

类别	内容
关键词	CAN/CANFD 接口函数库使用
摘 要	本软件可适用于广州致远电子有限公司出品的各种 CAN/CANFD 接口卡。接口函数库是提供给用户进行上位机二次开发，可以自行编程进行数据收发、处理等。

## 修订历史

版本	日期	原因
V1.00	2019/01/09	创建文档
V1.01	2019/03/18	更新文档页眉页脚、“销售与服务网络”内容和新增“免责声明”内容
V1.02	2019/09/06	调整云设备数据结构
V1.03	2019/09/24	统一波特率设置，添加代码示例
V1.04	2019/10/14	USBCAN-2E-U 属性表的滤波项添加说明
V1.05	2019/11/19	属性表添加调用顺序说明
V1.06	2020/07/08	添加设备类型号定义
V1.07	2020/07/31	USBCANFD 设备添加延时发送队列清除功能
V1.08	2021/01/07	更新 CANFDNET,PCIECANFD 属性表
V1.09	2021/03/12	更新部分文字引用，更新部分设备属性表描述
V1.10	2021/05/13	添加 ZCAN_TransmitData, ZCAN_ReceiveData, ZCAN_SetValue, ZCAN_GetValue 相关接口以及对应的数据结构说明
V1.11	2021/06/03	添加支持合并接收设备列表，更新合并接收 Demo

## 目 录

第 1 章 ZLGCAN 接口编程 .....	1
1.1 简介 .....	1
第 2 章 开发流程图 .....	2
2.1 普通 CAN 卡开发流程 .....	2
2.2 云设备 .....	4
第 3 章 数据结构及函数接口定义 .....	5
3.1 数据结构定义 .....	5
ZCAN_DEVICE_INFO .....	5
ZCAN_CHANNEL_INIT_CONFIG .....	6
ZCAN_CHANNEL_ERROR_INFO .....	8
ZCAN_CHANNEL_STATUS .....	8
can_frame .....	9
canfd_frame .....	10
ZCAN_Transmit_Data .....	10
ZCAN_TransmitFD_Data .....	11
ZCAN_Receive_Data .....	11
ZCAN_ReceiveFD_Data .....	12
ZCAN_AUTO_TRANSMIT_OBJ .....	12
ZCANFD_AUTO_TRANSMIT_OBJ .....	13
ZCAN_AUTO_TRANSMIT_OBJ_PARAM .....	13
ZCLOUD_DEVINFO .....	13
ZCLOUD_USER_DATA .....	14
ZCLOUD_GPS_FRAME .....	15
IProperty .....	16
ZCAN_LIN_MSG .....	16
ZCAN_LIN_INIT_CONFIG .....	17
ZCANCANFDData .....	18
ZCANErrorData .....	19
ZCANGPSData .....	21
ZCANLINData .....	23
ZCANDataObj .....	25
3.2 接口库函数说明 .....	26
ZCAN_OpenDevice .....	26
ZCAN_CloseDevice .....	26
ZCAN_GetDeviceInf .....	26
ZCAN_IsDeviceOnLine .....	27
ZCAN_InitCAN .....	27
ZCAN_StartCAN .....	27
ZCAN_ResetCAN .....	28
ZCAN_ClearBuffer .....	28
ZCAN_ReadChannelErrInfo .....	28

ZCAN_ReadChannelStatus .....	28
ZCAN_Transmit .....	29
ZCAN_TransmitFD .....	29
ZCAN_TransmitData .....	29
ZCAN_GetReceiveNum .....	30
ZCAN_Receive .....	30
ZCAN_ReceiveFD .....	31
ZCAN_ReceiveData .....	31
ZCAN_SetValue .....	32
ZCAN_GetValue .....	32
GetIProperty .....	32
ReleaseIProperty .....	33
ZCLOUD_SetServerInfo .....	33
ZCLOUD_ConnectServer .....	33
ZCLOUD_IsConnected .....	34
ZCLOUD_DisconnectServer .....	34
ZCLOUD_GetUserData .....	34
ZCLOUD_ReceiveGPS .....	34
ZCAN_InitLIN .....	34
ZCAN_StartLIN .....	35
ZCAN_ResetLIN .....	35
ZCAN_TransmitLIN .....	35
ZCAN_GetLINReceiveNum .....	36
ZCAN_ReceiveLIN .....	36
ZCAN_SetLINSlaveMsg .....	36
ZCAN_ClearLINSlaveMsg .....	37
3.3 设备功能和属性表 .....	37
3.3.1 USBCANFD 系列 .....	38
3.3.2 PCIECANFD 系列 .....	51
3.3.3 USBCAN-xE-U PCI-50x0-U 系列 .....	62
3.3.4 CANDTU-x00UR .....	69
3.3.5 以太网系列 1 .....	73
3.3.6 以太网系列 2 .....	78
3.3.7 其他接口卡 .....	90
第 4 章 附录 .....	94
附录 1 - 设备类型定义 .....	95
附录 2 - 支持合并接收设备列表 .....	97
附录 2 - 错误码定义 .....	98

# 第1章 ZLGCAN 接口编程

## 1.1 简介

为满足市场发展的需要，广州致远电子有限公司推出了各式各样的 CAN(FD)接口卡，例如 USBCANFD 系列、PCIECANFD 系列和 USBCAN 系列等等。除了必要的硬件支持，更是配备了功能完善的分析软件 ZCANPro，给 CAN(FD)开发和诊断带来了很大的便利。

为满足接口卡接入集成系统的需要，公司推出了统一的编程接口，同时支持 CAN 和 CANFD。除了简单易用的接口，还配以接口使用例程和接口使用说明。本文档将对编程接口的使用作详尽的描述，务必带给您更好的体验。

接口库以基于 window 系统的动态链接库(DLL)的方式提供，可实现设备打开、配置、报文收发、关闭等功能。接口库采用 visual studio 2008 开发，依赖运行库 2008 版本，需要确保计算机中已包含该运行库，否则可到微软官方网站下载安装。

如图 1.1 所示，资料包中包含了 zlgcan.dll、kerneldlls 和 zlgcan 文件夹，其中 kerneldlls 文件夹包含具体接口卡的操作库，zlgcan 文件夹主要包含 zlgcan.lib、zlgcan.h 以及一些其它头文件，可参考使用例程使用。

开发编程直接加载 zlgcan.dll 即可，zlgcan.h 为接口描述头文件，zlgcan.dll 和 kerneldlls 文件夹需要放在可执行程序同级目录下。

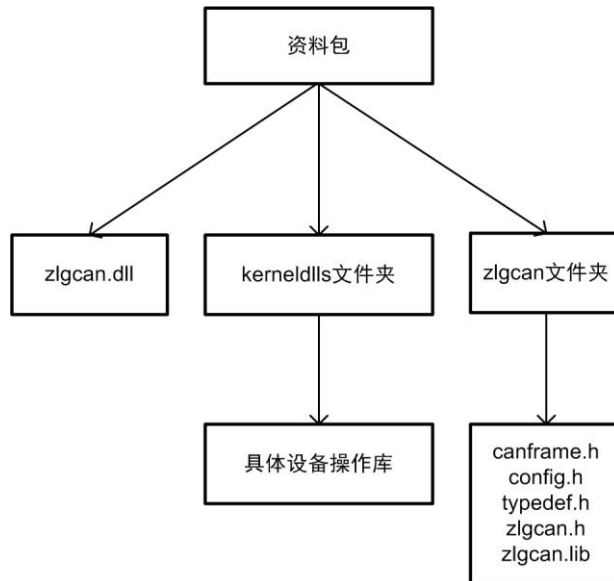


图 1.1 资料包结构

## 第2章 开发流程图

### 2.1 普通 CAN 卡开发流程

该开发流程适合我司出品的所有 CAN/CANFD 接口卡，请按照开发流程进行二次开发，流程如图 2.1 所示，开发代码示例如程序清单 2.1 所示。

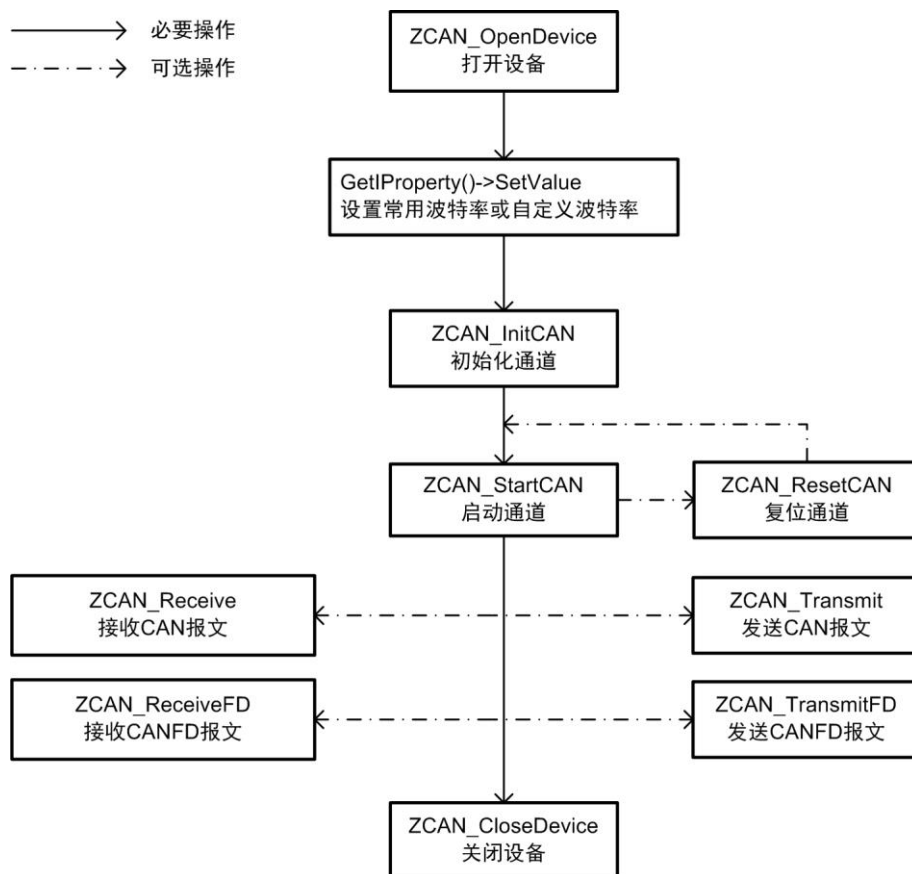


图 2.1 CAN 卡开发流程图

程序清单 2.1 开发代码示例

```
//ZCAN_USBCAN_2E_U 为设备类型，请根据实际修改
DEVICE_HANDLE dhandle = ZCAN_OpenDevice(ZCAN_USBCAN_2E_U, 0, 0);
if (INVALID_DEVICE_HANDLE == dhandle)
{
    std::cout << "打开设备失败" << std::endl;
    return 0;
}
IProperty* property = GetIProperty(dhandle);
if (NULL == property)
{
    std::cout << "属性指针为空" << std::endl;
    goto end;
}
```

//CAN 设备设置波特率的 key 为 baud\_rate, 值 1000000 为 1000kbps, 800000 为 800kbps, 其它请查看属性表

//若为 CANFD 设备, 设置仲裁域波特率的 key 为 canfd\_abit\_baud\_rate, 数据域波特率为 canfd\_dbit\_baud\_rate, 请注意区分 CAN 和 CANFD 设备设置波特率的差别。

```
if (property->SetValue("0/baud_rate", "1000000") != STATUS_OK)
{
    std::cout << "设置波特率失败" << std::endl;
    goto end;
}
ZCAN_CHANNEL_INIT_CONFIG cfg;
memset(&cfg, 0, sizeof(cfg));
cfg.can_type = TYPE_CAN; //CANFD 设备为 TYPE_CANFD
cfg.can.filter = 0;
cfg.can.mode = 0; //正常模式, 1 为只听模式
cfg.can.acc_code = 0;
cfg.can.acc_mask = 0xffffffff;
CHANNEL_HANDLE chHandle = ZCAN_InitCAN(dhandle, 0, &cfg);
if (INVALID_CHANNEL_HANDLE == chHandle)
{
    std::cout << "初始化通道失败" << std::endl;
    goto end;
}
if (ZCAN_StartCAN(chHandle) != STATUS_OK)
{
    std::cout << "启动通道失败" << std::endl;
    goto end;
}
ZCAN_Transmit_Data frame;
memset(&frame, 0, sizeof(frame));
frame.frame.can_id = MAKE_CAN_ID(0x100, 1, 0, 0);
frame.frame.can_dlc = 8;
BYTE data[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
memcpy(frame.frame.data, data, sizeof(data));
if (ZCAN_Transmit(chHandle, &frame, 1) != 1)
{
    std::cout << "发送数据失败" << std::endl;
    goto end;
}
end:
ReleaseProperty(property);
ZCAN_CloseDevice(dhandle);
```

## 2.2 云设备

本系列的开发流程图如图 2.2 所示，适用的设备有：ZCAN\_CLOUD (46)。

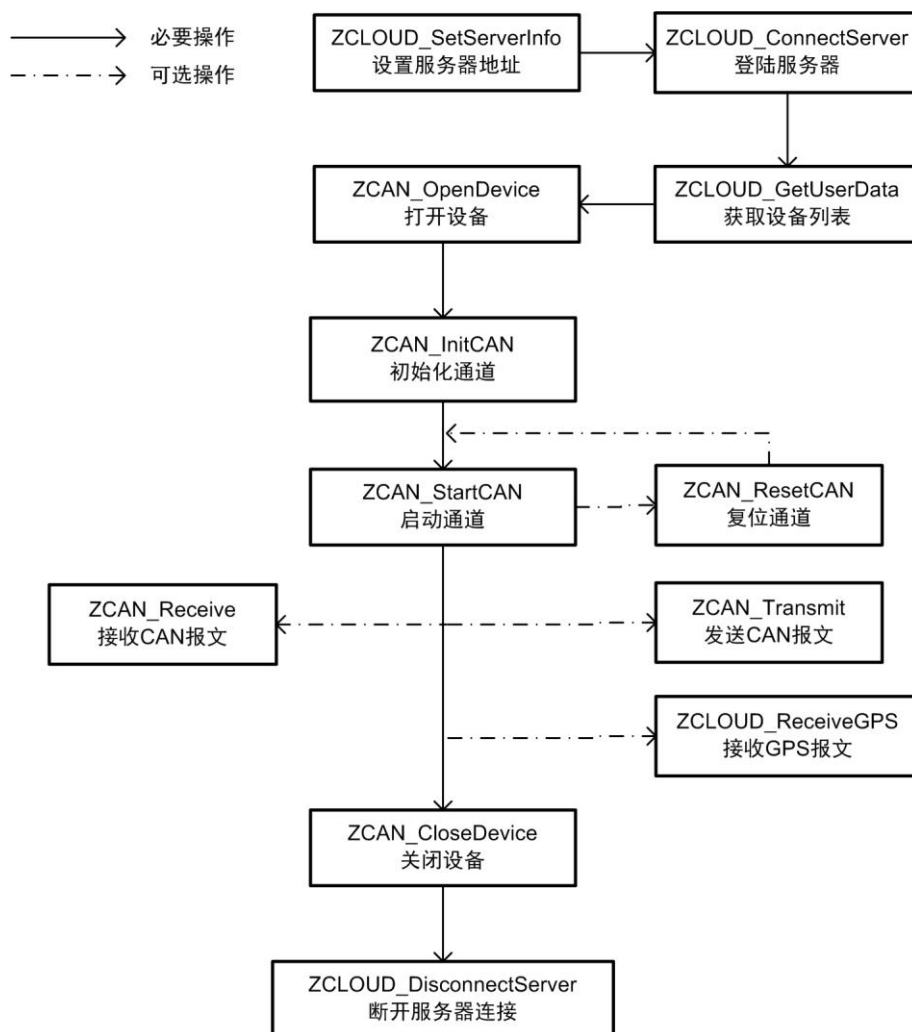


图 2.2 云设备流程图



## 第3章 数据结构及函数接口定义

### 3.1 数据结构定义

#### ZCAN\_DEVICE\_INFO

结构体详情见程序清单 3.1，包含设备的一些基本信息，在函数 ZCAN\_GetDeviceInf 中被填充。

程序清单 3.1 ZCAN\_DEVICE\_INFO 结构体成员

```
typedef struct tagZCAN_DEVICE_INFO {  
    USHORT hw_Version;  
    USHORT fw_Version;  
    USHORT dr_Version;  
    USHORT in_Version;  
    USHORT irq_Num;  
    BYTE   can_Num;  
    UCHAR  str_Serial_Num[20];  
    UCHAR  str_hw_Type[40];  
    USHORT reserved[4];  
}ZCAN_DEVICE_INFO;
```

#### 成员

##### hw\_Version

硬件版本号，16 进制，比如 0x0100 表示 V1.00。

##### fw\_Version

固件版本号，16 进制。

##### dr\_Version

驱动程序版本号，16 进制。

##### in\_Version

接口库版本号，16 进制。

##### irq\_Num

板卡所使用的中断号。

##### can\_Num

表示有几路通道。

##### str\_Serial\_Num

此板卡的序列号，比如”USBCAN V1.00”（注意：包括字符串结束符’\0’）。

##### str\_hw\_Type

硬件类型。

##### reserved

仅作保留，不设置。

## ZCAN\_CHANNEL\_INIT\_CONFIG

结构体详情见程序清单 3.2，定义了初始化配置的参数，调用 ZCAN\_InitCAN 之前，要先初始化该结构体。

程序清单 3.2 ZCAN\_CHANNEL\_INIT\_CONFIG 结构体成员

```
typedef struct tagZCAN_CHANNEL_INIT_CONFIG {
    UINT can_type; // 0:can 1:canfd
    union
    {
        struct
        {
            UINT acc_code;
            UINT acc_mask;
            UINT reserved;
            BYTE filter;
            BYTE timing0;
            BYTE timing1;
            BYTE mode;
        } can;
        struct
        {
            UINT acc_code;
            UINT acc_mask;
            UINT abit_timing;
            UINT dbit_timing;
            UINT brp;
            BYTE filter;
            BYTE mode;
            USHORT pad;
            UINT reserved;
        } canfd;
    };
}ZCAN_CHANNEL_INIT_CONFIG;
```

### 成员

#### **can\_type**

设备类型， 0 表示 CAN 设备， 1 表示 CANFD 设备。

#### ● CAN 设备

##### **acc\_code**

SJA1000 的帧过滤验收码，对经过屏蔽码过滤为“有关位”进行匹配，全部匹配成功后，此报文可以被接收，否则不接收。推荐设置为 0。

##### **acc\_mask**

SJA1000 的帧过滤屏蔽码，对接收的 CAN 帧 ID 进行过滤，位为 0 的是“有关位”，

位为 1 的是“无关位”。推荐设置为 0xFFFFFFFF，即全部接收。

**reserved**

仅作保留，不设置。

**filter**

滤波方式，=1 表示单滤波，=0 表示双滤波。

**timing0**

忽略，不设置。

**timing1**

忽略，不设置。

**mode**

工作模式，=0 表示正常模式（相当于正常节点），=1 表示只听模式（只接收，不影响总线）。

注：当设备类型为 PCI-5010-U、PCI-5020-U、USBCAN-E-U、USBCAN-2E-U、USBCAN-4E-U、CANDTU 时，帧过滤（acc\_code 和 acc\_mask 忽略）采用 GetIProperty 设置，详见 GetIProperty。

● **CANFD 设备**

**acc\_code**

验收码，同 CAN 设备。

**acc\_mask**

屏蔽码，同 CAN 设备。

**abit\_timing**

忽略，不设置。

**dbit\_timing**

忽略，不设置。

**brp**

波特率预分频因子，设置为 0。

**filter**

滤波方式，同 CAN 设备。

**mode**

模式，同 CAN 设备。

**pad**

数据对齐，不设置。

**reserved**

仅作保留，不设置。

注：当设备类型为 USBCANFD-100U、USBCANFD-200U、USBCANFD-MINI 时，帧过滤(acc\_code 和 acc\_mask 忽略)采用 GetIProperty 设置，详见 GetIProperty。

## ZCAN\_CHANNEL\_ERROR\_INFO

结构体详情见程序清单 3.3，包含总线错误信息，在函数 ZCAN\_ReadChannelErrInfo 中被填充。

程序清单 3.3 ZCAN\_CHANNEL\_ERROR\_INFO 结构体成员

```
typedef struct tagZCAN_CHANNEL_ERROR_INFO {  
    UINT error_code;  
    BYTE passive_ErrData[3];  
    BYTE arLost_ErrData;  
} ZCAN_CHANNEL_ERROR_INFO;
```

### 成员

#### **error\_code**

错误码，详见附录 2 - 错误码定义。

#### **passive\_ErrData**

当产生的错误中有消极错误时表示为消极错误的错误标识数据。

#### **arLost\_ErrData**

当产生的错误中有仲裁丢失错误时表示为仲裁丢失错误的错误标识数据。

## ZCAN\_CHANNEL\_STATUS

结构体详情见程序清单 3.4，包含控制器状态信息，在函数 ZCAN\_ReadChannelStatus 中被填充。

程序清单 3.4 ZCAN\_CHANNEL\_STATUS 结构体成员

```
typedef struct tagZCAN_CHANNEL_STATUS {  
    BYTE errInterrupt;  
    BYTE regMode;  
    BYTE regStatus;  
    BYTE regALCapture;  
    BYTE regECCapture;  
    BYTE regEWLimit;  
    BYTE regRECounter;  
    BYTE regTECounter;  
    UINT Reserved;  
} ZCAN_CHANNEL_STATUS;
```

### 成员

#### **errInterrupt**

中断记录，读操作会清除中断。

#### **regMode**

CAN 控制器模式寄存器值。

#### **regStatus**

CAN 控制器状态寄存器值。

**regALCapture**

CAN 控制器仲裁丢失寄存器值。

**regECCapture**

CAN 控制器错误寄存器值。

**regEWLimit**

CAN 控制器错误警告限制寄存器值。默认为 96。

**regRECounter**

CAN 控制器接收错误寄存器值。为 0-127 时，为错误主动状态；为 128-254 时，为错误被动状态；为 255 时，为总线关闭状态。

**regTECounter**

CAN 控制器发送错误寄存器值。为 0-127 时，为错误主动状态；为 128-254 时，为错误被动状态；为 255 时，为总线关闭状态。

**reserved**

仅作参考，不设置。

**can\_frame**

结构体详情见程序清单 3.5，包含了 CAN 报文信息。

程序清单 3.5 can\_frame 结构体成员

```
struct can_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8    can_dlc; /* frame payload length in byte (0 .. CAN_MAX_DLEN) */
    __u8    __pad; /* padding */
    __u8    __res0; /* reserved / padding */
    __u8    __res1; /* reserved / padding */
    __u8    data[CAN_MAX_DLEN]/* __attribute__((aligned(8)))*/;
};
```

**成员****can\_id**

帧 ID，32 位，高 3 位属于标志位，标志位含义如下：

第 31 位(最高位)代表扩展帧标志，=0 表示标准帧，=1 代表扩展帧，宏 IS\_EFF 可获取该标志；

第 30 位代表远程帧标志，=0 表示数据帧，=1 表示远程帧，宏 IS\_RTR 可获取该标志；

第 29 位代表错误帧标准，=0 表示 CAN 帧，=1 表示错误帧，目前只能设置为 0；

其余位代表实际帧 ID 值，使用宏 MAKE\_CAN\_ID 构造 ID，使用宏 GET\_ID 获取 ID。

**can\_dlc**

数据长度。

**\_\_pad**

对齐，忽略。

**\_\_res0**

仅作保留，不设置。

**\_\_res1**

仅作保留，不设置。

**data**

报文数据，有效长度为 can\_dlc。

## canfd\_frame

结构体详情见程序清单 3.6，包含了 CANFD 报文信息。

程序清单 3.6 canfd\_frame 结构体成员

```
struct canfd_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8 len; /* frame payload length in byte */
    __u8 flags; /* additional flags for CAN FD,i.e error code */
    __u8 __res0; /* reserved / padding */
    __u8 __res1; /* reserved / padding */
    __u8 data[CANFD_MAX_DLEN]/* __attribute__((aligned(8)))*/;
};
```

## 成员

**can\_id**

帧 ID，同 can\_frame 结构的 can\_id 成员。

**len**

数据长度。

**flags**

额外标志，比如使用 CANFD 加速，则设置为宏 CANFD\_BRS。

**\_\_res0**

仅作保留，不设置。

**\_\_res1**

仅作保留，不设置。

**data**

报文数据，有效长度为 len。

## ZCAN\_Transmit\_Data

结构体详情见程序清单 3.7，包含发送的 CAN 报文信息，在函数 ZCAN\_Transmit 中使用。

程序清单 3.7 ZCAN\_Transmit\_Data 结构体成员

```
typedef struct tagZCAN_Transmit_Data
{
    can_frame frame;
    UINT transmit_type;
```

```
}ZCAN_Transmit_Data;
```

## 成员

### frame

报文数据信息，详见 can\_frame 结构说明。

### transmit\_type

发送方式，0=正常发送，1=单次发送，2=自发自收，3=单次自发自收。

发送方式说明如下：

- **正常发送：**在ID仲裁丢失或发送出现错误时，CAN控制器会自动重发，直到发送成功，或发送超时，或总线关闭。
- **单次发送：**在一些应用中，允许部分数据丢失，但不能出现传输延迟时，自动重发就没有意义了。在这些应用中，一般会以固定的时间间隔发送数据，自动重发会导致后面的数据无法发送，出现传输延迟。使用单次发送，仲裁丢失或发送错误，CAN控制器不会重发报文。
- **自发自收：**产生一次带自接收特性的正常发送，在发送完成后，可以从接收缓冲区中读到已发送的报文。
- **单次自发自收：**产生一次带自接收特性的单次发送，在发送出错或仲裁丢失不会执行重发。在发送完成后，可以从接收缓冲区中读到已发送的报文。

## ZCAN\_TransmitFD\_Data

结构体详情见程序清单 3.8，包含发送的 CANFD 报文信息，在函数 ZCAN\_TransmitFD 中使用。

程序清单 3.8 ZCAN\_TransmitFD\_Data 结构体成员

```
typedef struct tagZCAN_TransmitFD_Data
{
    canfd_frame frame;
    UINT transmit_type;
}ZCAN_TransmitFD_Data;
```

## 成员

### frame

报文数据信息，详见 canfd\_frame 结构说明。

### transmit\_type

发送方式，同 ZCAN\_Transmit\_Data 结构的 transmit\_type 成员。

## ZCAN\_Receive\_Data

结构体详情见程序清单 3.9，包含接收的 CAN 报文信息，在函数 ZCAN\_Receive 中使用。

程序清单 3.9 ZCAN\_Receive\_Data 结构体成员

```
typedef struct tagZCAN_Receive_Data
{
    can_frame frame;
    UINT64 timestamp;
```

```
}ZCAN_Receive_Data;
```

## 成员

### frame

报文数据信息，详见 can\_frame 结构说明。

### timestamp

时间戳，单位微秒，基于设备启动时间。（如果为云设备，则基于 1970 年 1 月 1 日 0 时 0 分 0 秒）

## ZCAN\_ReceiveFD\_Data

结构体详情见程序清单 3.10，包含接收的 CANFD 报文信息，在函数 ZCAN\_ReceiveFD 中使用。

程序清单 3.10 ZCAN\_ReceiveFD\_Data 结构体成员

```
typedef struct tagZCAN_ReceiveFD_Data
{
    canfd_frame frame;
    UINT64      timestamp;
}ZCAN_ReceiveFD_Data;
```

## 成员

### frame

报文数据信息，详见 canfd\_frame 结构说明。

### timestamp

时间戳，单位微秒。

## ZCAN\_AUTO\_TRANSMIT\_OBJ

结构体详情见程序清单 3.11，包含定时发送 CAN 参数信息。

程序清单 3.11 ZCAN\_AUTO\_TRANSMIT\_OBJ 结构体成员

```
typedef struct tagZCAN_AUTO_TRANSMIT_OBJ{
    USHORT enable;
    USHORT index;
    UINT   interval;//定时发送时间。单位毫秒
    ZCAN_Transmit_Data obj;//报文
}ZCAN_AUTO_TRANSMIT_OBJ, *PZCAN_AUTO_TRANSMIT_OBJ;
```

## 成员

### enable

使能本条报文，0=禁能，1=使能。

### index

报文编号，从 0 开始，编号相同则使用最新的一条信息。

### interval

发送周期，单位毫秒。



### **obj**

发送的报文，详见 ZCAN\_Transmit\_Data 结构说明。

## **ZCANFD\_AUTO\_TRANSMIT\_OBJ**

结构体详情见程序清单 3.12，包含定时发送 CANFD 参数信息。

程序清单 3.12 ZCANFD\_AUTO\_TRANSMIT\_OBJ 结构体成员

```
typedef struct tagZCANFD_AUTO_TRANSMIT_OBJ{
    USHORT enable;
    USHORT index;
    UINT interval;
    ZCAN_TransmitFD_Data obj;//报文
}ZCANFD_AUTO_TRANSMIT_OBJ, *PZCANFD_AUTO_TRANSMIT_OBJ;
```

### **成员**

#### **enable**

使能本条报文，0=禁能，1=使能。

#### **index**

报文编号，从 0 开始，编号相同则使用最新的一条信息。

#### **interval**

发送周期，单位毫秒。

#### **obj**

发送的报文，详见 ZCAN\_TransmitFD\_Data 结构说明。

## **ZCAN\_AUTO\_TRANSMIT\_OBJ\_PARAM**

用于设置定时发送额外的参数，目前只支持 USBCANFD-X00U 系列设备，结构体 详情见程序清单 3.12。

程序清单 3.13 ZCAN\_AUTO\_TRANSMIT\_OBJ\_PARAM 结构体成员

```
typedef struct tagZCAN_AUTO_TRANSMIT_OBJ_PARAM
{
    USHORT index;    // 定时发送帧的索引
    USHORT type;     // 参数类型，目前类型只有 1：表示启动延时
    UINT value;      // 参数数值，单位 ms
}ZCAN_AUTO_TRANSMIT_OBJ_PARAM, *PZCAN_AUTO_TRANSMIT_OBJ_PARAM;
```

## **ZCLOUD\_DEVINFO**

结构体详情见程序清单 3.14，包含云设备的属性信息，在 ZCLOUD\_GetUserData 中被填充。

程序清单 3.14 ZCLOUD\_DEVINFO 结构体成员

```
typedef struct tagZCLOUD_DEVINFO
{
    int devIndex;
```

```

char type[64];
char id[64];
char owner[64];
char model[64];
char fwVer[16];
char hwVer[16];
char serial[64];
BYTE canNum;
int status;
BYTE bCanUploads[16];
BYTE bGpsUpload;
}ZCLOUD_DEVINFO;

```

## 成员

### **devIndex**

设备索引号，指该设备在该用户关联的所有设备中的索引序号。

### **type**

设备类型字符串。

### **id**

设备唯一识别号，字符串。

### **owner**

设备的拥有者

### **model**

模块型号字符串。

### **fwVer**

固件版本号字符串，如 V1.01。

### **hwVer**

硬件版本号字符串，如 V1.01。

### **serial**

设备序列号字符串。

### **canNum**

设备 CAN 通道数量。

### **status**

设备状态，0：设备在线，1：设备离线。

### **bCanUploads**

各通道数据云上送使能，0：不上送，1：上送。

### **bGpsUpload**

设备 GPS 数据云上送使能，0：不上送，1：上送。

## **ZCLOUD\_USER\_DATA**

结构体详情见程序清单 3.15，包含用户信息，包含用户基本信息以及用户拥有的设备信

息，通过 ZCLOUD\_GetUserData 获取。

程序清单 3.15 ZCLOUD\_USER\_DATA 结构体成员

```
typedef struct tagZCLOUD_USER_DATA
{
    char username[64];
    char mobile[64];
    ZCLOUD_DEVINFO devices[ZCLOUD_MAX_DEVICES];
    size_t devCnt;
}ZCLOUD_USER_DATA;
```

## 成员

### **username**

用户名字符串。

### **mobile**

用户手机号。

### **devices**

用户拥有的设备组，详见 ZCLOUD\_DEVINFO 结构说明。

### **devCnt**

设备个数。

## ZCLOUD\_GPS\_FRAME

结构体详情见程序清单 3.16，包含设备 GPS 数据，通过 ZCLOUD\_ReceiveGPS 获取。

程序清单 3.16 ZCLOUD\_GPS\_FRAME 结构体成员

```
typedef struct tagZCLOUD_GPS_FRAME
{
    float latitude;
    float longitude;
    float speed;
    struct __gps_time {
        USHORT    year;
        USHORT    mon;
        USHORT    day;
        USHORT    hour;
        USHORT    min;
        USHORT    sec;
    }tm;
} ZCLOUD_GPS_FRAME;
```

## 成员

### **latitude**

纬度。

### **longitude**

经度。

**speed**

速度。

**tm**

时间结构。

## IPROPERTY

结构体详情见程序清单 3.17, 用于获取/设置设备参数信息, 示例代码参考程序清单 3.18。

程序清单 3.17 IPROPERTY 结构体成员

```
typedef struct tagIPROPERTY
{
    SetValueFunc    SetValue;
    GetValueFunc    GetValue;
    GetPropertyFunc GetProperty;
}IPROPERTY;
```

### 成员

#### SetValue

设置设备属性值, 详见 3.3 小节。

#### GetValue

获取属性值。

#### GetProperty

用于返回设备包含的所有属性。

程序清单 3.18 IPROPERTY 示例代码

```
char path[50] = {0};
char value[50] = {0};
IPROPERTY* property_ = GetIPROPERTY(device_handle); // device_handle 为设备句柄
sprintf_s(path, "%d/canfd_abit_baud_rate", 0); // 0 代表通道 0
sprintf_s(value, "%d", 1000000); // 1Mbps 为 1000000
if (0 == property_>SetValue(path, value))
{
    return FALSE;
}
```

## ZCAN\_LIN\_MSG

结构体详情见程序清单 3.19, 该结构体定义了 LIN 消息的结构, 在设置从站响应信息和接收 LIN 数据接口中使用此结构表示单帧 LIN 消息。

程序清单 3.19 ZCAN\_LIN\_MSG 结构体成员

```
typedef struct _VCI_LIN_MSG{
    BYTE    ID;
    BYTE    DataLen;
```

```

    USHORT   Flag;
    UINT      TimeStamp;
    BYTE      Data[8];
}ZCAN_LIN_MSG, *PZCAN_LIN_MSG;

```

## 成员

### ID

消息 ID，LIN ID 中的 ID 部分，不包括校验。校验信息设备自动计算。

### DataLen

消息长度，表示后续 Data 的有效长度。

### Flag

消息 Flag，暂未使用。

### TimeStamp

时间戳，表示消息接收时间。

### Data

消息的负载数据，具体长度由 DataLen 成员给出。

## ZCAN\_LIN\_INIT\_CONFIG

结构体详情见程序清单 3.20，该结构体表示配置 LIN 的信息，在函数 VCI\_InitLIN 函数中调用。用于设置设备 LIN 的工作模式、波特率，是否使用增强校验等信息。

程序清单 3.20 ZCAN\_LIN\_INIT\_CONFIG 结构体成员

```

typedef struct _VCI_LIN_INIT_CONFIG
{
    BYTE      linMode;
    BYTE      linFlag;
    USHORT    reserved;
    UINT      linBaud;
}ZCAN_LIN_INIT_CONFIG, *PZCAN_LIN_INIT_CONFIG;

```

## 成员

### linMode

LIN 工作模式，主站为 0，从站为 1。

### linFlag

LIN 支持的特性，多个特性使用位或方式进行指定。

### Reserved

保留位。

### linBaud

LIN 作为主站时使用的波特率。

## 标志定义

```

#define LIN_MODE_MASTER      0
#define LIN_MODE_SLAVE      1

```

```
#define LIN_FLAG_CHK_ENHANCE    0x01
#define LIN_FLAG_VAR_DLC        0x02
```

## ZCANCANFDData

结构体详情见程序清单 3.21，该结构体表示 CAN/CANFD 帧结构，可以表示发送接收 CAN/CANFD 帧，目前仅作为 ZCANDataObj 结构的成员使用。

程序清单 3.21 ZCANCANFDData 结构体成员

```
typedef struct tagZCANCANFDData
{
    UINT64        timeStamp;
    union
    {
        struct{
            UINT    frameType : 2;
            UINT    txDelay : 2;
            UINT    transmitType : 4;
            UINT    txEchoRequest : 1;
            UINT    txEchoed : 1;
            UINT    reserved : 22;
        }unionVal;
        UINT        rawVal;
    }flag;
    BYTE          extraData[4];
    canfd_frame frame;
}ZCANCANFDData;
```

## 成员

### timeStamp

时间戳，作为接收帧时，时间戳单位微秒(us)。正常发送时，timeStamp 字段无意义。队列延迟发送数据时，timeStamp 字段存放发送当前帧后设备等待的时间，时间单位取决于 flag.unionVal.txDelay，等待时间结束后设备发送下一帧。

### flag

flag 字段表示 CAN/CANFD 帧的标记信息，长度 4 字节。flag 字段的含义如表 3.1 所示。

表 3.1 CAN/CANFD 帧 Flag

字段	发送/接收有效	描述
frameType	发送/接收有效	帧类型 0: CAN 帧 1: CANFD 帧
txDelay	发送有效	队列发送延时，延时时间存放在 timeStamp 字段 0: 不启用延时

		1: 启用延时，延时时间单位为 1 毫秒(1ms) 2: 启用延时，延时时间单位为 100 微秒(0.1ms) 队列发送延时支持情况取决于设备，具体见对应设备的队列发送章节。
transmitType	发送有效	发送类型 0: 正常发送 1: 单次发送 2: 自发自收 3: 单次自发自收 所有设备支持正常发送，其余发送类型参考具体设备说明
txEchoRequest	发送有效	发送回显请求 0: 不需要设备回显发送帧 1: 请求设备回显发送帧 支持发送回显的设备，发送数据时将此位置 1，设备可以通过接收数据接口收到发送出去的数据帧，接收到的发送数据使用 txEchoed 位标记 通过发送回显请求，可以获取帧发送到总线的准确时间
txEchoed	接收有效	报文是否是发送回显报文 0: 正常总线接收到的报文 1: 本设备发送回显报文
reserved	----	保留字段，暂未使用

### extraData

帧附加数据，暂未使用。

### frame

frame 成员用于保存 CAN/CANFD 帧的 ID，标准帧/扩展帧，数据帧/远程帧标记，帧长度，数据等信息。frame 结构可以参考 canfd\_frame 结构体说明部分。

### ZCANErrorData

结构体详情见程序清单 3.22，该结构体表示错误信息结构，可以表示总线错误、控制器错误、设备端错误等错误信息，目前仅作为 ZCANDataObj 结构的成员使用。

程序清单 3.22 ZCANErrorData 结构体成员

```
typedef struct tagZCANErrorData
{
    UINT64    timeStamp;
    BYTE      errType;
    BYTE      errSubType;
    BYTE      nodeState;
    BYTE      rxErrCount;
    BYTE      txErrCount;
    BYTE      errData;
```

```

    BYTE    reserved[2];
}ZCANErrorData;

```

## 成员

### timeStamp

时间戳，表示错误产生的时间，时间单位为微秒(us)。

### errType

错误类型，错误类型对应的数值如表 3.2 所示。

表 3.2 错误类型

错误类型	值	描述
ZCAN_ERR_TYPE_NO_ERR	0	无错误
ZCAN_ERR_TYPE_BUS_ERR	1	总线错误
ZCAN_ERR_TYPE_CONTROLLER_ERR	2	控制器错误
ZCAN_ERR_TYPE_DEVICE_ERR	3	终端设备错误

### errSubType

错误子类型，错误子类型的值根据错误类型不同表示不同的含义。总线错误的错误子类型如表 3.3 所示，控制器错误的错误子类型如表 3.4 所示，终端设备错误的错误子类型如表 3.5 所示。

表 3.3 总线错误子类型

错误子类型	值	描述
ZCAN_BUS_ERR_NO_ERR	0	无错误，nodeState 字段表示当前总线状态
ZCAN_BUS_ERR_BIT_ERR	1	位错误
ZCAN_BUS_ERR_ACK_ERR	2	应答错误
ZCAN_BUS_ERR_CRC_ERR	3	CRC 错误
ZCAN_BUS_ERR_FORM_ERR	4	格式错误
ZCAN_BUS_ERR_STUFF_ERR	5	填充错误
ZCAN_BUS_ERR_OVERLOAD_ERR	6	超载错误
ZCAN_BUS_ERR_ARBITRATION_LOST	7	仲裁丢失

表 3.4 控制器错误子类型

错误子类型	值	描述
ZCAN_CONTROLLER_RX_FIFO_OVERFLOW	1	控制器接收 FIFO 溢出
ZCAN_CONTROLLER_DRIVER_RX_BUFFER_OVERFLOW	2	驱动接收缓存溢出
ZCAN_CONTROLLER_DRIVER_TX_BUFFER_OVERFLOW	3	驱动发送缓存溢出



ZCAN_CONTROLLER_INTERNAL_ERROR	4	控制器内部错误
--------------------------------	---	---------

表 3.5 终端设备错误子类型

错误子类型	值	描述
ZCAN_DEVICE_APP_RX_BUFFER_OVERFLOW	1	终端应用接收缓存溢出
ZCAN_DEVICE_APP_TX_BUFFER_OVERFLOW	2	终端应用发送缓存溢出
ZCAN_DEVICE_APP_AUTO_SEND_FAILED	3	定时发送失败, errData 存放定时发送帧的索引。
ZCAN_CONTROLLER_TX_FRAME_INVALID	4	发送报文无效

### nodeState

节点状态, 显示当前节点的总线状态, 错误类型(errType)为总线错误(1)时有效。节点状态含义如表 3.6 所示。

表 3.6 节点状态

节点状态	值	描述
ZCAN_NODE_STATE_ACTIVE	1	总线积极
ZCAN_NODE_STATE_WARNNING	2	总线告警
ZCAN_NODE_STATE_PASSIVE	3	总线消极
ZCAN_NODE_STATE_BUSOFF	4	总线关闭

### rxErrCount

接收错误计数。

### txErrCount

发送错误计数。

### errData

错误数据, 数据的含义取决于当前的错误类型以及错误子类型, 具体请参考错误子类型说明。

### reserved

保留字段, 未使用。

## ZCANGPSData

结构体详情见程序清单 3.23, 该结构体表示 GPS 数据, 目前仅作为 ZCANDataObj 结构的成员使用。

程序清单 3.23 ZCANGPSData 结构体成员

```
typedef struct tagZCANGPSData
{
    struct {
        USHORT    year;
```

```

        USHORT    mon;
        USHORT    day;
        USHORT    hour;
        USHORT    min;
        USHORT    sec;
        USHORT    milsec;
    }        time;
union{
    struct{
        USHORT timeValid : 1;
        USHORT latlongValid : 1;
        USHORT altitudeValid : 1;
        USHORT speedValid : 1;
        USHORT courseAngleValid : 1;
        USHORT reserved:13;
    }unionVal;
    USHORT rawVal;
}flag;
double latitude;
double longitude;
double altitude;
double speed;
double courseAngle;
} ZCANGPSData;

```

## 成员

### time

UTC 时间，表示定位数据的时间。time 成员是结构体形式，采用年月日时分秒的形式表示时间，时间采用 UTC 时间。

### flag

数据标志位。主要用于标识定位数据的有效型，具体标志如表 3.7 所示。

表 3.7 GPS 数据标志字段

标志位	描述
timeValid	时间数据(time)是否有效 0: 无效; 1: 有效
latlongValid	经纬度数据(latitude/ longitude)是否有效 0: 无效; 1: 有效
altitudeValid	海拔数据(altitude)是否有效 0: 无效; 1: 有效
speedValid	速度数据(speed)是否有效 0: 无效; 1: 有效

courseAngleValid	航向角数据(courseAngle)是否有效 0: 无效; 1: 有效
reserved	保留

### latitude

纬度，正数表示北纬，负数表示南纬。

### longitude

经度，正数表示东经，负数表示西经。

### altitude

海拔，单位：米。

### speed

速度，单位： km/h。

### courseAngle

航向角。

## ZCANLINData

结构体详情见程序清单 3.24，该结构体表示 LIN 数据结构，目前仅作为 ZCANDataObj 结构的成员使用。

程序清单 3.24 ZCANLINData 结构体成员

```
typedef struct tagZCANLINData
{
    UINT64          timeStamp;
    union {
        struct {
            BYTE     ID:6;
            BYTE     Parity:2;
        }unionVal;
        BYTE        rawVal;
    }        PID;
    BYTE           dataLen;
    union{
        struct{
            USHORT tx : 1;
            USHORT rx : 1;
            USHORT noData : 1;
            USHORT chkSumErr : 1;
            USHORT parityErr : 1;
            USHORT syncErr : 1;
            USHORT bitErr : 1;
            USHORT wakeUp : 1;
            USHORT reserved : 8;
        }unionVal;
```

```

        USHORT rawVal;
    }flag;
    BYTE    chkSum;
    BYTE    reserved[3];
    BYTE    data[8];
}ZCANLINData;

```

## 成员

### timeStamp

时间戳，单位微秒(us)，表示数据帧接收时间。

### PID

受保护的帧 ID。PID 包含帧 ID(PID.uionVal.ID)和帧 ID 校验(PID.uionVal. Parity)两个部分。发送 LIN 数据时，用户填充 ID 时可以忽略 ID 校验部分，设备发送的时候会自动计算 ID 校验后发送。

### dataLen

数据长度。

### flag

数据标志位，2 个字节。标志位字段含义如表 3.8 所示。

表 3.8 LIN 数据标志字段

标志位	描述
tx	此位为 1 表示控制器发送在总线上的消息
rx	此位为 1 表示控制器接收总线上的消息
noData	此位为 1 表示无数据区
chkSumErr	此位为 1 表示校验和错误
parityErr	此位为 1 表示奇偶校验错误，此时消息中的 checksum 无效
syncErr	此位为 1 表示同步段错误
bitErr	此位为 1 表示发送时位错误
wakeUp	此位为 1 表示收到唤醒帧，此时消息 ID、数据长度、数据域、校验值无效
reserved	保留

### chkSum

数据校验，部分设备不支持校验数据的获取。

### reserved

保留。

### data

数据部分，有效数据长度参见 dataLen 成员。

## ZCANDataObj

结构体详情见程序清单 3.25，该结构作为合并接收使用的各种数据的载体，支持 CAN，CANFD，LIN，GPS，错误数据等各种不同类型的数据。

程序清单 3.25 ZCANDataObj 结构体成员

```
typedef struct tagZCANDataObj
{
    BYTE      dataType;
    BYTE      chnl;
    union{
        struct{
            USHORT reserved : 16;
        }unionVal;
        USHORT rawVal;
    }flag;
    BYTE      extraData[4];
    union
    {
        ZCANCANFDData      zcanCANFDData;
        ZCANErrorData      zcanErrData;
        ZCANGPSData        zcanGPSData;
        ZCANLINData         zcanLINData;
        BYTE                raw[92];
    } data;
}ZCANDataObj;
```

### 成员

#### dataType

数据类型，指示当前结构的数据类型。数据类型含义如表 3.9 所示。

表 3.9 数据类型

数据类型	值	描述
ZCAN_DT_ZCAN_CAN_CANFD_DATA	1	CAN/CANFD 数据, data. zcanCANFDData 有效
ZCAN_DT_ZCAN_ERROR_DATA	2	错误数据, data. zcanErrData 有效
ZCAN_DT_ZCAN_GPS_DATA	3	GPS 数据, data. zcanGPSData 有效
ZCAN_DT_ZCAN_LIN_DATA	4	LIN 数据, data. zcanLINData 有效

#### chnl

数据通道。数据类型表示 CAN/CANFD/错误数据时，通道表示的是 CAN 通道。数据类型表示的是 LIN 数据时，通道表示的是设备的 LIN 通道。

#### flag

数据标志，暂未使用。

#### extraData

额外数据，暂未使用。

### data

data 成员定义为联合体，包含具体数据结构。data 各成员的类型和有效性参见表 3.10 说明。

表 3.10 数据成员类型和有效性

数据成员	数据类型	描述
zcanCANFDData	ZCANCANFDData	CAN/CANFD 数据，dataType 值为 1 时有效
zcanErrData	ZCANErrorData	错误数据，dataType 值为 2 时有效
zcanGPSData	ZCANGPSData	GPS 数据，dataType 值为 3 时有效
zcanLINData	ZCANLINData	LIN 数据，dataType 值为 4 时有效

## 3.2 接口库函数说明

### ZCAN\_OpenDevice

该函数用于打开设备。一个设备只能被打开一次。

```
DEVICE_HANDLE ZCAN_OpenDevice(UINT device_type, UINT device_index, UINT reserved);
```

#### 参数

##### device\_type

设备类型，详见头文件 zlgcan.h 中的宏定义。

##### device\_index

设备索引号，比如当只有一个 USBCANFD-200U 时，索引号为 0，这时再插入一个 USBCANFD-200U，那么后面插入的这个设备索引号就是 1，以此类推。

##### reserved

仅作参考。

#### 返回值

为 INVALID\_DEVICE\_HANDLE 表示操作失败，否则表示操作成功，返回设备句柄值，请保存该句柄值，往后的操作需要使用。

### ZCAN\_CloseDevice

该函数用于关闭设备，关闭设备和打开设备一一对应。

```
UINT ZCAN_CloseDevice(DEVICE_HANDLE device_handle);
```

#### 参数

##### device\_handle

需要关闭的设备的句柄值，即 ZCAN\_OpenDevice 成功返回的值。

#### 返回值

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

### ZCAN\_GetDeviceInf

该函数用于获取设备信息。

```
UINT ZCAN_GetDeviceInf(DEVICE_HANDLE device_handle, ZCAN_DEVICE_INFO* pInfo);
```

#### 参数

##### **device\_handle**

设备句柄值。

##### **pInfo**

设备信息结构体，详见 ZCAN\_DEVICE\_INFO 结构说明。

#### 返回值

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

### **ZCAN\_IsDeviceOnLine**

该函数用于检测设备是否在线，仅支持 USB 系列设备。

```
UINT ZCAN_IsDeviceOnLine(DEVICE_HANDLE device_handle);
```

#### 参数

##### **device\_handle**

设备句柄值。

#### 返回值

设备在线=STATUS\_ONLINE，不在线=STATUS\_OFFLINE。

### **ZCAN\_InitCAN**

该函数用于初始化 CAN。

```
CHANNEL_HANDLE ZCAN_InitCAN(DEVICE_HANDLE device_handle, UINT can_index,  
ZCAN_CHANNEL_INIT_CONFIG* pInitConfig);
```

#### 参数

##### **device\_handle**

设备句柄值。

##### **can\_index**

通道索引号，通道 0 的索引号为 0，通道 1 的索引号为 1，以此类推。

##### **pInitConfig**

初始化结构，详见 ZCAN\_CHANNEL\_INIT\_CONFIG 结构说明。

#### 返回值

为 INVALID\_CHANNEL\_HANDLE 表示操作失败，否则表示操作成功，返回通道句柄值，请保存该句柄值，往后的操作需要使用。

### **ZCAN\_StartCAN**

该函数用于启动 CAN 通道。

```
UINT ZCAN_StartCAN(CHANNEL_HANDLE channel_handle);
```

#### 参数

##### **channel\_handle**

通道句柄值。

## 返回值

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

## ZCAN\_ResetCAN

该函数用于复位 CAN 通道，可通过 ZCAN\_StartCAN 恢复。

```
UINT ZCAN_ResetCAN(CHANNEL_HANDLE channel_handle);
```

## 参数

**channel\_handle**

通道句柄值。

## 返回值

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

## ZCAN\_ClearBuffer

该函数用于清除库接收缓冲区。

```
UINT ZCAN_ClearBuffer(CHANNEL_HANDLE channel_handle);
```

## 参数

**channel\_handle**

通道句柄值。

## 返回值

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

## ZCAN\_ReadChannelErrInfo

该函数用于读取通道的错误信息。

```
UINT ZCAN_ReadChannelErrInfo(CHANNEL_HANDLE channel_handle, ZCAN_CHANNEL_ERROR_INFO* pErrInfo);
```

## 参数

**channel\_handle**

通道句柄值。

**pErrInfo**

错误信息结构，详见 ZCAN\_CHANNEL\_ERROR\_INFO 结构说明。

## 返回值

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

## ZCAN\_ReadChannelStatus

该函数用于读取通道的状态信息。目前暂时没有设备支持使用此接口获取通道的状态信息，后续接口可能会被废弃。

```
UINT ZCAN_ReadChannelStatus(CHANNEL_HANDLE channel_handle, ZCAN_CHANNEL_STATUS* pCANStatus);
```

## 参数

**channel\_handle**



通道句柄值。

**pCANStatus**

状态信息结构，详见 ZCAN\_CHANNEL\_STATUS 结构说明。

**返回值**

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

**ZCAN\_Transmit**

该函数用于发送 CAN 报文。

```
UINT ZCAN_Transmit(CHANNEL_HANDLE channel_handle, ZCAN_Transmit_Data* pTransmit, UINT len);
```

**参数**

**channel\_handle**

通道句柄值。

**pTransmit**

结构体 ZCAN\_Transmit\_Data 数组的首指针。

**len**

报文数目

**返回值**

返回实际发送成功的报文数目。

**ZCAN\_TransmitFD**

该函数用于发送 CANFD 报文。

```
UINT ZCAN_TransmitFD(CHANNEL_HANDLE channel_handle, ZCAN_TransmitFD_Data* pTransmit, UINT len);
```

**参数**

**channel\_handle**

通道句柄值。

**pTransmit**

结构体 ZCAN\_TransmitFD\_Data 数组的首指针。

**len**

报文数目

**返回值**

返回实际发送成功的报文数目。

**ZCAN\_TransmitData**

该函数用于发送 CAN/CANFD 报文。

```
UINT ZCAN_TransmitData(DEVICE_HANDLE device_handle, ZCANDataObj* pTransmit, UINT len);
```

**参数**

**device\_handle**

设备句柄值。

### **pTransmit**

结构体 ZCANDataObj 数组的首指针。

### **len**

报文数目

### **返回值**

返回实际发送成功的报文数目。

注：只有支持合并接收的设备才可以使用此接口发送数据，支持合并接收功能的设备调用发送接口发送数据时，并不要求设备开启合并接收功能。支持合并接收的设备列表如表 4.2 所示。

## **ZCAN\_GetReceiveNum**

获取缓冲区中 CAN，CANFD 或者合并接收报文数目。

```
UINT ZCAN_GetReceiveNum(CHANNEL_HANDLE channel_handle, BYTE type);
```

### **参数**

#### **channel\_handle**

通道句柄值。

#### **type**

获取 CAN，CANFD 或者合并接收报文，0=CAN，1=CANFD，2=合并接收。设备支持合并接收功能并开启合并接收时，使用 type 参数 2 可以获取当前合并接收数据的总量。未开启合并接收时，channel\_handle 使用要获取数据的通道句柄，type 表示要获取的数据类型(CAN 或 CANFD)，调用函数可以获取缓存中指定通道指定类型的数据帧数量。支持合并接收的设备开启合并接收时，channel\_handle 可使用任意通道的句柄，type 使用值 2 表示获取合并接收帧数量。合并接收的数据可以通过调用接口 ZCAN\_ReceiveData 获取。支持合并接收的设备列表如表 4.2 所示。

### **返回值**

返回报文数目。

## **ZCAN\_Receive**

该函数用于接收 CAN 报文，建议使用 ZCAN\_GetReceiveNum 确保缓冲区有数据再使用。

```
UINT ZCAN_Receive(CHANNEL_HANDLE channel_handle, ZCAN_Receive_Data* pReceive, UINT len, INT wait_time = -1);
```

### **参数**

#### **channel\_handle**

通道句柄值。

#### **pReceive**

结构体 ZCAN\_Receive\_Data 数组的首指针。

#### **len**

数组长度（本次接收的最大报文数目，实际返回值小于等于这个值）。

#### **wait\_time**

缓冲区无数据，函数阻塞等待时间，单位毫秒。若为-1 则表示无超时，一直等待，默认为-1。

## 返回值

返回实际接收的报文数目。

## ZCAN\_ReceiveFD

该函数用于接收 CANFD 数据，建议使用 ZCAN\_GetReceiveNum 确保缓冲区有数据再使用。

```
UINT ZCAN_ReceiveFD(CHANNEL_HANDLE channel_handle, ZCAN_ReceiveFD_Data* pReceive, UINT len, INT wait_time = -1);
```

## 参数

### channel\_handle

通道句柄值。

### pReceive

结构体 ZCAN\_ReceiveFD\_Data 数组的首指针。

### len

数组长度（本次接收的最大报文数目，实际返回值小于等于这个值）。

### wait\_time

缓冲区无数据，函数阻塞等待时间，单位毫秒。若为-1 则表示无超时，一直等待，默认值为-1。

## 返回值

返回实际接收的报文数目。

## ZCAN\_ReceiveData

该函数用于接收 CAN、CANFD、LIN、GPS、错误数据等各种类型的数据，即合并接收功能。建议使用 ZCAN\_GetReceiveNum 接口使用合并接收参数获取缓冲区合并接收帧数量后确保缓冲区有数据再调用接口获取数据。

```
UINT ZCAN_ReceiveData (DEVICE_HANDLE device_handle, ZCANDataObj* pReceive, UINT len, INT wait_time = -1);
```

## 参数

### device\_handle

设备句柄值。

### pReceive

结构体 ZCANDataObj 数组的首指针。

### len

数组长度（本次接收的最大报文数目，实际返回值小于等于这个值）。

### wait\_time

缓冲区无数据，函数阻塞等待时间，单位毫秒。若为-1 则表示无超时，一直等待，默认值为-1。

## 返回值

返回实际接收的报文数目。

注：ZCAN\_ReceiveData 接口只有在设备支持合并接收功能并开启合并接收功能后才可以正常的接收到各种数据，设备不支持合并接收功能或者设备支持合并接收但是未开启合并接收时，请使用 ZCAN\_Receive/ZCAN\_ReceiveFD 等接口获取设备数据。支持合并接收的设备列表如表 4.2 所示。

## ZCAN\_SetValue

该函数设置设备属性。使用方法可以参考 IProperty 属性的 SetValue 函数，详见 3.3 小节。

```
UINT ZCAN_SetValue(DEVICE_HANDLE device_handle, const char* path, const void* value);
```

### 参数

#### **device\_handle**

设备句柄值。

#### **path**

设备属性路径。

#### **value**

要设置的属性值。

### 返回值

STATUS\_OK 表示设置成功，STATUS\_ERR 表示设置失败。

注：ZCAN\_SetValue 函数可以通过设备句柄，直接设置设备属性。旧接口要达到同样的目的，要先通过 GetIProperty 接口获取设备属性 IProperty，后通过 IProperty 的成员 SetValue 函数设置设备属性，属性设置完后还需要使用 ReleaseIProperty 释放对应的 IProperty。ZCAN\_SetValue 可以简化属性设置的步骤，方便用户使用。

## ZCAN\_GetValue

该函数获取设备属性。用方法可以参考 IProperty 属性的 GetValue 函数，详见 3.3 小节。

```
const void* ZCAN_GetValue (DEVICE_HANDLE device_handle, const char* path);
```

### 参数

#### **device\_handle**

设备句柄值。

#### **path**

设备属性路径。

### 返回值

返回的指针值非空表示操作成功，为空(NULL)则表示操作失败。

注：ZCAN\_GetValue 函数可以通过设备句柄，直接获取设备属性。旧接口要达到同样的目的，要先通过 GetIProperty 接口获取设备属性 IProperty，后通过 IProperty 的成员 GetValue 函数获取设备属性，属性获取完后还需要使用 ReleaseIProperty 释放对应的 IProperty。ZCAN\_GetValue 可以简化属性获取的步骤，方便用户使用。

## GetIProperty

该函数返回属性配置接口。

```
IProperty* GetIProperty(DEVICE_HANDLE device_handle);
```

### 参数

**device\_handle**

设备句柄值。

## 返回值

返回属性配置接口指针，详见 **IProperty** 结构说明，空则表示操作失败。

## ReleaseIProperty

释放属性接口，与 **GetIProperty** 结对使用。

```
UINT ReleaseIProperty(IProperty * pIProperty);
```

## 参数

**pIProperty**

**GetIProperty** 的返回值。

## 返回值

**STATUS\_OK** 表示操作成功，**STATUS\_ERR** 表示操作失败。

## ZCLOUD\_SetServerInfo

该函数用于设置云服务器相关连接信息。

```
void ZCLOUD_SetServerInfo(const char* httpSvr, unsigned short httpPort, const char* mqttSvr, unsigned short mqttPort);
```

## 参数

**httpSvr**

用户认证服务器地址，IP 地址或域名。

**httpPort**

用户认证服务器端口号。

**mqttSvr**

数据服务器地址，IP 地址或域名，一般与认证服务器相同。

**mqttPort**

数据服务器端口号。

## ZCLOUD\_ConnectServer

该函数用于连接云服务器，会先登录认证服务器，然后连接到数据服务器。

```
UINT ZCLOUD_ConnectServer(const char* username, const char* password);
```

## 参数

**username**

用户名。

**password**

密码。

## 返回值

0: 成功，1: 失败， 2: 认证服务器连接错误，3: 用户信息验证错误，4: 数据服务器连接错误。

## ZCLOUD\_IsConnected

该函数用于判断是否已经连接到云服务器。

```
bool ZCLOUD_IsConnected();
```

### 返回值

true: 已连接, false: 未连接。

## ZCLOUD\_DisconnectServer

该函数用于断开云服务器连接。

```
UINT ZCLOUD_DisconnectServer()
```

### 返回值

0: 成功, 1: 失败。

## ZCLOUD\_GetUserData

获取用户数据, 包括用户基本信息和所拥有设备信息。

```
const ZCLOUD_USER_DATA* ZCLOUD_GetUserData();
```

### 返回值

用户数据结构指针。

## ZCLOUD\_ReceiveGPS

该函数用于接收云设备 GPS 数据。

```
UINT ZCLOUD_ReceiveGPS(DEVICE_HANDLE device_handle, ZCLOUD_GPS_FRAME* pReceive, UINT len, int wait_time DEF(-1));
```

### 参数

#### **device\_handle**

设备句柄值。

#### **pReceive**

结构体 ZCLOUD\_GPS\_FRAME 数组的首指针。

#### **len**

数组长度（本次接收的最大报文数目，实际返回值小于等于这个值）。

#### **wait\_time**

缓冲区无数据，函数阻塞等待时间，单位毫秒，若为-1 则表示无超时，一直等待，默认值为-1。

### 返回值

返回实际接收的报文数目。

## ZCAN\_InitLIN

该函数用于对 LIN 进行初始化，指定设备工作模式，采用经典校验方式还是增强校验等参数，如果是主站模式，需要指定 LIN 工作的波特率。

```
CHANNEL_HANDLE FUNC_CALL ZCAN_InitLIN(DEVICE_HANDLE device_handle, UINT can_index, PZCAN_LIN_INIT_CONFIG pLINInitConfig);
```

## 参数

**device\_handle**

设备句柄值。

**can\_index**

通道索引号，通道 0 的索引号为 0，通道 1 的索引号为 1，以此类推。

**pLINInitConfig**

初始化结构，详见 ZCAN\_LIN\_INIT\_CONFIG 结构说明。

## 返回值

为 INVALID\_CHANNEL\_HANDLE 表示操作失败，否则表示操作成功，返回通道句柄值，请保存该句柄值，往后的操作需要使用。

## ZCAN\_StartLIN

该函数用于启动 LIN 通道。

```
UINT FUNC_CALL ZCAN_StartLIN(CHANNEL_HANDLE channel_handle);
```

## 参数

**channel\_handle**

通道句柄值。

## 返回值

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

## ZCAN\_ResetLIN

该函数用于复位对应的 LIN 通道，即停止此通道的数据发送和接收。复位之后如果需要继续接收或者发送数据，需要重新调用 VCI\_StartLIN 来启动 LIN 通道。

```
UINT FUNC_CALL ZCAN_ResetLIN(CHANNEL_HANDLE channel_handle);
```

## 参数

**channel\_handle**

通道句柄值。

## 返回值

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

## ZCAN\_TransmitLIN

该函数用来控制 LIN 发送 LIN 消息，只有 LIN 处于主站模式下才可以使用此函数进行数据发送。

```
ULONG FUNC_CALL ZCAN_TransmitLIN(CHANNEL_HANDLE channel_handle, PZCAN_LIN_MSG pSend, ULONG Len);
```

## 参数

**channel\_handle**

通道句柄值。

**pSend**

结构体 ZCAN\_LIN\_MSG 数组的首指针。

**Len**

报文数目

**返回值**

返回实际发送成功的报文数目。

**ZCAN\_GetLINReceiveNum**

该函数用于获取指定通道已经接收到的 LIN 消息数量。

```
ULONG FUNC_CALL ZCAN_GetLINReceiveNum(CHANNEL_HANDLE channel_handle);
```

**参数****channel\_handle**

通道句柄值。

**返回值**

返回报文数目。

**ZCAN\_ReceiveLIN**

该函数用来接收 LIN 消息，不论 LIN 处于主站还是从站模式，都可以使用该函数获取总线上的数据信息。

```
ULONG FUNC_CALL ZCAN_ReceiveLIN(CHANNEL_HANDLE channel_handle, PZCAN_LIN_MSG  
pReceive, ULONG Len,int WaitTime);
```

**参数****channel\_handle**

通道句柄值。

**pReceive**

结构体 ZCAN\_LIN\_MSG 数组的首指针。

**Len**

数组长度（本次接收的最大报文数目，实际返回值小于等于这个值）。

**wait\_time**

缓冲区无数据，函数阻塞等待时间，单位毫秒。若为-1 则表示无超时，一直等待，默认值为-1。

**返回值**

返回实际接收的报文数目。

**ZCAN\_SetLINSlaveMsg**

该函数用来设置 LIN 作为从站时候的响应信息，设置响应信息后，从站收到对应 ID 的请求时候会将预定义的数据发送出去作为响应。

```
UINT FUNC_CALL ZCAN_SetLINSlaveMsg(CHANNEL_HANDLE channel_handle, PZCAN_LIN_MSG  
pSend, UINT nMsgCount);
```

**参数****channel\_handle**

通道句柄值。



#### **pSend**

结构体 ZCAN\_LIN\_MSG 数组的首指针。

#### **nMsgCount**

数组长度。

#### **返回值**

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

### **ZCAN\_ClearLINSlaveMsg**

该函数用来清除 LIN 作为从站时候的响应信息，设置响应信息后，从站收到对应 ID 的请求时候会将预定义的数据发送出去作为响应，如果需要控制从站不再响应对应的 ID，需要调用此函数清除对特定 ID 的响应信息，清除后，从站不会在对此 ID 进行响应。

```
UINT FUNC_CALL ZCAN_ClearLINSlaveMsg(CHANNEL_HANDLE channel_handle, BYTE* pLINID, UINT nIDCount);
```

#### **参数**

##### **channel\_handle**

通道句柄值。

##### **pLINID**

LIN ID 数组的首指针。

##### **nIDCount**

数组长度。

#### **返回值**

STATUS\_OK 表示操作成功，STATUS\_ERR 表示操作失败。

## **3.3 设备功能和属性表**

本节列出了 CAN 接口卡的属性配置项，即 IProperty 的 SetValue 或 GetValue 的 path、value 配置项，函数原型如下：

```
/**
 * \brief 设置指定路径的属性的值。
 * \param[in] path : 属性的路径。
 * \param[in] value : 属性的值。
 * \retval 成功返回 1，失败返回 0。
 */
typedef int (*SetValueFunc)(const char* path, const char* value);

/**
 * \brief 获取指定路径的属性的值。
 * \param[in] path : 属性的路径。
 * \retval 成功返回属性的值，失败返回 NULL。
 */
typedef const char* (*GetValueFunc)(const char* path);
```

```
typedef struct tagIProperty
{
    SetValueFunc    SetValue;
    GetValueFunc    GetValue;
    GetPropertysFunc GetPropertys;
}IProperty;
```

### 3.3.1 USBCANFD 系列

本系列适用的设备：USBCANFD-100U、USBCANFD-200U、USBCANFD-MINI。

#### 1. CANFD 标准

CANFD 控制器标准类型，ISO 或非 ISO，通常使用 ISO 标准。

项	值	说明
path	n/canfd_standard	n 代表通道号，如 0 代表通道 0，1 代表通道 1，下同
value	0 – CANFD ISO 1 – CANFD Non-ISO	默认 0
Get/Set	Set	可用于 GetValue 或 SetValue，下同
注意点	需在 ZCAN_InitCAN 之前设置	

#### 2. 仲裁域波特率

CANFD 控制器仲裁域波特率，支持常用标准波特率，若列表中无所需值可使用自定义波特率。CANFD 控制器可接收 CAN 报文，仲裁域波特率对应 CAN 波特率。

项	值	说明
path	n/canfd_abit_baud_rate	n 代表通道号
value	【1000000, 800000, 500000, 250000, 125000, 100000, 50000】	单位 bps，如值为 500000 则表示波特率为 500k
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	

示例代码见数据域波特率示例。

#### 3. 数据域波特率

CANFD 控制器数据域波特率，支持支持常用标准波特率，若列表中无所需值可使用自定义波特率。

项	值	说明
path	n/canfd_dbit_baud_rate	n 代表通道号
value	【5000000, 4000000, 2000000, 1000000, 800000, 500000, 250000, 125000, 100000】	单位 bps，如值为 500000 则表示波特率为 500k

Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	

#### 4. 自定义波特率

设置 CANFD 控制器为任意有效波特率, 设置值可通过 ZCANPRO 的波特率计算器进行计算。如果设置了自定义波特率则无需再设置仲裁域以及数据域波特率。

项	值	说明
path	n/baud_rate_custom	n 代表通道号
value	自定义	通过 ZCANPRO 的波特率计算器计算
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	

#### 5. 终端电阻

USBCANFD 每通道内置 120  $\Omega$  终端电阻, 可通过属性设置选择使能或不使能。

项	值	说明
path	n/initenal_resistance	n 代表通道号
value	0 - 禁能 1 - 使能	
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	

#### 6. 发送超时时间

设备发送 CAN(FD)报文时总线上没有节点回复 ACK 时的超时时间, 单位毫秒, 默认值为 1500ms。

注: 通道进入 BUSOFF 会导致发送立即返回, 不会等待设置的超时时间。

项	值	说明
path	n/tx_timeout	n 代表通道号
value	自定义	“1000”, 单位毫秒, 最大值为 4000
Get/Set	Set	
注意点	需在 ZCAN_OpenDevice 之后设置	

#### 7. 定时发送

USBCANFD 支持每通道最大 100 条定时发送列表, 只需将待发送数据及周期设置到设备并使能, 设备将自动进行发送。相比于 PC 端的发送, 定时发送精度高, 周期准。在设备进行定时发送任务时, PC 端仍可调数据发送接口进行数据发送。列表添加完成并设置 apply\_auto\_send 开启定时发送后, 设备会按列表索引顺序依次启动发送, 如需延时启动, 可通过 auto\_send\_param 属性进行设置。

以下列表列举定时发送的多个相关属性。

设置定时发送 CAN 帧		
项	值	说明
path	n/auto_send	n 代表通道号
value	ZCAN_AUTO_TRANSMIT_OBJ 指针	需将指针强制转换为 char*
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
设置定时发送 CANFD 帧		
项	值	说明
path	n/auto_send_canfd	n 代表通道号
value	ZCANFD_AUTO_TRANSMIT_OBJ 指针	需将指针强制转换为 char*
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
定时发送附加参数（用于定时发送列表间的延时启动）		
项	值	说明
path	n/auto_send_param	n 代表通道号
value	ZCAN_AUTO_TRANSMIT_OBJ_PARAM 指针	需将指针强制转换为 char*
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
清空定时发送		
项	值	说明
path	n/clear_auto_send	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
应用定时发送（使能定时发送属性设置）		
项	值	说明
path	n/apply_auto_send	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	

## 8. 队列发送

通过队列发送，用户可以提前准备好多帧报文，设定报文之间的间隔，将准备好的报文发送给设备，设备按照预定义的帧间隔进行精准发送，通过此方式可提高发送帧之间的帧间隔精度。与定时发送相比，队列发送每帧只发送一次，需由用户不断准备报文并批量发送到设备。

用户需要使用队列发送时，需要将设置设备的发送模式设置为队列发送，当设备处于队列发送模式时，定时发送功能将被禁用。设置设备的发送模式的方法请参考设备属性表中对应的说明。

队列模式下的数据发送使用 ZCAN\_TransmitData 接口，CAN/CANFD 数据使用 ZCANDataObj 结构。USBCANFD 系列设备队列模式下延时时间单位支持毫秒(ms)和 100 微秒(0.1ms)两种时间单位，两种时间单位设备支持的最长延时时间(帧发送间隔时间)分别是 65535ms 和 6553.5ms。使用 100 微秒时间单位，设备的发送精度更高一些，但是支持的延时时间最长只有约 6.5 秒。时间单位可以通过帧标志为中的 txDelay 字段设置。队列模式下，txDelay 字段使用 0 表示直接发送数据到总线，txDelay 设置为 1 表示使用毫秒作为时间单位，txDelay 设置为 2 表示使用 100 微秒作为时间单位。延时时间存放在 timeStamp 字段中。

设备发送当前帧的同时会启动计时器按照当前帧设定的时间进行计时，计时时间结束会从队列取下一帧进行发送并重新开始计时。

队列模式下可以通过接口获取设备端可用的队列空间和清空设备队列发送的缓存，使用方式参考设备属性表。

设置设备发送模式		
项	值	说明
path	n/set_send_mode	n 代表通道号
value	0 – 正常模式 1 – 队列模式	设备默认为 0 正常模式
Get/Set	Set	
获取发送队列可用缓存数量（仅队列模式）		
项	值	说明
path	n/get_device_available_tx_count/1	n 代表通道号，最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 int*
value	无	
Get/Set	Get	
清空发送缓存（仅队列模式，缓存中未发送的帧将被清空，停止时使用）		
项	值	说明
path	n/clear_delay_send_queue	n 代表通道号
value	0	固定值

Get/Set	Set	
---------	-----	--

## 9. 合并接收

USBCANFD 系列设备支持数据合并接收，合并接收是指通过同一个接口 ZCAN\_ReceiveData 可以接收到设备支持的不同通道，不同类型的数据(ZCANDataObj 结构包含通道和数据类型等信息)。针对 USBCANFD 系列设备，通过合并接收可以接收到不同通道的 CAN 和 CANFD 数据。合并接收可以以统一的方式获取各种类型的数据，避免出现由于传统接口按照通道和类型获取 CAN/CANFD 数据导致的获取数据的时间戳可能不连续的问题。

合并接收功能需要通过设备属性表设置功能的开启或关闭，设备默认不开启合并接收功能。为了避免合并接收和普通接收状态切换可能产生数据错乱的问题，如果需要合并接收功能，则需要在启动设备第一个通道前打开合并接收，并且在后续使用过程中不要随意更改合并接收开关。

设备开启合并接收后，只能通过合并接收接口 ZCAN\_ReceiveData 获取设备数据。设备通道启动后，每个已经启动的通道的数据都可封装在 ZCANDataObj 数据结构中示，通过 ZCAN\_ReceiveData 接口接收数据。ZCANDataObj 的 dataType 成员表示数据的具体类型，ZCANDataObj 的 chnl 成员表示数据所在的通道。

合并接收功能开启/关闭		
项	值	说明
path	n/set_device_recv_merge	n 代表通道号
value	0 – 关闭合并接收功能 1 – 开启合并接收功能	设备默认为 0，不开启合并接收
Get/Set	Set	
查询设备当前是否开启了合并接收		
项	值	说明
path	n/get_device_recv_merge/1	n 代表通道号，使用任意合法通道号即可。最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 int*，指向的数值为 0 表示未开启合并接收，为 1 表示已开启合并接收
value	无	
Get/Set	Get	

## 10. 自定义序列号

自定义序列号通常用于多卡同时使用时对卡以及通道进行区分，比如 3 个 USBCAFD-100U 可分别设置序列号为 U-1, U-2, U-3，打开时获取序列号并加以判断。所设置的序列号可掉电保存。

设置自定义序列号		
项	值	说明

path	n/set_cn	n 代表通道号
value	自定义字符串	最多 128 字符
Get/Set	Set	
获取自定义序列号		
项	值	说明
path	n/get_cn/1	后面的 1 必须，也可以是任意数字
value	无	
Get/Set	Get	

## 11. 滤波

USBCANFD 系列每通道最多可设置 64 组滤波，为白名单模式，即如果不设置滤波 CAN 通道将接收所有报文，如果设置了滤波，CAN 通道将只接收滤波范围内的报文。添加一条滤波的标准顺序是：设置滤波模式，设置起始帧，设置结束帧。如果要添加多条就重复上述步骤，添加完滤波并不会立即生效，需设置 filter\_ack 使能所设的滤波表。

设置滤波模式		
项	值	说明
path	n/filter_mode	n 代表通道号
value	0 – 标准帧 1 – 扩展帧	
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
设置滤波起始帧 ID		
项	值	说明
path	n/filter_start	n 代表通道号
value	自定义	“0x00000000”，16 进制字符
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
设置滤波结束帧 ID		
项	值	说明
path	n/filter_end	n 代表通道号
value	自定义	“0x00000000”，16 进制字符
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	

滤波生效（全部滤波 ID 同时生效）		
项	值	说明
path	n/filter_ack	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
清除滤波		
项	值	说明
path	n/filter_clear	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	

## 12. 示例代码

```
// 以下代码以 USBCANFD-200U 设备型号为例
#include "stdafx.h"
#include "zlgcan.h"
#include <iostream>
#include <iomanip>
#include <windows.h>
#include <thread>

#define CH_COUNT    2
bool    g_thd_run = 1;

// 此函数仅用于构造示例 CAN 报文
void get_can_frame(ZCAN_Transmit_Data& can_data, canid_t id)
{
    memset(&can_data, 0, sizeof(can_data));
    can_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0);    // CAN ID + STD/EXT + DATA/RMT
    can_data.frame.can_dlc = 8;                        // CAN 数据长度 8
    can_data.transmit_type = 0;                        // 正常发送
    can_data.frame.__pad |= TX_ECHO_FLAG;              // 发送回显
    for (int i = 0; i < 8; ++i) {                      // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}

// 此函数仅用于构造示例 CANFD 报文
void get_canfd_frame(ZCAN_TransmitFD_Data& canfd_data, canid_t id)
```



```

{
    memset(&canfd_data, 0, sizeof(canfd_data));
    canfd_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0); // CAN ID + STD/EXT + DATA/RMT
    canfd_data.frame.len = 64;                        // CANFD 数据长度 64
    canfd_data.transmit_type = 0;                     // 正常发送
    canfd_data.frame.flags |= TX_ECHO_FLAG;           // 发送回显
    for (int i = 0; i < 64; ++i) {                   // 填充 CANFD 报文 DATA
        canfd_data.frame.data[i] = i;
    }
}

// 此函数仅用于构造示例队列发送报文
void get_can_frame_queue(ZCANDataObj& data, int ch, canid_t id, bool is_fd, UINT delay)
{
    memset(&data, 0, sizeof(data));
    data.dataType = ZCAN_DT_ZCAN_CAN_CANFD_DATA;
    data.chnl = ch;
    ZCANCANFDData & can_data = data.data.zcanCANFDData;
    can_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0); // CAN ID + STD/EXT + DATA/RMT
    can_data.frame.len = is_fd ? 64 : 8;             // 数据长度 8/64
    can_data.flag.unionVal.transmitType = 0;         // 正常发送
    can_data.flag.unionVal.txEchoRequest = 1;        // 设置发送回显
    can_data.flag.unionVal.frameType = is_fd ? 1 : 0; // CAN or CANFD
    can_data.flag.unionVal.txDelay = ZCAN_TX_DELAY_UNIT_MS; // 队列延时单位毫秒
    can_data.timeStamp = delay;                      // 队列延时时间，最大值 65535
    for (int i = 0; i < can_data.frame.len; ++i) {   // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}

// 此函数仅用于构造示例合并发送报文
void get_can_canfd_frame(ZCANDataObj& data, int ch, canid_t id, bool is_fd)
{
    memset(&data, 0, sizeof(data));
    data.dataType = ZCAN_DT_ZCAN_CAN_CANFD_DATA;
    data.chnl = ch;
    ZCANCANFDData & can_data = data.data.zcanCANFDData;
    can_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0); // CAN ID
    can_data.frame.len = is_fd ? 64 : 8;             // CAN 数据长度 8
    can_data.flag.unionVal.transmitType = 0;         // 正常发送
    can_data.flag.unionVal.txEchoRequest = 1;        // 设置发送回显
    can_data.flag.unionVal.frameType = is_fd ? 1 : 0; // CAN or CANFD
    can_data.flag.unionVal.txDelay = ZCAN_TX_DELAY_NO_DELAY; // 直接发送报文到总线
    for (int i = 0; i < can_data.frame.len; ++i) {   // 填充 CAN 报文 DATA

```

```

        can_data.frame.data[i] = i;
    }
}

void thread_task(DEVICE_HANDLE handle)
{
    std::cout << "Thread run, handle:0x" << std::hex << handle << std::endl;
    ZCANDataObj recv_data[100] = {0}; // 定义接收数据缓冲区, 100 仅用于举例, 根据实际情况定义
    while (g_thd_run) {
        int rcount = ZCAN_ReceiveData(handle, recv_data, 100, 1);
        int lcount = rcount;
        while (g_thd_run && lcount > 0) {
            for (int i = 0; i < rcount; ++i, --lcount) {
                if (recv_data[i].dataType != ZCAN_DT_ZCAN_CAN_CANFD_DATA) { //
只处理 CAN 或 CANFD 数据
                    continue;
                }
                std::cout << "CHNL: " << std::dec << (int)recv_data[i].chnl; // 打印通道
                ZCANCANFDData & can_data = recv_data[i].data.zcanCANFDData;
                std::cout << " TIME:" << std::fixed << std::setprecision(6) <<
                    can_data.timeStamp/1000000.0f << "s[" <<
                    std::dec << can_data.timeStamp << "];" // 打印时间戳
                if (can_data.flag.unionVal.txEchoed == 1) {
                    std::cout << " [TX] "; // 发送帧
                }
                else {
                    std::cout << " [RX] "; // 接收帧
                }
                std::cout << "ID: 0x" << std::hex << can_data.frame.can_id; // 打印 ID
                std::cout << " LEN " << std::dec << (int)can_data.frame.len; // 打印长度
                std::cout << " DATA " << std::hex; // 打印数据
                for (int ind = 0; ind < can_data.frame.len; ++ind) {
                    std::cout << std::hex << " " << (int)can_data.frame.data[ind];
                }
                std::cout << std::endl;
            }
        }
        Sleep(10);
    }

    std::cout << "Thread exit" << std::endl;
}

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    UINT dev_type = ZCAN_USBCANFD_200U;
    std::thread thd_handle;
    CHANNEL_HANDLE ch[CH_COUNT] = {};

    // 打开设备
    DEVICE_HANDLE device = ZCAN_OpenDevice(dev_type, 0, 0);
    if (INVALID_DEVICE_HANDLE == device) {
        std::cout << "open device failed!" << std::endl;
        return 0;
    }

    // 获取 IProperty 指针，用于配置参数
    IProperty* prop = GetIProperty(device);
    if (NULL == prop) {
        std::cout << "get property failed" << std::endl;
        ZCAN_CloseDevice(device);
        return 0;
    }

    for (int i = 0; i < CH_COUNT; ++i)
    {
        char path[64] = {};
        // 设置 CANFD 标准为 ISO;
        sprintf_s(path, "%d/canfd_standard", i);
        if (0 == prop->SetValue(path, "0")) {
            std::cout << "set canfd standard failed" << std::endl;
            goto end;
        }
        // 设置仲裁域波特率为 1M
        sprintf_s(path, "%d/canfd_abit_baud_rate", i);
        if (0 == prop->SetValue(path, "1000000")) {
            std::cout << "set abit baud rate failed" << std::endl;
            goto end;
        }
        // 设置通道 0 数据域波特率为 5M
        sprintf_s(path, "%d/canfd_dbit_baud_rate", i);
        if (0 == prop->SetValue(path, "5000000")) {
            std::cout << "set dbit baud rate failed" << std::endl;
            goto end;
        }
        // 设置通道 0 自定义波特率，此处仅演示调用方式
        /*

```

```

        if (0 ==
prop->SetValue("0/ baud_rate_custom", "1.0Mbps(75%),5.0Mbps(75%),(60,0080040D,00800001)")) {
            std::cout << "set custom baud rate failed" << std::endl;
            goto end;
        }
    */

    // 初始化通道
    ZCAN_CHANNEL_INIT_CONFIG config;
    memset(&config, 0, sizeof(config));
    config.can_type = 1;                // 0 - CAN, 1 - CANFD
    config.can.mode = 0;                // 0 - 正常模式, 1 - 只听模式
    ch[i] = ZCAN_InitCAN(device, i, &config);
    if (INVALID_CHANNEL_HANDLE == ch[i]) {
        std::cout << "init channel failed!" << std::endl;
        goto end;
    }

    // 使能通道终端电阻
    sprintf_s(path, "%d/initenal_resistance", i);
    if (0 == prop->SetValue(path, "1")) {
        std::cout << "enable terminal resistance failed" << std::endl;
        goto end;
    }

    // 设置通道发送超时时间为 100ms
    sprintf_s(path, "%d/tx_timeout", i);
    if (0 == prop->SetValue(path, "100")) {
        std::cout << "set send timeout failed" << std::endl;
        goto end;
    }
}

#if 0
// 仅对 0 通道设置滤波
if (0 == i)
{
    // 设置第一组滤波, 只接收 ID 范围在 0x100-0x200 之间的标准帧
    prop->SetValue("0/filter_mode", "0");                // 标准帧
    prop->SetValue("0/filter_start", "0x100");            // 起始 ID
    prop->SetValue("0/filter_end", "0x200");            // 结束 ID
    // 设置第二组滤波, 只接收 ID 范围在 0x1FFFF-0x2FFFF 之间的扩展帧
    prop->SetValue("0/filter_mode", "1");                // 扩展帧
    prop->SetValue("0/filter_start", "0x1FFFF");          // 起始 ID
    prop->SetValue("0/filter_end", "0x2FFFF");          // 结束 ID
    // 使能滤波
    prop->SetValue("0/filter_ack", "0");
}

```

```

        // 清除滤波,此处仅举例, 何时调用用户自由决定
        //prop->SetValue("0/filter_clear", "0");
    }
#endif

    // 设置合并接收标志, 启用合并发送, 接收接口 (只需设置 1 次)
    if (0 == i) {
        prop->SetValue("0/set_device_recv_merge", "1");
    }

    // 启动 CAN 通道
    if (0 == ZCAN_StartCAN(ch[i])) {
        std::cout << "start channel 0 failed" << std::endl;
        goto end;
    }
}

// 启动 CAN 通道的接收线程
thd_handle = std::thread(thread_task, device);

// 设置通道 0 自定义序列号为 abc
prop->SetValue("0/set_cn", "abc");
const char* pRet = prop->GetValue("0/get_cn/1");
std::cout << "sn: " << pRet << std::endl;
#endif

/*
   下列代码构造两条定时发送 CAN 报文以及两条定时发送 CANFD 报文,
   索引 0 的 CAN 报文周期 100ms 发送一次,
   索引 1 的 CAN 报文周期 200ms 发送一次, 并且延时 1s 启动,
   索引 2 的 CANFD 报文周期 500ms 发送一次,
   索引 3 的 CANFD 报文周期 600ms 发送一次,
   发送 5s 后停止发送
*/

ZCAN_AUTO_TRANSMIT_OBJ auto_can;
ZCANFD_AUTO_TRANSMIT_OBJ auto_canfd;
ZCAN_AUTO_TRANSMIT_OBJ_PARAM delay_param;
memset(&auto_can, 0, sizeof(auto_can));
auto_can.index = 0; // 定时列表索引 0
auto_can.enable = 1; // 使能此索引, 每条可单独设置
auto_can.interval = 100; // 定时发送间隔 100ms
get_can_frame(auto_can.obj, 0); // 构造 CAN 报文
prop->SetValue("0/auto_send", (const char*)&auto_can); // 设置定时发送
memset(&auto_can, 0, sizeof(auto_can));
auto_can.index = 1; // 定时列表索引 1
auto_can.enable = 1; // 使能此索引, 每条可单独设置

```

```

auto_can.interval = 200; // 定时发送间隔 200ms
get_can_frame(auto_can.obj, 1); // 构造 CAN 报文
prop->SetValue("0/auto_send", (const char*)&auto_can); // 设置定时发送
memset(&auto_canfd, 0, sizeof(auto_canfd));
auto_canfd.index = 2; // 定时列表索引 2
auto_canfd.enable = 1; // 使能此索引，每条可单独设置
auto_canfd.interval = 500; // 定时发送间隔 500ms
get_canfd_frame(auto_canfd.obj, 2); // 构造 CANFD 报文
prop->SetValue("0/auto_send_canfd", (const char*)&auto_canfd); // 设置定时发送
memset(&auto_canfd, 0, sizeof(auto_canfd));
auto_canfd.index = 3; // 定时列表索引 3
auto_canfd.enable = 1; // 使能此索引，每条可单独设置
auto_canfd.interval = 600; // 定时发送间隔 600ms
get_canfd_frame(auto_canfd.obj, 3); // 构造 CANFD 报文
prop->SetValue("0/auto_send_canfd", (const char*)&auto_canfd); // 设置定时发送
// 设置索引 1 的 CAN 帧延时 1s 启动发送
delay_param.index = 1; // 索引 1
delay_param.type = 1; // type 为 1 表示延时启动
delay_param.value = 1000; // 延时 1000ms
prop->SetValue("0/auto_send_param", (const char*)&delay_param); // 设置发送延时
prop->SetValue("0/apply_auto_send", "0"); // 使能定时发送
Sleep(5000); // 等待发送 5s
prop->SetValue("0/clear_auto_send", "0"); // 清除定时发送
system("pause");
#endif

#if 1
// 设置队列发送模式
prop->SetValue("0/set_send_mode", "1");
// 获取队列发送可用缓冲
int free_count = *((int*)prop->GetValue("0/get_device_available_tx_count/1"));
// 构造 50 条报文，报文 ID 从 0 递增，帧间隔 10ms 递增
if (free_count > 50) {
    ZCANDataObj tran_data[50] = { };
    for (int i = 0; i < 50; ++i) {
        get_can_frame_queue(tran_data[i], 0, i, i%2 ? true : false, i*10);
    }
    int ret_count = ZCAN_TransmitData(device, tran_data, 50);
}
// 5 秒后清空队列发送
Sleep(5000);
prop->SetValue("0/clear_delay_send_queue", "0");
system("pause");
#endif

```

```

// 构造 10 帧 CAN 报文(0 通道发送)以及 10 帧 CANFD 报文（1 通道发送）
ZCANDataObj trans_data[20] = {};
for (int i = 0; i < 20; ++i)
{
    int ch = i < 10 ? 0 : 1;
    bool is_fd = i < 10 ? false : true;
    get_can_canfd_frame(trans_data[i], ch, i + 0x100, is_fd);
}
int send_count = ZCAN_TransmitData(device, trans_data, 20);
std::cout << "send frame: " << std::dec << send_count << std::endl;

Sleep(500);
system("pause");

end:
ReleaseIProperty(prop);
g_thd_run = 0;
if (thd_handle.joinable())
    thd_handle.join();
std::cout << "Thread exited, close device" << std::endl;
if (INVALID_DEVICE_HANDLE != device)
    ZCAN_CloseDevice(device);
system("pause");
return 0;
}

```

### 3.3.2 PCIECANFD 系列

本系列的属性表如[错误!未找到引用源。](#)所示，适用的设备：PCIE-CANFD-100U、PCIE-CANFD-200U、PCIE-CANFD-400U。

#### 1. 仲裁域波特率

CANFD 控制器仲裁域波特率，支持常用标准波特率，若列表中无所需值可使用自定义波特率。CANFD 控制器可接收 CAN 报文，仲裁域波特率对应 CAN 波特率。

项	值	说明
path	n/canfd_abit_baud_rate	n 代表通道号
value	【1000000, 800000, 500000, 250000, 125000】	单位 bps，如值为 500000 则表示波特率为 500k
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	

#### 2. 数据域波特率

CANFD 控制器数据域波特率，支持支持常用标准波特率，若列表中无所需值可使用自

定义波特率。

项	值	说明
path	n/canfd_dbit_baud_rate	n 代表通道号
value	【8000000， 5000000， 4000000， 2000000， 1000000】	单位 bps，如值为 500000 则表示波特率为 500k
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	

### 3. 自定义波特率

设置 CANFD 控制器为任意有效波特率，设置值可通过 ZCANPRO 的波特率计算器进行计算。如果设置了自定义波特率则无需再设置仲裁域以及数据域波特率。

项	值	说明
path	n/ baud_rate_custom	n 代表通道号
value	自定义	通过 ZCANPRO 的波特率计算器计算
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	

### 4. 发送类型

设备发送 CAN(FD)报文时的类型，支持正常发送和自发自收。

项	值	说明
path	n/send_type	n 代表通道号
value	【0， 1】	0 表示正常发送；1 表示自发自收
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	

### 5. 发送超时时间

设备发送 CAN(FD)报文时的超时时间，单位毫秒，默认值为 2000ms。

注：设置发送超时时间需要二次开发库(zlgcan)，设备 FPGA 固件以及驱动程序的支持，具体版本要求如下：

FPGA 版本：V1.12 及以上版本

zlgcan 二次开发库版本：V1.0.0.20 及以上版本

驱动程序版本：V1.0.0.8\_20201109 及以上版本

项	值	说明
path	n/tx_timeout	n 代表通道号
value	自定义	“1000”，单位毫秒，最大值为 60000
Get/Set	Set	



注意点	需在 ZCAN_OpenDevice 之后设置
-----	-------------------------

## 6. 发送重试次数

设备发送 CAN(FD)报文时的发送重试次数，取值范围 0-255。

项	值	说明
path	n/retry	n 代表通道号
value	<b>【0-255】</b>	0-254 表示发送失败后的重试次数，255 表示一直重试，直到发送超时
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	

## 7. 定时发送

PCIECANFD 支持定时发送，只需将待发送数据及周期设置到设备并使能，设备将自动按照设置的帧信息和间隔时间进行发送。相比于 PC 端的发送，定时发送精度高，周期准。在设备进行定时发送任务时，PC 端仍可调用数据发送接口进行数据发送。添加完成设备会立即开启定时发送。可以使用 clear\_auto\_send 取消所有的定时发送数据。

以下列表列举定时发送的多个相关属性。

设置定时发送 CAN 帧		
项	值	说明
path	n/auto_send	n 代表通道号
value	ZCAN_AUTO_TRANSMIT_OBJ 指针	需将指针强制转换为 char*
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
设置定时发送 CANFD 帧		
项	值	说明
path	n/auto_send_canfd	n 代表通道号
value	ZCANFD_AUTO_TRANSMIT_OBJ 指针	需将指针强制转换为 char*
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
清空定时发送		
项	值	说明
path	n/clear_auto_send	n 代表通道号
value	0	固定值
Get/Set	Set	

注意点	在 ZCAN_StartCAN 之后设置
-----	----------------------

## 8. 队列发送

通过队列发送，用户可以提前准备好多帧报文，以及报文之间的间隔，将准备好的报文批量发送给设备，设备按照预定义的帧间隔进行精准发送，通过此方式可提高发送帧之间的帧间隔精度。区别与定时发送，队列发送每帧只发送一次，需由用户不断准备报文并批量发送到设备。

首先用户需设置设备发送模式为队列发送，当设备处于队列发送模式时，定时发送功能将被禁用。

队列模式下的数据发送使用 ZCAN\_Transmit/ZCAN\_TransmitFD 接口，返回值表示有多少帧已经加入到设备的发送队列中。

队列模式下可以通过接口获取设备端可用的队列空间。

队列模式下帧间隔单位 ms，长度 2 字节需要分别填入 can/canfd 帧中的 \_\_res0(帧间隔低 8 位)和 \_\_res1(帧间隔高 8 位)字段中，设备支持最大帧间隔时间为 65535ms，同时需要设置延时发送标志位(TX\_DELAY\_SEND\_FLAG)为 1 标识使用队列发送。

队列模式下，CAN 帧的 TX\_DELAY\_SEND\_FLAG 标志位在 frame.\_\_pad 字段的 Bit7 位，CANFD 帧的 TX\_DELAY\_SEND\_FLAG 标志位在 frame.flags 字段的 Bit7 位。标志位为 1 表示使用队列顺序发送数据。标志为 0 表示直接发送到总线。

队列模式下，单次 ZCAN\_Transmit/ZCAN\_TransmitFD 函数调用时，发送多帧数据会按照第 1 帧的 TX\_DELAY\_SEND\_FLAG 位决定此次调用采用直接发送到总线或者使用队列模式进行发送。

设备发送当前帧的同时会启动计时器按照当前帧设定的时间进行计时，计时时间结束会从队列取下一帧进行发送并重新开始计时

注：设置队列发送需要二次开发库(zlgcan)，设备 FPGA 固件以及驱动程序的支持，具体版本要求如下：

FPGA 版本：V1.12 及以上版本

zlgcan 二次开发库版本：V1.0.0.20 及以上版本

驱动程序版本：V1.0.0.8\_20201109 及以上版本

设置设备发送模式		
项	值	说明
path	n/set_send_mode	n 代表通道号
value	0 – 正常模式 1 – 队列模式	设备上电默认 0
Get/Set	Set	
获取设备当前的发送模式		
项	值	说明
Path	n/get_send_mode/1	n 代表通道号，最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 int*，表示设备当前的模式。0：正常模

		式；1：队列模式。
Value	无	设备上电默认 0
Get/Set	Get	
获取发送队列可用缓存数量（仅队列模式）		
项	值	说明
path	n/get_device_available_tx_count/1	n 代表通道号，最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 int*
value	无	
Get/Set	Get	
清空发送缓存（仅队列模式，缓存中未发送的帧将被清空，停止时使用）		
项	值	说明
path	n/clear_delay_send_queue	n 代表通道号
value	0	固定值
Get/Set	Set	

## 9. 滤波

PCIECANFD 系列每通道最多可设置 32 组滤波，为白名单模式，即如果不设置滤波 CAN 通道将接收所有报文，如果设置了滤波，CAN 通道将只接收滤波范围内的报文。添加一条滤波的标准顺序是：设置滤波模式，设置起始帧，设置结束帧。如果要添加多条就重复上述步骤，添加完滤波并不会立即生效，需设置 filter\_ack 使能所设的滤波表。

注：设置滤波需要二次开发库(zlgcan)，设备 FPGA 固件以及驱动程序的支持，具体版本要求如下：

FPGA 版本：V1.12 及以上版本

zlgcan 二次开发库版本：V1.0.0.20 及以上版本

驱动程序版本：V1.0.0.8\_20201109 及以上版本

设置滤波模式		
项	值	说明
path	n/filter_mode	n 代表通道号
value	0 – 标准帧 1 – 扩展帧	
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
设置滤波起始帧 ID		
项	值	说明
path	n/filter_start	n 代表通道号

value	自定义	“0x00000000”，16 进制字符
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
设置滤波结束帧 ID		
项	值	说明
path	n/filter_end	n 代表通道号
value	自定义	“0x00000000”，16 进制字符
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
滤波生效（全部滤波 ID 同时生效）		
项	值	说明
path	n/filter_ack	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
清除滤波		
项	值	说明
path	n/filter_clear	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	

## 10. 示例代码

```
// 以下代码以 PCIECANFD-200U 设备型号为例
#include "stdafx.h"
#include "zlgcan.h"
#include <iostream>
#include <windows.h>
#include <thread>

// 此函数仅用于构造示例 CAN 报文
void get_can_frame(ZCAN_Transmit_Data& can_data, canid_t id)
{
    memset(&can_data, 0, sizeof(can_data));
    can_data.frame.can_id = MAKE_CAN_ID(id,0,0,0); // CAN ID
    can_data.frame.can_dlc = 8;                  // CAN 数据长度 8
}
```

```

    can_data.transmit_type = 0; // 正常发送
    for (int i = 0; i < 8; ++i) { // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}
// 此函数仅用于构造示例 CANFD 报文
void get_canfd_frame(ZCAN_TransmitFD_Data& canfd_data, canid_t id)
{
    memset(&canfd_data, 0, sizeof(canfd_data));
    canfd_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0); // CANFD ID
    canfd_data.frame.len = 64; // CANFD 数据长度 64
    canfd_data.transmit_type = 0; // 正常发送
    canfd_data.frame.flags |= CANFD_BRS; // CANFD 加速
    for (int i = 0; i < 64; ++i) { // 填充 CANFD 报文 DATA
        canfd_data.frame.data[i] = i;
    }
}
// 此函数仅用于构造示例队列发送 CAN 报文
void get_can_frame_queue(ZCAN_Transmit_Data& can_data, canid_t id, UINT delay)
{
    memset(&can_data, 0, sizeof(can_data));
    can_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0); // CAN ID
    can_data.frame.can_dlc = 8; // CAN 数据长度 8
    can_data.frame.__pad |= TX_DELAY_SEND_FLAG; // 设置延时标志位
    can_data.frame.__res0 = LOBYTE(delay); // 帧间隔低字节
    can_data.frame.__res1 = HIBYTE(delay); // 帧间隔高字节
    can_data.transmit_type = 0; // 正常发送
    for (int i = 0; i < 8; ++i) { // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}
bool g_thd_run = 1; // 线程运行标记
// 线程函数，用于接收报文
void thread_task(CHANNEL_HANDLE handle)
{
    int nChnl = (unsigned int)handle & 0x000000FF;
    std::cout << "chnl: " << std::dec << nChnl << " thread run, handle:0x" << std::hex << handle << std::endl;
    ZCAN_Receive_Data data[100] = {};
    ZCAN_ReceiveFD_Data fd_data[100] = {};
    while (g_thd_run)
    {
        int count = ZCAN_GetReceiveNum(handle, 0); // 获取 CAN 报文（参数 2： 0 - CAN， 1 - CANFD）
        数量
        while (g_thd_run && count > 0)

```

```

    {
        int rcount = ZCAN_Receive(handle, data, 100, 10);
        for (int i = 0; i < rcount; ++i)
        {
            std::cout << "CHNL: " << std::dec << nChnl << " recv can ID: 0x" << std::hex <<
data[i].frame.can_id << std::endl;
        }
        count -= rcount;
    }
    count = ZCAN_GetReceiveNum(handle, 1);    // 获取 CANFD 报文 (参数 2: 0 - CAN, 1 - CANFD)
数量
    while (g_thd_run && count > 0)
    {
        int rcount = ZCAN_ReceiveFD(handle, fd_data, 100, 10);
        for (int i = 0; i < rcount; ++i)
        {
            std::cout << "CHNL: " << std::dec << nChnl << " recv canfd ID: 0x" << std::hex <<
fd_data[i].frame.can_id << std::endl;
        }
        count -= rcount;
    }
    Sleep(100);
}
std::cout << "chnl: " << std::dec << nChnl << " thread exit" << std::endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    UINT dev_type = ZCAN_PCIE_CANFD_200U;
    std::thread thd_handle;

    // 打开设备
    DEVICE_HANDLE device = ZCAN_OpenDevice(dev_type, 0, 0);
    if (INVALID_DEVICE_HANDLE == device) {
        std::cout << "open device failed!" << std::endl;
        return 0;
    }

    // 获取 IProperty 指针, 用于配置参数
    IProperty* prop = GetIProperty(device);
    if (NULL == prop) {
        std::cout << "get property failed" << std::endl;
        ZCAN_CloseDevice(device);
    }
}

```

```

        return 0;
    }

#if 1
    // 设置通道 0 仲裁域波特率为 1M
    if (0 == prop->SetValue("0/canfd_abit_baud_rate", "1000000")) {
        std::cout << "set abit baud rate failed" << std::endl;
        goto end;
    }
    // 设置通道 0 数据域波特率为 4M
    if (0 == prop->SetValue("0/canfd_dbit_baud_rate", "4000000")) {
        std::cout << "set dbit baud rate failed" << std::endl;
        goto end;
    }
#else
    // 设置通道 0 自定义波特率，此处仅演示调用方式
    if (0 == prop->SetValue("0/ baud_rate_custom", "1.0Mbps(80%),5.0Mbps(87%),(0,00000F3E,0C01010C)")) {
        std::cout << "set custom baud rate failed" << std::endl;
        goto end;
    }
#endif

    // 设置通道 0 发送类型,0-正常发送，1-自发自收,如果不需要设置使用自发自收，可以不用设置
    if (0 == prop->SetValue("0/send_type", "0")) {
        std::cout << "set send_type failed" << std::endl;
        goto end;
    }

    // 初始化通道 0
    ZCAN_CHANNEL_INIT_CONFIG config;
    memset(&config, 0, sizeof(config));
    config.can_type = 1;                // 0 - CAN， 1 - CANFD
    config.can.mode = 0;                // 0 - 正常模式， 1 - 只听模式
    CHANNEL_HANDLE channel_0 = ZCAN_InitCAN(device, 0, &config);
    if (INVALID_CHANNEL_HANDLE == channel_0) {
        std::cout << "init channel 0 failed!" << std::endl;
        goto end;
    }

    // 设置通道 0 发送超时时间为 100ms，默认时间 2000，
    if (0 == prop->SetValue("0/tx_timeout", "100")) {
        std::cout << "set send timeout failed" << std::endl;
        goto end;
    }
}

```

```

// 设置第一组滤波, 只接收 ID 范围在 0x100-0x200 之间的标准帧
prop->SetValue("0/filter_mode", "0");           // 标准帧
prop->SetValue("0/filter_start", "0x100");       // 起始 ID
prop->SetValue("0/filter_end", "0x200");         // 结束 ID
// 设置第二组滤波, 只接收 ID 范围在 0x1FFFF-0x2FFFF 之间的扩展帧
prop->SetValue("0/filter_mode", "1");           // 扩展帧
prop->SetValue("0/filter_start", "0x1FFFF");     // 起始 ID
prop->SetValue("0/filter_end", "0x2FFFF");       // 结束 ID
// 使能滤波
prop->SetValue("0/filter_ack", "0");
// 清除滤波, 此处仅举例, 何时调用用户自由决定
prop->SetValue("0/filter_clear", "0");
//std::cout << "filter cleared" << std::endl;

// 启动 CAN 通道 0
if (0 == ZCAN_StartCAN(channel_0)) {
    std::cout << "start channel 0 failed" << std::endl;
    goto end;
}
// 启动 CAN 通道 0 的接收线程
thd_handle = std::thread(thread_task, channel_0);

/* 下列代码构造两条定时发送 CAN 报文以及两条定时发送 CANFD 报文, 索引 0 的 CAN 报文周期
100ms 发送一次, 索引 1 的 CAN 报文周期 200ms 发送一次, 索引 2 的 CANFD 报文周期 500ms 发送一次,
索引 3 的 CANFD 报文周期 600ms 发送一次, 发送 5s 后停止发送 */
ZCAN_AUTO_TRANSMIT_OBJ auto_can;
ZCANFD_AUTO_TRANSMIT_OBJ auto_canfd;
memset(&auto_can, 0, sizeof(auto_can));
auto_can.index = 0;           // 定时列表索引 0
auto_can.enable = 1;          // 使能此索引, 每条可单独设置
auto_can.interval = 100;      // 定时发送间隔 100ms
get_can_frame(auto_can.obj, 0); // 构造 CAN 报文
prop->SetValue("0/auto_send", (const char*)&auto_can); // 设置定时发送
memset(&auto_can, 0, sizeof(auto_can));
auto_can.index = 1;           // 定时列表索引 1
auto_can.enable = 1;          // 使能此索引, 每条可单独设置
auto_can.interval = 200;      // 定时发送间隔 200ms
get_can_frame(auto_can.obj, 1); // 构造 CAN 报文
prop->SetValue("0/auto_send", (const char*)&auto_can); // 设置定时发送
memset(&auto_canfd, 0, sizeof(auto_canfd));
auto_canfd.index = 2;         // 定时列表索引 2
auto_canfd.enable = 1;        // 使能此索引, 每条可单独设置
auto_canfd.interval = 500;    // 定时发送间隔 500ms

```



```

get_canfd_frame(auto_canfd.obj, 2); // 构造 CANFD 报文
prop->SetValue("0/auto_send_canfd", (const char*)&auto_canfd); // 设置定时发送
memset(&auto_canfd, 0, sizeof(auto_canfd));
auto_canfd.index = 3; // 定时列表索引 3
auto_canfd.enable = 1; // 使能此索引，每条可单独设置
auto_canfd.interval = 600; // 定时发送间隔 600ms
get_canfd_frame(auto_canfd.obj, 3); // 构造 CANFD 报文
prop->SetValue("0/auto_send_canfd", (const char*)&auto_canfd); // 设置定时发送
prop->SetValue("0/apply_auto_send", "0"); // 使能定时发送
Sleep(5000); // 等待发送 5s
prop->SetValue("0/clear_auto_send", "0"); // 清除定时发送

// 设置队列发送模式
prop->SetValue("0/set_send_mode", "1");
// 获取队列发送可用缓冲
int free_count = *((int*)prop->GetValue("0/get_device_available_tx_count/1"));
// 构造 50 条报文，报文 ID 从 0 递增，帧间隔 10ms 递增
if (free_count > 50) {
    ZCAN_Transmit_Data tran_data[50] = {};
    for (int i = 0; i < 50; ++i) {
        get_can_frame_queue(tran_data[i], i, i * 10);
    }
    int ret_count = ZCAN_Transmit(channel_0, tran_data, 50);
}
// 5 秒后清空队列发送
Sleep(5000);
prop->SetValue("0/clear_delay_send_queue", "0");

Sleep(1000);
// 正常发送 10 帧 CAN 报文
ZCAN_Transmit_Data trans_data[10] = {};
for (int i = 0; i < 10; ++i){
    get_can_frame(trans_data[i], i);
}
int send_count = ZCAN_Transmit(channel_0, trans_data, 10);
std::cout << "send can frame: " << std::dec << send_count << std::endl;
// 正常发送 10 帧 CANFD 报文
ZCAN_TransmitFD_Data trans_datafd[10] = {};
for (int i = 0; i < 10; ++i){
    get_canfd_frame(trans_datafd[i], i);
}
send_count = ZCAN_TransmitFD(channel_0, trans_datafd, 10);
std::cout << "send canfd frame: " << std::dec << send_count << std::endl;

```

```

        system("pause");
end:
    ReleaseIProperty(prop);
    g_thd_run = 0;
    if (thd_handle.joinable())
        thd_handle.join();
    std::cout << "thread exit, close device" << std::endl;
    ZCAN_CloseDevice(device);
    system("pause");

    return 0;
}

```

### 3.3.3 USBCAN-xE-U PCI-50x0-U 系列

本系列适用的设备：PCI-5010-U、PCI-5020-U、USBCAN-E-U、USBCAN-2E-U、CANalyst-II+、USBCAN-4E-U、USBCAN-8E-U。

#### 1. 波特率

标准波特率		
项	值	说明
path	n/ baud_rate	n 代表通道号
value	【1000000, 800000, 500000, 250000, 125000, 100000, 50000】	单位 bps, 如值为 500000 则表示波特率为 500k
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	
自定义波特率		
项	值	说明
path	n/ baud_rate_custom	n 代表通道号
value	自定义	通过 ZCANPRO 的波特率计算器计算
Get/Set	Set	
注意点	1. 需在 ZCAN_InitCAN 之前设置 2. USBCAN-4E-U 输入要设置的波特率数值即可, 比如 200kbps, 则输入 200000; 310kbps, 则输入 310000, 其他型号均需通过波特率计算器计算	

#### 2. 滤波

本系列每通道最多可设置 32 组滤波, 即如果不设置滤波 CAN 通道将接收所有报文, 如果设置了滤波, CAN 通道将只接收或不接收滤波范围内的报文。添加一条滤波的标准顺序是: 设置滤波模式, 设置起始帧, 设置结束帧。如果要添加多条就重复上述步骤, 添加完滤波并不会立即生效, 需设置 filter\_ack 使能所设的滤波表。

注 1: USBCAN-8E-U 不支持此属性, 如需滤波需在 ZCAN\_InitCAN 时设置。

设置滤波模式		
项	值	说明
path	n/filter_mode	n 代表通道号
value	0 – 标准帧 1 – 扩展帧	
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
设置滤波起始帧 ID		
项	值	说明
path	n/filter_start	n 代表通道号
value	自定义	“0x00000000”，16 进制字符
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
设置滤波结束帧 ID		
项	值	说明
path	n/filter_end	n 代表通道号
value	自定义	“0x00000000”，16 进制字符
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之后设置	
滤波生效（全部滤波 ID 同时生效）		
项	值	说明
path	n/filter_ack	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	1. 需在 ZCAN_InitCAN 之后设置 2. USBCAN-2E-U 的滤波生效是所有通道的滤波生效，也就是设置了两个通道的滤波，只需要执行一次滤波生效即可	
清除滤波		
项	值	说明
path	n/filter_clear	n 代表通道号
value	0	固定值

Get/Set	Set	
注意点	1. 需在 ZCAN_InitCAN 之后设置 2. USBCAN-2E-U 的清除滤波在执行滤波生效时执行，如果没有执行滤波生效，则只是清除驱动的缓存数据，不清除实际设备的数据。	

### 3. 定时发送

本系列仅支持 USBCAN-2E-U、CANalyst-II+、USBCAN-4E-U、USBCAN-8E-U，每通道最大 32 条定时发送列表，只需将待发送数据及周期设置到设备并使能，设备将自动进行发送。相比于 PC 端的发送，定时发送精度高，周期准。

注 1：USBCAN-2E-U、CANalyst-II+ 开启定时发送后不能再调用 ZCAN\_Transmit 进行普通发送。

注 2：PCI-50X0-U 系列不支持定时发送。

设置定时发送 CAN 帧（设置后立即生效）		
项	值	说明
path	n/auto_send	n 代表通道号
value	ZCAN_AUTO_TRANSMIT_OBJ 指针	需将指针强制转换为 char*
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
清空定时发送		
项	值	说明
path	n/clear_auto_send	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	

### 4. 通道转发

USBCAN-4E-U 和 USBCAN-8E-U 支持通道间数据转发，其他型号不支持。通道转发即将某通道接收到的帧通过另一个通道发送出去。

path	value	说明
n/redirect	“x 1”=转发 “x 0”=不转发	path 中的 n 为转发通道，value 中的 x 为转发到的通道，value 的第二个参数 1 表示转发，0 表示不转发。如将通道 2 接收到的数据转发到通道 3，path 为“2/redirect”，value 为“3 1”
Get/Set	Set	
注意点	USBCAN-4E-U 支持 0-3 通道，USBCAN-8E-U 支持 0-7 通道，通道转发在 ZCAN_InitCAN 之后设置	

## 5. 示例代码

```
// 以下代码以 USBCAN-4E-U 为例

#include "stdafx.h"
#include "zlgcan.h"
#include <iostream>
#include <windows.h>
#include <thread>

#define USBCAN_8E_U 1
#define CH_COUNT 4
bool g_thd_run = 1;

// 接收数据线程
void thread_task(CHANNEL_HANDLE handle)
{
    int nChnl = (unsigned int)handle & 0x000000FF;
    std::cout << "chnl: " << std::dec << nChnl << " thread run, handle:0x" << std::hex << handle << std::endl;
    ZCAN_Receive_Data data[100] = {};
    while (g_thd_run)
    {
        int count = ZCAN_GetReceiveNum(handle, 0);    // 获取 CAN 报文(参数 2:0 - CAN, 1 - CANFD)
        数量
        while (g_thd_run && count > 0)
        {
            int rcount = ZCAN_Receive(handle, data, 100, 10);
            for (int i = 0; i < rcount; ++i)
            {
                std::cout << "CHNL: " << std::dec << nChnl << " recv can ID: 0x";
                std::cout << std::hex << data[i].frame.can_id << std::endl;
            }
            count -= rcount;
        }
        Sleep(100);
    }
    std::cout << "chnl: " << std::dec << nChnl << " thread exit" << std::endl;
}

// 此函数仅用于构造示例 CAN 报文
void get_can_frame(ZCAN_Transmit_Data& can_data, canid_t id)
{
    memset(&can_data, 0, sizeof(can_data));
    can_data.frame.can_id = id;                // CAN ID
}
```

```

        can_data.frame.can_dlc = 8;                // CAN 数据长度 8
        can_data.transmit_type = 0;                // 正常发送
        for (int i = 0; i < 8; ++i) {              // 填充 CAN 报文 DATA
            can_data.frame.data[i] = i;
        }
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    std::thread thd_handle[CH_COUNT];
    CHANNEL_HANDLE ch[CH_COUNT] = {};
    UINT dev_type = ZCAN_USBCAN_4E_U;

#ifdef USBCAN_8E_U
    dev_type = ZCAN_USBCAN_8E_U;
#endif

    // 打开设备
    DEVICE_HANDLE device = ZCAN_OpenDevice(dev_type, 0, 0);
    if (INVALID_DEVICE_HANDLE == device) {
        std::cout << "open device failed" << std::endl;
        return 0;
    }

    // 获取 IProperty 指针，用于配置参数
    IProperty* prop = GetIProperty(device);
    if (NULL == prop) {
        std::cout << "get property failed" << std::endl;
        ZCAN_CloseDevice(device);
        return 0;
    }

    // 循环设置、初始化、启动每个通道
    for (int i = 0; i < CH_COUNT; ++i) {
        char path[64] = {};

        // 设置波特率为 1M
        sprintf_s(path, "%d/ baud_rate", i);
        if (0 == prop->SetValue(path, "1000000")) {
            std::cout << "set baud rate failed" << std::endl;
            goto end;
        }
    }

    // 初始化并启动 CAN 通道 0、1、2

```

```

ZCAN_CHANNEL_INIT_CONFIG config;
memset(&config, 0, sizeof(config));
config.can_type = 0;           // 0 - CAN, 1 - CANFD
config.can.mode = 0;          // 0 - 正常模式, 1 - 只听模式
// USBCAN-8E-U 必须设置验收码和屏蔽码, 本系列其他型号可忽略
#if USBCAN_8E_U
    config.can.acc_code = 0;    // 验收码
    config.can.acc_mask = 0xffffffff; // 屏蔽码
#endif

ch[i] = ZCAN_InitCAN(device, i, &config);
if (INVALID_CHANNEL_HANDLE == ch[i]) {
    std::cout << "init channel failed!" << std::endl;
    goto end;
}

#if !USBCAN_8E_U
    // 滤波设置, 仅以 0 通道为例
    if (0 == i) {
        int ret = 1;
        // 设置第一组滤波, 只接收 ID 范围在 0x100-0x110 之间的标准帧
        ret &= prop->SetValue("0/filter_mode", "0");           // 标准帧
        ret &= prop->SetValue("0/filter_start", "0x100");       // 起始 ID
        ret &= prop->SetValue("0/filter_end", "0x110");         // 结束 ID
        // 设置第二组滤波, 只接收 ID 范围在 0x10000-0x10010 之间的扩展帧
        ret &= prop->SetValue("0/filter_mode", "1");           // 扩展帧
        ret &= prop->SetValue("0/filter_start", "0x10000");     // 起始 ID
        ret &= prop->SetValue("0/filter_end", "0x10010");       // 结束 ID
        // 滤波生效
        ret &= prop->SetValue("0/filter_ack", "0");
        if (0 == ret) {
            std::cout << "set filter failed!" << std::endl;
            goto end;
        }
        //// 清除滤波
        //if (0 == prop->SetValue("0/filter_clear", "0")) {
        //    std::cout << "clear filter failed!" << std::endl;
        //    goto end;
        //}
    }
#endif

// 启动 CAN 通道
if (0 == ZCAN_StartCAN(ch[i])) {
    std::cout << "start channel failed" << std::endl;
}

```

```

        goto end;
    }
    // 启动 CAN 通道的接收线程
    thd_handle[i] = std::thread(thread_task, ch[i]);
}

// 转发，将 1，2 通道转发到 0 通道
int ret = 1;
ret &= prop->SetValue("1/redirect", "0 1");
ret &= prop->SetValue("2/redirect", "0 1");
if (0 == ret) {
    std::cout << "set redirect failed!" << std::endl;
    goto end;
}

// 通道 0 定时发送 2 条 CAN 报文，ID 0 间隔 10ms，ID 1 间隔 100ms
ZCAN_AUTO_TRANSMIT_OBJ auto_can;
memset(&auto_can, 0, sizeof(auto_can));
auto_can.enable = 1; // 使能此索引，每条可单独设置
auto_can.interval = 10; // 定时发送间隔 10ms
auto_can.index = 0; // 定时列表索引 0
get_can_frame(auto_can.obj, 0); // 构造 CAN 报文
prop->SetValue("0/auto_send", (const char*)&auto_can); // 设置定时发送
memset(&auto_can, 0, sizeof(auto_can));
auto_can.enable = 1; // 使能此索引，每条可单独设置
auto_can.interval = 100; // 定时发送间隔 100ms
auto_can.index = 1; // 定时列表索引 1
get_can_frame(auto_can.obj, 1); // 构造 CAN 报文
prop->SetValue("0/auto_send", (const char*)&auto_can); // 设置定时发送

// 5 秒后停止定时发送
Sleep(1000);
prop->SetValue("0/clear_auto_send", "0");

// 通道 0 发送 10 帧报文
ZCAN_Transmit_Data trans_data[10] = {};
for (int i = 0; i < 10; ++i){
    get_can_frame(trans_data[i], i);
}
int send_count = ZCAN_Transmit(ch[0], trans_data, 10);
std::cout << "send frame: " << std::dec << send_count << std::endl;

system("pause");
end:

```



```

ReleaseIProperty(prop);
g_thd_run = 0;
for (int i = 0; i < CH_COUNT; ++i) {
    if (thd_handle[i].joinable())
        thd_handle[i].join();
}
std::cout << "thread exit, close device" << std::endl;
ZCAN_CloseDevice(device);
system("pause");

return 0;
}

```

### 3.3.4 CANDTU-x00UR

本系列适用的设备：CANDTU-100UR、CANDTU-200UR。

#### 1. 波特率

标准波特率		
项	值	说明
path	n/baud_rate	n 代表通道号
value	【1000000，800000，500000，250000，125000，100000，50000】	单位 bps，如值为 500000 则表示波特率为 500k
Get/Set	Set	
注意点	需在 ZCAN_StartCAN 之前设置	
自定义波特率		
项	值	说明
path	n/baud_rate_custom	n 代表通道号
value	自定义	通过 ZCANPRO 的波特率计算器计算
Get/Set	Set	
注意点	需在 ZCAN_StartCAN 之前设置	

#### 2. 终端电阻

CANDTU 每通道内置 120Ω 终端电阻, 可通过属性设置选择使能或不使能。

项	值	说明
path	n/initenal_resistance	n 代表通道号
value	0 - 禁能 1 - 使能	
Get/Set	Set	

注意点	需在 ZCAN_StartCAN 之前设置
-----	-----------------------

### 3. 工作模式

项	值	说明
path	n/work_mode	n 代表通道号
value	0 – 只听模式 1 – 正常模式	
Get/Set	Set	
注意点	需在 ZCAN_StartCAN 之前设置	

### 4. 验收码屏蔽码

区别于其他 CAN 卡，CANDTU 的验收码及屏蔽码不是在 ZCAN\_InitCAN 时设置，而是通过属性表进行设置。

验收码		
项	值	说明
path	n/acc_code	n 代表通道号
value	自定义	如“0x00000000”，16 进制字符
Get/Set	Set	
注意点	需在 ZCAN_StartCAN 之前设置	
屏蔽码		
项	值	说明
path	n/acc_mask	n 代表通道号
value	自定义	如“0xFFFFFFFF”，16 进制字符
Get/Set	Set	
注意点	需在 ZCAN_StartCAN 之前设置	

### 5. 示例代码

// 以下代码以 CANDTU-200UR 为例

```
#include "stdafx.h"
#include "zlgcan.h"
#include <iostream>
#include <windows.h>
#include <thread>

#define CH_COUNT 2
bool g_thd_run = 1;
```

```

// 接收数据线程
void thread_task(CHANNEL_HANDLE handle)
{
    int nChnl = (unsigned int)handle & 0x000000FF;
    std::cout << "chnl: " << std::dec << nChnl << " thread run, handle:0x" << std::hex << handle << std::endl;
    ZCAN_Receive_Data data[100] = {};
    while (g_thd_run)
    {
        int count = ZCAN_GetReceiveNum(handle, 0);    // 获取 CAN 报文(参数 2:0 - CAN, 1 - CANFD)
        数量
        while (g_thd_run && count > 0)
        {
            int rcount = ZCAN_Receive(handle, data, 100, 10);
            for (int i = 0; i < rcount; ++i)
            {
                std::cout << "CHNL: " << std::dec << nChnl << " recv can ID: 0x" << std::hex <<
data[i].frame.can_id << std::endl;
            }
            count -= rcount;
        }
        Sleep(100);
    }
    std::cout << "chnl: " << std::dec << nChnl << " thread exit" << std::endl;
}

// 此函数仅用于构造示例 CAN 报文
void get_can_frame(ZCAN_Transmit_Data& can_data, canid_t id)
{
    memset(&can_data, 0, sizeof(can_data));
    can_data.frame.can_id = id;                // CAN ID
    can_data.frame.can_dlc = 8;                // CAN 数据长度 8
    can_data.transmit_type = 0;                // 正常发送
    for (int i = 0; i < 8; ++i) {              // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    std::thread thd_handle;
    CHANNEL_HANDLE ch[CH_COUNT] = {};

    // 打开设备
    DEVICE_HANDLE device = ZCAN_OpenDevice(ZCAN_CANDTU_200UR, 0, 0);

```

```

if (INVALID_DEVICE_HANDLE == device) {
    std::cout << "open device failed" << std::endl;
    return 0;
}

// 获取 IProperty 指针，用于配置参数
IProperty* prop = GetIProperty(device);
if (NULL == prop) {
    std::cout << "get property failed" << std::endl;
    ZCAN_CloseDevice(device);
    return 0;
}

// 循环设置、初始化、启动每个通道
for (int i = 0; i < CH_COUNT; ++i) {
    char path[64] = {};

    // 设置波特率为 1M
    sprintf_s(path, "%d/aud_rate", i);
    if (0 == prop->SetValue(path, "1000000")) {
        std::cout << "set baud rate failed" << std::endl;
        goto end;
    }

    // 使能通道 120  $\Omega$  内置终端电阻
    sprintf_s(path, "%d/initenal_resistance", i);
    if (0 == prop->SetValue(path, "1")) {
        std::cout << "enable terminal resistance failed" << std::endl;
        goto end;
    }

    // 设置通道工作模式为正常模式
    sprintf_s(path, "%d/work_mode", i);
    if (0 == prop->SetValue(path, "1")) {
        std::cout << "set work mode failed" << std::endl;
        goto end;
    }

    // 设置通道 0、1 验收码屏蔽码,接收全部数据
    int ret = 1;
    sprintf_s(path, "%d/acc_code", i);
    ret &= prop->SetValue(path, "0");
    sprintf_s(path, "%d/acc_mask", i);
    ret &= prop->SetValue(path, "0xFFFFFFFF");
    if (0 == ret) {
        std::cout << "set acc param failed" << std::endl;
        goto end;
    }
}

```

```

    }
    // 初始化通道
    ZCAN_CHANNEL_INIT_CONFIG config;
    memset(&config, 0, sizeof(config));
    config.can_type = 0;           // 0 - CAN, 1 - CANFD
    ch[i] = ZCAN_InitCAN(device, i, &config);
    if (INVALID_CHANNEL_HANDLE == ch[i]) {
        std::cout << "init channel failed!" << std::endl;
        goto end;
    }
    // 启动 CAN 通道
    if (0 == ZCAN_StartCAN(ch[i])) {
        std::cout << "start channel failed" << std::endl;
        goto end;
    }
}

// 启动 CAN 通道 1 的接收线程
thd_handle = std::thread(thread_task, ch[1]);

// 通道 0 发送 10 帧报文
ZCAN_Transmit_Data trans_data[10] = {};
for (int i = 0; i < 10; ++i){
    get_can_frame(trans_data[i], i);
}
int send_count = ZCAN_Transmit(ch[0], trans_data, 10);
std::cout << "send frame: " << std::dec << send_count << std::endl;

system("pause");
end :
    ReleaseIPProperty(prop);
    g_thd_run = 0;
    if (thd_handle.joinable())
        thd_handle.join();
    std::cout << "thread exit, close device" << std::endl;
    ZCAN_CloseDevice(device);

    return 0;
}

```

### 3.3.5 以太网系列 1

区别于其他通信接口类型的设备，以太网系列 CAN 卡在本接口库中只进行通道链接以及数据收发，设备相关参数（如波特率、滤波、IP 等）的设置不在本接口库功能范畴。本系列每个通道对应一个网络连接，可理解为同一个设备的多个通道做为不同的单通道设备。

TCP 系列型号包括：CANDTU-NET、CANDTU-NET-400、CANET-TCP、CANWIFI-TCP、CANFDWIFI-TCP。

UDP 系列型号包括：CANET-UDP、CANWIFI-UDP、CANFDWIFI-UDP。

### 1. 工作模式

连接的工作模式，如设备作为服务器，二次开发时应设为客户端模式。

项	值	说明
path	n/work_mode	n 代表通道号
value	0 – 客户端 1 – 服务器	
Get/Set	Set	
注意点	1. 需在 ZCAN_StartCAN 之前设置 2. UDP 系列不设置此参数	

### 2. 本地端口

UDP 模式和 TCP Server 模式下的本地监听端口。

项	值	说明
path	n/local_port	n 代表通道号
value	自定义	如“4001”
Get/Set	Set	
注意点	1. 需在 ZCAN_StartCAN 之前设置 2. TCP 系列不设置此参数	

### 3. IP 地址

目标设备的 IP。

项	值	说明
path	n/ip	n 代表通道号
value	自定义	如“192.168.0.178”
Get/Set	Set	
注意点	需在 ZCAN_StartCAN 之前设置	

### 4. 工作端口

目标设备监听的端口。

项	值	说明
path	n/work_port	n 代表通道号
value	自定义	如“4001”
Get/Set	Set	

注意点	需在 ZCAN_StartCAN 之前设置
-----	-----------------------

## 5. 示例代码

```
// 以下代码以 CANET-2E-U 为例
// 对于本系列，每个通道为一个单独连接，可以将每个通道理解为独立的设备进行操作

#include "stdafx.h"
#include "zlgcan.h"
#include <iostream>
#include <windows.h>
#include <thread>

#define CH_COUNT 2
bool g_thd_run = 1;

// 此函数仅用于构造示例 CAN 报文
void get_can_frame(ZCAN_Transmit_Data& can_data, canid_t id)
{
    memset(&can_data, 0, sizeof(can_data));
    can_data.frame.can_id = id;           // CAN ID
    can_data.frame.can_dlc = 8;          // CAN 数据长度 8
    can_data.transmit_type = 0;          // 正常发送
    for (int i = 0; i < 8; ++i) {        // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}

void thread_task(CHANNEL_HANDLE handle, int ch)
{
    std::cout << "chnl: " << std::dec << ch << " thread run, handle:0x" << std::hex << handle << std::endl;
    ZCAN_Receive_Data data[100] = {};
    while (g_thd_run)
    {
        int count = ZCAN_GetReceiveNum(handle, 0);    // 获取 CAN 报文(参数 2:0 - CAN, 1 - CANFD)
        数量
        while (g_thd_run && count > 0)
        {
            int rcount = ZCAN_Receive(handle, data, 100, 10);
            for (int i = 0; i < rcount; ++i)
            {
                std::cout << "CHNL: " << std::dec << ch << " recv can ID: 0x" << std::hex <<
                data[i].frame.can_id << std::endl;
            }
            count -= rcount;
        }
    }
}
```

```

        Sleep(100);
    }

    std::cout << "chnl: " << std::dec << ch << " thread exit" << std::endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    UINT dev_type = ZCAN_CANETTCP;
    std::thread thd_handle[CH_COUNT];
    CHANNEL_HANDLE ch[CH_COUNT] = {};
    DEVICE_HANDLE device[CH_COUNT] = {INVALID_DEVICE_HANDLE};
    IProperty* prop[CH_COUNT] = {};

    // 循环打开、设置、初始化、启动每个通道
    for (int i = 0; i < CH_COUNT; ++i) {
        char path[64] = {};
        char port[2][16] = { "4001", "4002" };

        // 打开设备，即使为同一个设备，不同通道的设备索引也不是同一个
        device[i] = ZCAN_OpenDevice(dev_type, i, 0);
        if (INVALID_DEVICE_HANDLE == device[i]) {
            std::cout << "open device failed!" << std::endl;
            goto end;
        }

        // 获取 IProperty 指针，用于配置参数
        prop[i] = GetIProperty(device[i]);
        if (NULL == prop[i]) {
            std::cout << "get property failed" << std::endl;
            goto end;
        }

        // 设置工作模式为客户端
        if (0 == prop[i]->SetValue("0/work_mode", "0")) {
            std::cout << "set work mode failed" << std::endl;
            goto end;
        }

        // 设置目标 IP 地址
        if (0 == prop[i]->SetValue("0/ip", "172.16.9.221")) {
            std::cout << "set ip failed" << std::endl;
            goto end;
        }

        // 设置目标端口 通道 0-4001，通道 1-4002

```



```

        if (0 == prop[i]->SetValue("0/work_port", port[i])) {
            std::cout << "set port failed" << std::endl;
            goto end;
        }

        // 初始化通道，在 CANET 系列中初始化不做实际操作，仅用于获取通道句柄
        ZCAN_CHANNEL_INIT_CONFIG config;
        ch[i] = ZCAN_InitCAN(device[i], 0, &config);
        if (INVALID_CHANNEL_HANDLE == ch[i]) {
            std::cout << "init channel failed!" << std::endl;
            goto end;
        }

        // 启动 CAN 通道
        if (0 == ZCAN_StartCAN(ch[i])) {
            std::cout << "start channel failed" << std::endl;
            goto end;
        }
        // 启动 CAN 通道的接收线程
        thd_handle[i] = std::thread(thread_task, ch[i], i);
    }

    // 通道 0 发送 10 帧报文
    ZCAN_Transmit_Data trans_data[10] = {};
    for (int i = 0; i < 10; ++i){
        get_can_frame(trans_data[i], i);
    }
    int send_count = ZCAN_Transmit(ch[0], trans_data, 10);
    std::cout << "send frame: " << std::dec << send_count << std::endl;
    system("pause");

end:
    g_thd_run = 0;
    for (int i = 0; i < CH_COUNT; ++i) {
        if (thd_handle[i].joinable())
            thd_handle[i].join();
        std::cout << "thread exit, close device" << std::endl;
        if (NULL != prop[i])
            ReleaseIPProperty(prop[i]);
        if (NULL != device[i])
            ZCAN_CloseDevice(device[i]);
    }

    system("pause");

```

```

return 0;
}

```

### 3.3.6 以太网系列 2

本系列相对于系列 1 增加了定时发送、队列发送等功能，支持的型号包括：CANFDNET-TCP、CANFDNET-400U-TCP、CANFDNET-UDP、CANFDNET-400U-UDP。

#### 1. 工作模式

连接的工作模式，如设备作为服务器，二次开发时应设为客户端模式。

项	值	说明
path	n/work_mode	n 代表通道号
value	0 – 客户端 1 – 服务器	
Get/Set	Set	
注意点	1. 需在 ZCAN_StartCAN 之前设置 2. UDP 系列不设置此参数	

#### 2. 本地端口

UDP 模式和 TCP Server 模式下的本地监听端口。

项	值	说明
path	n/local_port	n 代表通道号
value	自定义	如“4001”
Get/Set	Set	
注意点	1. 需在 ZCAN_StartCAN 之前设置 2. TCP 系列不设置此参数	

#### 3. IP 地址

目标设备的 IP。

项	值	说明
path	n/ip	n 代表通道号
value	自定义	如“192.168.0.178”
Get/Set	Set	
注意点	需在 ZCAN_StartCAN 之前设置	

#### 4. 工作端口

目标设备监听的端口。

项	值	说明
path	n/work_port	n 代表通道号

value	自定义	如“4001”
Get/Set	Set	
注意点	需在 ZCAN_StartCAN 之前设置	

## 5. 定时发送

定时发送支持发送 CAN,CANFD 帧，用户需要设置要定时发送的帧后，开启定时发送，设备会按照设置的定时间隔周期发送数据，用户可以调用接口停止定时发送。

设置定时发送 CAN 帧		
项	值	说明
path	n/auto_send	n 代表通道号
value	ZCAN_AUTO_TRANSMIT_OBJ 指针	需将指针强制转换为 char*
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
设置定时发送 CANFD 帧		
项	值	说明
path	n/auto_send_canfd	n 代表通道号
value	ZCANFD_AUTO_TRANSMIT_OBJ 指针	需将指针强制转换为 char*
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
开始定时发送		
项	值	说明
path	n/apply_auto_send	n 代表通道号
value	1	固定值
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
清空定时发送		
项	值	说明
path	n/clear_auto_send	n 代表通道号
value	0	固定值
Get/Set	Set	
注意点	在 ZCAN_StartCAN 之后设置	
查询定时发送 CAN 帧数量		

项	值	说明
path	n/get_auto_send_can_count/1	n 代表通道号，最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 int*
value	函数返回指向 int 的指针表示定时发送 CAN 的数量	int nCount = *(int*)pRet; pRet 表示 GetValue 返回值，nCount 表示定时发送 CAN 数量
Get/Set	Get	
注意点	在 ZCAN_StartCAN 之后设置	
查询定时发送 CAN 帧信息		
项	值	说明
path	n/get_auto_send_can_data/1	n 代表通道号，最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 ZCAN_AUTO_TRANSMIT_OBJ*
value	函数返回指向的 ZCAN_AUTO_TRANSMIT_OBJ 指针 表示定时发送 CAN 的数据首地址	返回值类型转换后指向定时发送 CAN 帧
Get/Set	Get	
注意点	在 ZCAN_StartCAN 之后设置	
查询定时发送 CANFD 帧数量		
项	值	说明
path	n/get_auto_send_canfd_count/1	n 代表通道号，最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 int*
value	函数返回指向 int 的指针表示定时发送 CANFD 的数量	int nCount = *(int*)pRet; pRet 表示 GetValue 返回值，nCount 表示定时发送 CANFD 数量
Get/Set	Get	
注意点	在 ZCAN_StartCAN 之后设置	
查询定时发送 CANFD 帧信息		
项	值	说明
path	n/get_auto_send_canfd_data/1	n 代表通道号，最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 ZCANFD_AUTO_TRANSMIT_OBJ *
value	函数返回指向的	返回值转换类型后指向定时发送 CANFD

	ZCANFD_AUTO_TRANSMIT_OBJ 指针表示定时发送 CAN 的数据首地址	帧
Get/Set	Get	
注意点	在 ZCAN_StartCAN 之后设置	

## 6. 队列发送

CANFDNET 系列设备支持普通发送、队列发送。普通发送指用户发送多帧数据时，设备会尽力使用最快的速度进行发送，帧之间的时间间隔无法控制。队列发送指用户可以设定帧之间的发送间隔，设备会按照帧顺序，使用指定的间隔将数据发送到总线。

通过队列发送，用户可以提前准备好多帧报文，设定报文之间的间隔，将准备好的报文发送给设备，设备按照预定义的帧间隔进行精准发送。队列发送可提高发送帧之间的帧间隔精度。与定时发送相比，队列发送每帧只发送一次，需由用户不断准备报文并发送到设备。CANFDNET 设备可以同时使用队列发送，定时发送，普通发送。

队列发送的数据发送使用 ZCAN\_TransmitData 接口，CAN/CANFD 数据使用 ZCANDataObj 结构。CANFDNET 系列设备队列模式下延时时间单位支持毫秒(ms)和 100 微秒(0.1ms)两种时间单位，使用 100 微秒时间单位，设备的发送精度更高一些。时间单位可以通过帧标志为中的 txDelay 字段设置。，txDelay 字段使用 0 表示直接发送数据到总线，txDelay 设置为 1 表示使用毫秒作为时间单位，txDelay 设置为 2 表示使用 100 微秒作为时间单位。延时时间存放在 timeStamp 字段中。

设备发送当前帧的同时会启动计时器按照当前帧设定的时间进行计时，计时时间结束会从队列取下一帧进行发送并重新开始计时。

可以通过接口获取设备端可用的队列空间以及清空当前队列发送数据，使用方式参考设备属性表。

获取发送队列可用缓存数量		
项	值	说明
path	n/get_device_available_tx_count/1	n 代表通道号，最后的数字“1”只是内部标志，可以是任意数字，返回值 char*转换为 int*
value	无	
Get/Set	Get	
清空队列发送缓存（即队列缓存中未发送的帧将被清空）		
项	值	说明
path	n/clear_delay_send_queue	n 代表通道号
value	0	固定值
Get/Set	Set	

## 7. 合并接收

CANFDNET 系列设备支持数据合并接收，合并接收是指通过同一个接口 ZCAN\_ReceiveData 可以接收到设备支持的不同通道，不同类型的数据(ZCANDataObj 结构

包含通道和数据类型等信息)。针对 CANFDNET 系列设备,通过合并接收可以接收到不同通道的 CAN 和 CANFD 数据,设备支持 LIN 总线时可以也可以接收到 LIN 数据。合并接收可以以统一的方式获取各种类型的数据,避免出现由于传统接口按照通道和类型获取 CAN/CANFD/LIN 数据导致的获取数据的时间戳可能不连续的问题。

合并接收功能需要通过设备属性表设置功能的开启或关闭,设备默认不开启合并接收功能。为了避免合并接收和普通接收状态切换可能产生数据错乱的问题,如果需要合并接收功能,则需要在启动设备第一个通道前打开合并接收,并且在后续使用过程中不要随意更改合并接收开关。

设备开启合并接收后,只能通过合并接收接口 ZCAN\_ReceiveData 获取设备数据。设备通道启动后,每个已经启动的通道的数据都可封装在 ZCANDataObj 数据结构中示,通过 ZCAN\_ReceiveData 接口接收数据。ZCANDataObj 的 dataType 成员表示数据的具体类型,ZCANDataObj 的 chnl 成员表示数据所在的通道。

合并接收功能开启/关闭		
项	值	说明
path	n/set_device_rcv_merge	n 代表通道号
value	0 – 关闭合并接收功能 1 – 开启合并接收功能	设备默认为 0, 不开启合并接收
Get/Set	Set	
查询设备当前是否开启了合并接收		
项	值	说明
path	n/get_device_rcv_merge/1	n 代表通道号,使用任意合法通道号即可。最后的数字“1”只是内部标志,可以是任意数字,返回值 char*转换为 int*,指向的数值为 0 表示未开启合并接收,为 1 表示已开启合并接收
value	无	
Get/Set	Get	

## 8. 总线利用率

CANFDNET 设备如果开启了总线利用率上报,则可以通过接口获取总线利用率信息。

获取总线利用率（设备开启此选项后才能获取）		
项	值	说明
path	n/get_bus_usage/1	n 代表通道号,最后的数字“1”只是内部标志,可以是任意数字,返回值 char*转换为 BusUsage*
value	0	
Get/Set	Get	

## 9. 示例代码

```
// 以下代码以 CANFDNET-400 为例，对于本系列，每个连接均可操作全部通道
// 如设备做服务器，端口 8000，程序做客户端，只需在 start 任意通道之前设置 0 通道的工作模式、端口、
IP

#include "stdafx.h"
#include "zlscan.h"
#include <iostream>
#include <iomanip>
#include <windows.h>
#include <thread>

#define CH_COUNT    4
bool    g_thd_run = 1;

// 此函数仅用于构造示例 CAN 报文
void get_can_frame(ZCAN_Transmit_Data& can_data, canid_t id)
{
    memset(&can_data, 0, sizeof(can_data));
    can_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0);    // CAN ID + STD/EXT + DATA/RMT
    can_data.frame.can_dlc = 8;                        // CAN 数据长度 8
    can_data.transmit_type = 0;                        // 正常发送
    can_data.frame.__pad |= TX_ECHO_FLAG;              // 发送回显
    for (int i = 0; i < 8; ++i) {                      // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}

// 此函数仅用于构造示例 CANFD 报文
void get_canfd_frame(ZCAN_TransmitFD_Data& canfd_data, canid_t id)
{
    memset(&canfd_data, 0, sizeof(canfd_data));
    canfd_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0); // CAN ID + STD/EXT + DATA/RMT
    canfd_data.frame.len = 64;                          // CANFD 数据长度 64
    canfd_data.transmit_type = 0;                        // 正常发送
    canfd_data.frame.flags |= TX_ECHO_FLAG;              // 发送回显
    for (int i = 0; i < 64; ++i) {                      // 填充 CANFD 报文 DATA
        canfd_data.frame.data[i] = i;
    }
}

// 此函数仅用于构造示例队列发送报文
void get_can_frame_queue(ZCANDataObj& data, int ch, canid_t id, bool is_fd, UINT delay)
{
    memset(&data, 0, sizeof(data));
```

```

data.dataType = ZCAN_DT_ZCAN_CAN_CANFD_DATA;
data.chnl = ch;
ZCANCANFDData & can_data = data.data.zcanCANFDData;
can_data.frame.can_id = MAKE_CAN_ID(id, 0, 0, 0);    // CAN ID + STD/EXT + DATA/RMT
can_data.frame.len = is_fd ? 64 : 8;                // 数据长度 8/64
can_data.flag.unionVal.transmitType = 0;            // 正常发送
can_data.flag.unionVal.txEchoRequest = 1;           // 设置发送回显
can_data.flag.unionVal.frameType = is_fd ? 1 : 0;    // CAN or CANFD
can_data.flag.unionVal.txDelay = ZCAN_TX_DELAY_UNIT_MS; // 队列延时单位毫秒
can_data.timeStamp = delay;                          // 队列延时时间，最大值 65535
for (int i = 0; i < can_data.frame.len; ++i) {        // 填充 CAN 报文 DATA
    can_data.frame.data[i] = i;
}
}

// 此函数仅用于构造示例合并发送报文
void get_can_canfd_frame(ZCANDataObj& data, int ch, canid_t id, bool is_fd)
{
    memset(&data, 0, sizeof(data));
    data.dataType = ZCAN_DT_ZCAN_CAN_CANFD_DATA;
    data.chnl = ch;
    ZCANCANFDData & can_data = data.data.zcanCANFDData;
    can_data.frame.can_id = MAKE_CAN_ID(id,0,0,0);    // CAN ID + STD/EXT + DATA/RMT
    can_data.frame.len = is_fd ? 64:8;                // CAN 数据长度 8
    can_data.flag.unionVal.transmitType = 0;            // 正常发送
    can_data.flag.unionVal.txEchoRequest = 1;           // 设置发送回显
    can_data.flag.unionVal.frameType = is_fd ? 1 : 0;    // CAN or CANFD
    can_data.flag.unionVal.txDelay = ZCAN_TX_DELAY_NO_DELAY; // 直接发送报文到总线
    for (int i = 0; i < can_data.frame.len; ++i) {        // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}

void thread_task(DEVICE_HANDLE handle)
{
    std::cout << "Thread run, handle:0x" << std::hex << handle << std::endl;
    ZCANDataObj recv_data[100] = {0}; // 定义接收数据缓冲区，100 仅用于举例，用户根据实际情况自己
    定义
    while (g_thd_run) {
        int rcount = ZCAN_ReceiveData(handle, recv_data, 100, 1);
        int lcount = rcount;
        while (g_thd_run && lcount > 0) {
            for (int i = 0; i < rcount; ++i,--lcount) {

```



```

        if (recv_data[i].dataType != ZCAN_DT_ZCAN_CAN_CANFD_DATA) { //
只处理 CAN 或 CANFD 数据
            continue;
        }
        std::cout << "CHNL: " << std::dec << (int)recv_data[i].chnl; // 打印通道
        ZCANCANFDData & can_data = recv_data[i].data.zcanCANFDData;
        std::cout << " TIME:" << std::fixed << std::setprecision(6) <<
            can_data.timeStamp/1000000.0f <<"s[" <<
            std::dec << can_data.timeStamp <<"]"; // 打印时间戳
        if (can_data.flag.unionVal.txEchoed == 1) {
            std::cout << " [TX] "; // 发送帧
        }
        else {
            std::cout << " [RX] "; // 接收帧
        }
        std::cout << "ID: 0x" << std::hex << can_data.frame.can_id; // 打印 ID
        std::cout << " LEN " << std::dec << (int)can_data.frame.len; // 打印长度
        std::cout << " DATA " << std::hex; // 打印数据
        for (int ind = 0; ind < can_data.frame.len; ++ind) {
            std::cout << std::hex << " " << (int)can_data.frame.data[ind];
        }
        std::cout << std::endl;
    }
    }
    Sleep(10);
}

std::cout << "Thread exit" << std::endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    UINT dev_type = ZCAN_CANFDNET_400U_TCP;
    std::thread thd_handle;
    CHANNEL_HANDLE ch[CH_COUNT] = {};

    // 打开设备
    DEVICE_HANDLE device = ZCAN_OpenDevice(dev_type, 0, 0);
    if (INVALID_DEVICE_HANDLE == device) {
        std::cout << "open device failed!" << std::endl;
        return 0;
    }
}

```

```

// 获取 IProperty 指针，用于配置参数
IProperty* prop = GetIProperty(device);
if (NULL == prop) {
    std::cout << "get property failed" << std::endl;
    ZCAN_CloseDevice(device);
    return 0;
}

// 本系列同一设备多通道只需设置一次 工作模式、IP、端口，并需在全部通道 start 之前设置
// 设置工作模式为客户端
if (0 == prop->SetValue("0/work_mode", "0")) {
    std::cout << "set work mode failed" << std::endl;
    goto end;
}

// 设置目标 IP 地址
if (0 == prop->SetValue("0/ip", "172.16.9.221")) {
    std::cout << "set ip failed" << std::endl;
    goto end;
}

// 设置目标通道端口
if (0 == prop->SetValue("0/work_port", "8000")) {
    std::cout << "set port failed" << std::endl;
    goto end;
}

// 循环初始化、启动每个通道
for (int i = 0; i < CH_COUNT; ++i)
{
    char path[64] = {};
    // 初始化通道，在 CANFD ET 系列中初始化不做实际操作，仅用于获取通道句柄
    ZCAN_CHANNEL_INIT_CONFIG config;
    memset(&config, 0, sizeof(config));
    ch[i] = ZCAN_InitCAN(device, i, &config);
    if (INVALID_CHANNEL_HANDLE == ch[i]) {
        std::cout << "init channel failed!" << std::endl;
        goto end;
    }

    // 设置合并接收标志，启用合并发送，接收接口（只需设置 1 次）
    if (0 == i) {
        prop->SetValue("0/set_device_rcv_merge", "1");
    }
}

```

```

        // 启动 CAN 通道
        if (0 == ZCAN_StartCAN(ch[i])) {
            std::cout << "start channel 0 failed" << std::endl;
            goto end;
        }
    }

    // 启动 CAN 通道的接收线程
    thd_handle = std::thread(thread_task, device);

#ifdef 0
    /*
    下列代码构造两条定时发送 CAN 报文以及两条定时发送 CANFD 报文，
    索引 0 的 CAN 报文周期 100ms 发送一次，
    索引 1 的 CAN 报文周期 200ms 发送一次，
    索引 2 的 CANFD 报文周期 500ms 发送一次，
    索引 3 的 CANFD 报文周期 600ms 发送一次，
    发送 5s 后停止发送
    */
    ZCAN_AUTO_TRANSMIT_OBJ auto_can;
    ZCANFD_AUTO_TRANSMIT_OBJ auto_canfd;
    memset(&auto_can, 0, sizeof(auto_can));
    auto_can.index = 0; // 定时列表索引 0
    auto_can.enable = 1; // 使能此索引，每条可单独设置
    auto_can.interval = 100; // 定时发送间隔 100ms
    get_can_frame(auto_can.obj, 0); // 构造 CAN 报文
    prop->SetValue("0/auto_send", (const char*)&auto_can); // 设置定时发送
    memset(&auto_can, 0, sizeof(auto_can));
    auto_can.index = 1; // 定时列表索引 1
    auto_can.enable = 1; // 使能此索引，每条可单独设置
    auto_can.interval = 200; // 定时发送间隔 200ms
    get_can_frame(auto_can.obj, 1); // 构造 CAN 报文
    prop->SetValue("0/auto_send", (const char*)&auto_can); // 设置定时发送
    memset(&auto_canfd, 0, sizeof(auto_canfd));
    auto_canfd.index = 2; // 定时列表索引 2
    auto_canfd.enable = 1; // 使能此索引，每条可单独设置
    auto_canfd.interval = 500; // 定时发送间隔 500ms
    get_canfd_frame(auto_canfd.obj, 2); // 构造 CANFD 报文
    prop->SetValue("0/auto_send_canfd", (const char*)&auto_canfd); // 设置定时发送
    memset(&auto_canfd, 0, sizeof(auto_canfd));
    auto_canfd.index = 3; // 定时列表索引 3
    auto_canfd.enable = 1; // 使能此索引，每条可单独设置
    auto_canfd.interval = 600; // 定时发送间隔 600ms
    get_canfd_frame(auto_canfd.obj, 3); // 构造 CANFD 报文

```

```

prop->SetValue("0/auto_send_canfd", (const char*)&auto_canfd);    // 设置定时发送

prop->SetValue("0/apply_auto_send", "0");                          // 使能定时发送

// 获取 0 通道定时发送 CAN 报文列表
int auto_count = *(prop->GetValue("0/get_auto_send_can_count/1"));
if (auto_count > 0)
{
    ZCAN_AUTO_TRANSMIT_OBJ* pAuto;
    pAuto = (ZCAN_AUTO_TRANSMIT_OBJ*)prop->GetValue("0/get_auto_send_can_data/1");
    std::cout << "ch0 can auto send list count: " << auto_count << std::endl;
    for (int i = 0; i < auto_count; ++i, pAuto++)
    {
        std::cout << "index " << pAuto->index << "\tID " << pAuto->obj.frame.can_id << std::endl;
    }
}
// 获取 0 通道定时发送 CANFD 报文列表
auto_count = *(prop->GetValue("0/get_auto_send_canfd_count/1"));
if (auto_count > 0)
{
    ZCANFD_AUTO_TRANSMIT_OBJ* pAuto;
    pAuto = (ZCANFD_AUTO_TRANSMIT_OBJ*)prop->GetValue("0/get_auto_send_canfd_data/1");
    std::cout << "ch0 canfd auto send list count: " << auto_count << std::endl;
    for (int i = 0; i < auto_count; ++i, pAuto++)
    {
        std::cout << "index " << pAuto->index << "\tID " << pAuto->obj.frame.can_id << std::endl;
    }
}

Sleep(5000);                                                        // 等待发送 5s
prop->SetValue("0/clear_auto_send", "0");                          // 清除定时发送
system("pause");

#endif

// 队列发送及获取总线利用率
#if 1
// 获取队列发送可用缓冲
int free_count = *((int*)prop->GetValue("0/get_device_available_tx_count/1"));
// 构造 50 条报文，报文 ID 从 0 递增，帧间隔 10ms 递增
if (free_count > 50) {
    ZCANDataObj tran_data[50] = {};
    for (int i = 0; i < 50; ++i) {
        get_can_frame_queue(tran_data[i], 0, i, (i % 2 ? true : false), i * 10);
    }
}

```

```

        int ret_count = ZCAN_TransmitData(device, tran_data, 50);
    }

    // 获取通道 0 总线利用率, 5 次, 1000ms 一次
    for (int i = 0; i < 5; ++i) {
        BusUsage* pUse = (BusUsage*)prop->GetValue("0/get_bus_usage/1");
        if (NULL != pUse) {
            std::cout << "busload: " << std::dec << (int)pUse->nBusUsage << "%" << std::endl;
        }
        Sleep(1000);
    }

    // 清空队列发送
    prop->SetValue("0/clear_delay_send_queue", "0");
    system("pause");
#endif

    // 通道 0 发送 50 帧报文,使用 ZCAN_Transmit 接口发送
    {
        ZCAN_Transmit_Data trans_data[50] = {};
        for (int i = 0; i < 50; ++i){
            get_can_frame(trans_data[i], i);
        }
        int send_count = ZCAN_Transmit(ch[0], trans_data, 50);
        std::cout << "send frame: " << std::dec << send_count << std::endl;
    }

    // 构造 10 帧 CAN 报文(0 通道发送)以及 10 帧 CANFD 报文 (1 通道发送)
    // 使用 ZCAN_TransmitData 接口发送
    {
        ZCANDataObj trans_data[20] = {};
        for (int i = 0; i < 20; ++i)
        {
            int ch = i < 10 ? 0 : 1;
            bool is_fd = i < 10 ? false : true;
            get_can_canfd_frame(trans_data[i], ch, i + 0x100, is_fd);
        }
        int send_count = ZCAN_TransmitData(device, trans_data, 20);
        std::cout << "send frame: " << std::dec << send_count << std::endl;
    }

    Sleep(500);
    system("pause");

```

```

end:
    ReleaseIProperty(prop);
    g_thd_run = 0;

    if (thd_handle.joinable())
        thd_handle.join();
    std::cout << "Thread exited, close device" << std::endl;

    if (INVALID_DEVICE_HANDLE != device)
        ZCAN_CloseDevice(device);
    system("pause");

    return 0;
}

```

### 3.3.7 其他接口卡

本系列适用的设备：USBCAN-I、USBCAN-II、PCI9810、PCI9820、PCI5110、PCIe-9110I、PCI9820I、PCIE-9221、PCIe-9120I、PCI5121。

#### 1. 波特率

标准波特率		
项	值	说明
path	n/ baud_rate	n 代表通道号
value	【1000000，800000，500000，250000，125000，100000，50000】	单位 bps，如值为 500000 则表示波特率为 500k
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	
自定义波特率		
项	值	说明
path	n/ baud_rate_custom	n 代表通道号
value	自定义	通过 ZCANPRO 的波特率计算器计算
Get/Set	Set	
注意点	需在 ZCAN_InitCAN 之前设置	

#### 2. 示例代码

```

// 以下代码以 USBCAN-II 为例

#include "stdafx.h"
#include "zlscan.h"
#include <iostream>
#include <windows.h>

```

```

#include <thread>

#define CH_COUNT 2
bool g_thd_run = 1;

// 接收数据线程
void thread_task(CHANNEL_HANDLE handle)
{
    int nChnl = (unsigned int)handle & 0x000000FF;
    std::cout << "chnl: " << std::dec << nChnl << " thread run, handle:0x" << std::hex << handle << std::endl;
    ZCAN_Receive_Data data[100] = { };
    while (g_thd_run)
    {
        int count = ZCAN_GetReceiveNum(handle, 0);    // 获取 CAN 报文(参数 2:0 - CAN, 1 - CANFD)
        数量
        while (g_thd_run && count > 0)
        {
            int rcount = ZCAN_Receive(handle, data, 100, 10);
            for (int i = 0; i < rcount; ++i)
            {
                std::cout << "CHNL: " << std::dec << nChnl << " recv can ID: 0x" << std::hex <<
data[i].frame.can_id << std::endl;
            }
            count -= rcount;
        }
        Sleep(100);
    }

    std::cout << "chnl: " << std::dec << nChnl << " thread exit" << std::endl;
}

// 此函数仅用于构造示例 CAN 报文
void get_can_frame(ZCAN_Transmit_Data& can_data, canid_t id)
{
    memset(&can_data, 0, sizeof(can_data));
    can_data.frame.can_id = id;                // CAN ID
    can_data.frame.can_dlc = 8;                // CAN 数据长度 8
    can_data.transmit_type = 0;                // 正常发送
    for (int i = 0; i < 8; ++i) {              // 填充 CAN 报文 DATA
        can_data.frame.data[i] = i;
    }
}

int _tmain(int argc, _TCHAR* argv[])

```

```

{
    std::thread thd_handle;
    CHANNEL_HANDLE ch[CH_COUNT] = {};

    // 打开设备
    DEVICE_HANDLE device = ZCAN_OpenDevice(ZCAN_USBCAN2, 0, 0);
    if (INVALID_DEVICE_HANDLE == device) {
        std::cout << "open device failed" << std::endl;
        return 0;
    }

    // 获取 IProperty 指针，用于配置参数
    IProperty* prop = GetIProperty(device);
    if (NULL == prop) {
        std::cout << "get property failed" << std::endl;
        ZCAN_CloseDevice(device);
        return 0;
    }

    // 设置通道 0 自定义波特率为 1M，此处仅举例
    /*if (0 == prop->SetValue("0/ baud_rate_custom", "1.0Mbps(75%),(00,14)") {
        std::cout << "set ch0 custom baud rate failed" << std::endl;
        goto end;
    }*/

    // 循环设置、初始化、启动每个通道
    for (int i = 0; i < CH_COUNT; ++i) {
        char path[64] = {};
        sprintf_s(path, "%d/ baud_rate", i);
        // 设置波特率为 1M
        if (0 == prop->SetValue(path, "1000000")) {
            std::cout << "set baud rate failed" << std::endl;
            goto end;
        }
        // 初始化通道
        ZCAN_CHANNEL_INIT_CONFIG config;
        memset(&config, 0, sizeof(config));
        config.can_type = 0; // 0 - CAN, 1 - CANFD
        config.can.mode = 0; // 0 - 正常模式, 1 - 只听模式
        config.can.acc_code = 0x0;
        config.can.acc_mask = 0xFFFFFFFF; // 接收全部数据
        ch[i] = ZCAN_InitCAN(device, i, &config);
        if (INVALID_CHANNEL_HANDLE == ch[i]) {
            std::cout << "init channel failed!" << std::endl;

```



```

        goto end;
    }
    // 启动通道
    if (0 == ZCAN_StartCAN(ch[i])) {
        std::cout << "start channel failed" << std::endl;
        goto end;
    }
}

// 启动 CAN 通道 1 的接收线程
thd_handle = std::thread(thread_task, ch[1]);

// 通道 0 发送 10 帧报文
ZCAN_Transmit_Data trans_data[10] = {};
for (int i = 0; i < 10; ++i){
    get_can_frame(trans_data[i], i);
}
int send_count = ZCAN_Transmit(ch[0], trans_data, 10);
std::cout << "send frame: " << std::dec << send_count << std::endl;

system("pause");
end:
ReleaseIPProperty(prop);
g_thd_run = 0;
if (thd_handle.joinable())
    thd_handle.join();
std::cout << "thread exit, close device" << std::endl;
ZCAN_CloseDevice(device);

return 0;
}

```

## 第4章 附录

## 附录 1 - 设备类型定义

我司所有的 CAN 相关设备的类型号如表 4.1 所示。

表 4.1 设备类型号定义

产品型号	动态库中的设备名称	类型号
PCI-9810I	PCI9810	2
USBCAN-I/I+	USBCAN1	3
USBCAN-II/II+	USBCAN2	4
PCI-9820	PCI9820	5
CANET 系列的 UDP 工作方式	CANET-UDP	12
PCI-9840I	PCI9840	14
PC104-CAN2I	PC104-CAN2	15
PCI-9820I	PCI9820I	16
CANET 系列的 TCP 工作方式	CANET-TCP	17
PCI-5010-U	PCI-5010-U	19
USBCAN-E-U	USBCAN-E-U	20
USBCAN-2E-U	USBCAN-2E-U	21
PCI-5020-U	PCI-5020-U	22
PCIE-9221	PCIE-9221	24
CANWiFi-200T 的 TCP 工作方式	CANWIFI_TCP	25
CANWiFi-200T 的 UDP 工作方式	CANWIFI_UDP	26
PCIE-9120I	PCIE-9120I	27
PCIE-9110I	PCIE-9110I	28
PCIE-9140I	PCIE-9140I	29
USBCAN-4E-U	USBCAN-4E-U	31
CANDTU	CANDTU	32
USBCAN-8E-U	USBCAN-8E-U	34
CANDTU-NET	CANDTU-NET	36
CANDTU-100UR	CANDTU-100UR	37
PCIE-CANFD-100U	PCIE-CANFD-100U	38
PCIE-CANFD-200U	PCIE-CANFD-200U	39
PCIE-CANFD-400U	PCIE-CANFD-400U	40

USBCANFD-200U	USBCANFD-200U	41
USBCANFD-100U	USBCANFD-100U	42
USBCANFD-MINI	USBCANFD-MINI	43
CANSCOPE	CANSCOPE	45
CLOUD	CLOUD	46
CANDTU-NET-400	CANDTU-NET-400	47
CANFDNET-200U 的 TCP 工作方式	CANFDNET-TCP	48
CANFDNET-200U 的 UDP 工作方式	CANFDNET-UDP	49
CANFDWIFI 的 TCP 工作方式	CANFDWIFI-TCP	50
CANFDWIFI 的 UDP 工作方式	CANFDWIFI-UDP	51
CANFDNET-400U 的 TCP 工作方式	CANFDNET-400U -TCP	52
CANFDNET-400U 的 UDP 工作方式	CANFDNET-400U -UDP	53
CANFDNET-100U 的 TCP 工作方式	CANFDNET-100U -TCP	55
CANFDNET-100U 的 UDP 工作方式	CANFDNET-100U -UDP	56

## 附录 2 - 支持合并接收设备列表

我司支持合并接收设备列表如表 4.2 所示。

表 4.2 支持合并接收设备列表

产品系列	产品型号	动态库中的设备名称	类型号
USBCANFD 系列	USBCANFD-200U	USBCANFD-200U	41
	USBCANFD-100U	USBCANFD-100U	42
	USBCANFD-MINI	USBCANFD-MINI	43
CANFDNET 系列	CANFDNET-200U 的 TCP 工作方式	CANFDNET-TCP	48
	CANFDNET-200U 的 UDP 工作方式	CANFDNET-UDP	49
	CANFDNET-400U 的 TCP 工作方式	CANFDNET-400U-TCP	52
	CANFDNET-400U 的 UDP 工作方式	CANFDNET-400U-UDP	53
	CANFDNET-100U 的 TCP 工作方式	CANFDNET-100U-TCP	55
	CANFDNET-100U 的 UDP 工作方式	CANFDNET-100U-UDP	56
CANFDWIFI 系列	CANFDWIFI 的 TCP 工作方式	CANFDWIFI-TCP	50
	CANFDWIFI 的 UDP 工作方式	CANFDWIFI-UDP	51

## 附录 2 - 错误码定义

表 4.3 错误码定义

名称	值	描述
CAN 错误码		
ZCAN_ERROR_CAN_OVERFLOW	0x0001	CAN 控制器内部 FIFO 溢出
ZCAN_ERROR_CAN_ERRALARM	0x0002	CAN 控制器错误报警
ZCAN_ERROR_CAN_PASSIVE	0x0004	CAN 控制器消极错误
ZCAN_ERROR_CAN_LOSE	0x0008	CAN 控制器仲裁丢失
ZCAN_ERROR_CAN_BUSERR	0x0010	CAN 控制器总线错误
ZCAN_ERROR_CAN_BUSOFF	0x0020	CAN 控制器总线关闭
ZCAN_ERROR_CAN_BUFFER_OVERFLOW	0x0040	CAN 缓存溢出
通用错误码		
ZCAN_ERROR_DEVICEOPENED	0x0100	设备已经打开
ZCAN_ERROR_DEVICEOPEN	0x0200	打开设备错误
ZCAN_ERROR_DEVICENOTOPEN	0x0400	设备没有打开
ZCAN_ERROR_BUFFEROVERFLOW	0x0800	缓冲区溢出
ZCAN_ERROR_DEVICENOTEXIST	0x1000	此设备不存在
ZCAN_ERROR_LOADKERNELDLL	0x2000	装载动态库失败
ZCAN_ERROR_CMDFAILED	0x4000	执行命令失败错误码
ZCAN_ERROR_BUFFERCREATE	0x8000	内存不足