBuffer[0]); //revCnt的大小为
FlexrayBuffer的长度, 可以小, 当一定不能比FlexrayBuffer大
//注: 每次传入 tsfifo_receive_can_msgs 的 revCnt 需要重新赋值, 因为该变量为一个输入输出量, 返回读到的报文数量, 如果存在没有读到的情况, 该变量会变为 0, 此时再往里面传入, 将一直读不到数据
tsfifo_receive_flexray_msgs(
    FlexrayBuffer,
    &revCnt,
    CH1, //读取通道1的报文数据
    READ_TX_RX_DEF. ONLY_RX_MESSAGES);
```

获取 Rx Tx 报文:

```
TFlexray FlexrayBuffer[100];
int revCnt = sizeof(FlexrayBuffer)/sizeof(FlexrayBuffer[0]); //revCnt 的大小为
FlexrayBuffer 的长度, 可以小, 当一定不能比 FlexrayBuffer 大
#注: 每次传入 tsfifo_receive_can_msgs 的 revCnt 需要重新赋值, 因为该变量为一个输入输出量, 返回读到的报文数量, 如果存在没有读到的情况, 该变量会变为 0, 此时再往里面传入, 将一直读不到数据
tsfifo_receive_flexray_msgs(
    FlexrayBuffer,
    &revCnt,
    CH1, //读取通道1的报文数据
    READ_TX_RX_DEF. ONLY_RX_MESSAGES);
```

获取 fifo 报文数量:

```
#读取通道 0 fifo 的报文数量
s32 ACount=0;
tsfifo_read_flexray_buffer_frame_count(0,ACount);
```

获取 fifo Tx 报文数量:

```
#读取通道 0 fifo Tx 的报文数量
s32 ACount = 0;
```

```
tsfifo_read_flexray_tx_buffer_frame_count(0, ACount);
```

获取 fifo Rx 报文数量：

#读取通道 0 fifo Rx 的报文数量

```
s32 ACount = 0;
```

```
tsfifo_read_flexray_rx_buffer_frame_count(0, ACount);
```

清空 fifo 报文：

#读取通道 0 fifo Rx 的报文数量

```
tsfifo_clear_flexray_receive_buffers(0);
```

7. 接口函数介绍

1. initialize_lib_tscan

函数名称	void initialize_lib_tscan(bool AEnableFIFO, bool AEnableErrorFrame, bool AUseHWTTime)
功能介绍	初始化 libTSCANOnLinux 模块。
调用位置	初始化此模块过后，其他 API 函数才能被调用。
输入参数	AEnableFIFO：是否开启 FIFO 机制，建议设置为 True，否则用户无法通过 tsfifo_receive_xx 函数读取报文。 AEnableErrorFrame：是否接收错误帧。如果设置为 False，则驱动直接把错误帧抛弃掉。 AUseHWTTime：直接设置为 False 即可。
返回值	无
示例	initialize_lib_tscan(true, false, false);

2. finalize_lib_tscan

函数名称	void finalize_lib_tscan (void)
功能介绍	释放 libTSCANOnLinux 模块。
调用位置	此函数跟 initialize_lib_tscan 函数是对应的。在使用完 API 模块过后，退出程序之前，一定要调用此函数释放掉驱动模块。
输入参数	无
返回值	无
示例	finalize_lib_tscan();

3. tscan_scan_devices

函数名称	u32 tscan_scan_devices(uint32_t* ADeviceCount)
功能介绍	扫描当前电脑上存在的 TSCAN 设备数目
调用位置	当用户想知道当前 PC 上 TSCAN 设备数的场合

输入参数	ADeviceCount: 指针参数, 指向设备数量
返回值	==0: 获取成功 其他值: 获取失败
示例	<pre>uint32_t ADeviceCount; tscan_scan_devices(&ADeviceCount); //扫描设备数量, 并存储在变//量 ADeviceCount 中</pre>

4. tscan_connect

函数名称	u32 tscan_connect(const char* ADeviceSerial, size_t* AHandle)
功能介绍	连接 TSCAN 工具, 并获取该工具的唯一句柄
调用位置	使用 TSCAN 工具之前, 先调用此函数连接设备
输入参数	ADeviceSerial: != NULL, 获取指定序列号的设备 ==NULL, 获取任意处于连接状态的设备 AHandle: 设备句柄
返回值	==0: 连接成功 ==5: 设备已经连接
示例	<pre>u32 retValue = tscan_connect("", &ADeviceHandle); //连接字符串为空, 则设备连接默认设备 u32 retValue = tscan_connect("512356EF32CD", &ADeviceHandle); //连 接字符串不为空, 则连接指定串行号设备</pre>

5. tscan_disconnect_by_handle

函数名称	u32 tscan_disconnect_by_handle(const size_t ADeviceHandle)
功能介绍	根据设备句柄, 断开该 TSCAN 设备
调用位置	不需要使用设备, 调用此函数断开设备连接
输入参数	ADeviceHandle: 设备句柄
返回值	==0: 断开设备成功 其他值: 断开设备失败
示例	<pre>tscan_disconnect_by_handle(ADeviceHandle);</pre>

6. tscan_config_can_by_baudrate

函数名称	u32 tscan_config_can_by_baudrate(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const double ARateKbps, const u32 A1200hmConnected)
功能介绍	配置 CAN 总线波特率
调用位置	在调用 CAN 报文收发 API 之前, 调用此 API 初始化总线波特率等。
输入参数	ADeviceHandle: 设备句柄 AChnIdx: 通道参数 ARateKbps: 波特率参数, 比如 500 代表 500kbps A1200hmConnected: 是否使能 120Ω 终端电阻
返回值	==0: 函数执行成功 其他值: 函数执行失败

示例	<code>tscan_config_can_by_baudrate (ADeviceHandle, CHN1, 500, 1);</code> //配置通道 1 的波特率参数为 500kbps，并且使能终端电阻
----	--

7. tscan_register_event_can

函数名称	<code>u32 tscan_register_event_can(const size_t ADeviceHandle, const TCANQueueEvent_Win32_t ACallback)</code>
功能介绍	注册 can 数据包接收回调函数
调用位置	在 CAN 工具连接成功后，调用此函数注册接收数据的函数
输入参数	ADeviceHandle[IN]：设备句柄； ACallback[IN]：接收数据的处理函数
返回值	==0：注册成功 其他值：注册失败
示例	<pre>retValue = tscan_register_event_can(ADeviceHandle, ReceiveCANMessage); ReceiveCANMessage 是报文处理函数，其定义如下： void ReceiveCANMessage(const TLibCAN* AData) { if(AData->FProperties.bits.istx){ qDebug()<<"tx frame with id 0x"<<QString::number(AData->FIdentifier, 16);}else{ qDebug()<<"rx frame with id 0x"<<QString::number(AData->FIdentifier, 16);} }</pre>

8. tscan_unregister_event_can

函数名称	<code>u32 tscan_unregister_event_can(const size_t ADeviceHandle, const TCANQueueEvent_Win32_t ACallback)</code>
功能介绍	反注册 CAN 数据接收函数
调用位置	在不需要通过回调函数的形式接收 CAN 报文的时候，调用此函数
输入参数	ADeviceHandle：设备句柄； ACallback：接收数据处理函数委托
返回值	==0：反注册成功 其他值：反注册失败
示例	<pre>retValue = tscan_unregister_event_can(ADeviceHandle, ReceiveCANMessage);</pre>

9. tsfifo_add_can_canfd_pass_filter

函数名称	<code>u32 tsfifo_add_can_canfd_pass_filter(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const s32 AIdentifier, const bool AIsStd)</code>
功能介绍	添加 can/canfd 过滤报文
调用位置	用户如果只想接收特定 ID 报文的时候，需要调用此函数

输入参数	ADeviceHandle: 设备句柄; AChnIdx: 通道索引 AIdentifier: 报文标识符 AIsStd: 是否标准帧
返回值	==0: 执行成功 其他值: 执行失败
示例	tsfifo_add_can_canfd_pass_filter(ADeviceHandle, CHN1, 0x123, true); //把 0x123 报文添加到过滤器中

10. tsfifo_receive_can_msgs

函数名称	u32 tsfifo_receive_can_msgs(const size_t ADeviceHandle, TLibCAN* ACANBuffers, s32* ACANBufferSize, u8 AChn, u8 ARXTX)
功能介绍	从 fifo 中读取收到的 CAN 报文
调用位置	用户读取收到(包含发送出去的和接收到的)的 CAN 报文。如果用读取 fifo 的方式读取 CAN 报文, 需要在 initialize_lib_tscan 函数中第一个参数 EnableFiFo 设置为 true, 否则无法从 fifo 中读取数据。
输入参数	ADeviceHandle[IN]: 设备句柄。 ACANBuffers[OUT]: 报文数组首地址, 该首地址表示用于存储读取的报文的首地址。 ACANBufferSize[IN,OUT]:该参数是一个 IN, OUT 参数。 IN:表示传入的报文数组的尺寸,驱动内部才知道一次性最多读取多少个数据, 否则造成内存越界。 OUT: 表示实际读取的报文数量。比如*ACANBufferSize 在传进去的时候等于 20, 函数执行过后变成了 10, 表示实际读取了 10 个报文。 AChn[IN]: 需要读取的报文通道 ARxTx[IN]: =0: 只读取从其他节点接收到的报文; 1: 把自己发出去的和从其他节点收到的报文都读取出来。
返回值	==0: 读取数据成功 其他值: 读取数据失败
示例	<pre>TLibCAN readDataBuffer[20]; //首先创建一个 20 个元素的报文数组, 用于存//储从 fifo 读取的报文 int realDataSize = 20; //报文大小是 IN, OUT 参数, 所以要先设置 //初始值 //Reveive data from FIFO of Driver if(tsfifo_receive_can_msgs(ADeviceHandle, readDataBuffer, &realDataSize, CHN1, 1) == 0x00) { for(int i = 0; i< realDataSize; i++) { qDebug()<<"read frame from fifo with id 0x"<<QString::number(readDataBuffer[i].FIdentifier, 16); } } }</pre>

11. tsfifo_clear_can_receive_buffers

函数名称	u32 tsfifo_clear_can_receive_buffers(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx)
功能介绍	清除指定通道 FIFO 里面的 CAN 报文
调用位置	需要清除指定通道 FIFO 内部的报文，仿真内部缓存的报文太多影响了最新报文的接收。
输入参数	ADeviceHandle[IN]：设备句柄。 AChnIdx[IN]：CAN 报文通道，清除指定通道 FIFO 内部的报文
返回值	==0：函数执行成功 其他值：函数执行失败
示例	tsfifo_clear_can_receive_buffers(ADeviceHandle, CHN1); //清除指定通道的 FIFO

12. tscan_transmit_can_async

函数名称	u32 tscan_transmit_can_async(const size_t ADeviceHandle, const TLibCAN* ACAN)
功能介绍	异步方式发送 CAN 报文
调用位置	在需要发送 CAN 报文的场合
输入参数	ADeviceHandle[IN]：设备句柄。 ACAN[IN]：CAN 报文
返回值	==0：发送报文成功 其他值：函数执行失败
示例	<pre>TLibCAN msg; //首先定义 CAN 报文 msg.FIdentifier = 0x03; //设置报文 ID msg.FProperties.bits.remoteframe = 0x00; //not remote frame, standard frame msg.FProperties.bits.extframe = 0; //设置是否数据帧，远程帧等属性 msg.FDLC = 3; //设置要发送的报文长度 msg.FIdxChn = CHN1; //设置报文发送的通道 tscan_transmit_can_async (ADeviceHandle, &msg);</pre>

13. tscan_config_canfd_by_baudrate

函数名称	u32 tscan_config_canfd_by_baudrate(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const double AArbRateKbps, const double ADataRateKbps, const TLIBCANFDControllerType AControllerType, const TLIBCANFDControllerMode AControllerMode, const u32 A1200hmConnected)
功能介绍	配置 CANFD 总线波特率
调用位置	在调用 CANFD 报文收发 API 之前，调用此 API 初始化总线波特率等。

输入参数	<p>ADeviceHandle[IN]:设备句柄</p> <p>AChnIdx[IN]:通道参数</p> <p>AArbRateKbps[IN]:仲裁场波特率参数, 比如 500 代表 500kbps</p> <p>ADataRateKbps[IN]:数据场波特率参数, 比如 2000 代表 2000kbps</p> <p>AControllerType[IN]:控制器类型, 主要包含:</p> <ul style="list-style-type: none">lfdtCAN: 普通 CAN 模式lfdtISOCAN: ISO-CANFD 模式lfdtNonISOCAN: NoISO-CANFD 模式 <p>AControllerMode[IN]:控制器模式, 主要包含:</p> <ul style="list-style-type: none">lfdmNormal: 正常工作模式lfdmACKOff: 关闭 ACK 应答模式lfdmRestricted: 受限模式lfdmInternalLoopback: 设备内循环模式lfdmExternalLoopback: 设备外循环模式 <p>A1200hmConnected[IN]:是否使能 120 Ω 终端电阻</p>
返回值	<p>==0:函数执行成功</p> <p>其他值:函数执行失败</p>
示例	<pre>tscan_config_canfd_by_baudrate(ADeviceHandle, CHN1, 500, 2000, lfdtISOCAN, lfdmNormal, 1); //配置通道 1 的波特率参数为仲裁场 500kbps, 数据场 2000kbps, ISO-CANFD 模式, 正常工作模式, 并且使能终端电阻。</pre>

14. tscan_register_event_canfd

函数名称	<code>u32 tscan_register_event_canfd(const size_t ADeviceHandle, const TCANFDQueueEvent_Win32_t ACallback)</code>
功能介绍	注册 can 数据包接收回调函数
调用位置	在 CAN 工具连接成功后, 调用此函数注册接收数据的函数
输入参数	<p>ADeviceHandle[IN]: 设备句柄;</p> <p>ACallback[IN]: 接收数据的处理函数</p>
返回值	<p>==0: 注册成功</p> <p>其他值: 注册失败</p>
示例	<pre>retValue = tscan_register_event_canfd(ADeviceHandle, ReceiveCANMessage); ReceiveCANFDMessage 是报文处理函数, 其定义如下: void ReceiveCANFDMessage(const TLibCANFD* AData) { if(AData->FProperties.bits.istx){ qDebug()<<"tx frame with id 0x"<<QString::number(AData->FIdentifier, 16);}else{ qDebug()<<"rx frame with id 0x"<<QString::number(AData->FIdentifier, 16);} }</pre>

15. tscan_unregister_event_canfd

函数名称	u32 tscan_unregister_event_canfd(const size_t ADeviceHandle, const TCANFDQueueEvent_Win32_t ACallback)
功能介绍	反注册 CAN 数据接收函数
调用位置	在不需要通过回调函数的形式接收 CAN 报文的时候，调用此函数
输入参数	ADeviceHandle：设备句柄； ACallback：接收数据处理函数委托
返回值	==0：反注册成功 其他值：反注册失败
示例	retValue = tscan_unregister_event_canfd(ADeviceHandle, ReceiveCANFDMessage);

16. tsfifo_receive_canfd_msgs

函数名称	u32 tsfifo_receive_canfd_msgs(const size_t ADeviceHandle, const TLibCANFD* ACANFDBuffers, s32* ACANFDBufferSize, u8 AChn, u8 ARXTX)
功能介绍	从 fifo 中读取收到的 CANFD 报文
调用位置	用户读取收到(包含发送出去的和接收到的)的 CANFD 报文。如果用读取 fifo 的方式读取 CAN 报文，需要在 initalize_lib_tscan 函数中第一个参数 EnableFiFo 设置为 true，否则无法从 fifo 中读取数据。
输入参数	ADeviceHandle[IN]：设备句柄。 ACANFDBuffers[OUT]：报文数组首地址，该首地址表示用于存储读取的报文的首地址。 ACANFDBufferSize[IN,OUT]:该参数是一个 IN, OUT 参数。 IN:表示传入的报文数组的尺寸，驱动内部才知道一次性最多读取多少个数据，否则造成内存越界。 OUT: 表示实际读取的报文数量。比如*ACANBufferSize 在传进去的时候等于 20，函数执行过后变成了 10，表示实际读取了 10 个报文。 AChn[IN]：需要读取的报文通道 ARxTx[IN]：=0：只读取从其他节点接收到的报文；1：把自己发出去的和从其他节点收到的报文都读取出来。
返回值	==0：读取数据成功 其他值：读取数据失败
示例	<pre>TLibCANFD readDataBuffer[20]; //首先创建一个 20 个元素的报文数组，用于存储从 fifo 读取的报文 int realDataSize = 20; //报文大小是 IN, OUT 参数，所以要先设置初始值 //Reveive data from FIFO of Driver if(tsfifo_receive_canfd_msgs(ADeviceHandle, readDataBuffer, &realDataSize, CHN1, 1) == 0x00) { for(int i = 0; i< realDataSize; i++)</pre>

```
{
    qDebug()<<"read frame from fifo with id
0x"<<QString::number(readDataBuffer[i].FIdentifier, 16);
}
}
```

17. tsfifo_clear_can_receive_buffers

函数名称	u32 tsfifo_clear_can_receive_buffers(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx)
功能介绍	清除指定通道 FIFO 里面的 CAN 报文
调用位置	需要清除指定通道 FIFO 内部的报文，仿真内部缓存的报文太多影响了最新报文的接收。
输入参数	ADeviceHandle[IN]：设备句柄。 AChnIdx[IN]：CAN 报文通道，清除指定通道 FIFO 内部的报文
返回值	==0：函数执行成功 其他值：函数执行失败
示例	tsfifo_clear_can_receive_buffers(ADeviceHandle, CHN1); //清除指定通道的 FIFO

18. tscan_transmit_canfd_async

函数名称	u32 tscan_transmit_canfd_async(const size_t ADeviceHandle, const TLIBCANFD* ACANFD)
功能介绍	异步方式发送 CANFD 报文
调用位置	在需要发送 CANFD 报文的场合
输入参数	ADeviceHandle[IN]：设备句柄。 ACANFD[IN]：CANFD 报文
返回值	==0：发送报文成功 其他值：函数执行失败
示例	<pre>TLibCANFD fdmsg; //首先定义 CANFD 报文 fdmsg.FIdentifier = 0x03; //然后设置 CAN 报文 ID fdmsg.FProperties.bits.remoteframe = 0x00; //not remote frame, standard frame fdmsg.FProperties.bits.extframe = 0; //是否扩展帧，远程帧等属性 fdmsg.FDLC = 3; //报文数据长度 fdmsg.FIdxChn = CHN1; //报文发送到哪一个通道中 fdmsg.FFDProperties.bits.EDL = 1; //FDMode:是否 FD 模式报文 fdmsg.FFDProperties.bits.BRS = 1; //Open baudrate speed: 是否波特率可变 tscan_transmit_canfd_async (ADeviceHandle, &canmsg);</pre>

19. tslin_set_node_funtiontype

函数名称	u32 tslin_set_node_funtiontype(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const u8 AFunctionType)
功能介绍	设置设备的功能类型：0：主节点（Master Node）1：从节点（Slave Node）2：监听节点（Monitor Node）
调用位置	LIN 总线工作之前，需要设置该节点的工作模式
输入参数	ADeviceHandle：设备句柄； AChnIdx:通道编号 AFunctionType：功能类型
返回值	==0：函数执行成功 其他值：执行失败
示例	tslin_set_node_funtiontype(ADeviceHandle, CHN1, MasterNode) //设置设备的 LIN 通道 1 工作在主节点模式下

20. tslin_config_baudrate

函数名称	u32 tslin_config_baudrate(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const double ARateKbps, TLINProtocol AProtocol)
功能介绍	配置 LIN 总线波特率
调用位置	在调用 LIN 报文收发 API 之前，调用此 API 初始化总线波特率等。
输入参数	ADeviceHandle[IN]:设备句柄 AChnIdx[IN]:通道参数 ARateKbps[IN]:波特率参数，比如 20 代表 20kbps AProtocol [IN]:LIN 总线协议版本。包括如下定义： LIN_Protocol_13: LIN1.3 版本 LIN_Protocol_20: LIN 协议 2.0 版本 LIN_Protocol_21: LIN 协议 2.1 版本 LIN_Protocol_J2602: LIN 协议 J2602 版本
返回值	==0:函数执行成功 其他值：函数执行失败
示例	tscan_config_lin_by_baudrate (ADeviceHandle, CHN1, 20, LIN_Protocol_21); //配置通道 1 的波特率参数为 20kbps，协议版本为 2.1 版本

21. tslin_register_event_lin

函数名称	u32 tslin_register_event_lin(const size_t ADeviceHandle, const TLINQueueEvent_Win32_t ACallback)
功能介绍	注册 LIN 数据包接收回调函数
调用位置	在 LIN 工具连接成功后，调用此函数注册接收数据的函数
输入参数	ADeviceHandle[IN]：设备句柄；

	ACallback[IN]: 接收数据的处理函数
返回值	==0: 注册成功 其他值: 注册失败
示例	<pre>retValue = tslin_register_event_lin(ADeviceHandle, ReceiveLINMessage); //ReceiveLINMessage 是报文处理函数, 其定义如下: void ReceiveLINMessage(const TLibLIN* AData) { if(AData->FProperties.bits.istx){ qDebug()<<"tx frame with id 0x"<<QString::number(AData->FIdentifier, 16);}else{ qDebug()<<"rx frame with id 0x"<<QString::number(AData->FIdentifier, 16);}}</pre>

22. tslin_unregister_event_lin

函数名称	u32 tslin_unregister_event_lin(const size_t ADeviceHandle, const TLINQueueEvent_Win32_t ACallback)
功能介绍	反注册 LIN 数据接收函数
调用位置	在不需要通过回调函数的形式接收 LIN 报文的时候, 调用此函数
输入参数	ADeviceHandle: 设备句柄; ACallback: 接收数据处理函数委托
返回值	==0: 反注册成功 其他值: 反注册失败
示例	<pre>retValue = tslin_unregister_event_lin(ADeviceHandle, ReceiveLINMessage);</pre>

23. tsfifo_add_lin_pass_filter

函数名称	u32 tsfifo_add_lin_pass_filter(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const s32 AIdentifier)
功能介绍	添加 can/canfd 过滤报文
调用位置	用户如果只想接收特定 ID 报文的时候, 需要调用此函数
输入参数	ADeviceHandle: 设备句柄; AChnIdx: 通道索引 AIdentifier: 报文 ID
返回值	==0: 执行成功 其他值: 执行失败
示例	<pre>tsfifo_add_lin_pass_filter(ADeviceHandle, CHN1, 0x12); //把 0x12 报文添加到过滤器中</pre>

24. tsfifo_receive_lin_msgs

函数名称	u32 tsfifo_receive_lin_msgs(const size_t ADeviceHandle, const TLibLIN* ALINBuffers, s32* ALINBufferSize, u8 AChn, u8 ARXTX)
功能介绍	从 fifo 中读取收到的 lin 报文
调用位置	用户读取收到(包含发送出去的和接收到的)的 lin 报文。如果用读取 fifo 的方式读取 lin 报文，需要在 initalize_lib_tscan 函数中第一个参数 EnableFiFo 设置为 true，否则无法从 fifo 中读取数据。
输入参数	<p>ADeviceHandle[IN]：设备句柄。</p> <p>ALINBuffers[OUT]：报文数组首地址，该首地址表示用于存储读取的报文的首地址。</p> <p>ALINBufferSize[IN,OUT]:该参数是一个 IN, OUT 参数。</p> <p>IN:表示传入的报文数组的尺寸，驱动内部才知道一次性最多读取多少个数据，否则造成内存越界。</p> <p>OUT: 表示实际读取的报文数量。比如*ALINBufferSize 在传进去的时候等于 20，函数执行过后变成了 10，表示实际读取了 10 个报文。</p> <p>AChn[IN]：需要读取的报文通道</p> <p>ARxTx[IN]：=0：只读取从其他节点接收到的报文；1：把自己发出去的和从其他节点收到的报文都读取出来。</p>
返回值	<p>==0：读取数据成功</p> <p>其他值：读取数据失败</p>
示例	<pre>TLibLIN readDataBuffer[20]; //首先创建一个 20 个元素的报文数组，用于存// 储从 fifo 读取的报文 int realDataSize = 20; //报文大小是 IN, OUT 参数，所以要先设置 //初始值 //Reveive data from FIFO of Driver if(tsfifo_receive_lin_msgs(ADeviceHandle, readDataBuffer, &realDataSize, CH N1, 1) == 0x00) { for(int i = 0; i< realDataSize; i++) { qDebug()<<"read frame from fifo with id 0x"<<QString::number(readDataBuffer[i].FIdentifier, 16); } }</pre>

25. tsfifo_clear_lin_receive_buffers

函数名称	u32 tsfifo_clear_lin_receive_buffers(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx)
功能介绍	清除指定通道 FIFO 里面的 LIN 报文
调用位置	需要清除指定通道 FIFO 内部的报文，仿真内部缓存的报文太多影响了最新报文的接收。
输入参数	ADeviceHandle[IN]：设备句柄。

	AChnIdx[IN]: LIN 报文通道, 清除指定通道 FIFO 内部的报文
返回值	==0: 函数执行成功 其他值: 函数执行失败
示例	<code>tsfifo_clear_lin_receive_buffers(ADeviceHandle, CHN1);</code> //清除指定通道的 FIFO

26. tslin_transmit_lin_async

函数名称	<code>u32 tslin_transmit_lin_async(const size_t ADeviceHandle, const TLibLIN* ALIN)</code>
功能介绍	异步方式发送 LIN 报文
调用位置	在需要发送 LIN 报文的场合
输入参数	ADeviceHandle[IN]: 设备句柄。 ALIN[IN]: LIN 报文
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	<pre>TLibLIN msg; //首先定义 LIN 报文 msg.FIdentifier = 0x03; //设置报文 ID msg.FDLC = 3; //设置要发送的报文长度 msg.FIdxChn = CHN1; //设置报文发送的通道 tslin_transmit_lin_async (ADeviceHandle, &msg);</pre>

27. tscan_register_event_canfd_whandle

函数名称	<code>u32 tscan_register_event_canfd_whandle(const size_t ADeviceHandle, const TCANFDQueueEvent_whandle ACallback)</code>
功能介绍	注册接收不同硬件 canfd 数据的回调函数
调用位置	在 canfd 工具连接成功后, 调用此函数注册接收数据的函数
输入参数	ADeviceHandle[IN]: 设备句柄。 ACallback[IN]: 接收数据处理函数委托
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	<pre>void ReceiveCANFDMessage(const TLIBCANFD * AData) { //处理报文数据 } tscan_register_event_canfd_whandle(ADeviceHandle, ReceiveCANFDMessage);</pre>

28. tscan_unregister_event_canfd_whandle

函数名称	<code>u32 tscan_unregister_event_canfd_whandle(const size_t ADeviceHandle, const TCANFDQueueEvent_whandle ACallback)</code>
功能介绍	注销接收不同硬件 canfd 数据的回调函数

调用位置	在 canfd 工具连接成功后，调用此函数注销接收数据的函数
输入参数	ADeviceHandle[IN]: 设备句柄。 ACallback[IN]: 接收数据处理函数委托
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	<code>tscan_unregister_event_canfd_whatle(ADeviceHandle, ReceiveCANFDMessag);</code>

29. tslin_register_event_lin_whatle

函数名称	<code>u32 tslin_register_event_lin_whatle(const size_t ADeviceHandle, const TLINQueueEvent_whatle ACallback);</code>
功能介绍	注册接收不同硬件 lin 数据的回调函数
调用位置	在 lin 工具连接成功后，调用此函数注册接收数据的函数
输入参数	ADeviceHandle[IN]: 设备句柄。 ACallback[IN]: 接收数据处理函数委托
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	<pre>void ReceiveLINMessage(const TLIBLIN* AData) { //处理接收报文信息 } tslin_register_event_lin_whatle(ADeviceHandle, ReceiveLINMessage);</pre>

30. tslin_unregister_event_lin_whatle

函数名称	<code>u32 tslin_unregister_event_lin_whatle(const size_t ADeviceHandle, const TLINQueueEvent_whatle ACallback);</code>
功能介绍	注销接收不同硬件 lin 数据的回调函数
调用位置	在 lin 工具连接成功后，调用此函数注销接收数据的函数
输入参数	ADeviceHandle[IN]: 设备句柄。 ACallback[IN]: 接收数据处理函数委托
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	<code>tslin_unregister_event_lin_whatle(ADeviceHandle, ReceiveLINMessage);</code>

31. tsflexray_unregister_event_flexray_whatle

函数名称	<code>u32 tslin_unregister_event_flexray_whatle(const size_t ADeviceHandle, const TLINQueueEvent_whatle ACallback);</code>
功能介绍	注销接收不同硬件 flexray 数据的回调函数

调用位置	在 flexray 工具连接成功后，调用此函数注销接收数据的函数
输入参数	ADeviceHandle[IN]: 设备句柄。 ACallback[IN]: 接收数据处理函数委托
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	<pre>void ReceiveFlexRayMessage(const TLIBFlexRay* AData) { //处理接收到的报文数据 } tslin_unregister_event_lin_whatle(ADeviceHandle, ReceiveFlexRayMessage);</pre>

32. tsflexray_register_event_flexray_whatle

函数名称	u32 tsflexray_register_event_flexray_whatle(const size_t ADeviceHandle, const TFlexRayQueueEvent_whatle ACallback);
功能介绍	注册接收不同硬件 flexray 数据的回调函数
调用位置	在 flexray 工具连接成功后，调用此函数注销接收数据的函数
输入参数	ADeviceHandle[IN]: 设备句柄。 ACallback[IN]: 接收数据处理函数委托
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	<pre>tsflexray_register_event_flexray_whatle(ADeviceHandle, ReceiveFlexRayMessage);</pre>

33. tscan_register_event_fastlin

函数名称	u32 tscan_register_event_fastlin(const size_t ADeviceHandle, const TLINQueueEvent_Win32_t ACallback);
功能介绍	注册接收 fastlin 数据的回调函数
调用位置	在 lin 工具连接成功后，调用此函数注册接收数据的函数
输入参数	ADeviceHandle[IN]: 设备句柄。 ACallback[IN]: 接收数据处理函数委托
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	<pre>void ReceiveLINMessage(const TLIBLIN* AData) { //处理接收到的报文数据 } tscan_register_event_fastlin(ADeviceHandle, ReceiveLINMessage);</pre>

34. tscan_unregister_event_fastlin

函数名称	u32 tscan_unregister_event_fastlin(const size_t ADeviceHandle, const TLINQueueEvent_Win32_t ACallback);
------	---

功能介绍	注销接收 fastlin 数据的回调函数
调用位置	在 lin 工具连接成功后，调用此函数注销接收数据的函数
输入参数	ADeviceHandle[IN]：设备句柄。 ACallback[IN]:接收数据处理函数委托
返回值	==0：发送报文成功 其他值：函数执行失败
示例	tscan_unregister_event_fastlin(ADeviceHandle, ReceiveLINMessage);

35. tscan_get_device_info

函数名称	u32 tscan_get_device_info(const s32 ADeviceIndex, char** AFManufacturer, char** AFProduct, char** AFSerial);
功能介绍	获取设备信息
调用位置	在 lin 工具连接成功后，调用此函数注销接收数据的函数
输入参数	ADeviceHandle[IN]：设备句柄。 AFManufacturer[OUT]:生产商名称 AFProduct[OUT]：设备名称 AFSerial[OUT]：设备序列号
返回值	==0：发送报文成功 其他值：函数执行失败
示例	char *AFManufacturer = “ ” ; char *AFProduct= “ ” ; char *AFSerial= “ ” ; tscan_get_device_info(ADeviceIndex, &AFManufacturer,&AFProduct, &AFSerial);

36. tscan_get_can_channel_count

函数名称	u32 tscan_get_can_channel_count(const size_t AHandle, s32* ACount);
功能介绍	获取 can 通道数量
调用位置	在 can 工具连接成功后
输入参数	AHandle：句柄 ACount:返回的数量
返回值	==0：发送报文成功 其他值：函数执行失败
示例	s32 ACount = 0; tscan_get_can_channel_count(Handle, &ACount);

37. tscan_get_lin_channel_count

函数名称	u32 tscan_get_lin_channel_count(const size_t AHandle, s32* ACount);
功能介绍	获取 lin 通道数量
调用位置	在 lin 工具连接成功后
输入参数	AHandle：句柄 ACount:返回的数量

返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	s32 ACount = 0; tscan_get_lin_channel_count(Handle, &ACount);

38. tscan_get_flexray_channel_count

函数名称	u32 tscan_get_flexray_channel_count(const size_t AHandle, s32* ACount);
功能介绍	获取 flexray 通道数量
调用位置	在 flexray 工具连接成功后
输入参数	AHandle: 句柄 ACount: 返回的数量
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	s32 ACount = 0; tscan_get_flexray_channel_count(Handle, &ACount);

39. tscan_disconnect_by_handle

函数名称	u32 tscan_disconnect_by_handle(const size_t ADeviceHandle);
功能介绍	通过设备句柄断开设备连接
调用位置	
输入参数	ADeviceHandle: 设备句柄
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	tscan_disconnect_by_handle(ADeviceHandle);

40. tscan_disconnect_all_devices

函数名称	u32 tscan_disconnect_all_devices(void);
功能介绍	断开所有设备连接
调用位置	
输入参数	无
返回值	==0: 发送报文成功 其他值: 函数执行失败
示例	tscan_disconnect_all_devices();

41. initialize_lib_tscan

函数名称	u32 initialize_lib_tscan(bool AEnableFIFO, bool AEnableErrorFrame, bool AUseHWTTime);
功能介绍	初始化 can 模块
调用位置	程序开始时
输入参数	AEnableFIFO: 开启接收 FIFO, 推荐设置为 true;

	AEnableErrorFrame:开启接受错误帧; AUseHwTime: 是否开启性能模式, 推荐设置为 false;
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>initialize_lib_tscan(true, true, false);</code>

42. finalize_lib_tscan

函数名称	<code>u32 finalize_lib_tscan(void);</code>
功能介绍	释放 can 模块
调用位置	程序结束时, 和 <code>initialize_lib_tscan</code> 配套使用
输入参数	AEnableFIFO:开启接收 FIFO, 推荐设置为 true; AEnableErrorFrame:开启接受错误帧; AUseHwTime: 是否开启性能模式, 推荐设置为 false;
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>initialize_lib_tscan(true, true, false);</code>

43. tscan_transmit_can_sync

函数名称	<code>u32 tscan_transmit_can_sync(const size_t ADeviceHandle, const TLIBCAN* ACAN, const u32 ATimeoutMS);</code>
功能介绍	同步发送 can 报文
调用位置	连接 can 设备后
输入参数	ADeviceHandle:设备句柄 ACAN:can 报文结构体指针 ATimeoutMS: 超时时间
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCAN ACAN; <code>tscan_transmit_can_sync(ADeviceHandle, &ACAN, 100);</code>

44. tscan_transmit_can_sequence

函数名称	<code>u32 tscan_transmit_can_sequence(const size_t ADeviceHandle, const TLIBCAN* ACANSeq, const s32 ASize);</code>
功能介绍	连续发送 can 报文
调用位置	连接 can 设备后
输入参数	ADeviceHandle:设备句柄 ACANSeq:can 报文结构体指针 ASize: 大小
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCAN ACAN;

```
tscan_transmit_can_sync(ADeviceHandle, &ACAN, 100);
```

45. tscan_transmit_can_async

函数名称	u32 tscan_transmit_can_async(const size_t ADeviceHandle, const TLIBCAN* ACAN);
功能介绍	异步发送 can 报文
调用位置	连接 can 设备后
输入参数	ADeviceHandle:设备句柄 ACAN:can 报文结构体指针
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCAN ACAN; tscan_transmit_can_async(ADeviceHandle, &ACAN);

46. tscan_config_can_by_baudrate

函数名称	u32 tscan_config_can_by_baudrate(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const double ARateKbps, const u32 A1200hmConnected);
功能介绍	配置设备 can 波特率
调用位置	连接 can 设备后
输入参数	ADeviceHandle:设备句柄 AChnIdx:通道索引 ARateKbps: 波特率 A1200hmConnected: 是否使能终端电阻: 1: 使能; 0: 不使能
返回值	==0: 执行成功 其他值: 函数执行失败
示例	tscan_config_can_by_baudrate(ADeviceHandle, 0, 1000, 1);

47. tscan_add_cyclic_msg_can

函数名称	u32 tscan_add_cyclic_msg_can(const size_t ADeviceHandle, const TLIBCAN* ACAN, const float APeriodMS);
功能介绍	添加周期 can 报文
调用位置	连接 can 设备后
输入参数	ADeviceHandle:设备句柄 ACAN:can 数据包 APeriodMS: 周期值
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCAN ACAN; tscan_add_cyclic_msg_can(ADeviceHandle, &ACAN, 100);

48. tscan_delete_cyclic_msg_can

函数名称	u32 tscan_delete_cyclic_msg_can(const size_t ADeviceHandle, const TLIBCAN* ACAN);
功能介绍	删除周期 can 报文
调用位置	连接 can 设备后
输入参数	ADeviceHandle:设备句柄 ACAN:can 数据包
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCAN ACAN; tscan_delete_cyclic_msg_can(ADeviceHandle, &ACAN);

49. tscan_add_cyclic_msg_canfd

函数名称	u32 tscan_add_cyclic_msg_canfd(const size_t ADeviceHandle, const TLIBCANFD* ACANFD, const float APeriodMS);
功能介绍	添加周期 canfd 报文
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 ACANFD:canfd 数据包 APeriodMS: 周期值
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCANFD ACANFD; tscan_add_cyclic_msg_canfd(ADeviceHandle, &ACANFD, 100);

50. tscan_delete_cyclic_msg_canfd

函数名称	u32 tscan_delete_cyclic_msg_canfd(const size_t ADeviceHandle, const TLIBCANFD* ACANFD);
功能介绍	删除周期 canfd 报文
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 ACANFD:canfd 数据包 APeriodMS: 周期值
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCANFD ACANFD; tscan_delete_cyclic_msg_canfd(ADeviceHandle, &ACANFD);

51. tscan_transmit_canfd_sync

函数名称	u32 tscan_transmit_canfd_sync(const size_t ADeviceHandle, const TLIBCANFD* ACAN, const u32 ATimeoutMS);
功能介绍	同步发送 canfd 报文
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 ACANFD:canfd 数据包 ATimeoutMS: 超时时间
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCANFD ACANFD; tscan_transmit_canfd_sync(ADeviceHandle, &ACANFD, 100);

52. tscan_transmit_canfd_sequence

函数名称	u32 tscan_transmit_canfd_sequence(const size_t ADeviceHandle, const TLIBCANFD* ACANFDSeq, const s32 ASize);
功能介绍	发送 canfd 报文序列
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 ACANFDSeq:canfd 数据包 ASize: 大小
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCANFD ACANFDSeq; tscan_transmit_canfd_sync(ADeviceHandle, &ACANFDSeq, 100);

53. tscan_transmit_canfd_async

函数名称	u32 tscan_transmit_canfd_sequence(const size_t ADeviceHandle, const TLIBCANFD* ACANFDSeq, const s32 ASize);
功能介绍	连续发送 canfd 报文序列
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 ACANFDSeq:canfd 数据包 ASize: 大小
返回值	==0: 执行成功 其他值: 函数执行失败
示例	TLIBCANFD ACANFDSeq; tscan_transmit_canfd_async(ADeviceHandle, &ACANFDSeq, 100);

54. tscan_config_canfd_by_baudrate

函数名称	u32 tscan_config_canfd_by_baudrate(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const double AArbRateKbps, const double ADataRateKbps, const TLIBCANFDControllerType AControllerType, const TLIBCANFDControllerMode AControllerMode, const u32 A1200hmConnected);
功能介绍	配置 canfd 波特
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 AChnIdx:通道索引 AArbRateKbps: 仲裁段波特率 ADataRateKbps: 数据段波特率 AControllerType: 控制器类型 0: can 1:isocanfd 2:non-isocanfd AControllerMode: 控制器模式 0:正常模式 1:关闭应答 2;限制模式 A1200hmConnected: 是否激活 120 Ω 终端电阻
返回值	==0: 执行成功 其他值: 函数执行失败
示例	tscan_config_canfd_by_baudrate(ADeviceHandle, 0, 500, 2000, 1, 0, true);

55. tsfifo_receive_canfd_msgs

函数名称	u32 tsfifo_receive_canfd_msgs(const size_t ADeviceHandle, TLIBCANFD* ACANBuffers, s32* ACANBufferSize, u8 AChn, u8 ARXTX);
功能介绍	接收 canfd 报文
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 ACANBuffers:canfd 缓冲区 ACANBufferSize: 缓冲区大小 AChn: 通道 ARXTX: 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
返回值	==0: 执行成功 其他值: 函数执行失败
示例	tscan_config_canfd_by_baudrate(ADeviceHandle, 0, 500, 2000, 1, 0, true);

56. tsfifo_clear_canfd_receive_buffers

函数名称	u32 tsfifo_clear_canfd_receive_buffers(const size_t ADeviceHandle, const s32 AIdxChn);
功能介绍	清除 canfd 接收 fifo 缓冲区
调用位置	连接 canfd 设备后

输入参数	ADeviceHandle:设备句柄 AIdxChn:通道索引
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>tsfifo_clear_canfd_receive_buffers(ADeviceHandle, 0);</code>

57. tsflexray_set_controller_frametrigger

函数名称	<code>u32 tsflexray_set_controller_frametrigger(const size_t ADeviceHandle, const int ANodeIndex, const PLibFlexray_controller_config AControllerConfig, const int* AFrameLengthArray, const int AFrameNum, const PLibTrigger_def AFrameTrigger, const int AFrameTriggerNum, const int ATimeoutMs);</code>
功能介绍	设置 flexray 报文触发控制器
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 ANodeIndex:通道索引 AControllerConfig: 控制器配置指针 AFrameLengthArray: 帧长度数组指针 AFrameNum: 帧数量 AFrameTrigger: 帧触发 AFrameTriggerNum: 帧触发数 ATimeoutMs: 超时时间(ms)
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>tsflexray_set_controller_frametrigger(ADeviceHandle, 0);</code>

58. tsflexray_set_controller

函数名称	<code>u32 tsflexray_set_controller(const size_t ADeviceHandle, const int ANodeIndex, const PLibFlexray_controller_config AControllerConfig, const int ATimeoutMs);</code>
功能介绍	设置 flexray 报文控制器
调用位置	连接 flexray 设备后
输入参数	ADeviceHandle:设备句柄 ANodeIndex:节点索引 AControllerConfig: 控制器配置结构体指针 ATimeoutMs: 超时时间(ms)
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>TLIBFlexray_controller_config AControllerConfig; tsflexray_set_controller(ADeviceHandle, 0, &AControllerConfig,</code>

```
100);
```

59. tsflexray_set_frametrigger

函数名称	u32 tsflexray_set_frametrigger(const size_t ADeviceHandle, const int ANodeIndex, const int* AFrameLengthArray, const int AFrameNum, const PLibTrigger_def AFrameTrigger, const int AFrameTriggerNum, const int ATimeoutMs);
功能介绍	设置 flexray 报文触发控制器
调用位置	连接 canfd 设备后
输入参数	ADeviceHandle:设备句柄 ANodeIndex:节点索引 AFrameLengthArray: 帧长度数组 AFrameNum: 帧数 AFrameTrigger: 帧触发结构体指针 AFrameTriggerNum: 帧触发数量 ATimeoutMs: 超时时间
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<pre>int AFrameLengthArray[]; TLIBTrigger_def AFrameTrigger; tsflexray_set_frametrigger(ADeviceHandle, ANodeIndex, AFrameLengthArray, AFrameNum, &AFrameTrigger, AFrameTriggerNum, 100);</pre>

60. tsflexray_transmit_sync

函数名称	u32 tsflexray_transmit_sync(const size_t ADeviceHandle, const PLibFlexRay AData, const int ATimeoutMs);
功能介绍	同步发送 flexray 报文
调用位置	
输入参数	ADeviceHandle:设备句柄 AData:flexray 结构体指针 ATimeoutMs: 超时时间
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<pre>TLIBFlexRay AData; tsflexray_transmit_sync(ADeviceHandle, &AData, 100);</pre>

61. tsflexray_transmit_async

函数名称	u32 tsflexray_transmit_async(const size_t ADeviceHandle, const PLibFlexRay AData);
功能介绍	异步发送 flexray 报文

调用位置	
输入参数	ADeviceHandle:设备句柄 AData:flexray 结构体指针
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>TLIBFlexRay AData;</code> <code>tsflexray_transmit_async(ADeviceHandle,&AData);</code>

62. tsfifo_receive_flexray_msgs

函数名称	<code>u32 tsfifo_receive_flexray_msgs(const size_t ADeviceHandle, PLibFlexRay ADataBuffers, s32* ADataBufferSize, u8 AIdxChn, u8 ARxTx);</code>
功能介绍	异步发送 flexray 报文
调用位置	
输入参数	ADeviceHandle:设备句柄 ADataBuffers:数据缓冲区 ADataBufferSize: 缓冲区大小 AIdxChn: 通道索引 ARxTx: 1:接收 TX/RX 所有报文, 如果只读取接收端的报文, 则修改为 0:READ_TX_RX_DEF.ONLY_RX_MESSAGES
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>TLIBFlexray FlexrayBuffer[100];</code> <code>int revCnt = sizeof(FlexrayBuffer)/sizeof(FlexrayBuffer[0]);</code> <code>tsfifo_receive_flexray_msgs(ADeviceHandle, FlexrayBuffer,</code> <code>&revCnt , CH1, 1);</code>

63. tsfifo_clear_flexray_receive_buffers

函数名称	<code>u32 tsfifo_clear_flexray_receive_buffers(const size_t ADeviceHandle, const s32 AIdxChn);</code>
功能介绍	清除 flexray 接收缓冲区
调用位置	
输入参数	ADeviceHandle:设备句柄 AIdxChn: 通道索引
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>tsfifo_clear_flexray_receive_buffers(ADeviceHandle, 0);</code>

64. tsflexray_start_net

函数名称	<code>u32 tsflexray_start_net(const size_t ADeviceHandle, const int</code>
------	--

	<code>AChnIdx, const int ATimeoutMs);</code>
功能介绍	Flexray 设备起始网
调用位置	
输入参数	ADeviceHandle:设备句柄 AChnIdx: 通道索引 ATimeoutMs: 超时时间
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>tsflexray_start_net(ADeviceHandle, 0, 100);</code>

65. tsflexray_stop_net

函数名称	<code>u32 tsflexray_stop_net(const size_t ADeviceHandle, const int AChnIdx, const int ATimeoutMs);</code>
功能介绍	Flexray 设备中止网
调用位置	
输入参数	ADeviceHandle:设备句柄 AChnIdx: 通道索引 ATimeoutMs: 超时时间
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>tsflexray_stop_net(ADeviceHandle, 0, 100);</code>

66. tsfifo_read_flexray_buffer_frame_count

函数名称	<code>u32 tsfifo_read_flexray_buffer_frame_count(const size_t ADeviceHandle, const int AIdxChn, int* ACount);</code>
功能介绍	读取设备 flexray 缓冲区报文数量
调用位置	
输入参数	ADeviceHandle:设备句柄 AChnIdx: 通道索引 ACount: 返回计数
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<code>int ACount = 0; tsfifo_read_flexray_buffer_frame_count(ADeviceHandle, 0, &ACount);</code>

67. tsfifo_read_flexray_tx_buffer_frame_count

函数名称	<code>u32 tsfifo_read_flexray_tx_buffer_frame_count(const size_t ADeviceHandle, const int AIdxChn, int* ACount);</code>
功能介绍	读取设备 flexray tx 缓冲区报文数量
调用位置	

输入参数	ADeviceHandle:设备句柄 AIdxChn: 通道索引 ACount: 返回计数
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<pre>int ACount = 0; tsfifo_read_flexray_tx_buffer_frame_count(ADeviceHandle, 0, &ACount);</pre>

68. tsfifo_read_flexray_rx_buffer_frame_count

函数名称	<code>u32 tsfifo_read_flexray_rx_buffer_frame_count(const size_t ADeviceHandle, const int AIdxChn, int* ACount);</code>
功能介绍	读取设备 flexray rx 缓冲区报文数量
调用位置	
输入参数	ADeviceHandle:设备句柄 AIdxChn: 通道索引 ACount: 返回计数
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<pre>int ACount = 0; tsfifo_read_flexray_rx_buffer_frame_count(ADeviceHandle, 0, &ACount);</pre>

69. tslin_set_node_funtiontype

函数名称	<code>u32 tslin_set_node_funtiontype(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const u8 AFunctionType);</code>
功能介绍	设置 lin 节点函数类型
调用位置	
输入参数	ADeviceHandle:设备句柄 AChnIdx: 通道索引 AFunctionType: 函数类型 0: T_MasterNode 1: T_SlaveNode 2: T_MonitorNode
返回值	==0: 执行成功 其他值: 函数执行失败
示例	<pre>tslin_set_node_funtiontype(ADeviceHandle, 0, T_MASTER_NODE);</pre>

70. tslin_clear_schedule_tables

函数名称	<code>u32 tslin_clear_schedule_tables(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx);</code>
功能介绍	清除 lin 进度表
调用位置	

输入参数	ADeviceHandle: 设备句柄 AChnIdx: 通道索引
返回值	==0: 成功 其他值: 失败
示例	<code>tslin_clear_schedule_tables(ADeviceHandle, 0);</code>

71. tslin_transmit_lin_sync

函数名称	<code>u32 tslin_transmit_lin_sync(const size_t ADeviceHandle, const TLIBLIN* ALIN, const u32 ATimeoutMS);</code>
功能介绍	同步发送 lin 报文
调用位置	
输入参数	ADeviceHandle: 设备句柄 ALIN: lin 数据包 ATimeoutMS: 超时时间
返回值	==0: 成功 其他值: 失败
示例	<code>TLIBLIN ALIN;</code> <code>tslin_transmit_lin_sync(ADeviceHandle, &ALIN, 100);</code>

72. tslin_transmit_lin_async

函数名称	<code>u32 tslin_transmit_lin_async(const size_t ADeviceHandle, const TLIBLIN* ALIN);</code>
功能介绍	异步发送 lin 报文
调用位置	
输入参数	ADeviceHandle: 设备句柄 ALIN: lin 数据包
返回值	==0: 成功 其他值: 失败
示例	<code>TLIBLIN ALIN;</code> <code>tslin_transmit_lin_async(ADeviceHandle, &ALIN);</code>

73. tslin_transmit_fastlin_async

函数名称	<code>u32 tslin_transmit_fastlin_async(const size_t ADeviceHandle, const TLIBLIN* ALIN);</code>
功能介绍	异步发送 fastlin 报文
调用位置	
输入参数	ADeviceHandle: 设备句柄 ALIN: lin 数据包
返回值	==0: 成功 其他值: 失败
示例	<code>TLIBLIN ALIN;</code> <code>tslin_transmit_fastlin_async(ADeviceHandle, &ALIN);</code>

74. tslin_config_baudrate

函数名称	<code>u32 tslin_config_baudrate(const size_t ADeviceHandle, const APP_CHANNEL AChnIdx, const double ARateKbps, TLINProtocol AProtocol);</code>
功能介绍	配置 lin 波特
调用位置	连接 lin 设备后
输入参数	ADeviceHandle: 设备句柄 AChnIdx: 通道索引 ARateKbps: 波特率 AProtocol: lin 协议 包括 {0: LIN_PROTOCOL_13, 1: LIN_PROTOCOL_20, 2: LIN_PROTOCOL_21, 3: LIN_PROTOCOL_J2602}
返回值	==0: 成功 其他值: 失败
示例	<pre>TLIBLIN ALIN; tslin_transmit_fastlin_async(ADeviceHandle, 0, 2000, LIN_PROTOCOL_13);</pre>

75. tsfifo_receive_lin_msgs

函数名称	<code>u32 tsfifo_receive_lin_msgs(const size_t ADeviceHandle, const TLIBLIN* ALINBuffers, s32* ALINBufferSize, u8 AChn, u8 ARXTX);</code>
功能介绍	Fifo 接收 lin 报文
调用位置	连接 lin 设备后
输入参数	ADeviceHandle: 设备句柄 ALINBuffers: lin 缓冲区 ALINBufferSize: 缓冲区大小 AChn: 通道 ARXTX: 1: 接收 TX/RX 所有报文, 如果只读取接收端的报文, 则修改为 0: READ_TX_RX_DEF. ONLY_RX_MESSAGES
返回值	==0: 成功 其他值: 失败
示例	<pre>TLIBLIN ALINBuffers[100]; s32 revCnt = sizeof(ALINBuffers)/sizeof(ALINBuffers[0]); tsfifo_receive_lin_msgs(ADeviceHandle, ALINBuffers, &revCnt, 0, 1);</pre>

76. tsfifo_receive_fastlin_msgs

函数名称	<code>u32 tsfifo_receive_fastlin_msgs(const size_t ADeviceHandle, const TLIBLIN* ALINBuffers, s32* ALINBufferSize, u8 AChn, u8 ARXTX);</code>
功能介绍	接收 Fifo fastlin 报文
调用位置	连接 lin 设备后
输入参数	ADeviceHandle: 设备句柄 ALINBuffers: lin 缓冲区

	ALINBufferSize: 缓冲区大小 AChn: 通道 ARXTX: 1:接收 TX/RX 所有报文, 如果只读取接收端的报文, 则修改为 0:READ_TX_RX_DEF_ONLY_RX_MESSAGES
返回值	==0: 成功 其他值: 失败
示例	TLIBLIN ALINBuffers[100]; s32 revCnt = sizeof(ALINBuffers)/sizeof(ALINBuffers[0]); tsfifo_receive_fastlin_msgs(ADeviceHandle, ALINBuffers, &revCnt, 0, 1);

77. tscan_get_error_description

函数名称	<code>u32 tscan_get_error_description(const u32 ACode, char** ADesc);</code>
功能介绍	根据函数执行的返回值编码 ACode, 查询该函数的执行结果
调用位置	TSCAN 的 API 函数执行过后, 会返回一个 ErrorCode 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 ADesc: 存储返回的错误详细信息的地址
返回值	==0: 成功 其他值: 失败
示例	char *ADesc = ""; tscan_get_error_description(23, &ADesc);

78. tsreplay_add_channel_map

函数名称	<code>u32 tsreplay_add_channel_map(const size_t ADeviceHandle, APP_CHANNEL ALogicChannel, APP_CHANNEL AHardwareChannel);</code>
功能介绍	添加重放通道映射
调用位置	
输入参数	ADeviceHandle: 设备句柄 ALogicChannel: 逻辑通道 AHardwareChannel: 硬件通道
返回值	==0: 成功 其他值: 失败
示例	tsreplay_add_channel_map(ADeviceHandle, CH1, CH2);

79. tsreplay_clear_channel_map

函数名称	<code>u32 tsreplay_clear_channel_map(const size_t ADeviceHandle);</code>
功能介绍	清除重放通道映射
调用位置	
输入参数	ADeviceHandle: 设备句柄

返回值	==0: 成功 其他值: 失败
示例	<code>tsreplay_clear_channel_map(ADeviceHandle);</code>

80. tsreplay_start_blf

函数名称	<code>u32 tsreplay_start_blf(const size_t ADeviceHandle, char* ABlfFilePath, int ATriggerByHardware, u64 AStartUs, u64 AEndUs);</code>
功能介绍	打开重放 blf 文件
调用位置	
输入参数	ADeviceHandle: 设备句柄 ABlfFilePath: blf 文件路径 ATriggerByHardware: 硬件触发 AStartUs: 起始时间(us) AEndUs: 中止时间(us)
返回值	==0: 成功 其他值: 失败
示例	<code>tsreplay_start_blf(ADeviceHandle, "log1", ATriggerByHardware, 100, 500);</code>

81. tsreplay_stop

函数名称	<code>u32 tsreplay_stop(const size_t ADeviceHandle);</code>
功能介绍	停止重放
调用位置	
输入参数	ADeviceHandle: 设备句柄
返回值	==0: 成功 其他值: 失败
示例	<code>tsreplay_stop(ADeviceHandle);</code>

82. tsdiag_can_create

函数名称	<code>u32 tsdiag_can_create(int* pDiagModuleIndex, u32 AChnIndex, byte ASupportFDCAN, byte AMaxDLC, u32 ARequestID, bool ARequestIDIsStd, u32 AResponseID, bool AResponseIDIsStd, u32 AFunctionID, bool AFunctionIDIsStd);</code>
功能介绍	创建 can 诊断模块

调用位置	
输入参数	pDiagModuleIndex:诊断模块索引 AChnIndex:通道索引 ASupportFDCAN: 支持的 CANFD AMaxDLC:最大 DLC 长度 ARequestID:请求 ID ARequestIDIsStd:请求 ID 是否标准 AResponseID:响应 ID AResponseIDIsStd:响应 ID 是否标准 AFunctionID:功能 ID AFunctionIDIsStd:功能 ID 是否标准
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>int pDiagModuleIndex = 0 ; tsdiag_can_create(&pDiagModuleIndex, 0, ASupportFDCAN, AMaxDLC, ARequestID, true, AResponseID, true, AFunctionID, true);</pre>

83. tsdiag_can_delete

函数名称	<code>u32 tsdiag_can_delete(int ADiagModuleIndex);</code>
功能介绍	删除 can 诊断
调用位置	
输入参数	ADiagModuleIndex: 诊断模块索引
返回值	==0: 成功 其他值: 失败
示例	<code>tsdiag_can_delete(10);</code>

84. tsdiag_can_delete_all

函数名称	<code>u32 tsdiag_can_delete_all(void);</code>
功能介绍	删除所有 can 诊断
调用位置	
输入参数	无
返回值	==0: 成功 其他值: 失败
示例	<code>tsdiag_can_delete_all();</code>

85. tsdiag_can_attach_to_tscan_tool

函数名称	<code>u32 tsdiag_can_attach_to_tscan_tool(int ADiagModuleIndex, size_t ACANToolHandle);</code>
------	--

功能介绍	固定 can 诊断模块到 tscan 工具上
调用位置	
输入参数	ADiagModuleIndex: 诊断模块索引 ACANToolHandle: can 工具句柄
返回值	==0: 成功 其他值: 失败
示例	tsdiag_can_attach_to_tscan_tool(0,0);

86. tstp_can_send_functional

函数名称	<code>u32 tstp_can_send_functional(int ADiagModuleIndex, byte* AReqArray, int AReqArraySize);</code>
功能介绍	功能 id 请求数据
调用位置	
输入参数	ADiagModuleIndex: uds 模块的句柄 AReqArray: 指向请求数据源的数据指针 AReqArraySize: 数据源数组的大小 (字节数)
返回值	==0: 成功 其他值: 失败
示例	<pre>byte [] reqdataArray= {0x3E, 0x80}; if(tstp_can_send_functional(ADiagModuleIndex, reqdataArray, 2)==0x00) { app.log_text("send functional payload Success!",lv10K); }</pre>

87. tstp_can_send_request

函数名称	<code>u32 tstp_can_send_request(int ADiagModuleIndex, byte* AReqArray, int AReqArraySize);</code>
功能介绍	请求 id 请求数据
调用位置	
输入参数	ADiagModuleIndex: 诊断模块索引 AReqArray: 指向请求数据源的数据指针 AReqArraySize: 数据源数组的大小 (字节数)
返回值	==0: 成功 其他值: 失败
示例	<pre>byte [] reqdataArray= {0x10,0x03}; if(tstp_can_send_request(udsHandle, reqdataArray, 2) == 0x00) { app.log_text("send diagnostic payload Success!", lv10K); }</pre>

88. tstp_can_request_and_get_response

函数名称	<code>u32 tstp_can_request_and_get_response(int ADiagModuleIndex, byte* AReqArray, int AReqArraySize, byte* AReturnArray, int* AReturnArraySize);</code>
功能介绍	请求 id 请求数据并获取响应数据
调用位置	
输入参数	ADiagModuleIndex:uds 模块的句柄 AReqArray:指向请求数据源的数据指针 AReqArraySize:数据源数组的大小（字节数） AReturnArray:指向响应数据缓冲区的数据指针 AReturnArraySize:指向整数值的数据指针，用于保存响应数据大小
返回值	==0: 成功 其他值: 失败
示例	<pre>byte reqDataArray= {0x10,0x03}; byte responseArray[10]; int responseArraySize = 10; if(tstp_can_request_and_get_response(udsHandle, reqDataArray, 2, responseArray, &responseArraySize) == 0x00) { app.log_text("send diagnostic payload and get response success!",lv10K); }</pre>

89. tsdiag_can_session_control

函数名称	<code>int tsdiag_can_session_control(int ADiagModuleIndex, byte ASubSession)</code>
功能介绍	设置会话控制
调用位置	
输入参数	ADiagModuleIndex:诊断模块索引 ASubSession:子会话值
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>if(tsdiag_can_session_control(udsHandle, 2) == 0x00) { app.log_text("switch sesion:2 success",lv10K); } else { app.log_text("switch sesion:2 failed",lv1Error); }</pre>

90. tsdiag_can_routine_control

函数名称	<code>int tsdiag_can_routine_control(int ADiagModuleIndex, byte AARoutineControlType, uint64 ARoutintID);</code>
功能介绍	设置 can 常规控制类型值
调用位置	
输入参数	ADiagModuleIndex:诊断模块索引 ARoutineControlType:例行控制类型 ARoutintID:例行 id
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>if(tsdiag_can_routine_control(udsHandle, 0x01, 0x9178) == 0x00) { app.log_text("routine control [type:0x01; id:0x9178] success", lv10K); } else { app.log("routine control:%d failed", ENABLE_RX_TX); }</pre>

91. tsdiag_can_communication_control

函数名称	<code>int tsdiag_can_communication_control(int ADiagModuleIndex, byte AControlType);</code>
功能介绍	设置 can 通信控制类型
调用位置	
输入参数	ADiagModuleIndex:诊断模块索引 AControlType:通信控制类型
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>if(tsdiag_can_communication_control(udsHandle, 0x00) == 0x00) { app.log_text("communication control:0x00 success", lv10K); } else { app.log_text("communication control:0x00 failed", lv1Error); }</pre>

92. tsdiag_can_security_access_request_seed

函数名称	<code>int tsdiag_can_security_access_request_seed(int ADiagModuleIndex, int ALevel, byte* ARecSeed, int* ARecSeedSize);</code>
------	--

功能介绍	27 seed 获取 seed
调用位置	连接硬件之后，需要创建诊断模块后，不再需要该诊断模块
输入参数	ADiagModuleIndex:诊断模块索引 ALevel:安全级别 ARecSeed:用于保存种子值的数据指针 ARecSeedSize:种子值的大小
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>byte seedBuffer[4]; //seedBuffer to save the seed received from the DUT int ARecSeedSize= 4; //byte num of the seed value if(tsdiaq_can_security_access_request_seed(ADiagModuleIndex, 0x01, seedBuffer,&ARecSeedSize) == 0x00) { app.log_text("get seed success!",lv1OK); } else { app.log_text("get seed failed!",lv1Error); }</pre>

93. tsdiag_can_security_access_send_key

函数名称	<code>int tsdiag_can_security_access_send_key(int ADiagModuleIndex, int ALevel, byte* ASeed, int ASeedSize);</code>
功能介绍	27 key 发送 key
调用位置	
输入参数	ADiagModuleIndex:诊断模块索引 ALevel:安全级别 ASeed:用于保存键值的数据指针 ARecSeedSize:键值大小
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>byte ASeed[4]; //Calculated from seed value if(tsdiaq_can_security_access_send_key(udsHandle, 0x02, ASeed, 4) == 0x00) { app.log_text("unlock the ecu success!",lv1OK); } else { app.log_text("unlock the ecu failed",lv1Error); }</pre>

94. tsdiag_can_request_download

函数名称	<code>int tsdiag_can_request_download(int ADiagModuleIndex, u32 AMemAddr, u32 AMemSize);</code>
功能介绍	客户端使用请求下载服务来启动从客户端到服务器的 can 数据传输
调用位置	
输入参数	ADiagModuleIndex:诊断模块索引 AMemAddr:存储器地址 AMemSize:存储器大小
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>if(tsdiag_can_request_download(udsHandle, 0x08000000, 0x00010000) == 0x00) { app.log_text("require download success!",lv10K); } else { app.log_text("require download failed!",lv10K); }</pre>

95. tsdiag_can_request_upload

函数名称	<code>int tsdiag_can_request_upload(int ADiagModuleIndex, u32 AMemAddr, u32 AMemSize);</code>
功能介绍	请求上传 CAN 数据包
调用位置	
输入参数	ADiagModuleIndex:诊断模块索引 AMemAddr:存储器地址 AMemSize:存储器大小
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>if(tsdiag_can_request_upload(udsHandle, 0x08000000, 0x00010000) == 0x00) { app.log_text("require upload success!",lv10K); } else { app.log_text("require upload failed!",lv1Error); }</pre>

96. tsdiag_can_transfer_data

函数名称	<code>int tsdiag_can_transfer_data(int ADiagModuleIndex, byte* ASourceDatas, int ASize, int AReqCase);</code>
功能介绍	启动传输数据包
调用位置	
输入参数	ADiagModuleIndex: 诊断模块索引 ASourceDatas: 指向数据缓冲区的数据指针 ASize: 数据集大小（字节数） AReqCase: req 情况
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>byte ASourceDatas[0x10000]; //buffer saved hex data if(tsdiag_can_transfer_data(ADiagModuleIndex, ASourceDatas, 0x10000, 0x00) == 0x00) { app.log_text("transfer data success!",lv10K); } else { app.log_text("transfer data failed!",lv1Error); }</pre>

97. tsdiag_can_request_transfer_exit

函数名称	<code>int tsdiag_can_request_transfer_exit(int ADiagModuleIndex)</code>
功能介绍	请求停止并退出 can 数据传输
调用位置	连接硬件之后，需要创建诊断模块后，不再需要该诊断模块
输入参数	ADiagModuleIndex: 诊断模块索引
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>if(tsdiag_can_request_transfer_exit(udsHandle) == 0x00) { app.log_text("request transfer data exit success!",lv10K); } else { app.log_text("request transfer data exit failed!",lv1Error); }</pre>

98. tsdiag_can_write_data_by_identifier

函数名称	<code>int tsdiag_can_write_data_by_identifier(int ADiagModuleIndex, u16 ADataIdentifier, byte* AWriteData, int AWriteDataSize)</code>
功能介绍	通过标识符写入数据
调用位置	
输入参数	ADiagModuleIndex: 诊断模块索引 ADataIdentifier: 要写入的数据的标识符 AWriteData: 指向数据缓冲区的数据指针 AWriteDataSize: 要写入的数据编号
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>byte writeData[10]; if(tsdiag_can_write_data_by_identifier(udsHandle, 0xF198, writeData, 10) == 0x00) { app.log_text("write data by id:0xF198 success!", lv10K); } else { app.log_text("write data by id:0xF198 failed!", lv1Error); }</pre>

99. tsdiag_can_read_data_by_identifier

函数名称	<code>int tsdiag_can_read_data_by_identifier(int ADiagModuleIndex, u16 ADataIdentifier, byte* AReturnArray, int AReturnArraySize)</code>
功能介绍	通过标识符读取数据
调用位置	
输入参数	ADiagModuleIndex: 诊断模块索引 ADataIdentifier: 要读取的数据的标识符 AReturnArray: 指向数据缓冲区的数据指针 AReturnArraySize: 从 dut 读取的实际数据编号
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>byte readData[20]; //Buffer to save datas read from ECU int readDataSize = 20; if(tsdiag_can_read_data_by_identifier(udsHandle, 0xF198, readData, &readDataSize) == 0x00) { app.log_text("read data by id:DATA_ID success!", lv10K); } else { </pre>

	<pre>app.log_text("read data by id:DATA_ID failed!",lvlError); }</pre>
--	--

100.tsflexray_register_event_flexray

函数名称	<code>int tsflexray_register_event_flexray(const size_t ADeviceHandle, const TFlexRayQueueEvent_Win32_t ACallback);</code>
功能介绍	注册 flexray 回调函数
调用位置	
输入参数	ADeviceHandle:设备句柄 ACallback:要处理接收报文的的委托函数
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>void ReceiveFlexRayMessage(const TLibFlexRay* AData) { //处理收到的报文数据 AData } tsflexray_register_event_flexray(ADeviceHandle,ReceiveFlexRayMessage);</pre>

101.tsflexray_unregister_event_flexray

函数名称	<code>int tsflexray_unregister_event_flexray(const size_t ADeviceHandle, const TFlexRayQueueEvent_Win32_t ACallback);</code>
功能介绍	注销 flexray 回调函数
调用位置	
输入参数	ADeviceHandle:设备句柄 ACallback:要处理接收报文的的委托函数
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>void ReceiveFlexRayMessage(const TLibFlexRay* AData) { //处理收到的报文数据 AData } tsflexray_unregister_event_flexray(ADeviceHandle,ReceiveFlexRayMessage);</pre>

102.tsflexray_unregister_pretx_event_flexray

函数名称	<code>int tsflexray_unregister_pretx_event_flexray(const size_t ADeviceHandle, const TFlexRayQueueEvent_Win32_t ACallback);</code>
功能介绍	注销预发送 flexray 事件
调用位置	

输入参数	ADeviceHandle:设备句柄 ACallback:要处理接收报文的的委托函数
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>void ReceivePreTxFlexRayMessage(const TLibFlexRay* AData) { //处理收到的报文数据 AData } tsflexray_unregister_pretx_event_flexray(ADeviceHandle, ReceiveFlexRayMessage);</pre>

103.tsflexray_register_pretx_event_flexray

函数名称	<code>int tsflexray_register_pretx_event_flexray(const size_t ADeviceHandle, const TFlexRayQueueEvent_Win32_t ACallback);</code>
功能介绍	注册预发送 flexray 事件
调用位置	
输入参数	ADeviceHandle:设备句柄 ACallback:要处理接收报文的的委托函数
返回值	==0: 执行成功 其他值: 查询错误码
示例	<pre>void ReceivePreTxFlexRayMessage(const TLibFlexRay* AData) { //处理收到的报文数据 AData } tsflexray_register_pretx_event_flexray(ADeviceHandle, ReceivePreTxFlexRayMessage);</pre>

8. 示例 工程

本工程演示了调用 API，实现加载 DLL，扫描设备，连接设备，注册回调函数，设置波特率，发送 LIN 报文，接收 LIN 报文等过程，运行效果图：

```
find libSCAN.dll!  
ISCAN Device Count:1  
Device with handle:81784864 connected  
Device with handle:81784864 register rev callback success  
Device with handle:81784864 set baudrate success  
Device with handle:81784864 sync send can message failed  
Device with handle:81784864 async send can message success  
Device with handle:81784864 set function type success  
Device with handle:81784864 apply new ldf callback success  
Device with handle:81784864 register rev callback success  
Device with handle:81784864 set baudrate success  
Device with handle:81784864 sync send lin message success  
Receive Recall  
Translate message:DataLength:3 Datas: 12 34 56  
1 messages received  
Translate message:DataLength:3 Datas: 12 34 56  
Device with handle:81784864 async send lin message success  
Receive Recall  
Receive message:DataLength:0 Datas:  
1 messages received  
Receive message:DataLength:0 Datas:  
Device with handle:81784864 unregister rev callback success  
Disconnect device with handle:81784864 success  
end dll!
```

为了同步监测 TSLIN 报文通讯情况，可以打开同星公司的 TSMaster 软件，打开 LIN Trace 窗口，监测 LIN 报文通讯过程，如下所示：

LIN Trace

Filter String:

Absolute Time	Chn	#	Identifier	FPS	Message Name	f0	Checksum	Dir	DLC	Message Data Bytes
> 4754.566786	1	3C		0	MasterReq	63		Tx	3	12 34 56
> 4754.575786	1	3D		0	SlaveResp	00		Rx	0	
> 5232.313286	1	3C		0	MasterReq	63		Tx	3	12 34 56
> 5232.323285	1	3D		0	SlaveResp	00		Rx	0	
> 5250.195285	1	3C		0	MasterReq	63		Tx	3	12 34 56
> 5250.204285	1	3D		0	SlaveResp	00		Rx	0	
> 5276.119785	1	3C		0	MasterReq	63		Tx	3	12 34 56
> 5276.128785	1	3D		0	SlaveResp	00		Rx	0	
> 5329.847784	1	3C		0	MasterReq	63		Tx	3	12 34 56
> 5329.854784	1	3D		0	SlaveResp	00		Rx	0	

注意，TSMaster 软件在监测过程中，尽量不要打开 PC 端的调度表下载仿真功能，这样会造成设备端默认维护一张调度表，影响用户程序调试。如下按钮：

The screenshot shows the LIN Transmitt software interface. On the left, there is a sidebar with icons for various functions and a list of items: LIN Schedule Tables, main, DiagRequest, and DiagResponse. The main window is titled 'CH1/testLDF/BCM2_LIN1 [Master]' and contains two tables.

The top table is a message list with columns: Row, Enable, Message Name, ID, Len, Dir, DLC, D0, D1, D2, D3, D4, D5, D6, D7, Delay Time (ms), and Comment. It contains 7 rows of data, including messages like SADSchalters_01, DWAE_01, UGDO_Be_01, IRUEs_01, NGSS_01, and SADSchalters_01.

The bottom table is a signal list with columns: Signal Name, Signal Gen., Generator, Raw Value, Raw Step, Physical Value, Phys Step, and Comment. It contains 18 rows of data, including signals like Dummy_pID, Schalter_MD1Vorne_Druck, Schalter_MD1Vorne_Zug, Schalter_MD1Vorne_Vorwahl, Schalter_MD1Vorne_Fehler, Schalter_MRolloVorne_auf, Schalter_MRolloVorne_zu, Schalter_MRolloVorne_Fehler, Schalter_MD3Vorne_auf, Schalter_MD3Vorne_zu, Schalter_MD3Vorne_Fehler, Schalter_MRollohinten_auf, Schalter_MRollohinten_zu, Schalter_MRollohinten_Fehler, and Schalter_MD3Vorne_Fehler.