SysMaxAutomation®

CANable Z(PRO)用户手册

CANable Z(PRO)

用户手册

V1.3

修订历史

版本	修改内容	日期
V1.0	初始版本	2020-03-02
V1.1	修正通讯示例代码	2020-03-23
V1.2	更新 linux 下 cangaroo 内容	2020-10-31
V1.3	增加 BusMaster 软件使用介绍	2020-02-17

目录

第1章 特性与概述

1.1 系列型号

型号	CAN 版本	CAN 通道数	CAN 最大比特率	电气隔离	最大接收帧数
CANable Z	CAN 2.0B	1	1Mb/s	N/A	15000 帧/s
CANable Z PRO	CAN 2.0B	1	1Mb/s	2.5KV rms	15000 帧/s
USB-CAN-II	CAN 2.0B	2	1Mb/s	2.5KV rms	
USB-CAN-FD	CAN FD	1	8Mb/s	5KV rms	

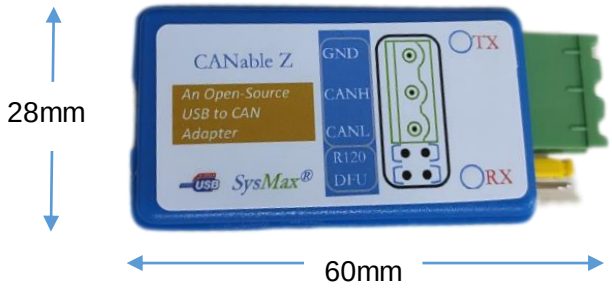
1.2 基本特性

CANable Z(PRO)是一款低成本的，简单好用的 USB-CAN 转换模块，支持 Windos、Linux、树莓派等系统。USB 通信基于 USB buck 传输，保证了高速通信下的速度和稳定，并且在 Windos、Linux 系统均无需安装驱动。因为我们在 Windows 系统适配了微软自带 WCID(**W**indows **C**ompatible **ID**)驱动。在 Linux 系统适配了 socketcan 接口，您可以直接使用 can-utils 工具进行操作。

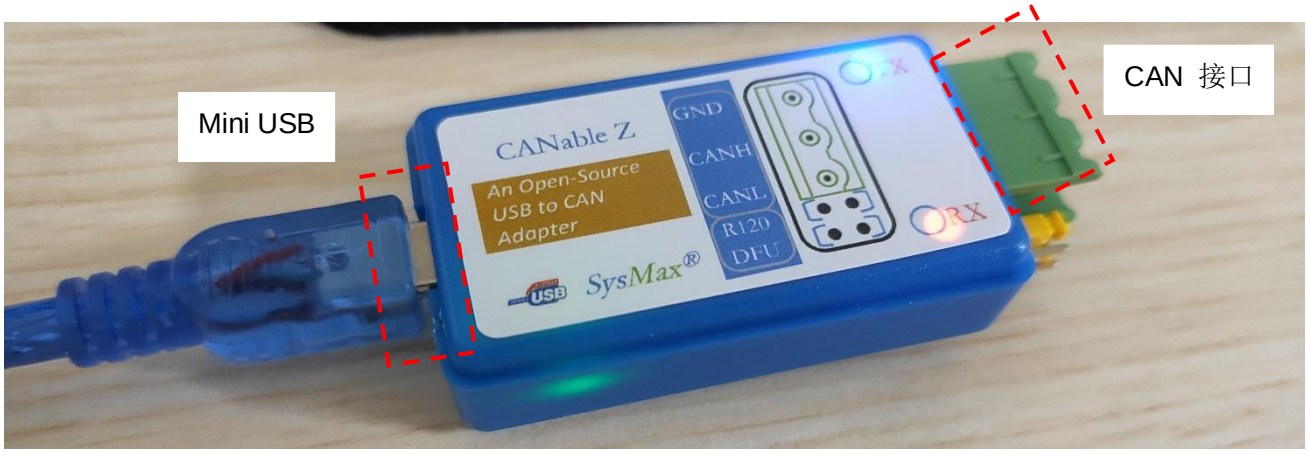
CANable Z PRO 是 ***CANable Z*** 的升级版本，使用了高度集成的全隔离芯片 2.5KV rms 信号和 1500VDC 的电源隔离，同时增加了更多的硬件保护电路，抗干扰能力更强，适合工业调试应用或电机类应用。功能和软件支持与 ***CANable Z*** 完全兼容。

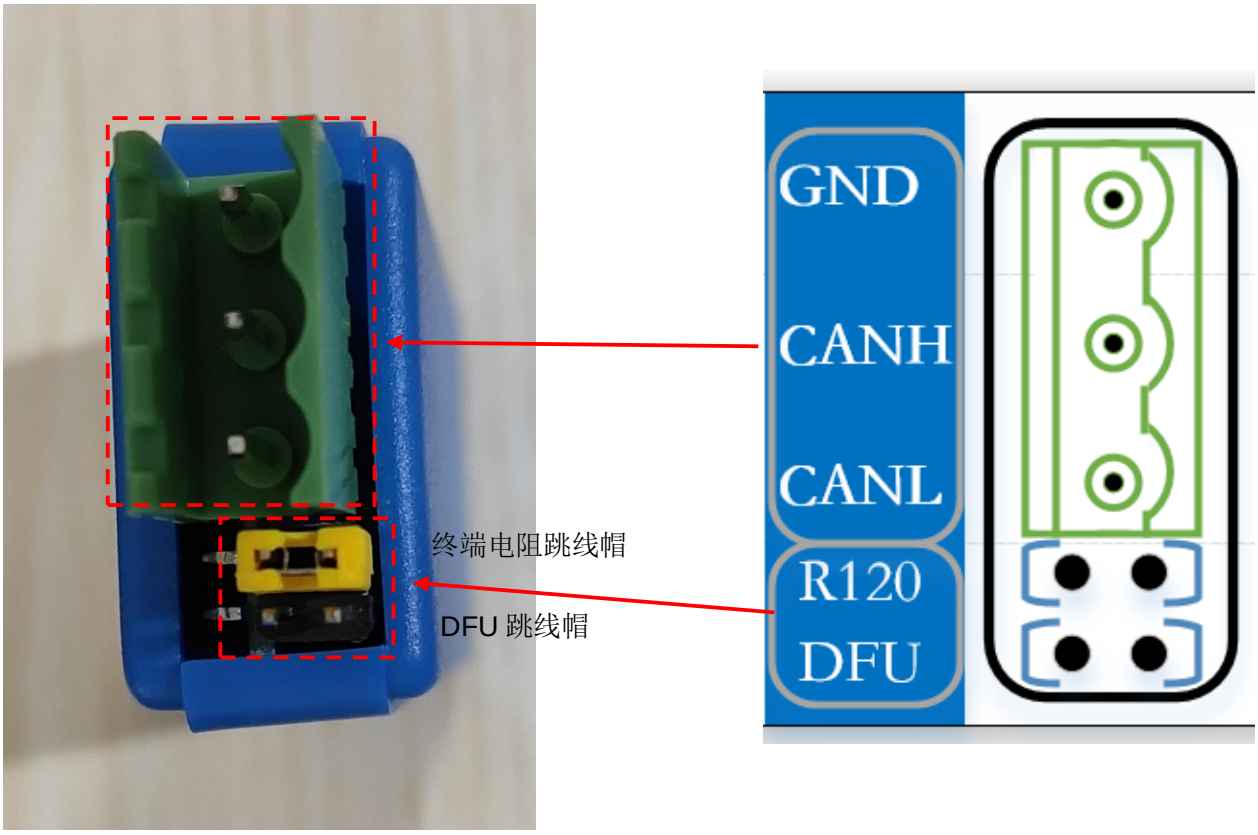
- CPU 32bit Cortex-M0
- 电源 LED 指示灯，通信 LED 指示灯
- 采用 USB 接口供电和通信
- 内部自恢复保险丝，防止损坏主机 USB 口
- USB 与 CAN BUS 间采用 2.5KVrms 信号和电源隔离（***CANable Z PRO only***）
- 螺旋接线端子 CANH, CANL
- 2.54 跳线帽 BOOT 选择
- 2.54 跳线帽 CAN 终端电阻选择
- USB2.0 Full Speed (12Mbps)
- 支持最大 32 个设备同时连接电脑工作
- 支持测试模式：Silent、Loopback、One-shot
- 支持用户自定义 CAN 波特率和采样点
- 支持 CAN 2.0A (11-bit ID) 和 2.0B (29-bit ID) 最大波特率 1Mbps
- 配套的调试软件 cangaroo
- 提供 Windows 二次开发 dll 和 demo (Qt)
- 提供 Python 二次开发包，支持 Windows、Linux、OS X、树莓派、Windows CE、Android

1.3 外形尺寸



1.4 CANable Z 接口介绍





1.4.1 CAN

CAN 接口采用 KF2EDGK 3PIN 接口，间距 5.08MM。
接线：CAN_H 和 CAN_L 为必接，GND 为选接。
注：不接 GND 不影响正常通讯，带屏蔽层的线缆可将屏蔽层接 GND。

1.4.2 指示灯

- 侧面绿色指示灯：电源
- 红色指示灯：数据发送指示灯
- 蓝色指示灯：数据接收指示灯

注：CANable Z 第一版贴纸印刷错误，TX/RX 印字标反，以本文档为准。

1.4.3 终端电阻及 BOOT 跳线帽

- R120：CAN 总线 120Ω 终端电阻跳线帽，短接后接入 CANable 内部 120Ω 电阻。（出厂默认短接）
- DFU：USB 固件升级跳线帽。短接后重新插上 USB 口，CANable 会进入固件升级模式。非固件升级使用请勿短接跳线。

1.5 CANable Z PRO 接口介绍

1.5.1 CAN

CAN 接口采用 KF2EDGK 5PIN 接口，间距 5.08MM。
接线：CAN_H 和 CAN_L 为必接，GND 为选接。
注：不接 GND 不影响正常通讯，带屏蔽层的线缆可将屏蔽层接 GND。

1.5.2 指示灯

- PWR：电源指示灯
- TX：数据发送指示灯
- RX：数据接收指示灯

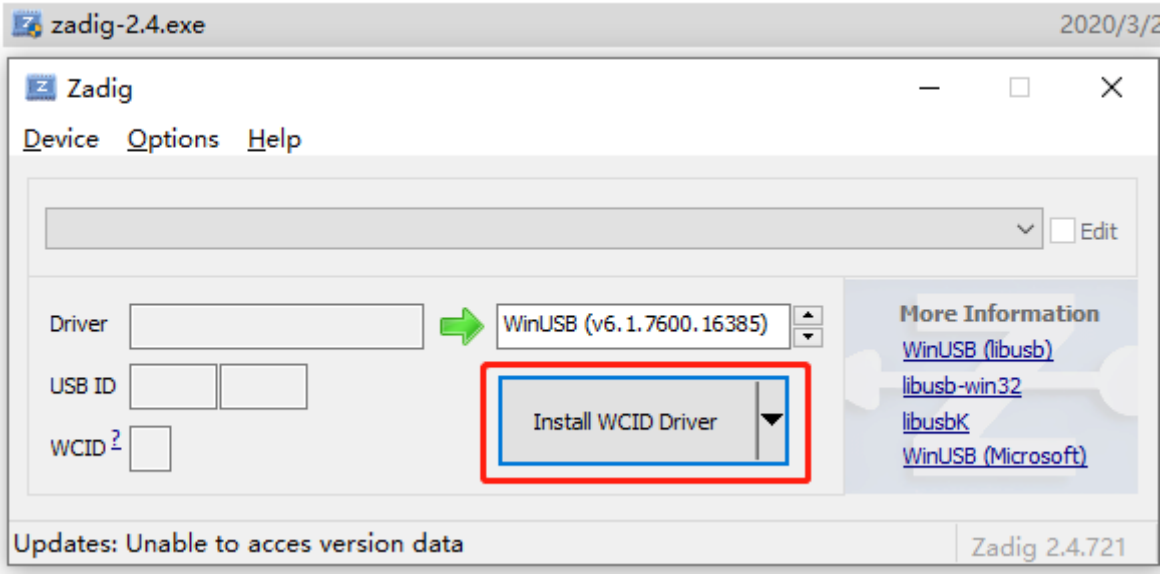
1.5.3 终端电阻及 BOOT 跳线帽

- TRES：CAN 总线 120Ω 终端电阻跳线帽，短接后接入 CANable 内部 120Ω 电阻。（出厂默认短接）
- DFU：USB 固件升级跳线帽。短接后重新插上 USB 口，CANable 会进入固件升级模式。非固件升级使用请勿短接 DFU 跳线。

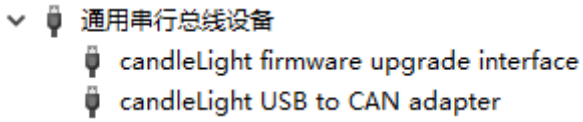
第2章 调试软件

2.1 驱动安装

1. Win8 以上的版本均不需要额外安装驱动，插上 USB 口后系统会自动识别。
2. 由于 win8 以前的系统默认不支持 WCID(**W**indows**C**ompatible **ID**)，所以模块接入电脑后显示黄色感叹号，需要使用工具软件 Zadig 自动安装一下 WCID 驱动。



3. 驱动识别成功后，设备管理器中可以查看到 candleLight 设备如下图：



2.2 cangaroo 简介

cangaroo 是 ***CANable&CANable _pro*** 专用的、简单易用的 can 总线调试软件，麻雀虽小五脏俱全，对于一般的 can 调试开发完全够用，而且专门针对汽车逆向工程增加了通过 can id 分类接收到的 can 数据帧，并且当某个 id 的数据帧活跃时会进行高亮显示以便于观察分析。

- 支持 Windos/Linux (Ubuntu 发行版本)
- 最大支持同时接入 32 个 ***CANable&CANable _pro*** 模块
- 数据帧时间戳
- 支持数据日志保存
- 支持 CAN DBC 文件协议解析
- 软件绿色免安装

2.3 cangaroo 使用说明

2.3.1 将 CANable Z 或 CANable Z_pro 连接电脑

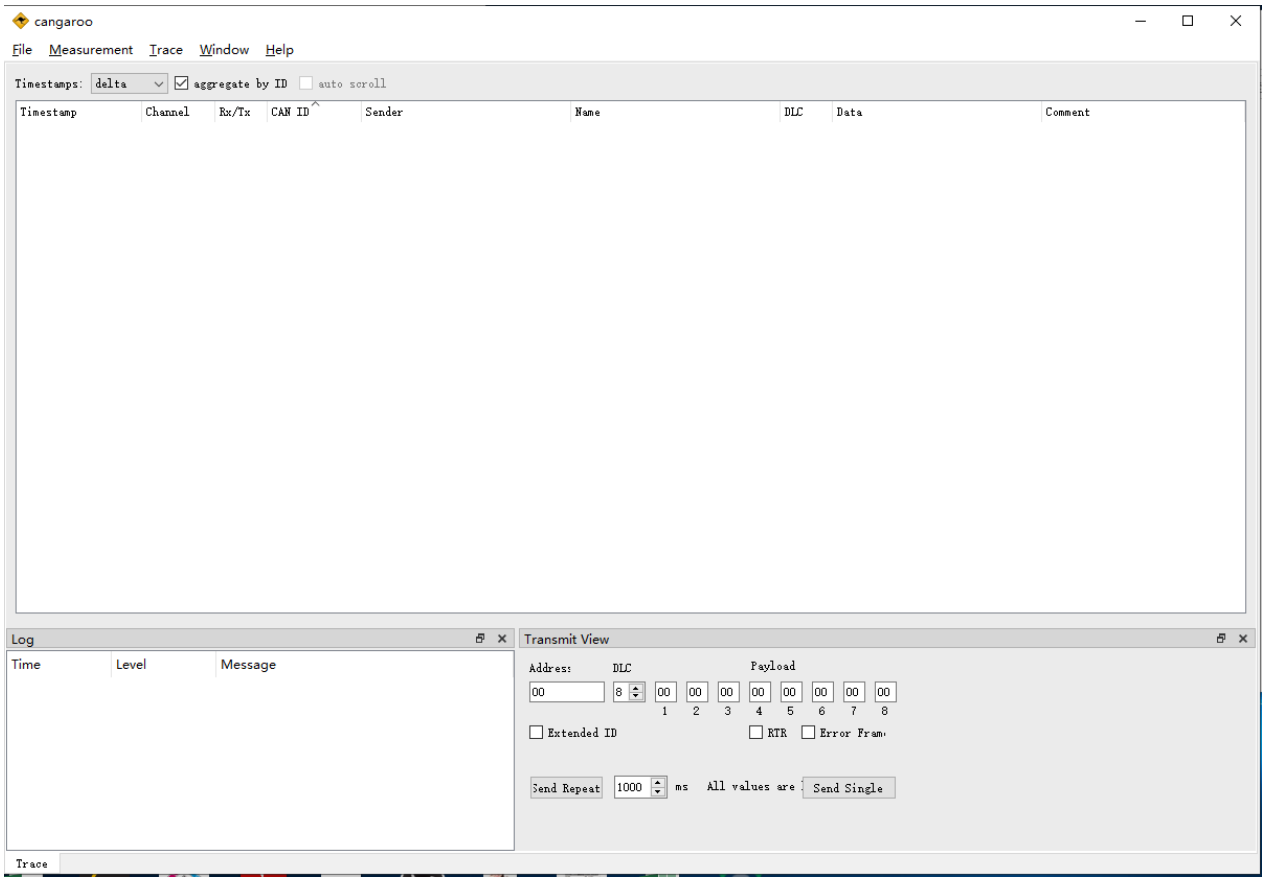
使用 usb 数据线连接 ***CANable Z*** 或 ***CANable Z_pro*** 和电脑，PWR 指示灯常亮。

2.3.2 运行 cangaroo

2.3.2.1 Windows

1. 解压 cangaroo_X_X_X_winXX.zip
2. 在 cangaroo_X_X_X_winXX 文件夹中，双击 cangaroo.exe 运行软件。

Windows 启动后界面：



2.3.2.2 Ubuntu

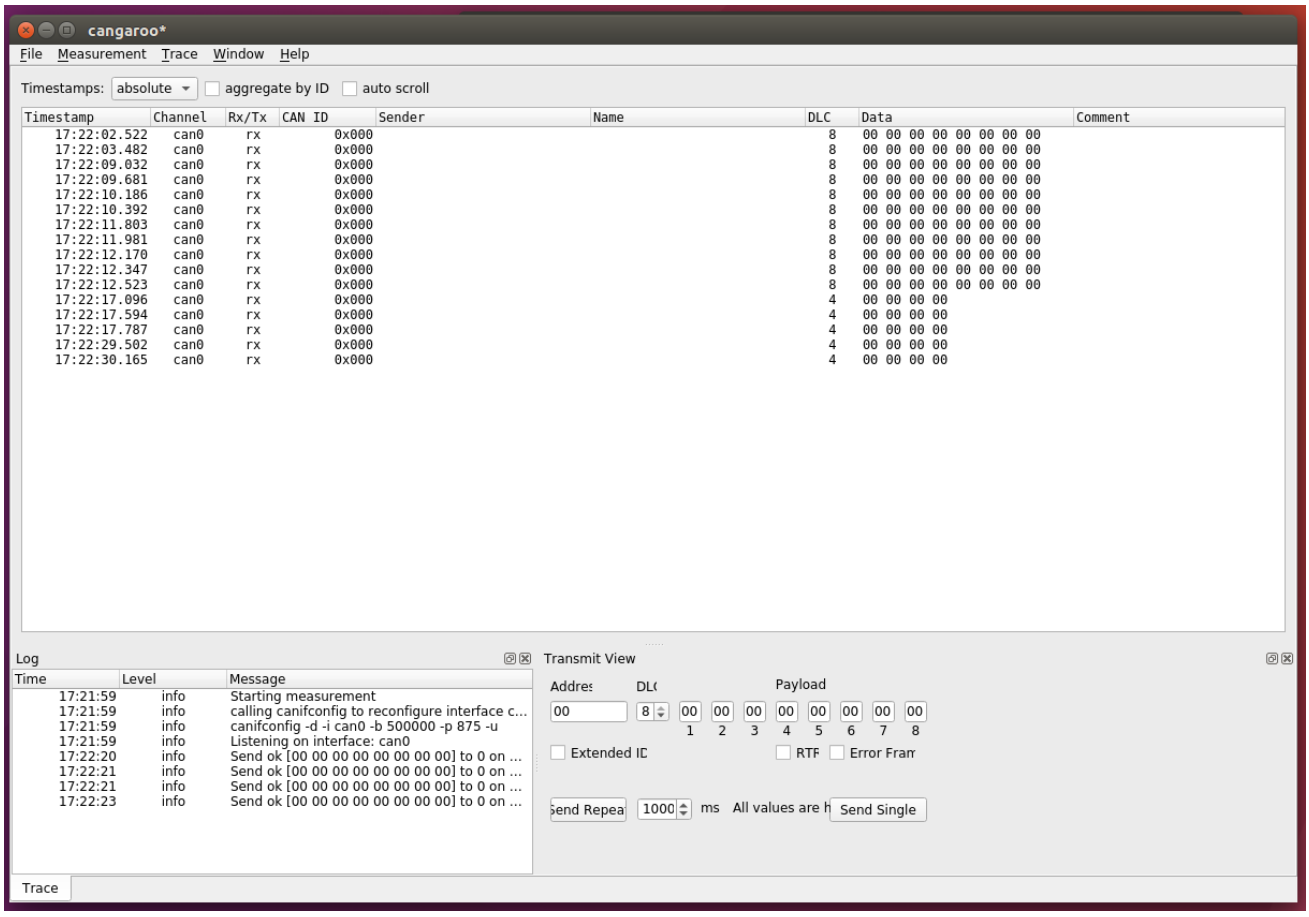
下载 cangaroo_x_x_x_ubuntu.zip 完成后解压生成 cangaroo_x_x_x_ubuntu.zip 文件夹。
在命令行输入以下命令回车进行 libnl 库的安装。

```
sudo apt-get install libnl-route-3-dev
```

进入 cangaroo_x_x_x_ubuntu 文件夹下，将 canifconfig、cangaroo 文件增加可**执行权限**，然后以**管理员权限**运行 cangaroo。

```
sudo ./cangaroo
```

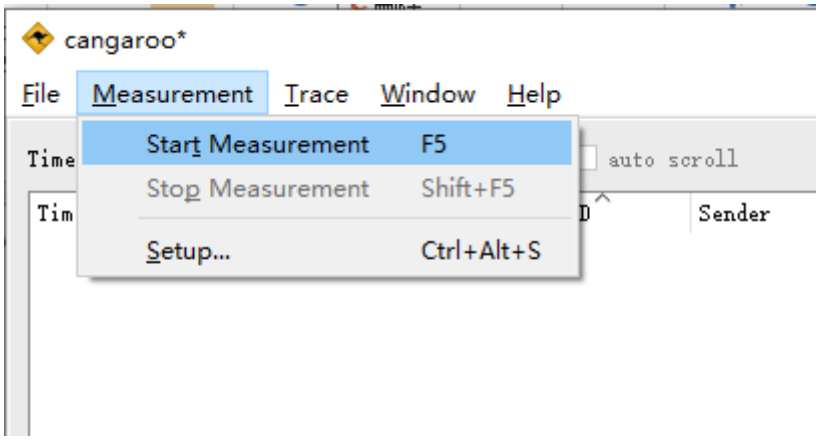
Ubuntu 下运行界面：



2.3.3 启动 CANable Z 或 CANable Z_pro

cangaroo 启动后的界面：

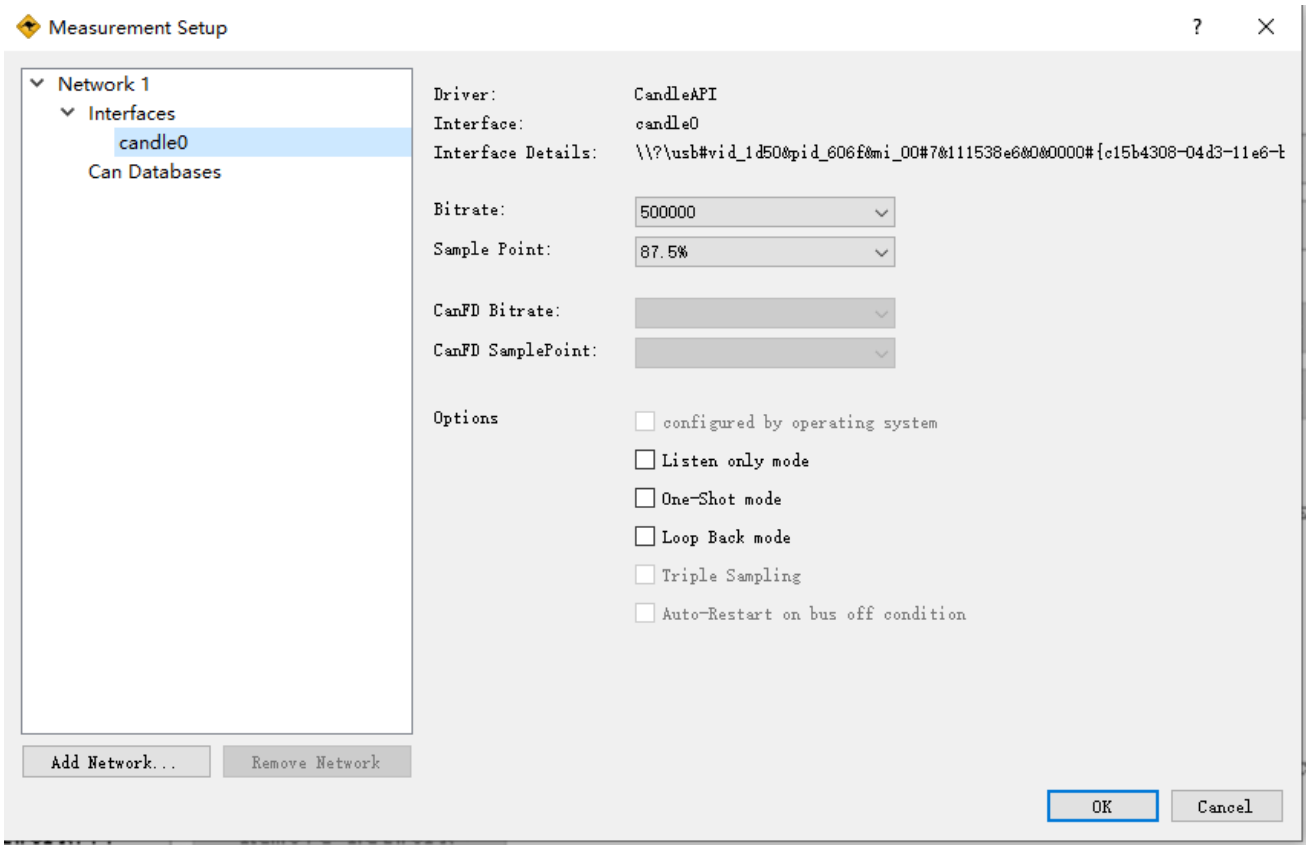
- 点击左上角 Measurement -> Start Measurement 开始按钮进入设置界面：



- 点击左侧列表中的 candle0 ，然后进行波特率、采样点、工作模式等相关设置，

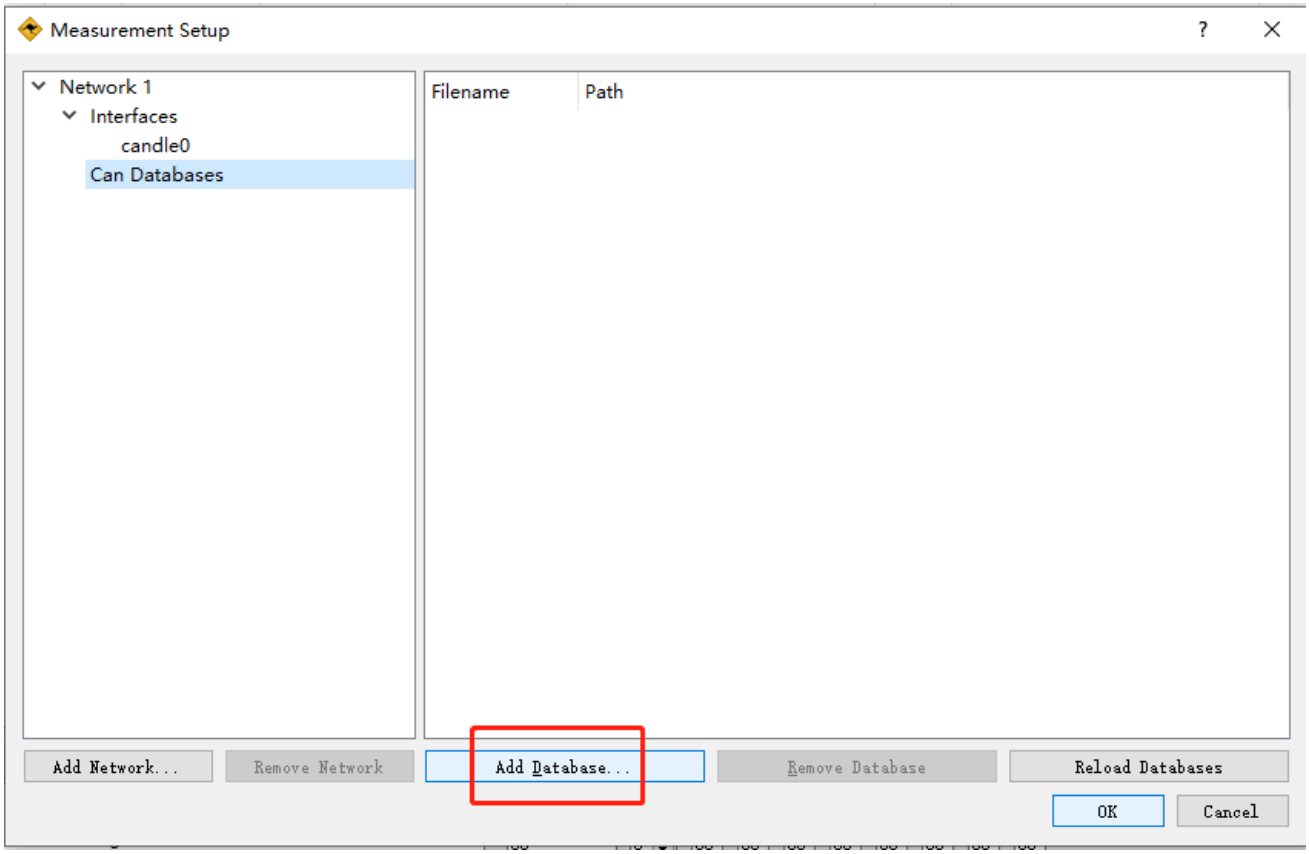
- Bitrate ： 比特率设置
- Sample Point ： 采样点设置，一般常用 87.5%
- Listen only mode ： 只听模式，该模式下 CANable 不能进行任何信号的发送，收到 CAN 帧后也不会发送 ACK 应答信号。（总线诊断、监听一般用该模式，避免引入干扰）
- One-Shot mode ：单次发送模式，即关闭 CAN 帧发送失败重发功能，无论是否收到 ACK 应答都只发送一次。
- Loop Back mode ： 自发自收的回环模式。

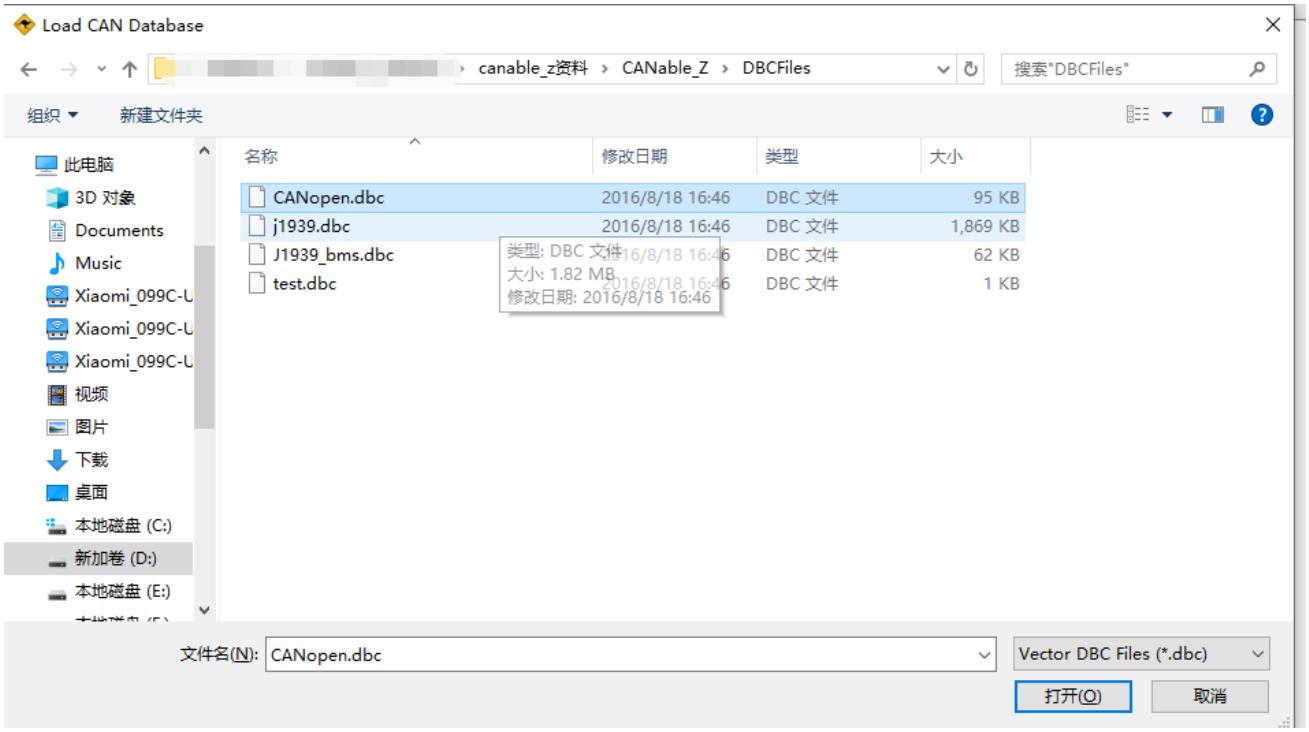
- 设置完成然后点击 OK，此时 CANable Z 模块上的 TX/RX 指示灯亮起，指示端口已处于工作状态，此时就可以进行 CAN 数据的收发操作了。



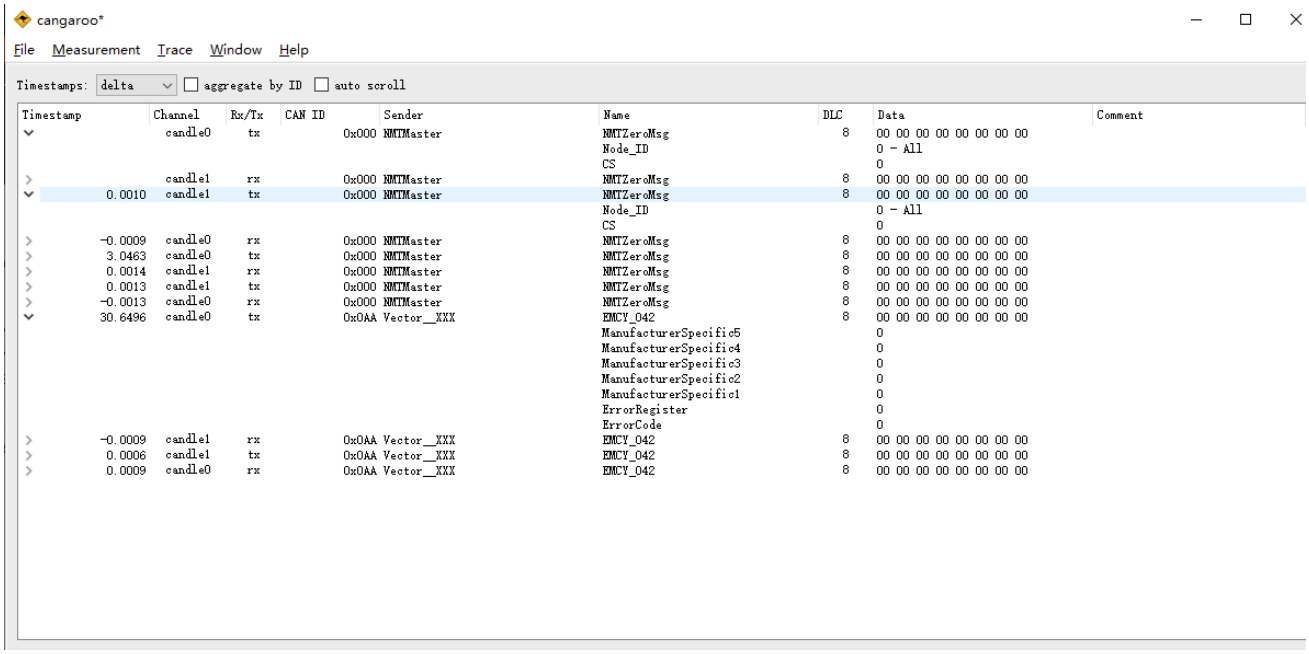
2.3.4 加载 CAN DBC 文件

- 点击 Measurement -> Setup，进入设置界面：
- 点击左侧列表中 Interfaces 下的 Can Databases，然后点击右侧 Add Database... 按钮，添加 DBC 文件：





添加完成后当接收到相应的数据帧时将在接收窗口中显示解析到的相关信息：



可以通过点击接收窗口中的各个数据帧展开查看详细信息。

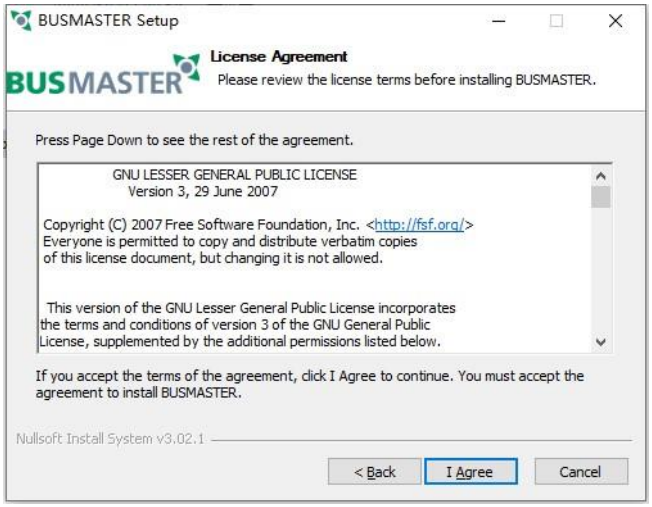
2.4 BUSMASTER 简介

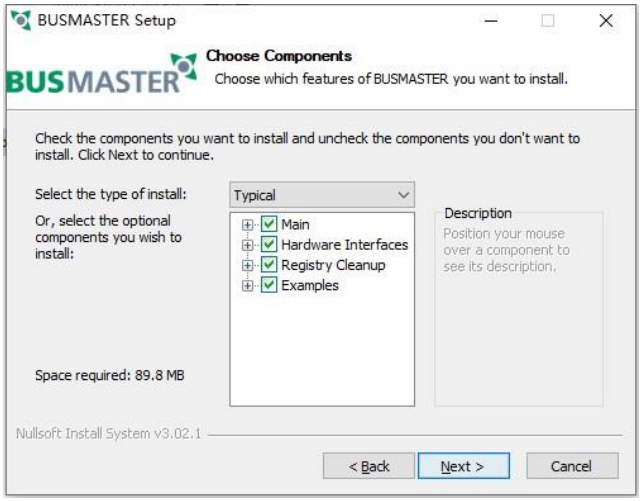
BUSMASTER 是一款功能强大的专业级 CAN 总线调试软件，不仅能够应对一般的 CAN 调试开发，还一个用于设计,监测,分析与模拟 CAN 网络的开源的开放式总线 PC 软件。BUSMASTER 已被 RBEI 概念化,设计与开发,同时基于 CANvas 软件工具。

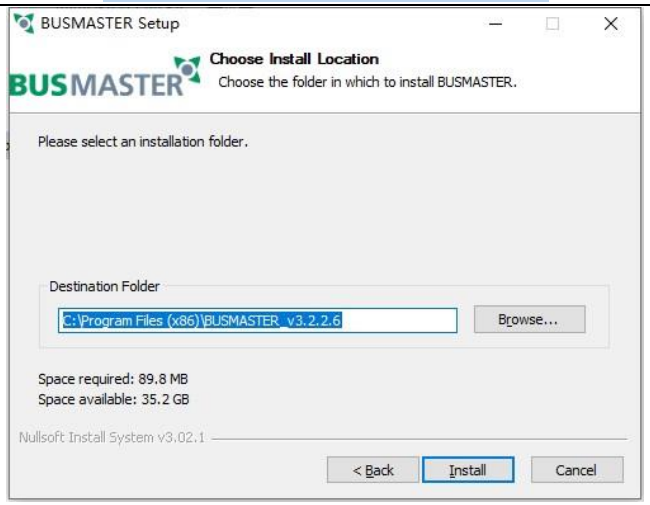
- 支持 ES581 CAN 总线 USB 接口的模块
- 创建,编辑 CAN 资料库
- 用硬件或软件过滤信息
- 记录,重放 CAN 信息
- 使用 ANSI C 功能编辑器建立可编程的结点
- 输入 DBC 资料库文件与 CAPL 程序的过滤
- BUSMASTER 仅支持 Windows 系统，需安装在 Windows7 以上版本

2.4.1 安装 BUSMASTER

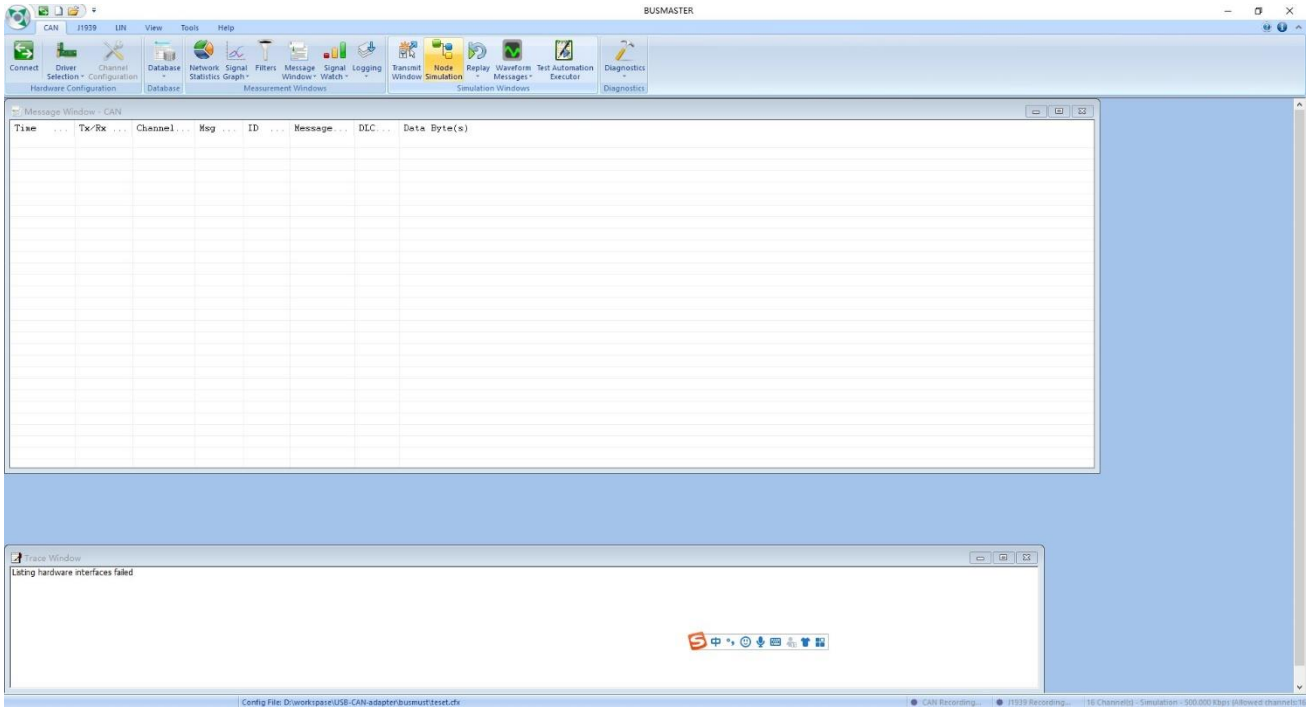
1. 双击 BUSMASTER_Installer_Ver_xxx-cantact.exe
2. 按默认配置安装即可





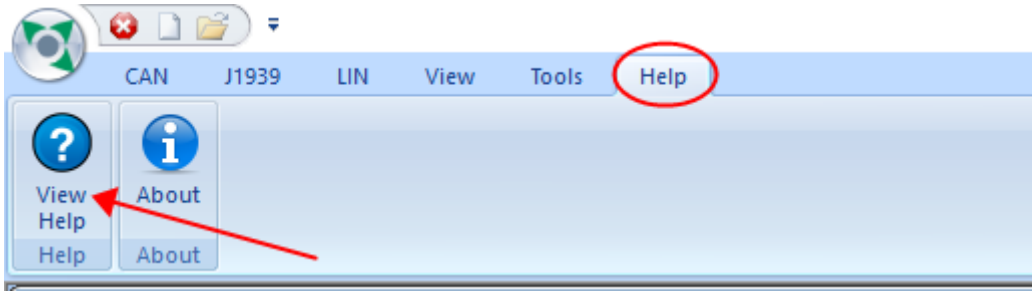


Windows 启动后界面：



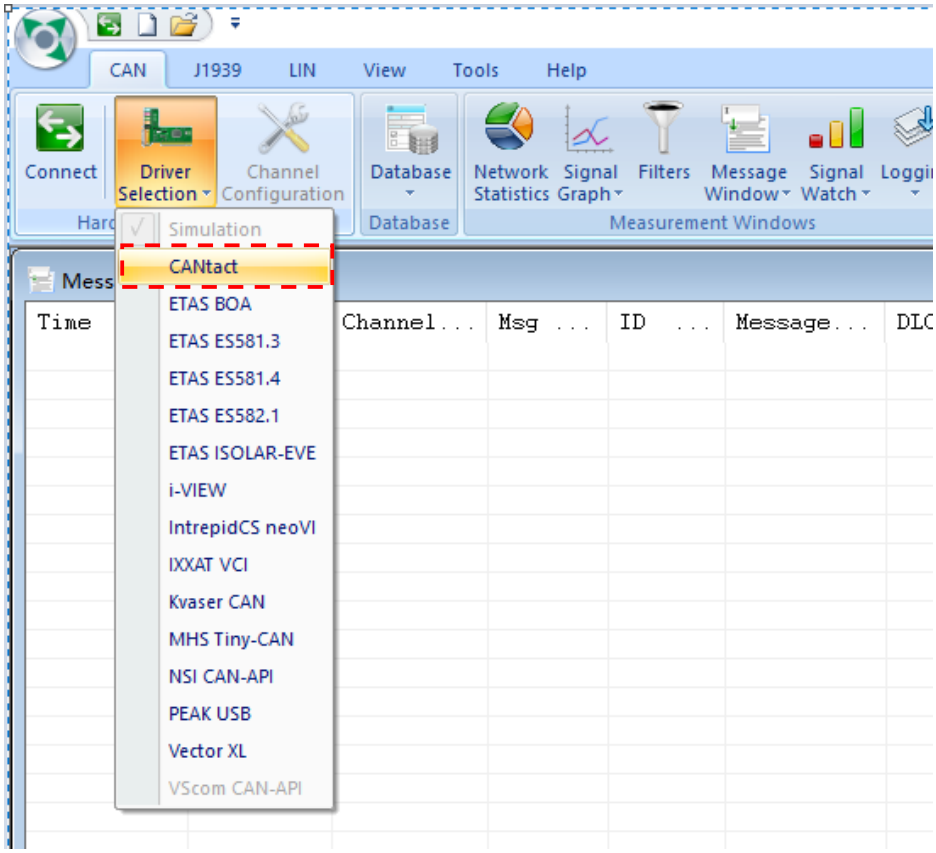
2.5 获取使用帮助

BUSMASTER 软件功能丰富，本文档只介绍基本的功能，更多使用说明可点击 Help-> View Help 查看完整的帮助说明。

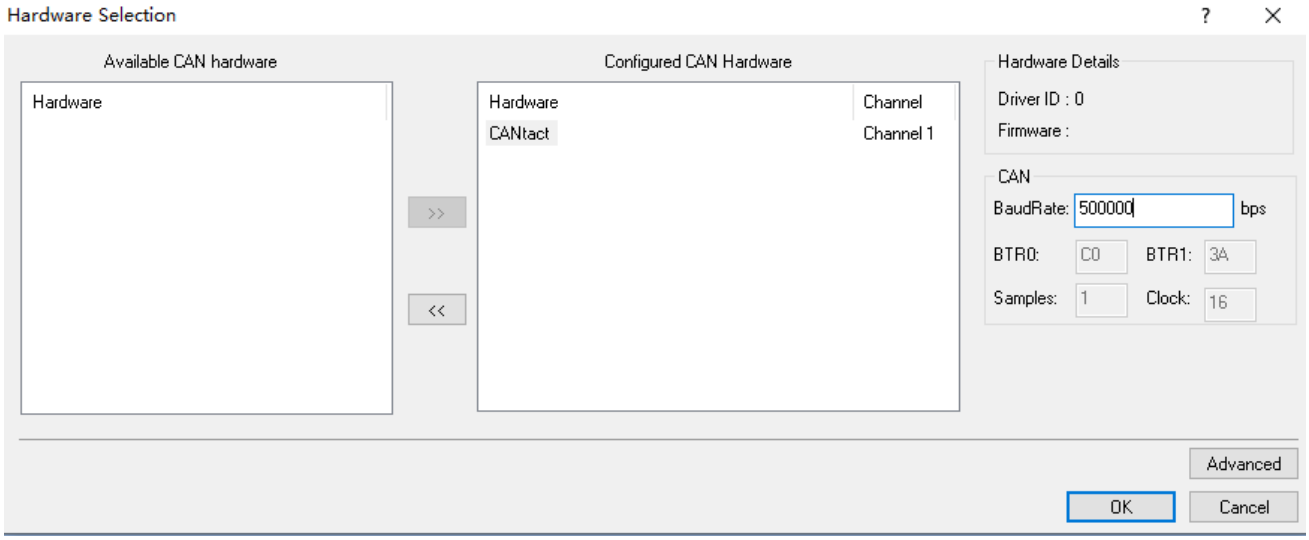


2.5.1 启动

- 先插入 CANable Z（PRO）
- 打开 BUSMASTER 软件
- 点击左上角 Driver Selection，选择 CANtact



- 点击 CANtact 后进入设置界面：



- 选择使用的设备，在右侧“BaudRate”框进行波特率配置

- 配置完成后点击 OK
- 回到主界面点击左上角 Connect，成功后 CANable Z(PRO)的两个指示灯亮起，此时已经可以开始正常收发数据

2.6 消息窗口

BUSMASTER 通过此窗口报告各种消息。消息可能来自以下两种类别中的任何一种

- 通过 CAN 总线传输的消息，包括 BUSMASTER 生成的消息。
- 错误讯息

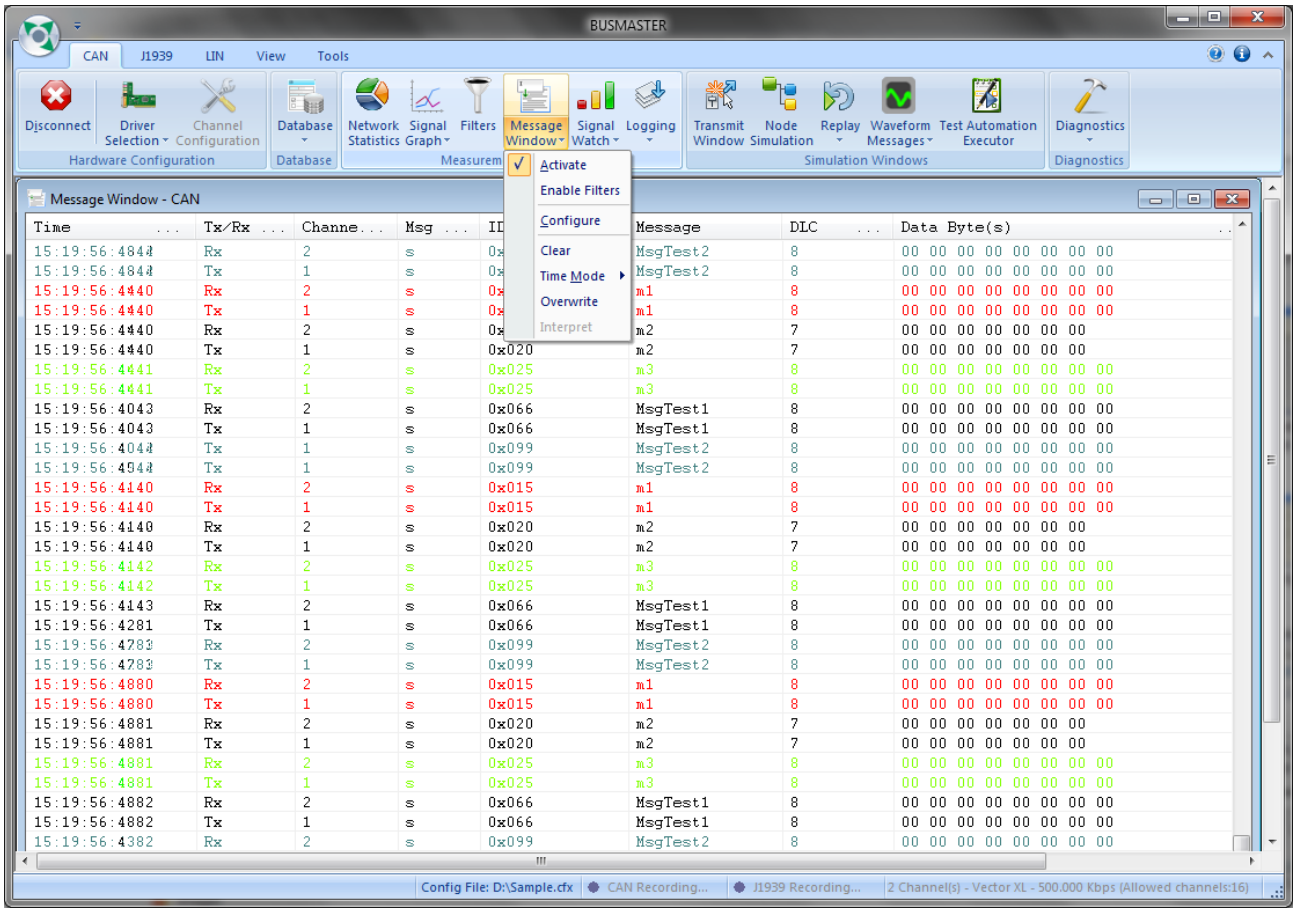
每条消息都显示在单独的行中，该行分别包含下面列出的以下五个字段

- 时间（**Time**）－可以通过三种不同的模式查看时间，即
  - 系统－在此模式下，消息将与 PC /系统时间一起显示。
  - 相对－在此模式下，自接收到以前具有相同标识符的消息以来，消息将随时间显示。
  - 绝对－在此模式下，参考是连接时间。自从与设备（ES581）建立逻辑连接以来，消息将随着时间显示。

在所有情况下，时间格式均保持为 HH：MM：SS：MS，其中 MS 代表毫秒，显示单位为 24 小时制。

- Tx / Rx-从 BUSMASTER 发送的消息被标记为 Tx，而对于接收到的消息，标记为 Rx。
- 类型－指示消息是标准，扩展还是 RTR 类型，遵循的约定是
  - S －标准帧
  - X －扩展帧
  - Sr －标准 RTR 帧
  - Xr －扩展的 RTR 帧
- 消息-此部分包含消息 ID。但是，BUSMASTER 可以为消息指定名称和颜色。如果特定的消息代码具有名称和颜色，则消息名称将代替消息 ID 出现，并且消息将以指定的颜色显示。
- DLC －数据长度计数的缩写。它显示消息正文中的数据字节数。
- 数据字节－数据字节以十六进制或十进制模式显示。请参阅“切换数字模式”部分以了解如何切换数字模式。发生错误时，将以红色显示适当的错误消息

以下小节介绍了用于更改消息条目显示的各种工具栏按钮。



2.6.1 更改时间显示

这是一个工具栏按钮，会弹出一个菜单，其中包含三个选项：系统，绝对时间和相对时间模式显示。

2.6.2 切换消息覆盖

这是一个切换工具栏按钮。此按钮用于在消息覆盖模式和附加模式之间切换。在消息覆盖模式下，消息窗口中将只有一个消息 ID 实例。随后收到的具有相同 ID 的消息将覆盖消息条目。在附加模式下，将添加新添加的消息条目。请参阅“消息窗口”部分中显示的图的说明 B。

2.6.3 切换数字模式

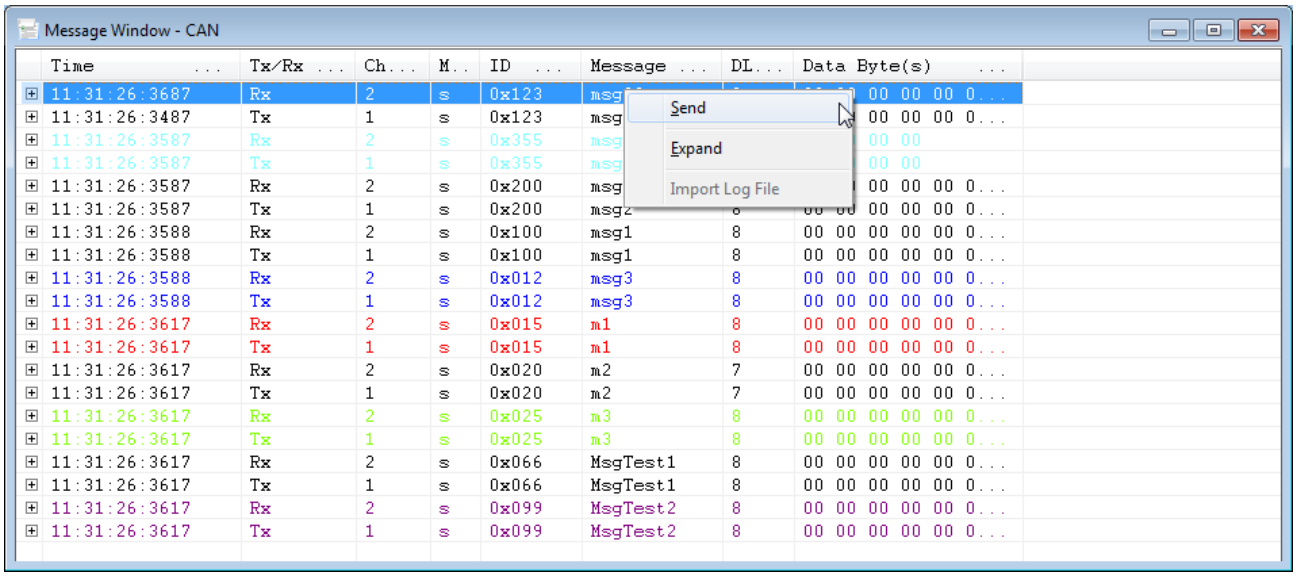
这是一个切换工具栏按钮。此按钮用于在十进制模式和十六进制模式之间切换。在十进制模式下，数据字节将以十进制格式显示。在十六进制模式下，数据字节将以十六进制格式显示。默认情况下，消息将以十六进制模式显示。请参阅“消息窗口”部分中显示的图形的说明 C。

2.6.4 切换消息解释

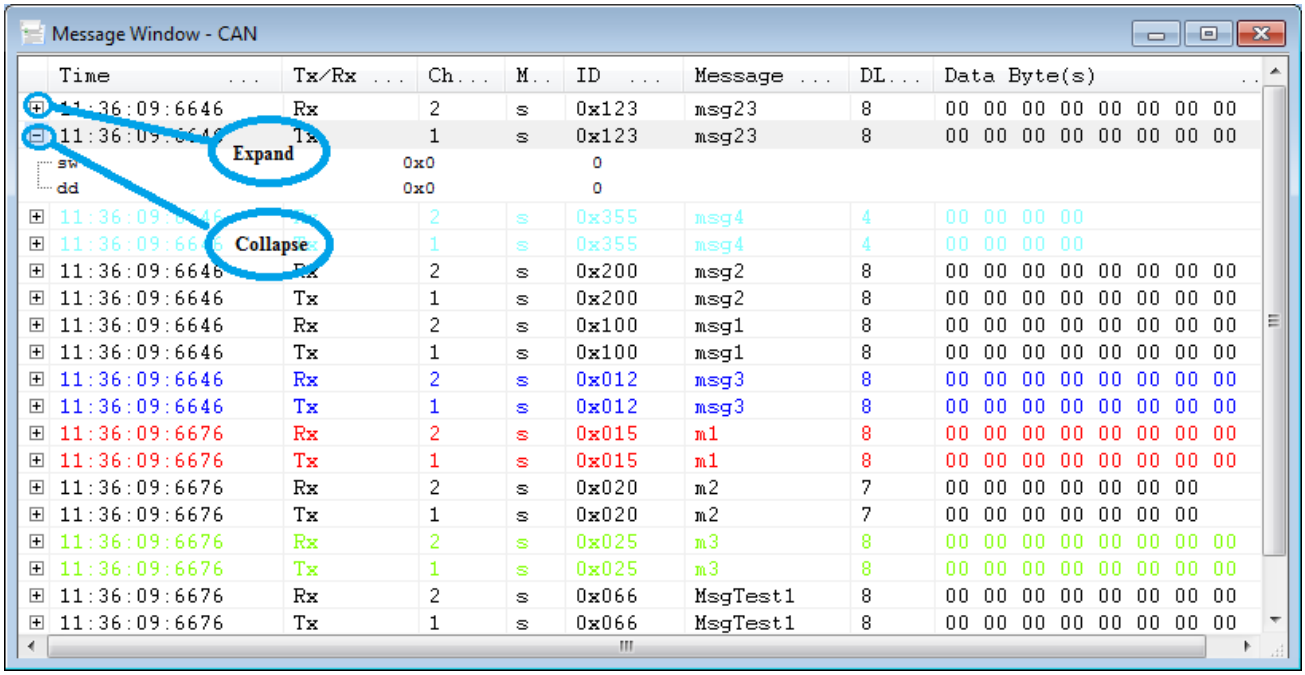
这是一个切换工具栏按钮。此按钮用于启用或禁用消息在线解释。仅在消息覆盖模式下启用此按钮。如果启用了对消息的在线解释，则将在消息条目后跟接收到的消息的文本描述。请参阅“消息窗口”部分中显示的图形的描述 D
仅当在数据库中找到消息 ID 时，才进行上述操作。否则，该消息之后将显示一条通知消息，指出“在数据库中找不到消息”。
双击消息即可获得消息解释。

2.6.5 选择性消息解释

可以在覆盖显示模式下选择性地解释消息。选择一个数据库消息条目，然后单击鼠标右键。BUSMASTER 将弹出此菜单。

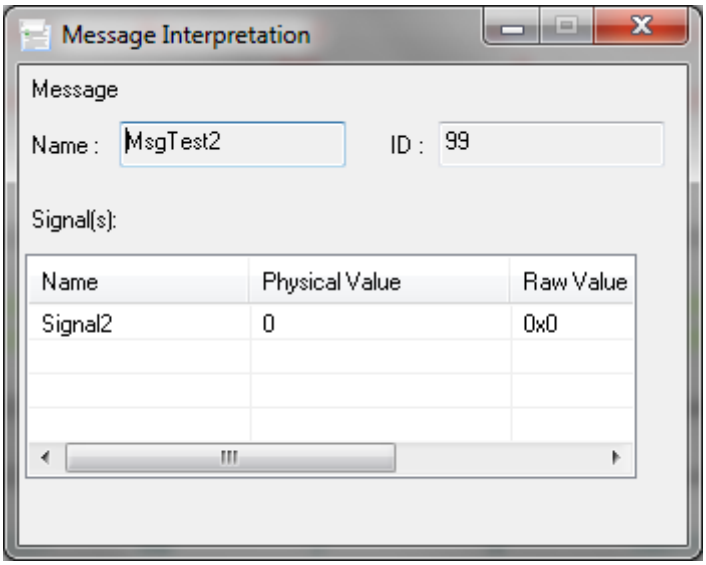


选择扩展以使用其信号值扩展消息条目。信号值将以原始值和物理值显示。可以通过右键单击该条目并选择“折叠”来关闭展开的条目。可以通过右键单击该条目并选择“折叠”来关闭展开的条目。或者，我们可以单击每个可解释消息上显示的“ +”符号以展开消息条目，然后单击“-”符号将折叠展开的条目，如下图所示。



2.6.6 解释对话框

可以使用“解释”对话框在弹出窗口中分别解释消息。双击消息条目将显示带有消息详细信息的解释对话框。这将具有信号列表及其原始值和物理值。



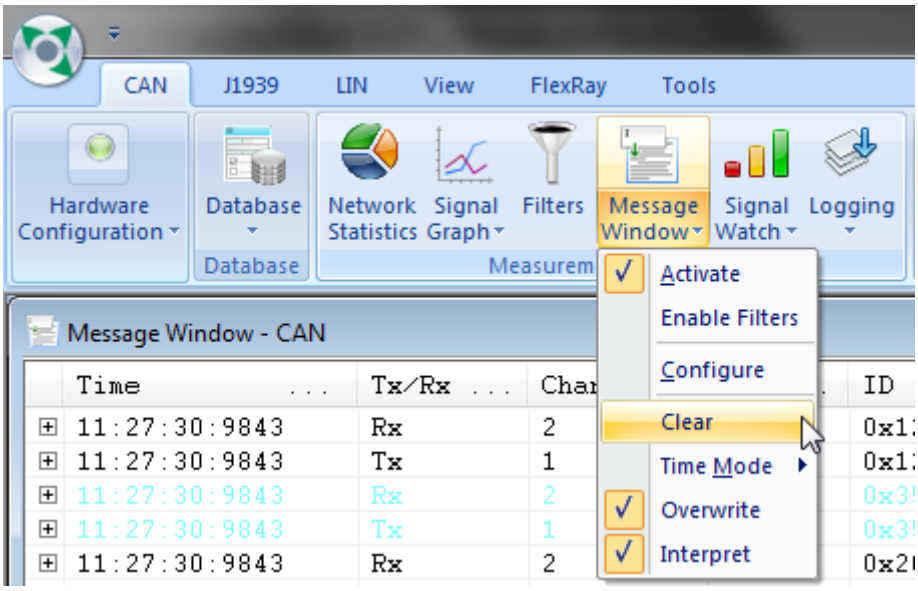
左键单击消息条目将更改消息选择。

2.6.7 从消息显示发送消息

可以从“消息显示”条目直接发送消息。选择一条消息条目，然后单击鼠标右键。这将弹出消息操作菜单（参见图 1）。选择发送以在 CAN 总线上传输选定的消息条目。

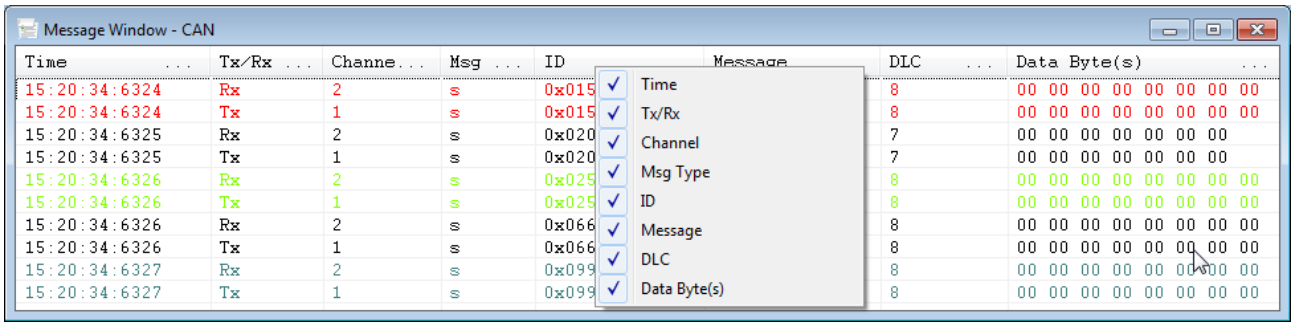
2.6.8 清除消息窗口

按下下面所示的工具栏按钮后，消息窗口将被清除。



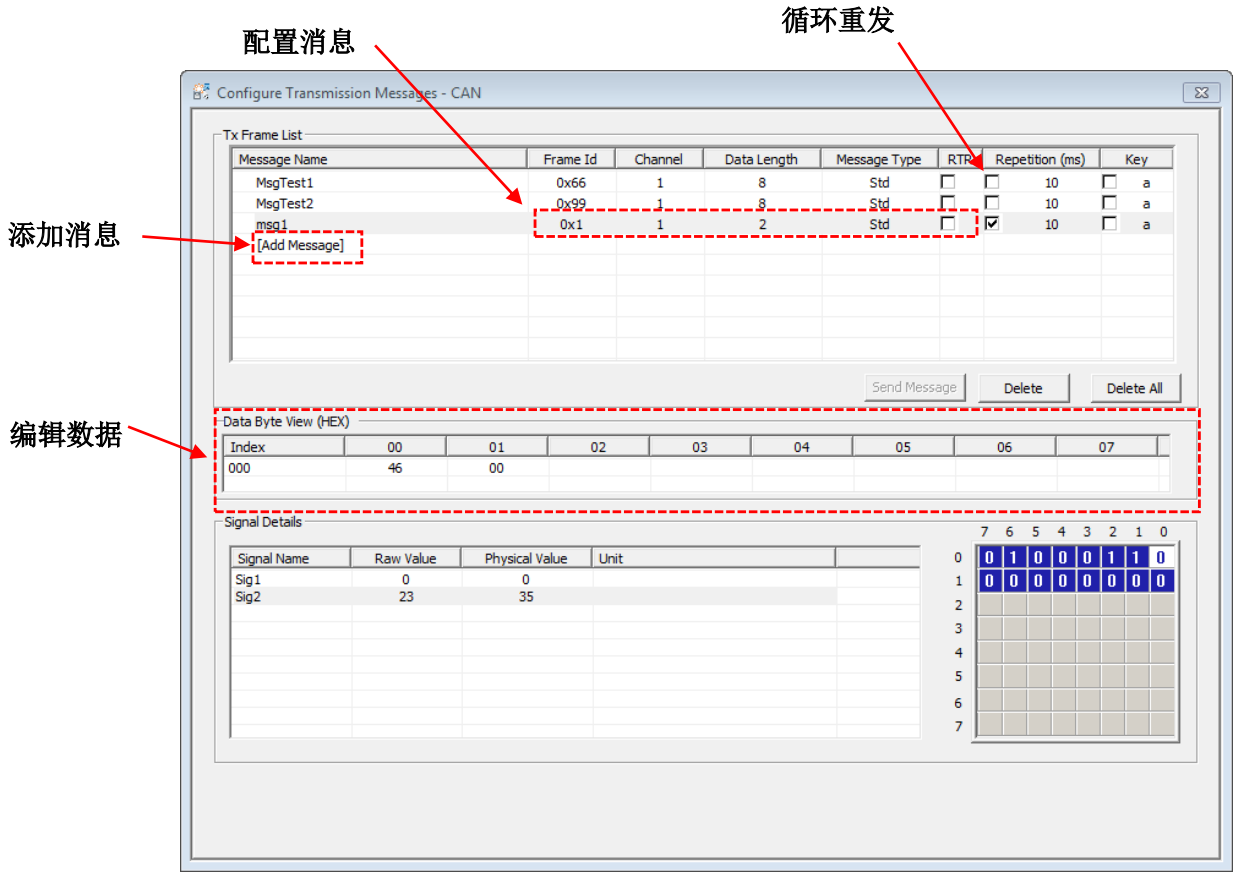
2.6.9 消息列的排序和可见性

可以根据用户的喜好将消息列拖放到消息窗口中的任何列位置。这些列也可以显示或隐藏。要显示/隐藏列，请在列标题上单击鼠标右键，将显示一个带有所有列标题名称的弹出菜单，如图 5 所示。6。在此菜单中，当前显示的列在其上带有复选标记。如果用户希望隐藏一列，只需从菜单中取消选中该列，该列将被隐藏。这些列的顺序和可见性与配置一起保存。



2.7 数据发送

可以按照以下步骤通过 CAN 总线发送消息.选择“**CAN --> Transmit Window**”菜单选项.这将显示数据发送对话框,如下图所示.



2.7.1 配置消息

导入 DBF 文件后,将在“**Tx Frame list** ”列表列中填充 DBF 文件中的消息(DB 消息).双击**[Add Message]**选择数据库消息.也可以通过键入消息 ID 添加非数据库消息。

如果选择了来自数据库的消息 ID /名称,则 DLC 和帧类型将使用数据库信息进行更新.将使用数据库中定义的信号启用信号列表.信号原始值或物理值可以直接在此列表中输入.验证后,数据将被更新.

信号描述符可用于输入物理值.双击获得描述符的信号的物理值单元将显示信号描述符的列表.

如果消息 ID 不是数据库消息,请输入 DLC,消息字节.在这种情况下,信号列表将被禁用.

可以通过选中“ RTR”复选框将 RTR 消息添加到其中

信号矩阵将显示数据字节的位模式.

2.7.2 消息的循环传输

可以通过启用重复来定期发送消息.循环发送对于定期发送带有不同数据字节的消息很有用.一旦 BUSMASTER 连接到 BUS,发送将自动开始,并在断开连接时停止.

2.7.3 事件传输

可以通过按键将消息发送到网络.可以为 Tx 帧列表中的每条消息分配一个字母数字键.

第3章 SocketCAN (只适用于 Linux)

SocketCAN 是 Linux 的 CAN 驱动程序和网络工具的集合。它允许以与其他网络设备类似的方式与 CAN 总线设备连接。这使开发人员可以编写支持多种 CAN 总线接口的代码。不幸的是，SocketCAN 仅在 Linux 上有效。

3.1 Linux 下 CAN 设备的基本操作

您可以在命令终端中通过命令来查看、配置、启动、关闭 can 设备。

3.1.1 查看 can 设备

在命令终端中输入：

```
ifconfig -a
```

得到如下结果：

```
can0: flags=193<UP,RUNNING,NOARP> mtu 16

unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 txqueuelen 10 (UNSPEC)

RX packets 14 bytes 112 (112.0 B)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 0 bytes 0 (0.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp9s0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500

ether 78:45:c4:b8:d2:b5 txqueuelen 1000 (Ethernet)

RX packets 0 bytes 0 (0.0 B)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 0 bytes 0 (0.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

3.1.2 设置 can 设备的波特率

在命令终端中输入：(以下命令将 can0 设备的波特率设定为 500000 bps)

```
sudo ip link set can0 type can bitrate 500000
```

3.1.3 启动 can 设备

在命令终端中输入：

```
sudo ip link set can0 up
```

3.1.4 关闭 can 设备

在命令终端中输入：

```
sudo ip link set can0 down
```

3.2 SocketCAN 实用程序：can-utils

can-utils 是 linux 下操作 socketcan 的一个实用工具集，包含多个收发 can 的小工具，如 candump，cansend 等。首先我们通过在命令终端中输入以下命令来安装 can-utils：

```
sudo apt-get install can-utils
```

3.2.1 candump

candump 可以实时显示接收到的 can 消息。要实时显示设备 can0 上的所有 can 消息，在命令终端中输入以下命令：

```
candump can0
```

candump 还可以使用掩码和标识符对接收到的 can 信息进行过滤。有两种过滤器类型：

- [can_id]:[can_mask] ： 当[received_can_id] & [can_mask] == [can_id] & [mask] 被显示
- [can_id]~[can_mask] ： 当[received_can_id] & [can_mask] != [can_id] & [mask] 被显示

例如：
仅显示 can0 上收到的 ID 为 0x123 的消息：

```
candump vcan0,0x123:0x7FF
```

仅显示 can0 上收到的 ID 为 0x123 或 0x456 的消息：

```
candump vcan0,0x123:0x7FF,0x456:0x7FF
```

3.2.2 cansend

cansend 可以将单个 CAN 帧发送到总线上。您将必须指定设备，标识符和要发送的数据字节。
例如：

```
cansend can0 123#1122334455667788
```

此条指令将在接口 can0 上发送一条消息，其标识符为 0x123，数据字节为[0x11、0x22、0x33、0x44、0x55、0x66、0x77、0x88]。请注意，此工具始终假定值以十六进制给出。

3.2.3 cangen

cangen 可以生成随机的 CAN 数据，这对于测试很有用。有关更多的用法信息，请在命令终端中输入：

```
cangen --help
```

3.2.4 cansniffer

cansniffer 可以显示总线上接收到的 CAN 消息，而且可以过滤掉数据不变的帧。这对于逆向工程 CAN 总线系统非常有用。有关更多信息，请在命令终端中输入：

```
cansniffer --help
```

3.3 使用 SocketCAN 二次开发

由于 CANable Z 和 CANable Z_pro 完全兼容 SocketCAN，所以我们通过 SocketCAN 对 CANable Z 和 CANable Z_pro 二次开发变得非常方便。它使用标准的网络套接字，这意味着可以用许多不同的语言进行开发。

3.3.1 使用系统 API (C 语言)

应用程序首先通过初始化一个套接字（与 TCP / IP 通信中的情况非常类似），然后将该套接字绑定到一个接口（或所有接口，如果应用程序需要），来设置对 CAN 接口的访问。一旦绑定，套接字就可以进行读取，写入等操作，像 UDP 套接字一样使用。
二次开发前需要安装 can_dev 模块并配置 CAN 总线波特率，然后启用。例如：

```
$ modprobe can_dev
$ modprobe can
$ modprobe can_raw
$ sudo ip link set can0 type can bitrate 500000
$ sudo ip link set up can0
```

还有一个用于测试目的的虚拟 CAN 驱动程序，可以使用以下命令在 Linux 中加载和创建：

```
$ modprobe can
$ modprobe can_raw
$ modprobe vcan
$ sudo ip link add dev vcan0 type vcan
$ sudo ip link set up vcan0
$ ip link show vcan0
3: vcan0: <NOARP,UP,LOWER_UP> mtu 16 qdisc noqueue state UNKNOWN link/can
```

以下代码段是 SocketCAN API 的工作示例，该 API 使用原始接口发送数据包：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <string.h>

#include <net/if.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/ioctl.h>

#include <linux/can.h>

#include <linux/can/raw.h>

int main(void)

{

int s;

int nbytes;

struct sockaddr_can addr;

struct can_frame frame;

struct ifreq ifr;

const char *ifname = "vcan0";

if((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {

perror("Error while opening socket");

return -1;

}

strcpy(ifr.ifr_name, ifname);

ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;

addr.can_ifindex = ifr.ifr_ifindex;

printf("%s at index %d\n", ifname, ifr.ifr_ifindex);

if(bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {

perror("Error in socket bind");

return -2;

}

frame.can_id = 0x123;
```

```
frame.can_dlc = 2;

frame.data[0] = 0x11;

frame.data[1] = 0x22;

nbytes = write(s, &frame, sizeof(struct can_frame));

printf("Wrote %d bytes\n", nbytes);

return 0;

}
```

3.3.2 使用 Python

Python 3.3 以后的版本增加了对 SocketCAN 的支持。开源库 python-can 可为 python 2 和 python 3 提供 SocketCAN 支持。具体使用详情请参考 python-can 官方文档。

第4章 使用 cando.dll 二次开发(只适用于 Windos)

cando.dll 是 Cando 用于二次开发的动态链接库，Cando 是 CANable 的衍生产物，cando.dll 同样适用于 CANable 及 CANable Z(PRO)。cando.dll 里边封装了对 CANable 的所有操作接口，方便我们快速进行二次开发。cando.dll 是使用 MinGW-win32 进行编译构建的。
注意：cando.dll 只适用于 Windos 下使用。

4. 1 4.1 内部变量和结构体

4.1.1 4.1.1 CAN 工作模式标志位

CANDO_MODE_NORMAL 正常工作模式
CANDO_MODE_LISTEN_ONLY CAN 侦听模式
CANDO_MODE_LOOP_BACK CAN 回环模式
CANDO_MODE_ONE_SHOT CAN 发送失败后不自动重新发送模式
CANDO_MODE_NO_ECHO_BACK CAN 发送数据帧后不向电脑返回 echo 帧(默认为返回 echo 帧)

4.1.2 4.1.2 CAN ID 标志位

CANDO_ID_MASK 用于和 Frame.can_id 按位与 运算，得到 can id
CANDO_ID_EXTENDED 用于和 Frame.can_id 按位与 运算，判断是否为扩展帧
CANDO_ID_RTR 用于和 Frame.can_id 按位与 运算，判断是否为远程帧
CANDO_ID_ERR 用于和 Frame.can_id 按位与 运算，判断是否错误帧

4.1.3 4.1.3 CAN 总线错误标志位

CAN_ERR_BUSOFF 离线错误
CAN_ERR_RX_TX_WARNING 发送或接收错误报警
CAN_ERR_RX_TX_PASSIVE 发送或接收被动错误
CAN_ERR_OVERLOAD 总线过载
CAN_ERR_STUFF 填充规则错误
CAN_ERR_FORM 格式错误
CAN_ERR_ACK 应答错误
CAN_ERR_BIT_RECESSIVE 位隐性错误
CAN_ERR_BIT_DOMINANT 位显性错误
CAN_ERR_CRC CRC 校验错误

4.1.4 cando_frame_t 数据帧结构体

```
uint32_t echo_id // 判断是否为发送的 ECHO 帧，ECHO 帧时值为 0，否则为 0xFFFFFFFF
uint32_t can_id // 帧ID，用于判断帧类型，和`CANDO_ID_MASK`按位`与`运算得到can id
uint8_t can_dlc // can数据长度，0~8
uint8_t channel // 用于内部通信，用户无需理会
uint8_t flags; // 用于内部通信，用户无需理会
uint8_t reserved; // 暂未使用，用户无需理会
uint8_t data[8]; // can数据
uint32_t timestamp_us; // can 数据时间戳，单位为 us
```

4.1.5 cando_bittiming_t CAN 波特率配置结构体

```
uint32_t prop_seg // propagation Segment (固定为 1)
uint32_t phase_seg1 // phase segment 1 (1~15)
uint32_t phase_seg2 // phase segment 2 (1~8)
uint32_t sjw // synchronization segment (1~4)
uint32_t brp // CAN 时钟分频 (1~1024)，内部 CAN 时钟为 48MHz
```

如果您不需要自己设定特殊的 can 波特率或采样点请参考 cando_example 源码中的常见波特率配置表。

4.2 接口说明

4.2.1 设备列表相关接口

4.2.1.1 cando_list_malloc

```
bool cando_list_malloc(cando_list_handle *list)
```

函数说明：创建设备列表
***list：**设备列表句柄指针，类型为 cando_list_handle *
返回值：True，成功；False，失败

4.2.1.2 cando_list_free

```
bool cando_list_free(cando_list_handle list)
```

函数说明：释放设备列表
list：设备列表句柄，类型为 cando_list_handle
返回值：True，成功；False，失败

4.2.1.3 cando_list_scan

```
bool cando_list_scan(cando_list_handle list)
```

函数说明：扫描当前电脑连接的所有 CANable 或 CANable_pro 设备
list：设备列表句柄，类型为 cando_list_handle
返回值：True，成功；False，失败

4.2.1.4 cando_list_num

```
bool cando_list_num(cando_list_handle list, uint8_t *num)
```

函数说明：获取扫描到的 CANable 或 CANable_pro 设备的数量
list：设备列表句柄，类型为 cando_list_handle
***num：**存储设备数量的变量指针
返回值：True，成功；False，失败

4.2.2 设备操作相关接口

4.2.2.1 cando_malloc

```
bool cando_malloc(cando_list_handle list, uint8_t index, cando_handle *hdev)
```

函数说明：创建设备实例
list：设备列表句柄，类型为 cando_list_handle
index：想要创建设备位于设备列表中的索引值，从 0 开始
***hdev：**设备句柄指针，类型为 cando_handle *
返回值：True，成功；False，失败

4.2.2.2 cando_free

```
bool cando_free(cando_handle hdev)
```

函数说明：释放设备实例
hdev：设备句柄，类型为 cando_handle
返回值：True，成功；False，失败

4.2.2.3 cando_open

```
bool cando_open(cando_handle hdev)
```

函数说明：打开设备 USB 通信通道
hdev：设备句柄，类型为 cando_handle
返回值：True，成功；False，失败

4.2.2.4 cando_close

```
bool cando_close(cando_handle hdev)
```

函数说明：关闭设备 USB 通信通道
hdev：设备句柄，类型为 cando_handle
返回值：True，成功；False，失败

4.2.2.5 cando_get_serial_number_str

```
wchar_t cando_get_serial_number_str(cando_handle hdev)
```

函数说明：获取设备序列号字符串
hdev：设备句柄，类型为 cando_handle
返回值：设备序列号字符串，类型为 wchar_t

4.2.2.6 cando_get_dev_info

```
bool cando_get_dev_info(cando_handle hdev, uint32_t *fw_version, uint32_t *hw_version)
```

函数说明：获取设备的固件和硬件版本信息
hdev：设备句柄，类型为 cando_handle
***fw_wersion：**存储设备固件版本的变量指针，如：32 表示 v3.2
***hw_wersion：**存储设备硬件版本的变量指针，如：12 表示 v3.2
返回值：True，成功；False，失败

4.2.2.7 cando_set_timing

```
bool cando_set_timing(cando_handle hdev, cando_bittiming_t *timing)
```

函数说明：配置 CAN 波特率相关信息
hdev：设备句柄，类型为 cando_handle
***timing：**CAN 波特率配置结构体指针，参考 cando_bittiming_t
返回值：True，成功；False，失败

4.2.2.8 cando_start

```
bool cando_start(cando_handle hdev, uint32_t mode)
```

函数说明：启动 Cando 或 Cando_pro
hdev：设备句柄，类型为 cando_handle
mode：CAN 工作模式，参考 CANDO_MODE_*

返回值：True，成功；False，失败

4.2.2.9 cando_stop

```
bool cando_stop(cando_handle hdev)
```

函数说明：停止 Cando 或 Cando_pro
hdev：设备句柄，类型为 cando_handle
返回值：True，成功；False，失败

4.2.2.10 cando_frame_send

```
bool cando_frame_send(cando_handle hdev, cando_frame_t *frame)
```

函数说明：发送 CAN 数据帧
hdev：设备句柄，类型为 cando_handle
***frame：**数据帧结构体指针，参考 cando_frame_t
返回值：True，成功；False，失败

4.2.2.11 cando_frame_read

```
bool cando_frame_read(cando_handle hdev, cando_frame_t *frame, uint32_t timeout_ms)
```

函数说明：读取 CAN 数据帧
hdev：设备句柄，类型为 cando_handle
***frame：**数据帧结构体指针，参考 cando_frame_t
返回值：True，成功；False，失败

4.2.3 辅助功能接口

4.2.3.1 cando_parse_err_frame

```
bool cando_parse_err_frame(cando_frame_t *frame, uint32_t *err_code, uint8_t *err_tx, uint8_t *err_rx)
```

函数说明：解析错误数据帧的错误信息
***frame：**错误数据帧结构体指针，参考 cando_frame_t
***err_code：**CAN 总线错误代码，参考 CAN_ERR_*
***err_tx：**CAN 总线发送错误计数

***err_rx：**CAN 总线接收错误计数

返回值：True，成功；False，失败

4.3 cando_example (cando.dll 应用 Demo)

为了方便大家对 cando.dll 的使用有一个更直观的了解，cando_example 一个（简单的 can 调试助手）基于 cando.dll 的工程示例，cando_example 是使用 Qt 进行编写的，并且源代码开放，以供大家参考使用 cando.dll 并编写自己的 can 调试应用软件。

cando_example 运行界面：

第5章 使用 Python 库二次开发(适用于 Windows,Linux)

cando Python 库是基于 Python3 编写的，通过几个简单的函数便可以完成和 usb 转 can 模块（CANable 或者 CANable _pro）的通信，进行高效的 CAN 工具开发。cando Python 库同样适用于 CANable 及 CANable Z(PRO)

cando 后台 usb 通信是基于 libusb 进行的，所以使用前请首先安装 libusb 驱动。
Windows 推荐使用 Zadig 工具进行安装。

1. 下载 Zadig
2. 将 CANable 或 CANable _pro 连接电脑
3. 双击运行 zadig-x.x.exe
4. 点击菜单栏中的 Options ->List All Devices 然后点击菜单栏下方的下拉列表，选择 CANable

5. 选择下方的驱动为 libusb-win32 ，然后点击 Replace Driver ，等待安装完成即可

Linux Ubuntu18.04 默认已安装 libusb 无需安装，其他发行版本请根据情况自行安装。

5.1 cando Python 库安装

推荐通过 Python 包管理工具 **pip** 进行安装，pip 的安装请自行 Google。
在命令终端中输入如下命令进行 cando Python 库安装：

```
pip install cando
```

5.2 内部常量

5.2.1 CAN 工作模式标志位

CANDO_MODE_NORMAL 正常工作模式
CANDO_MODE_LISTEN_ONLY CAN 侦听模式
CANDO_MODE_LOOP_BACK CAN 回环模式
CANDO_MODE_ONE_SHOT CAN 发送失败后不自动重新发送模式
CANDO_MODE_NO_ECHO_BACK CAN 发送数据帧后不向电脑返回 echo 帧(默认为返回 echo 帧)

5.2.2 CAN ID 标志位

CANDO_ID_MASK 用于和 Frame.can_id 按位与 运算，得到 can id
CANDO_ID_EXTENDED 用于和 Frame.can_id 按位与 运算，判断是否为扩展帧
CANDO_ID_RTR 用于和 Frame.can_id 按位与 运算，判断是否为远程帧
CANDO_ID_ERR 用于和 Frame.can_id 按位与 运算，判断是否错误帧

5.2.3 CAN 错误标志位

CAN_ERR_BUSOFF 离线错误
CAN_ERR_RX_TX_WARNING 发送或接收错误报警
CAN_ERR_RX_TX_PASSIVE 发送或接收被动错误
CAN_ERR_OVERLOAD 总线过载
CAN_ERR_STUFF 填充规则错误
CAN_ERR_FORM 格式错误
CAN_ERR_ACK 应答错误
CAN_ERR_BIT_RECESSIVE 位隐性错误
CAN_ERR_BIT_DOMINANT 位显性错误
CAN_ERR_CRC CRC 校验错误

5.2.4 数据帧类

Class Frame 内部只有以下成员变量：

- echo_id : 判断是否为发送的 ECHO 帧，ECHO 帧时值为 0，否则为 0xFFFFFFFF
- can_id : 帧 ID，用于判断帧类型，和 CANDO_ID_MASK 按位与 运算得到 can id
- can_dlc : can 数据长度，0~8
- channel : 用于内部通信，用户无需理会
- flags : 用于内部通信，用户无需理会
- reserved : 暂未使用，用户无需理会
- data : can 数据，类型为长度为 8 的列表
- timestamp_us : can 数据时间戳，单位为 us

5.3 内部函数

```
list_scan()
```

扫描当前连接到电脑的所有设备
:return: 设备句柄的列表

```
dev_start(dev, mode=0)
```

启动设备，启动后 CANable_pro 上的 RUN 灯亮起
:param dev: 设备句柄
:param mode: 启动模式标志，可以是 ***CAN*** 工作模式标志位 的任意按位或 运算组合，默认为正常工作模式
:return: 无

```
dev_stop(dev)
```

关闭设备，关闭后 CANable_pro 上的 RUN 灯熄灭
:param dev: 设备句柄
:return: 无

```
dev_set_timing(dev, prop_seg, phase_seg1, phase_seg2, sjw, brp)
```

设置 CAN 的波特率和采样点
:param dev: 设备句柄
:param prop_seg: propagation Segment (固定为 1)
:param phase_seg1: phase segment 1 (1~15)
:param phase_seg2: phase segment 2 (1~8)
:param sjw: synchronization segment (1~4)
:param brp: CAN 时钟分频(1~1024)，内部 CAN 时钟为 48MHz
:return: 无

```
dev_get_serial_number_str(dev)
```

获取设备序列号字符串
:param dev: 设备句柄
:return: 设备序列号字符串

```
dev_get_dev_info_str(dev)
```

获取设备固件、硬件版本信息字符串
:param dev: 设备句柄
:return: 设备固件、硬件版本信息字符串

```
parse_err_frame(frame)
```

解析错误帧的错误信息
:param frame: 错误帧
:return: (错误代码，发送错误计数，接收错误计数)，错误代码参考 ***CAN*** 错误标志位

```
dev_frame_send(dev, frame)
```

发送帧数据
:param dev: 设备句柄
:param frame: 数据帧类，参考 ***can*** 数据帧类
:return: 无

```
dev_frame_read(dev, frame, timeout_ms)
```

读取帧数据，读取到的数据帧将会赋值到传入的 frame 中
:param dev: 设备句柄

:param frame: 数据帧类，参考 ***can*** 数据帧类
:param timeout_ms: 读取超时时间，单位为 ms
:return: 如果读取成功返回 True 否则返回 False

5.4 开始编程

5.4.1 列出连接的设备

```
import sys

from cando import *

# 获取设备列表

dev_lists = list_scan()

# 打印扫描到的Cando或Cando_pro的设备信息

if len(dev_lists):

for dev in dev_lists:

# 获取设备序列号

serial_number = dev_get_serial_number_str(dev)

# 获取设备版本信息

dev_info = dev_get_dev_info_str(dev)

# 打印设备信息

print("Serial Number: " + serial_number + ', Dev Info: ' + dev_info)

else:

print("Device not found!")

sys.exit(0)
```

运行得到输出结果：(电脑连接两个 CANable Z(PRO))

```
Serial Number: 004F00295734571020343132, Dev Info: fw: 3.2 hw: 1.2

Serial Number: 0028002B5734571020343132, Dev Info: fw: 3.2 hw: 1.2

Process finished with exit code 0
```

5.4.2 发送数据

```
import sys

import time

from cando import *

# 获取设备列表

dev_lists = list_scan()

# 判断是否发现设备

if len(dev_lists) == 0:

print("Device not found!")

sys.exit(0)

# 设置波特率：500K 采样点：87.5%

dev_set_timing(dev_lists[0], 1, 12, 2, 1, 6)

# 启动设备

dev_start(dev_lists[0], 0)

# 设置发送的数据帧

send_frame = Frame()

send_frame.can_id = 0x12

# send_frame.can_id |= CANDO_ID_EXTENDED

# send_frame.can_id |= CANDO_ID_RTR

send_frame.can_dlc = 8

send_frame.data = [0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08]

# 循环发送 500 条数据帧

for i in range(500):

# 发送数据帧

dev_frame_send(dev_lists[0], send_frame)

time.sleep(0.001) # 睡眠 1 ms

# 读取数据帧，因为发送成功后 Cando 将返回 ECHO 帧，如果不进行读取，会阻塞通道，造成无法继续发送
```

```
rec_frame = Frame()

dev_frame_read(dev_lists[0], rec_frame, 100)

# 停止设备

dev_stop(dev_lists[0])
```

5.4.3 接收数据

```
import sys

from cando import *

# 获取设备列表

dev_lists = list_scan()

# 判断是否发现设备

if len(dev_lists) == 0:

print("Device not found!")

sys.exit(0)

# 设置波特率：500K 采样点：87.5%

dev_set_timing(dev_lists[0], 1, 12, 2, 1, 6)

# 启动设备

dev_start(dev_lists[0], 0)

# 创建接收数据帧

rec_frame = Frame()

# 阻塞读取数据

print("Reading...")

while True:

if dev_frame_read(dev_lists[0], rec_frame, 10):

break

if rec_frame.can_id & CANDO_ID_ERR: # 错误帧处理

error_code, err_tx, err_rx = parse_err_frame(rec_frame)
```

```
print("Error: ")
if error_code & CAN_ERR_BUSOFF:
print(" CAN_ERR_BUSOFF")
if error_code & CAN_ERR_RX_TX_WARNING:
print(" CAN_ERR_RX_TX_WARNING")
if error_code & CAN_ERR_RX_TX_PASSIVE:
print(" CAN_ERR_RX_TX_PASSIVE")
if error_code & CAN_ERR_OVERLOAD:
print(" CAN_ERR_OVERLOAD")
if error_code & CAN_ERR_STUFF:
print(" CAN_ERR_STUFF")
if error_code & CAN_ERR_FORM:
print(" CAN_ERR_FORM")
if error_code & CAN_ERR_ACK:
print(" CAN_ERR_ACK")
if error_code & CAN_ERR_BIT_RECESSIVE:
print(" CAN_ERR_BIT_RECESSIVE")
if error_code & CAN_ERR_BIT_DOMINANT:
print(" CAN_ERR_BIT_DOMINANT")
if error_code & CAN_ERR_CRC:
print(" CAN_ERR_CRC")
print(" err_tx: " + str(err_tx))
print(" err_rx: " + str(err_rx))
else: # 数据帧处理
print("Rec Frame: ")
print(" is_extend : " + ("True" if rec_frame.can_id & CANDO_ID_EXTENDED else "False"))
print(" is_rtr : " + ("True" if rec_frame.can_id & CANDO_ID_RTR else "False"))
print(" can_id : " + str(rec_frame.can_id & CANDO_ID_MASK))
print(" can_dlc : " + str(rec_frame.can_dlc))
```

```
print(" data : " + str(rec_frame.data))

print(" timestamp_us : " + str(rec_frame.timestamp_us))

# 停止设备

dev_stop(dev_lists[0])
```

第6章 FAQ

6.1.1 为什么安装 libusb 驱动后，还是无法识别设备

1. 重新插拔设备，使驱动生效。
2. 检查所使用的 usb 数据线是否正常，某些 usb 线内部只有 VCC 和 GND 只是用来充电使用，没有数据通信能力。(建议使用配套的数据线)
3. 重启电脑再次进行尝试。

6.1.2 使用 Zadig 安装的 libusb-win32 时 Zadig 无响应

1. Zadig 安装过程可能会有假死现象，只需要耐心等待几分钟，当弹出安装成功对话框时说明安装完成，即可关闭。
2. 强制退出 Zadig ，尝试重新操作。

6.1.3 怎么看 Ubuntu 系统中是否已安装 libusb

1. 在命令终端中输入 sudo dpkg -l 回车，在列表中查看是否有 libusb。

6.1.4 Linux 系统运行时提示 Access denied

1. 使用管理员权限运行即可，因为需要进行 USB 通信，所以会提示权限不足问题。