**zlgcan**

　　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。本章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。  
　　依赖 Microsoft Visual C++运行库版本（必须具备）： 2005、2008、2010、2012、2013 。  
　　如果发现打开设备失败，请根据软件架构，检查电脑里面的运行环境是否缺失。

**接口函数**

**DEMO里面的函数库不一定是最新的，使用DEMO前，请务必下载使用最新的函数库进行替换。**

* zlgcan二次开发接口函数库（20210108）[zlgcan\_dll.zip](http://manual.zlg.cn/Public/Uploads/2021-02-02/6018ed0909340.zip)

**例程DEMO**

zlgcan接口函数提供4种常用的例程演示，包含开发环境（语言）：C++、C#、Labview、python。

| **开发环境（语言）** | **DEMO** | **说明文档** |
| --- | --- | --- |
| C++ | [zlgcan\_c++.zip](http://manual.zlg.cn/Public/Uploads/2020-11-04/5fa22bb90c6da.zip) | 点击访问 |
| C# | [zlgcan\_c#.zip](http://manual.zlg.cn/Public/Uploads/2021-03-10/6048c4db5c27b.zip) | 点击访问 |
| LabVIEW | [zlgcan\_labview2013.zip](http://manual.zlg.cn/Public/Uploads/2021-02-02/6018eb5393984.zip) | [点击访问](https://manual.zlg.cn/web/#/149?page_id=5273) |
| Python | [zlgcan\_python.zip](http://manual.zlg.cn/Public/Uploads/2020-11-17/5fb337d3d6917.zip) | 点击访问 |

**注：如果单击无法下载，请试下右击——链接另存为**

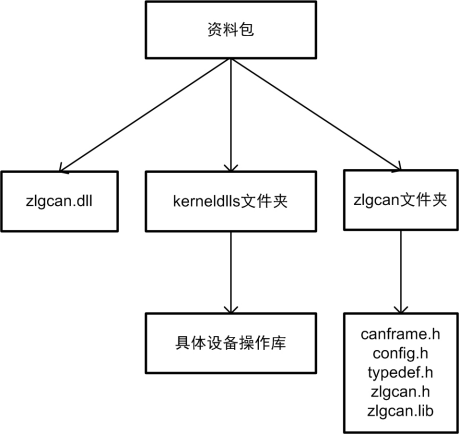
# ZLGCAN 接口函数简介

　　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。

　　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。

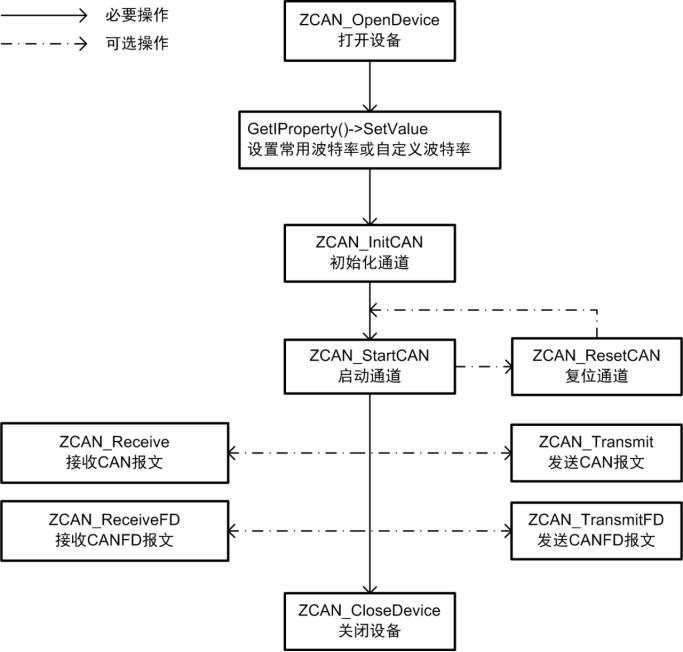


**图１ 资料包结构**

　　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生成目录下。

**开发流程图**

　　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1所示。



**图1 CAN卡开发流程图**

　　开发代码示例：

1. //ZCAN\_USBCAN\_2E\_U为设备类型, 请根据实际修改
2. DEVICE\_HANDLE dhandle = ZCAN\_OpenDevice(ZCAN\_USBCAN\_2E\_U, 0, 0);
3. if (INVALID\_DEVICE\_HANDLE == dhandle)
4. {
5. std::cout << "打开设备失败" << std::endl;
6. return 0;
7. }
8. IProperty\* property = GetIProperty(dhandle);
9. if (NULL == property)
10. {
11. std::cout << "属性指针为空" << std::endl;
12. goto end;
13. }
14. //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
15. //若为CANFD设备, 设置冲裁域波特率的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设备设置波特率的区别。
16. if (property->SetValue("0/baud\_rate", "1000000") != STATUS\_OK)
17. {
18. std::cout << "设置波特率失败" << std::endl;
19. goto end;
20. }
21. ZCAN\_CHANNEL\_INIT\_CONFIG cfg;
22. memset(&cfg, 0, sizeof(cfg));
23. cfg.can\_type = TYPE\_CAN;//CANFD设备为TYPE\_CANFD
24. cfg.can.filter = 0;
25. cfg.can.mode = 0; //正常模式, 1为只听模式
26. cfg.can.acc\_code = 0;
27. cfg.can.acc\_mask = 0xffffffff;
28. CHANNEL\_HANDLE chHandle = ZCAN\_InitCAN(dhandle, 0, &cfg);
29. if (INVALID\_CHANNEL\_HANDLE == chHandle)
30. {
31. std::cout << "初始化通道失败" << std::endl;
32. goto end;
33. }
34. if (ZCAN\_StartCAN(chHandle) != STATUS\_OK)
35. {
36. std::cout << "启动通道失败" << std::endl;
37. goto end;
38. }
39. ZCAN\_Transmit\_Data frame;
40. memset(&frame, 0, sizeof(frame));
41. frame.frame.can\_id = MAKE\_CAN\_ID(0x100, 1, 0, 0);
42. frame.frame.can\_dlc = 8;
43. BYTE data[] = {1, 2, 3, 4, 5, 6, 7, 8};
44. memcpy(frame.frame.data, data, sizeof(data));
45. if (ZCAN\_Transmit(chHandle, &frame, 1) != 1)
46. {
47. std::cout << "发送数据失败" << std::endl;
48. goto end;
49. }
50. end:
51. ReleaseIProperty(property);
52. ZCAN\_CloseDevice(dhandle);

## 数据结构定义

### ZCAN\_DEVICE\_INFO

　　该结构体包含设备的一些基本信息，在函数ZCAN\_GetDeviceInf中被填充。

1. typedef struct tagZCAN\_DEVICE\_INFO {
2. USHORT hw\_Version;
3. USHORT fw\_Version;
4. USHORT dr\_Version;
5. USHORT in\_Version;
6. USHORT irq\_Num;
7. BYTE can\_Num;
8. UCHAR str\_Serial\_Num[20];
9. UCHAR str\_hw\_Type[40];
10. USHORT reserved[4];
11. }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

　　该结构体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。

1. typedef struct tagZCAN\_CHANNEL\_INIT\_CONFIG {
2. UINT can\_type; // 0:can 1:canfd
3. union
4. {
5. struct
6. {
7. UINT acc\_code;
8. UINT acc\_mask;
9. UINT reserved;
10. BYTE filter;
11. BYTE timing0;
12. BYTE timing1;
13. BYTE mode;
14. }can;
15. struct
16. {
17. UINT acc\_code;
18. UINT acc\_mask;
19. UINT abit\_timing;
20. UINT dbit\_timing;
21. UINT brp;
22. BYTE filter;
23. BYTE mode;
24. USHORT pad;
25. UINT reserved;
26. }canfd;
27. };
28. }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）。

注意：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

* **CANFD设备**  
  **acc\_code**  
  验收码，同CAN设备。

**cc\_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。

注意：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。

### ZCAN\_CHANNEL\_ERROR\_INFO

　　该结构体包含总线错误信息，在函数ZCAN\_ReadChannelErrInfo中被填充。

1. typedef struct tagZCAN\_CHANNEL\_ERROR\_INFO {
2. UINT error\_code;
3. BYTE passive\_ErrData[3];
4. BYTE arLost\_ErrData;
5. } ZCAN\_CHANNEL\_ERROR\_INFO;

**成员**

**error\_code**  
　　错误码。

**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

　　该结构体包含控制器状态信息，在函数ZCAN\_ReadChannelStatus中被填充。

1. typedef struct tagZCAN\_CHANNEL\_STATUS {
2. BYTE errInterrupt;
3. BYTE regMode;
4. BYTE regStatus;
5. BYTE regALCapture;
6. BYTE regECCapture;
7. BYTE regEWLimit;
8. BYTE regRECounter;
9. BYTE regTECounter;
10. UINT Reserved;
11. }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**  
　　仅作保留，不设置。

### can\_frame

　　该结构体包含了CAN报文信息。

1. struct can\_frame {
2. canid\_t can\_id; /\* 32 bit CAN\_ID + EFF/RTR/ERR flags \*/
3. \_\_u8 can\_dlc; /\* frame payload length in byte (0 .. CAN\_MAX\_DLEN) \*/
4. \_\_u8 \_\_pad; /\* padding \*/
5. \_\_u8 \_\_res0; /\* reserved / padding \*/
6. \_\_u8 \_\_res1; /\* reserved / padding \*/
7. \_\_u8 data[CAN\_MAX\_DLEN]/\* \_\_attribute\_\_((aligned(8)))\*/;
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

　　该结构体包含了CANFD报文信息。

1. struct canfd\_frame {
2. canid\_t can\_id; /\* 32 bit CAN\_ID + EFF/RTR/ERR flags \*/
3. \_\_u8 len; /\* frame payload length in byte \*/
4. \_\_u8 flags; /\* additional flags for CAN FD,i.e error code \*
5. \_\_u8 \_\_res0; /\* reserved / padding \*/
6. \_\_u8 \_\_res1; /\* reserved / padding \*/
7. \_\_u8 data[CANFD\_MAX\_DLEN]/\* \_\_attribute\_\_((aligned(8)))\*/;
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。

**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

　　该结构体包含发送的CAN报文信息，在函数ZCAN\_Transmit中使用。

1. typedef struct tagZCAN\_Transmit\_Data
2. {
3. can\_frame frame;
4. UINT transmit\_type;
5. }ZCAN\_Transmit\_Data;

**成员**

**frame**  
　　报文数据信息，参见can\_frame。  
　　  
　　**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

　　该结构体包含发送的CANFD报文信息，在函数ZCAN\_TransmitFD中使用。

1. typedef struct tagZCAN\_TransmitFD\_Data
2. {
3. canfd\_frame frame;
4. UINT transmit\_type;
5. }ZCAN\_TransmitFD\_Data;

**成员**

**frame**  
　　报文数据信息，参见canfd\_frame。  
　　  
　　**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\_Receive\_Data

　　该结构体包含接收的CAN报文信息，在函数ZCAN\_Receive中使用。

1. typedef struct tagZCAN\_Receive\_Data
2. {
3. can\_frame frame;
4. UINT64 timestamp;
5. }ZCAN\_Receive\_Data;

**成员**

**frame**  
　　报文数据信息，详见can\_frame结构体。

**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

　　该结构体包含接收的CANFD报文信息，在函数ZCAN\_ReceiveFD中使用。

1. typedef struct tagZCAN\_ReceiveFD\_Data
2. {
3. canfd\_frame frame;
4. UINT64 timestamp;
5. }ZCAN\_ReceiveFD\_Data;

**成员**

**frame**  
　　报文数据信息，详见canfd\_frame结构体。

**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\_AUTO\_TRANSMIT\_OBJ

　　该结构体包含定时发送CAN参数信息。

1. typedef struct tagZCAN\_AUTO\_TRANSMIT\_OBJ{
2. USHORT enable;
3. USHORT index;
4. UINT interval;//定时发送时间。单位毫秒
5. ZCAN\_Transmit\_Data obj;//报文
6. }ZCAN\_AUTO\_TRANSMIT\_OBJ, \*PZCAN\_AUTO\_TRANSMIT\_O

**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。

### ZCANFD\_AUTO\_TRANSMIT\_OBJ

　　该结构体包含定时发送CANFD参数信息。

1. typedef struct tagZCANFD\_AUTO\_TRANSMIT\_OBJ{
2. USHORT enable;
3. USHORT index;
4. UINT interval;
5. ZCAN\_TransmitFD\_Data obj;//报文
6. }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。

### ZCLOUD\_DEVINFO

　　该结构体包含云设备的属性信息，在ZCLOUD\_GetUserData中被填充。

1. typedef struct tagZCLOUD\_DEVINFO
2. {
3. int devIndex;
4. char type[64];
5. char id[64];
6. char owner[64];
7. char model[64];
8. char fwVer[16];
9. char hwVer[16];
10. char serial[64];
11. BYTE canNum;
12. int status;
13. BYTE bCanUploads[16];
14. BYTE bGpsUpload;
15. }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

　　该结构体包含用户信息，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。

1. typedef struct tagZCLOUD\_USER\_DATA
2. {
3. char username[64];
4. char mobile[64];
5. ZCLOUD\_DEVINFO devices[ZCLOUD\_MAX\_DEVICES];
6. size\_t devCnt;
7. }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。

**devCnt**  
　　设备个数。

### ZCLOUD\_GPS\_FRAME

　　该结构体包含设备GPS数据，通过ZCLOUD\_ReceiveGPS获取。

1. typedef struct tagZCLOUD\_GPS\_FRAME
2. {
3. float latitude;
4. float longitude;
5. float speed;
6. struct \_\_gps\_time {
7. USHORT year;
8. USHORT mon;
9. USHORT day;
10. USHORT hour;
11. USHORT min;
12. USHORT sec;
13. }tm;
14. } ZCLOUD\_GPS\_FRAME;

**成员**

**latitude**  
　　纬度。

**longitude**  
　　经度。

**speed**  
　　速度。

**tm**  
　　时间结构。

### USBCANFDTxTimeStamp

　　该结构体用于获取发送帧的时间戳。

1. typedef struct tagUSBCANFDTxTimeStamp
2. {
3. UINT\* pTxTimeStampBuffer; //allocated by user, size:nBufferTimeStampCount\*4,unit:100us
4. UINT nBufferTimeStampCount; //buffer size
5. }USBCANFDTxTimeStamp;

**成员**

**pTxTimeStampBuffer**  
　　用户申请的内存，用于存放返回的时间戳，大小为nBufferTimeStampCount\*4。

**nBufferTimeStampCount**  
　　用户申请内存的大小，可以存放时间戳的个数。

### IProperty

　　该结构体用于获取/设置设备参数信息。

1. typedef struct tagIProperty
2. {
3. SetValueFunc SetValue;
4. GetValueFunc GetValue;
5. GetPropertysFunc GetPropertys;
6. }IProperty;

**成员**

**SetValue**  
　　设置设备属性值，详见属性表。

**GetValue**  
　　获取属性值。

**GetPropertys**  
　　用于返回设备包含的所有属性。

　　IProperty示例代码：

1. char path[50] = {0};
2. char value[50] = {0};
3. IProperty\* property\_ = GetIProperty(device\_handle);// device\_handle为设备句柄
4. sprintf\_s(path, "%d/canfd\_abit\_baud\_rate", 0);//0代表通道0
5. sprintf\_s(value, "%d", 1000000);//1Mbps为1000000
6. if (0 == property\_->SetValue(path, value))
7. {
8. return FALSE;
9. }

### ZCAN\_LIN\_MSG

　　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。

1. typedef struct \_VCI\_LIN\_MSG{
2. BYTE ID;
3. BYTE DataLen;
4. USHORT Flag;
5. UINT TimeStamp;
6. BYTE Data[8];
7. }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

　　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。

1. typedef struct \_VCI\_LIN\_INIT\_CONFIG
2. {
3. BYTE linMode;
4. BYTE linFlag;
5. USHORT reserved;
6. UINT linBaud;
7. }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**

1. #define LIN\_MODE\_MASTER 0
2. #define LIN\_MODE\_SLAVE 1
3. #define LIN\_FLAG\_CHK\_ENHANCE 0x01
4. #define LIN\_FLAG\_VAR\_DLC 0x02

### ZCANCANFDData

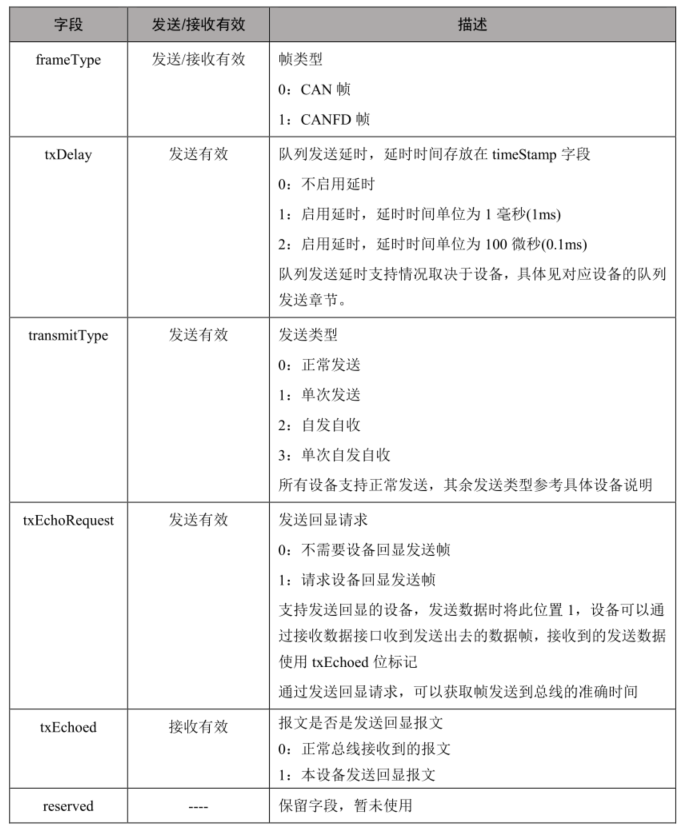
　　结构体详情见程序清单 3.22，该结构体表示 CAN/CANFD 帧结构，可以表示发送接收CAN/CANFD 帧，目前仅作为 ZCANDataObj 结构的成员使用

1. typedef struct tagZCANCANFDData
2. {
3. UINT64 timeStamp;
4. union
5. {
6. struct{
7. UINT frameType : 2;
8. UINT txDelay : 2;
9. UINT transmitType : 4;
10. UINT txEchoRequest : 1;
11. UINT txEchoed : 1;
12. UINT reserved : 22;
13. }unionVal;
14. UINT rawVal;
15. }flag;
16. BYTE extraData[4];
17. canfd\_frame frame;
18. }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 字段的含义如表所示



**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

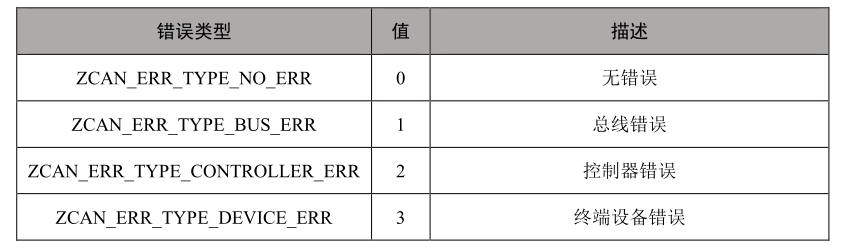
结构体详情见程序清单，该结构体表示错误信息结构，可以表示总线错误、控制器错误、设备端错误等错误信息，目前仅作为 ZCANDataObj 结构的成员使用

1. typedef struct tagZCANErrorData
2. {
3. UINT64 timeStamp;
4. BYTE errType;
5. BYTE errSubType;
6. BYTE nodeState;
7. BYTE rxErrCount;
8. BYTE txErrCount;
9. BYTE errData;
10. BYTE reserved[2];
11. }ZCANErrorData;

成员

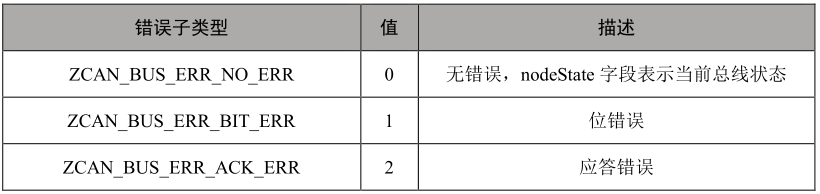
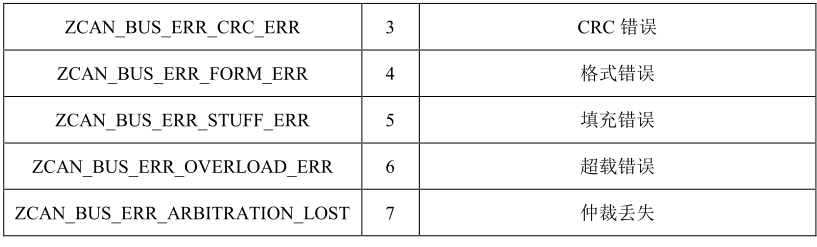
**timeStamp**  
时间戳，表示错误产生的时间，时间单位为微秒(us)。

**errType**  
错误类型，错误类型对应的数值

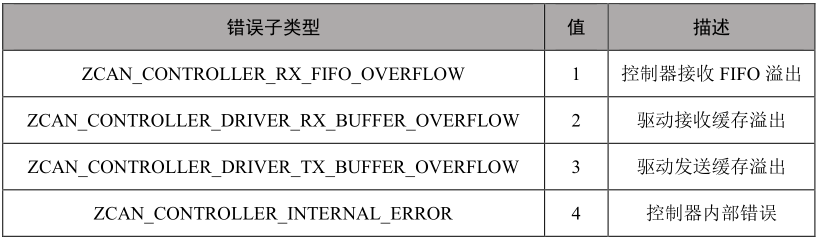


**errSubType**  
错误子类型，错误子类型的值根据错误类型不同表示不用的含义。总线错误的错误子类型如表所示：

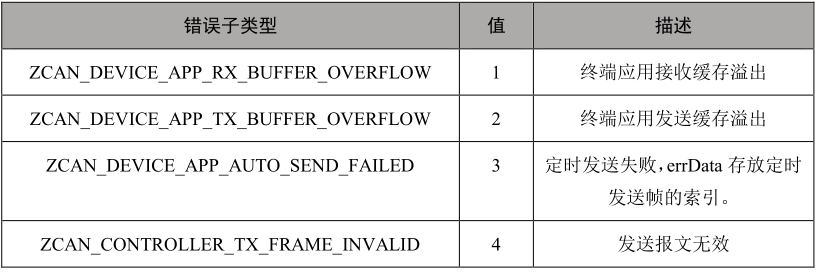
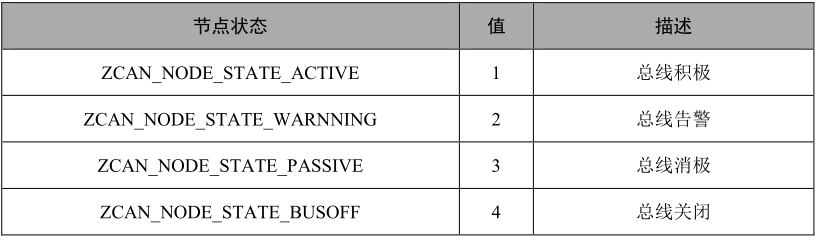
**总线错误子类型**

**控制器错误子类型**



**终端设备错误子类型**

  
  
**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。节点状态含义如表所示。  
  
  
  
**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，该结构体表示 GPS 数据，目前仅作为 ZCANDataObj 结构的成员使用

**ZCANGPSData 结构体成员**

1. typedef struct tagZCANGPSData
2. {
3. struct {
4. USHORT year;
5. USHORT mon;
6. USHORT day;
7. USHORT hour;
8. USHORT min;
9. USHORT sec;
10. USHORT milsec;
11. } time;
12. union{
13. struct{
14. USHORT timeValid : 1;
15. USHORT latlongValid : 1;
16. USHORT altitudeValid : 1;
17. USHORT speedValid : 1;
18. USHORT courseAngleValid : 1;
19. USHORT reserved:13;
20. }unionVal;
21. USHORT rawVal;
22. }flag;
23. double latitude;
24. double longitude;
25. double altitude;
26. double speed;
27. double courseAngle;
28. } 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，具体标志如表所示。

**GPS 数据标志字段**



**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，该结构体表示 LIN 数据结构，目前仅作为 ZCANDataObj结构的成员使用

**ZCANLINData 结构体成员**

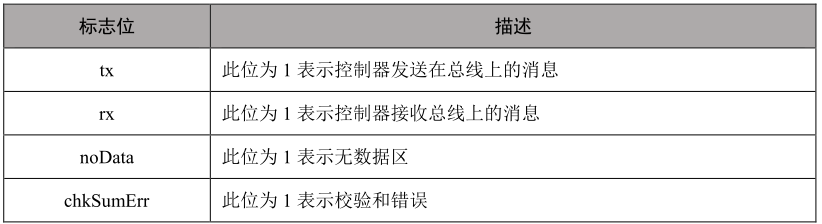
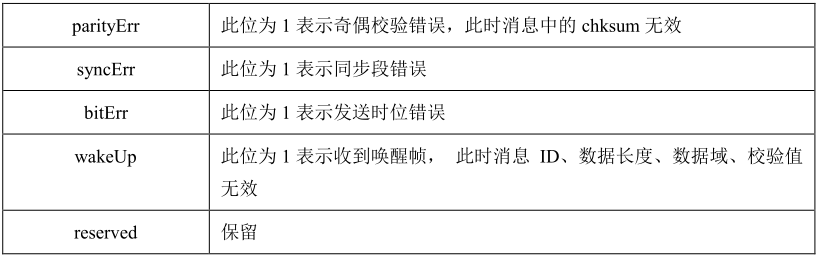
1. typedef struct tagZCANLINData
2. {
3. UINT64 timeStamp;
4. union {
5. struct {
6. BYTE ID:6;
7. BYTE Parity:2;
8. }unionVal;
9. BYTE rawVal;
10. } PID;
11. BYTE dataLen;
12. union{
13. struct{
14. USHORT tx : 1;
15. USHORT rx : 1;
16. USHORT noData : 1;
17. USHORT chkSumErr : 1;
18. USHORT parityErr : 1;
19. USHORT syncErr : 1;
20. USHORT bitErr : 1;
21. USHORT wakeUp : 1;
22. USHORT reserved : 8;
23. }unionVal;
24. USHORT rawVal;
25. }flag;
26. BYTE chkSum;
27. BYTE reserved[3];
28. BYTE data[8];
29. }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所示

**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。

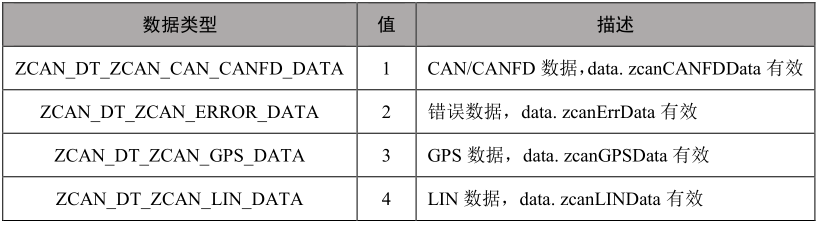
### ZCANDataObj

结构体详情见程序清单，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

1. typedef struct tagZCANDataObj
2. {
3. BYTE dataType;
4. BYTE chnl;
5. union{
6. struct{
7. USHORT reserved : 16;
8. }unionVal;
9. USHORT rawVal;
10. }flag;
11. BYTE extraData[4];
12. union
13. {
14. ZCANCANFDData zcanCANFDData;
15. ZCANErrorData zcanErrData;
16. ZCANGPSData zcanGPSData;
17. ZCANLINData zcanLINData;
18. BYTE raw[92];
19. } data;
20. }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

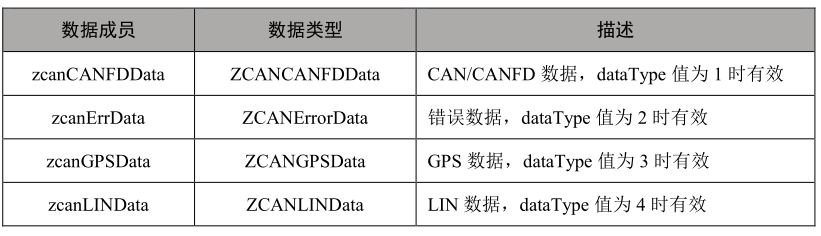


**chnl**  
数据通道。数据类型表示CAN/CANFD/错误数据时，通道表示的是CAN通道。数据类  
型表示的是 LIN 数据时，通道表示的是设备的 LIN 通道。

**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说明



* [函数说明](https://manual.zlg.cn/web/#%E5%87%BD%E6%95%B0%E8%AF%B4%E6%98%8E)
  + [ZCAN\_OpenDevice](https://manual.zlg.cn/web/#ZCAN_OpenDevice)
  + [ZCAN\_CloseDevice](https://manual.zlg.cn/web/#ZCAN_CloseDevice)
  + [ZCAN\_GetDeviceInf](https://manual.zlg.cn/web/#ZCAN_GetDeviceInf)
  + [ZCAN\_IsDeviceOnLine](https://manual.zlg.cn/web/#ZCAN_IsDeviceOnLine)
  + [ZCAN\_InitCAN](https://manual.zlg.cn/web/#ZCAN_InitCAN)
  + [ZCAN\_StartCAN](https://manual.zlg.cn/web/#ZCAN_StartCAN)
  + [ZCAN\_ResetCAN](https://manual.zlg.cn/web/#ZCAN_ResetCAN)
  + [ZCAN\_ClearBuffer](https://manual.zlg.cn/web/#ZCAN_ClearBuffer)
  + [ZCAN\_ReadChannelErrInfo](https://manual.zlg.cn/web/#ZCAN_ReadChannelErrInfo)
  + [ZCAN\_ReadChannelStatus](https://manual.zlg.cn/web/#ZCAN_ReadChannelStatus)
  + [ZCAN\_Transmit](https://manual.zlg.cn/web/#ZCAN_Transmit)
  + [ZCAN\_TransmitFD](https://manual.zlg.cn/web/#ZCAN_TransmitFD)
  + [ZCAN\_TransmitData](https://manual.zlg.cn/web/#ZCAN_TransmitData)
  + [ZCAN\_GetReceiveNum](https://manual.zlg.cn/web/#ZCAN_GetReceiveNum)
  + [ZCAN\_Receive](https://manual.zlg.cn/web/#ZCAN_Receive)
  + [ZCAN\_ReceiveFD](https://manual.zlg.cn/web/#ZCAN_ReceiveFD)
  + [ZCAN\_ReceiveData](https://manual.zlg.cn/web/#ZCAN_ReceiveData)
  + [GetIProperty](https://manual.zlg.cn/web/#GetIProperty)
  + [ReleaseIProperty](https://manual.zlg.cn/web/#ReleaseIProperty)
* [云设备接口库函数](https://manual.zlg.cn/web/#%E4%BA%91%E8%AE%BE%E5%A4%87%E6%8E%A5%E5%8F%A3%E5%BA%93%E5%87%BD%E6%95%B0)
  + [ZCLOUD\_SetServerInfo](https://manual.zlg.cn/web/#ZCLOUD_SetServerInfo)
  + [ZCLOUD\_ConnectServer](https://manual.zlg.cn/web/#ZCLOUD_ConnectServer)
  + [ZCLOUD\_IsConnected](https://manual.zlg.cn/web/#ZCLOUD_IsConnected)
  + [ZCLOUD\_DisconnectServer](https://manual.zlg.cn/web/#ZCLOUD_DisconnectServer)
  + [ZCLOUD\_GetUserData](https://manual.zlg.cn/web/#ZCLOUD_GetUserData)
  + [ZCLOUD\_ReceiveGPS](https://manual.zlg.cn/web/#ZCLOUD_ReceiveGPS)
* [LIN接口函数](https://manual.zlg.cn/web/#LIN%E6%8E%A5%E5%8F%A3%E5%87%BD%E6%95%B0)
  + [ZCAN\_InitLIN](https://manual.zlg.cn/web/#ZCAN_InitLIN)
  + [ZCAN\_StartLIN](https://manual.zlg.cn/web/#ZCAN_StartLIN)
  + [ZCAN\_ResetLIN](https://manual.zlg.cn/web/#ZCAN_ResetLIN)
  + [ZCAN\_TransmitLIN](https://manual.zlg.cn/web/#ZCAN_TransmitLIN)
  + [ZCAN\_GetLINReceiveNum](https://manual.zlg.cn/web/#ZCAN_GetLINReceiveNum)
  + [ZCAN\_ReceiveLIN](https://manual.zlg.cn/web/#ZCAN_ReceiveLIN)
  + [ZCAN\_SetLINSlaveMsg](https://manual.zlg.cn/web/#ZCAN_SetLINSlaveMsg)
  + [ZCAN\_ClearLINSlaveMsg](https://manual.zlg.cn/web/#ZCAN_ClearLINSlaveMsg)

**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。

1. DEVICE\_HANDLE ZCAN\_OpenDevice(
2. UINT device\_type,
3. UINT device\_index,
4. 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**  
　　仅作保留。

**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。

1. 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。

1. UINT ZCAN\_GetDeviceInf(
2. DEVICE\_HANDLE device\_handle,
3. 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结构体。

**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。

1. 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。

1. CHANNEL\_HANDLE ZCAN\_InitCAN(
2. DEVICE\_HANDLE device\_handle,
3. UINT can\_index,
4. 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结构体。

**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。

1. 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。

1. 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。

1. 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。

1. UINT ZCAN\_ReadChannelErrInfo(
2. CHANNEL\_HANDLE channel\_handle,
3. 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 结构体。

**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。

1. UINT ZCAN\_ReadChannelStatus(
2. CHANNEL\_HANDLE channel\_handle,
3. 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 结构体。

**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。

1. UINT ZCAN\_Transmit(
2. CHANNEL\_HANDLE channel\_handle,
3. 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。

1. UINT ZCAN\_TransmitFD(
2. CHANNEL\_HANDLE channel\_handle,
3. ZCAN\_TransmitFD\_Data\* pTransmit,
4. 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

1. 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。

## ZCAN\_GetReceiveNum

　　获取缓冲区中CAN或CANFD报文数目。

1. UINT ZCAN\_GetReceiveNum(
2. CHANNEL\_HANDLE channel\_handle,
3. BYTE type);

**参数**  
　　  
　　**channel\_handle**  
　　通道句柄值。

**type**  
　　获取CAN或CANFD报文，0=CAN，1=CANFD。  
　　  
**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。

1. UINT ZCAN\_Receive(
2. CHANNEL\_HANDLE channel\_handle,
3. ZCAN\_Receive\_Data\* pReceive,
4. UINT len,
5. 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，默认值为-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。

1. UINT ZCAN\_ReceiveFD(
2. CHANNEL\_HANDLE channel\_handle,
3. ZCAN\_ReceiveFD\_Data\* pReceive,
4. UINT len,
5. 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，建议使用ZCAN\_GetReceiveNum 确保缓冲区有数据再使用

1. UINT ZCAN\_ReceiveData (DEVICE\_HANDLE device\_handle, ZCANDataObj\* pReceive, UINT len, INTwait\_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的情况下，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**

## GetIProperty

　　该函数返回属性配置接口。

1. 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结构体，返回空则表示操作失败。

## ReleaseIProperty

　　释放属性接口，与GetIProperty结对使用。

1. 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。

# 云设备接口库函数

## ZCLOUD\_SetServerInfo

　　该函数用于设置云服务器相关连接信息。

1. void ZCLOUD\_SetServerInfo(
2. const char\* httpSvr,
3. unsigned short httpPort,
4. const char\* mqttSvr,
5. 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。

1. UINT ZCLOUD\_ConnectServer(
2. const char\* username,
3. 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。

1. 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。

1. 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。

1. 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。

1. UINT ZCLOUD\_ReceiveGPS(
2. DEVICE\_HANDLE device\_handle,
3. ZCLOUD\_GPS\_FRAME\* pReceive,
4. UINT len,
5. 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。

# LIN接口函数

　　目前只有CANDTU-200UR，CANDTU-200UWG，CANDTU-200UWGR型号支持。

## 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。

1. CHANNEL\_HANDLE FUNC\_CALL ZCAN\_InitLIN(
2. DEVICE\_HANDLE device\_handle,
3. UINT can\_index,
4. 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结构体。

**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。

1. 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。

1. 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。

1. ULONG FUNC\_CALL ZCAN\_TransmitLIN(
2. CHANNEL\_HANDLE channel\_handle,
3. PZCAN\_LIN\_MSG pSend,
4. 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。

1. ULONG FUNC\_CALL ZCAN\_GetLINReceiveNum(CHANNEL\_HANDLE channel\_handle);

**参数**

**channel\_handle**  
　　通道句柄值。

**返回值**  
　　返回接收报文数目。

## 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。

1. ULONG FUNC\_CALL ZCAN\_ReceiveLIN(
2. CHANNEL\_HANDLE channel\_handle,
3. PZCAN\_LIN\_MSG pReceive,
4. ULONG Len,
5. 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。

1. UINT FUNC\_CALL ZCAN\_SetLINSlaveMsg(
2. CHANNEL\_HANDLE channel\_handle,
3. PZCAN\_LIN\_MSG pSend,
4. 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。

1. UINT FUNC\_CALL ZCAN\_ClearLINSlaveMsg(
2. CHANNEL\_HANDLE channel\_handle,
3. BYTE\* pLINID,
4. 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。

# 错误码定义

| **名称** | **值** | **描述** |
| --- | --- | --- |
| 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 |
| 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 |

# 设备类型号

| **CAN卡型号** | **设备类型号** |
| --- | --- |
| ZCAN\_PCI5121 | 1 |
| ZCAN\_PCI9810 | 2 |
| ZCAN\_USBCAN1 | 3 |
| ZCAN\_USBCAN2 | 4 |
| ZCAN\_PCI9820 | 5 |
| ZCAN\_CAN232 | 6 |
| ZCAN\_PCI5110 | 7 |
| ZCAN\_CANLITE | 8 |
| ZCAN\_ISA9620 | 9 |
| ZCAN\_ISA5420 | 10 |
| ZCAN\_PC104CAN | 11 |
| ZCAN\_CANETUDP | 12 |
| ZCAN\_CANETE | 12 |
| ZCAN\_DNP9810 | 13 |
| ZCAN\_PCI9840 | 14 |
| ZCAN\_PC104CAN2 | 15 |
| ZCAN\_PCI9820I | 16 |
| ZCAN\_CANETTCP | 17 |
| ZCAN\_PCIE\_9220 | 18 |
| ZCAN\_PCI5010U | 19 |
| ZCAN\_USBCAN\_E\_U | 20 |
| ZCAN\_USBCAN\_2E\_U | 21 |
| ZCAN\_PCI5020U | 22 |
| ZCAN\_EG20T\_CAN | 23 |
| ZCAN\_PCIE9221 | 24 |
| ZCAN\_WIFICAN\_TCP | 25 |
| ZCAN\_WIFICAN\_UDP | 26 |
| ZCAN\_PCIe9120 | 27 |
| ZCAN\_PCIe9110 | 28 |
| ZCAN\_PCIe9140 | 29 |
| ZCAN\_USBCAN\_4E\_U | 31 |
| ZCAN\_CANDTU\_200UR | 32 |
| ZCAN\_CANDTU\_MINI | 33 |
| ZCAN\_USBCAN\_8E\_U | 34 |
| ZCAN\_CANREPLAY | 35 |
| ZCAN\_CANDTU\_NET | 36 |
| ZCAN\_CANDTU\_100UR | 37 |
| ZCAN\_PCIE\_CANFD\_100U | 38 |
| ZCAN\_PCIE\_CANFD\_200U | 39 |
| ZCAN\_PCIE\_CANFD\_400U | 40 |
| ZCAN\_USBCANFD\_200U | 41 |
| ZCAN\_USBCANFD\_100U | 42 |
| ZCAN\_USBCANFD\_MINI | 43 |
| ZCAN\_CANFDCOM\_100IE | 44 |
| ZCAN\_CANSCOPE | 45 |
| ZCAN\_CLOUD | 46 |
| ZCAN\_CANDTU\_NET\_400 | 47 |
| ZCAN\_CANFDNET\_TCP | 48 |
| ZCAN\_CANFDNET\_UDP | 49 |
| ZCAN\_CANFDWIFI\_TCP | 50 |
| ZCAN\_CANFDWIFI\_UDP | 51 |
| ZCAN\_CANFDNET\_400U\_TCP | 52 |
| ZCAN\_CANFDNET\_400U\_UDP | 53 |
| ZCAN\_CANFDBLUE\_200U | 54 |
| ZCAN\_CANFDNET\_100U\_TCP | 55 |
| ZCAN\_CANFDNET\_100U\_UDP | 56 |

# 设备属性表

　 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：

1. /\*\*
2. \* \brief 设置指定路径的属性的值。
3. \* \param[in] path : 属性的路径。
4. \* \param[in] value : 属性的值。
5. \* \retval 成功返回 1，失败返回 0。
6. \*/
7. typedef int (\*SetValueFunc)(const char\* path, const char\* value);
8. /\*\*
9. \* \brief 获取指定路径的属性的值。
10. \* \param[in] path : 属性的路径。
11. \* \retval 成功返回属性的值，失败返回 NULL。
12. \*/
13. typedef const char\* (\*GetValueFunc)(const char\* path);
14. typedef struct tagIProperty
15. {
16. SetValueFunc SetValue;
17. GetValueFunc GetValue;
18. GetPropertysFunc GetPropertys;
19. }IProperty;

## USBCANFD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：需要在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：需要在ZCAN\_InitCAN之前设置

### 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：需要在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 |
| Get/Set | Set |  |

注意点：需要在ZCAN\_InitCAN之前设置

### 5.终端电阻

　　USBCANFD 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-不使能 1-使能 |  |
| Get/Set | Set |  |

注意点：需要在ZCAN\_InitCAN之前设置

### 6.发送超时时间

　　设备发送CAN(FD)报文时的超时时间，单位毫秒，默认为15000ms。  
　　注：通道进入BUSOFF也会导致发送立即返回，不会等设置的超时时间。

| **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：在打开设备以后任意位置设置都会生效

### 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 端仍可调用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。

**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：需要在ZCAN\_StartCAN之后设置

**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：需要在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：需要在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：需要在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

**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.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是必须的，也可以试任意数字 |
| value | 无 |  |
| Get/Set | Get |  |

### 10.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：需要再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：需要再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：需要再ZCAN\_InitCAN之后设置

**滤波生效（全部滤波ID段同时生效）**

| **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：需要再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：需要再ZCAN\_InitCAN之后设置

### 11.示例代码

1. // 以下代码以 USBCANFD-mini 设备型号为例
2. #include "stdafx.h"
3. #include "zlgcan.h"
4. #include <iostream>
5. #include <windows.h>
6. #include <thread>
7. // 此函数仅用于构造示例 CAN 报文
8. void get\_can\_frame(ZCAN\_Transmit\_Data& can\_data, canid\_t id)
9. {
10. memset(&can\_data, 0, sizeof(can\_data));
11. can\_data.frame.can\_id = id; // CAN ID
12. can\_data.frame.can\_dlc = 8; // CAN 数据长度 8
13. can\_data.transmit\_type = 0; // 正常发送
14. for (int i = 0; i < 8; ++i) { // 填充 CAN 报文 DATA
15. can\_data.frame.data[i] = i;
16. }
17. }
18. // 此函数仅用于构造示例 CANFD 报文
19. void get\_canfd\_frame(ZCAN\_TransmitFD\_Data& canfd\_data, canid\_t id)
20. {
21. memset(&canfd\_data, 0, sizeof(canfd\_data));
22. canfd\_data.frame.can\_id = id; // CANFD ID
23. canfd\_data.frame.len = 64; // CANFD 数据长度 64
24. canfd\_data.transmit\_type = 0; // 正常发送
25. for (int i = 0; i < 64; ++i) { // 填充 CANFD 报文 DATA
26. canfd\_data.frame.data[i] = i;
27. }
28. }
29. // 此函数仅用于构造示例队列发送 CAN 报文
30. void get\_can\_frame\_queue(ZCAN\_Transmit\_Data& can\_data, canid\_t id, UINT delay)
31. {
32. memset(&can\_data, 0, sizeof(can\_data));
33. can\_data.frame.can\_id = id; // CAN ID
34. can\_data.frame.can\_dlc = 8; // CAN 数据长度 8
35. can\_data.frame.\_\_pad |= TX\_DELAY\_SEND\_FLAG; // 设置延时标志位
36. can\_data.frame.\_\_res0 = LOBYTE(delay); // 帧间隔低字节
37. can\_data.frame.\_\_res1 = HIBYTE(delay); // 帧间隔高字节
38. can\_data.transmit\_type = 0; // 正常发送
39. for (int i = 0; i < 8; ++i) { // 填充 CAN 报文 DATA
40. can\_data.frame.data[i] = i;
41. }
42. }
43. bool g\_thd\_run = 1; // 线程运行标记
44. // 线程函数，用于接收报文
45. void thread\_task(CHANNEL\_HANDLE handle)
46. {
47. int nChnl = (unsigned int)handle & 0x000000FF;
48. std::cout << "chnl: " << std::dec << nChnl << " thread run, handle:0x" << std::hex << handle << std::endl;
49. ZCAN\_Receive\_Data data[100] = {};
50. ZCAN\_ReceiveFD\_Data fd\_data[100] = {};
51. while (g\_thd\_run)
52. {
53. int count = ZCAN\_GetReceiveNum(handle, 0); // 获取 CAN 报文（参数 2：0 - CAN，1 - CANFD）
54. 数量
55. while (g\_thd\_run && count > 0)
56. {
57. int rcount = ZCAN\_Receive(handle, data, 100, 10);
58. for (int i = 0; i < rcount; ++i)
59. {
60. std::cout << "CHNL: " << std::dec << nChnl << " recv can ID: 0x" << std::hex <<
61. data[i].frame.can\_id << std::endl;
62. }
63. count -= rcount;
64. }
65. count = ZCAN\_GetReceiveNum(handle, 1); // 获取 CANFD 报文（参数 2：0 - CAN，1 - CANFD）
66. 数量
67. while (g\_thd\_run && count > 0)
68. {
69. int rcount = ZCAN\_ReceiveFD(handle, fd\_data, 100, 10);
70. for (int i = 0; i < rcount; ++i)
71. {
72. std::cout << "CHNL: " << std::dec << nChnl << " recv canfd ID: 0x" << std::hex <<
73. fd\_data[i].frame.can\_id << std::endl;
74. }
75. count -= rcount;
76. }
77. Sleep(100);
78. }
79. std::cout << "chnl: " << std::dec << nChnl << " thread exit" << std::endl;
80. }
81. int \_tmain(int argc, \_TCHAR\* argv[])
82. {
83. UINT dev\_type = ZCAN\_USBCANFD\_MINI;
84. std::thread thd\_handle;
85. // 打开设备
86. DEVICE\_HANDLE device = ZCAN\_OpenDevice(dev\_type, 0, 0);
87. if (INVALID\_DEVICE\_HANDLE == device) {
88. std::cout << "open device failed!" << std::endl;
89. return 0;
90. }
91. // 获取 IProperty 指针，用于配置参数
92. IProperty\* prop = GetIProperty(device);
93. if (NULL == prop) {
94. std::cout << "get property failed" << std::endl;
95. ZCAN\_CloseDevice(device);
96. return 0;
97. }
98. // 设置通道 0 CANFD 标准为 ISO;
99. if (0 == prop->SetValue("0/canfd\_standard", "0")) {
100. std::cout << "set canfd standard failed" << std::endl;
101. goto end;
102. }
103. // 设置通道 0 仲裁域波特率为 1M
104. if (0 == prop->SetValue("0/canfd\_abit\_baud\_rate", "1000000")) {
105. std::cout << "set abit baud rate failed" << std::endl;
106. goto end;
107. }
108. // 设置通道 0 数据域波特率为 5M
109. if (0 == prop->SetValue("0/canfd\_dbit\_baud\_rate","5000000")) {
110. std::cout << "set dbit baud rate failed" << std::endl;
111. goto end;
112. }
113. // 设置通道 0 自定义波特率，此处仅演示调用方式
114. /\*if (0 ==
115. prop->SetValue("0/baud\_rate\_custom","1.0Mbps(75%),5.0Mbps(75%),(60,0080040D,00800001)")) {
116. std::cout << "set custom baud rate failed" << std::endl;
117. goto end;
118. }\*/
119. // 初始化通道 0
120. ZCAN\_CHANNEL\_INIT\_CONFIG config;
121. memset(&config, 0, sizeof(config));
122. config.can\_type = 1; // 0 - CAN，1 - CANFD
123. config.can.mode = 0; // 0 - 正常模式，1 - 只听模式
124. CHANNEL\_HANDLE channel\_0 = ZCAN\_InitCAN(device, 0, &config);
125. if (INVALID\_CHANNEL\_HANDLE == channel\_0) {
126. std::cout << "init channel 0 failed!" << std::endl;
127. goto end;
128. }
129. // 使能通道 0 终端电阻
130. if (0 == prop->SetValue("0/initenal\_resistance", "1")) {
131. std::cout << "enable terminal resistance failed" << std::endl;
132. goto end;
133. }
134. // 设置通道 0 发送超时时间为 100ms
135. if (0 == prop->SetValue("0/tx\_timeout","100")) {
136. std::cout << "set send timeout failed" << std::endl;
137. goto end;
138. }
139. // 设置通道 0 自定义序列号为 abc
140. prop->SetValue("0/set\_cn", "abc");
141. const char\* pRet = prop->GetValue("0/get\_cn/1");
142. std::cout << "sn: " << pRet << std::endl;
143. // 设置第一组滤波，只接收 ID 范围在 0x100-0x200 之间的标准帧
144. prop->SetValue("0/filter\_mode", "0"); // 标准帧
145. prop->SetValue("0/filter\_start", "0x100"); // 起始 ID
146. prop->SetValue("0/filter\_end", "0x200"); // 结束 ID
147. // 设置第二组滤波，只接收 ID 范围在 0x1FFFF-0x2FFFF 之间的扩展帧
148. prop->SetValue("0/filter\_mode", "1"); // 扩展帧
149. prop->SetValue("0/filter\_start", "0x1FFFF"); // 起始 ID
150. prop->SetValue("0/filter\_end", "0x2FFFF"); // 结束 ID
151. // 使能滤波
152. prop->SetValue("0/filter\_ack", "0");
153. // 清除滤波,此处仅举例，何时调用用户自由决定
154. //prop->SetValue("0/filter\_clear", "0");
155. // 启动 CAN 通道 0
156. if (0 == ZCAN\_StartCAN(channel\_0)) {
157. std::cout << "start channel 0 failed" << std::endl;
158. goto end;
159. }
160. // 启动 CAN 通道 0 的接收线程
161. thd\_handle = std::thread(thread\_task, channel\_0);
162. /\* 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
163. 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
164. 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 \*/
165. ZCAN\_AUTO\_TRANSMIT\_OBJ auto\_can;
166. ZCANFD\_AUTO\_TRANSMIT\_OBJ auto\_canfd;
167. ZCAN\_AUTO\_TRANSMIT\_OBJ\_PARAM delay\_param;
168. memset(&auto\_can, 0, sizeof(auto\_can));
169. auto\_can.index = 0; // 定时列表索引 0
170. auto\_can.enable = 1; // 使能此索引，每条可单独设置
171. auto\_can.interval = 100; // 定时发送间隔 100ms
172. get\_can\_frame(auto\_can.obj, 0); // 构造 CAN 报文
173. prop->SetValue("0/auto\_send", (const char\*)&auto\_can); // 设置定时发送
174. memset(&auto\_can, 0, sizeof(auto\_can));
175. auto\_can.index = 1; // 定时列表索引 1
176. auto\_can.enable = 1; // 使能此索引，每条可单独设置
177. auto\_can.interval = 200; // 定时发送间隔 200ms
178. get\_can\_frame(auto\_can.obj, 1); // 构造 CAN 报文
179. prop->SetValue("0/auto\_send", (const char\*)&auto\_can); // 设置定时发送
180. memset(&auto\_canfd, 0, sizeof(auto\_canfd));
181. auto\_canfd.index = 2; // 定时列表索引 2
182. auto\_canfd.enable = 1; // 使能此索引，每条可单独设置
183. auto\_canfd.interval = 500; // 定时发送间隔 500ms
184. get\_canfd\_frame(auto\_canfd.obj, 2); // 构造 CANFD 报文
185. prop->SetValue("0/auto\_send\_canfd", (const char\*)&auto\_canfd); // 设置定时发送
186. memset(&auto\_canfd, 0, sizeof(auto\_canfd));
187. auto\_canfd.index = 3; // 定时列表索引 3
188. auto\_canfd.enable = 1; // 使能此索引，每条可单独设置
189. auto\_canfd.interval = 600; // 定时发送间隔 600ms
190. get\_canfd\_frame(auto\_canfd.obj, 3); // 构造 CANFD 报文
191. prop->SetValue("0/auto\_send\_canfd", (const char\*)&auto\_canfd); // 设置定时发送
192. // 设置索引 1 的 CAN 帧延时 1s 启动发送
193. delay\_param.index = 1; // 索引 1
194. delay\_param.type = 1; // 固定值 1
195. delay\_param.value = 1000; // 延时 1000ms
196. prop->SetValue("0/auto\_send\_param", (const char\*)&delay\_param); // 设置发送延时
197. prop->SetValue("0/apply\_auto\_send", "0"); // 使能定时发送
198. Sleep(5000); // 等待发送 5s
199. prop->SetValue("0/clear\_auto\_send", "0"); // 清除定时发送
200. // 设置队列发送模式
201. prop->SetValue("0/set\_send\_mode", "1");
202. // 获取队列发送可用缓冲
203. int free\_count = \*((int\*)prop->GetValue("0/get\_device\_available\_tx\_count/1"));
204. // 构造 50 条报文，报文 ID 从 0 递增，帧间隔 10ms 递增
205. if (free\_count > 50) {
206. ZCAN\_Transmit\_Data tran\_data[50] = {};
207. for (int i = 0; i < 50; ++i) {
208. get\_can\_frame\_queue(tran\_data[i], i, i \* 10);
209. }
210. int ret\_count = ZCAN\_Transmit(channel\_0, tran\_data, 50);
211. }
212. // 5 秒后清空队列发送
213. Sleep(5000);
214. prop->SetValue("0/clear\_delay\_send\_queue", "0");
215. Sleep(1000);
216. // 正常发送 10 帧报文
217. ZCAN\_Transmit\_Data trans\_data[10] = {};
218. for (int i = 0; i < 10; ++i){
219. get\_can\_frame(trans\_data[i], i);
220. }
221. int send\_count = ZCAN\_Transmit(channel\_0, trans\_data, 10);
222. std::cout << "send frame: " << std::dec << send\_count << std::endl;
223. system("pause");
224. end:
225. ReleaseIProperty(prop);
226. g\_thd\_run = 0;
227. if (thd\_handle.joinable())
228. thd\_handle.join();
229. std::cout << "thread exit, close device" << std::endl;
230. ZCAN\_CloseDevice(device);
231. system("pause");
232. return 0;
233. }