TTCANopen

CAN(FD)转 SPI

使用说明书(v3.7)

(三合一 学习型)

TTCANopen 工作室

目　录

产品升级说明

1）由于该版本 CANFD 转 USB 的内容直接继承 v3.7 版，故此模块的整体最初版本也被定义为 v3.7

2）

1. 产品介绍

本品支持CAN和CANFD应用，可做为CAN(FD)转SPI、CAN(FD)转USB、CAN(FD)转TTL模块使用，模块状态切换自如，可最大限度的适用用户的应用场景。

关于CAN(FD)转USB的内容请参考《CAN(FD)转USB使用说明书v3.7》。

关于CAN(FD)转TTL串口的内容请参考《CAN(FD)转TTL使用说明书v3.3》。

本说明书后续主要介绍CAN(FD)转SPI的相关内容。

由于市面上大多数单片机和微处理器都不直接具备CANFD接口，但其可以通过自身的SPI接口，使用本模块进行CAN或CANFD接口扩展，使其具备接入CAN总线的能力，我们称之为“微接入”。本品广泛应用于汽车电子，工业自动化监控，医疗仪器，机器人，纺织机械，安防系统，水文气象等领域。

TTCANopen CAN(FD)转SPI产品分为两种类型。

第一类：学习型，主要面向青年学生和初学者，转换器主控芯片为GD32E（C）103或STM32G431，CAN端口为非隔离

方式，集成多种功能，可极大降低学习成本。

第二类：车规定制型，CAN 端口为隔离型。

本品遵循 CAN 总线协议 2.0A、2.0B 和 ISO11891-1:2015 CANFD 规范。 本品 SPI 接口为从模式，通讯速率最高为 15Mbps，能够满足在 6M（或 8M 车规型） CANFD 总线 100%满载的条件下，SPI 端口不会丢帧。

本着“开放，共享”的应用理念，本品准许用户非商业性 DIY，提供资料中包含设备的“电路逻辑图”“PCB 印制板图”和“ISP 装载固件”，用户可以根据这些文件非常方便的复制出一模一样的作品，从而解除了用户应用的后顾之忧。同样用户也可以将本品相关内容融入到自身产品中进行二次开发应用。

2. 接口和主要指标

学习型: USB 端口链接器：A 型链接器

USB 端口通讯：USB2.0 全速， 虚拟串口。

用于 CANFD 转 USB，采用 USB 端口+5V 供电

SPI 端口：2*6p 孔接线排（含供电端子）

SPI 通讯：双向，从模式，最高速率 15Mbps

用于 CANFD 转 SPI， SPI 端口+3.3V 供电

TTL 端口：三线 TTL 串口（3.3V 供电）

TTL 串口配置： 波特率：256000

奇偶校验：无

数据位：8

停止位：1

CAN 端口链接器：单排 3P 插针（2.54mm）

CAN 收发器：QBD1044

终端电阻：平衡式终端电阻

CAN 通讯速率 kbps：10、20、50、100、125、250、500、800、1000

CANFD 数据速率：为 CAN 速率的整倍数，

最高 6Mbps（或 5Mbps）

核心 ARM 芯片：GD32E103（或 STM32G431）

外观尺寸：63×28×12 mm

模块集成：集成三类转换器，支持 CAN 和 CANFD

1. CAN(FD)转 SPI
2. CAN(FD)转 USB
3. CAN(FD)转 TTL

在三类转换器中，都集成有五种子模块

1).普通调试模块、

2).普通工作模块、

3).TTCANopen 调试模块、

4).TTCANopen 工作模块、

5).TTCANopen 心跳器模块

车规定制型:　（具体参考车规定制型使用说明书）

3. 设备接口说明

1）ISP 固件加载更新接口

本品为用户提供了 ISP 固件加载更新接口，见图 1 所示，用户可以从 www.ttcanopen.com 网站获取产品的最新升级版本，使用 ISP 下载器将最新的固件安装到设备上（当然用户也可以将自己研发的固件安装到设备上）。

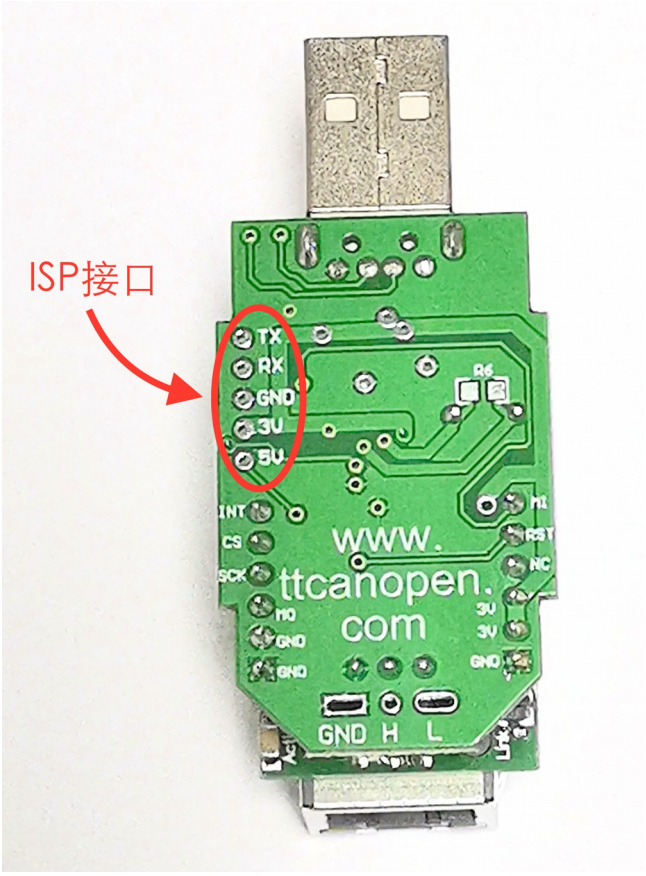


图 1　设备 ISP 固件更新接口

ISP 接点从上至下分别为：TX,RX,GND,3.3V,5V。

下载资料中包含 ISP 运行程序，解压后可以直接运行，请不要擅自更改 ISP 串口配置参数。

用户可以使用市面上多数 ISP 下载器安装设备固件，使用时请注意插针的排序，连接不当可能会损毁设备。

用户也可以到 TTCANopen 淘宝网店购买设备专用的 ISP 下载器，其使用非常方便，只需将专用 ISP 下载器的 5 根插针直接从背面插入到设备的对应 ISP 接口上即可，见图 2 所示。

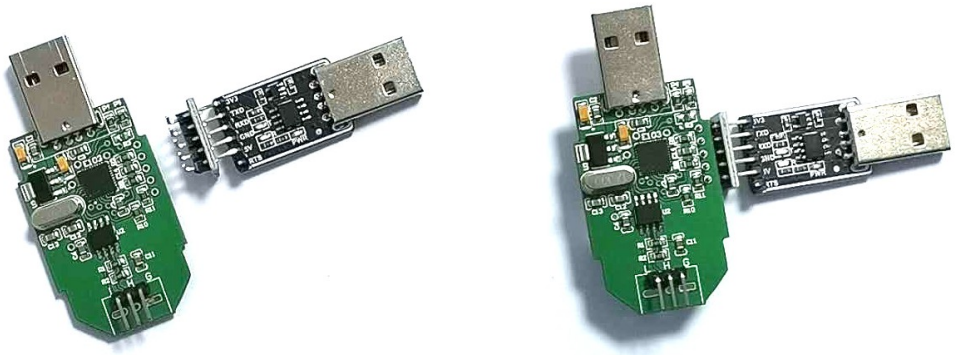


图 2 专用 ISP 下载器的连接

（ISP 专用下载器以淘宝店实物为准）

当产品工作在 CAN(FD)转 TTL 串口时（即：不连接 SPI 接口和不连接 USB 接口），设备复用了 ISP 接口的 TX,RX,GND 做为 TTL 串行通讯接口；复用了 ISP 接口的 3.3V,GND 做为设备供电电源接口。

2）SPI 接口

设备 SPI 接口是两个单排 6p 孔座，见图 3（左）所示。这个接口原是开发以太网使用的 SPI 接口，其接点是按 WIZ820io 设计的。为了节约开发成本，这里将 WIZ820io 拔除，并对其 SPI 部分接点进行了重新定义，用于 CANFD 转 SPI。

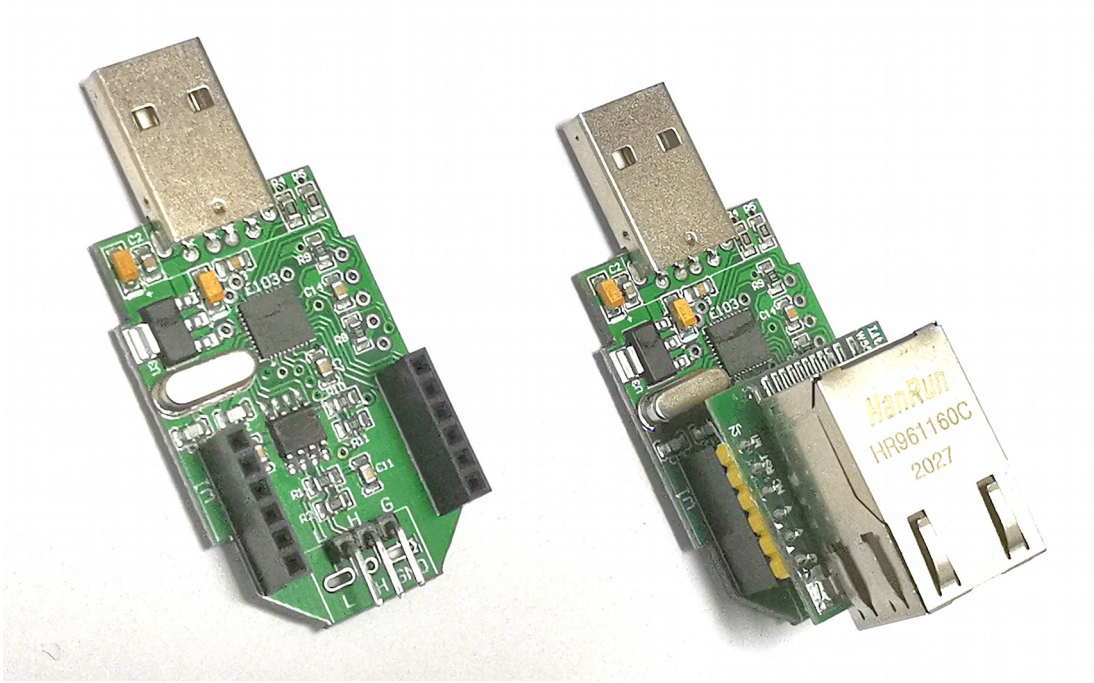


图 3 SPI 接口（左）

参照左侧图，其左边 6p 插座自上而下，各接点标识分别为：MI,RST,NC,3V,3V,GND。其右边 6p 插座自上而下，各接点标识为：INT,CS,SCK,MO,GND,GND。

在上述接点中，CANFD 转 SPI 模块，修改了 RST 和 INT 两个接点的用途和含义。其中 RST 用于接收握手；INT 用于指令握手。

在使用 CANFD 转 SPI 功能时，需使用 SPI 接口上的接点供电，供电电压 3.3V。本模块 SPI 工作在从模式下，最高通讯速率为 15Mbps。（SPI 接口通讯协议见后续章节）

3）CAN(FD)端口

模块 CAN(FD)接口有三个接线端子，见图 4 所示，分别是 CAN_L（左）、CAN_H（中）总线信号和 GND（右）地线，通常总线信号线为双绞线，地线可选（多为屏蔽地）。

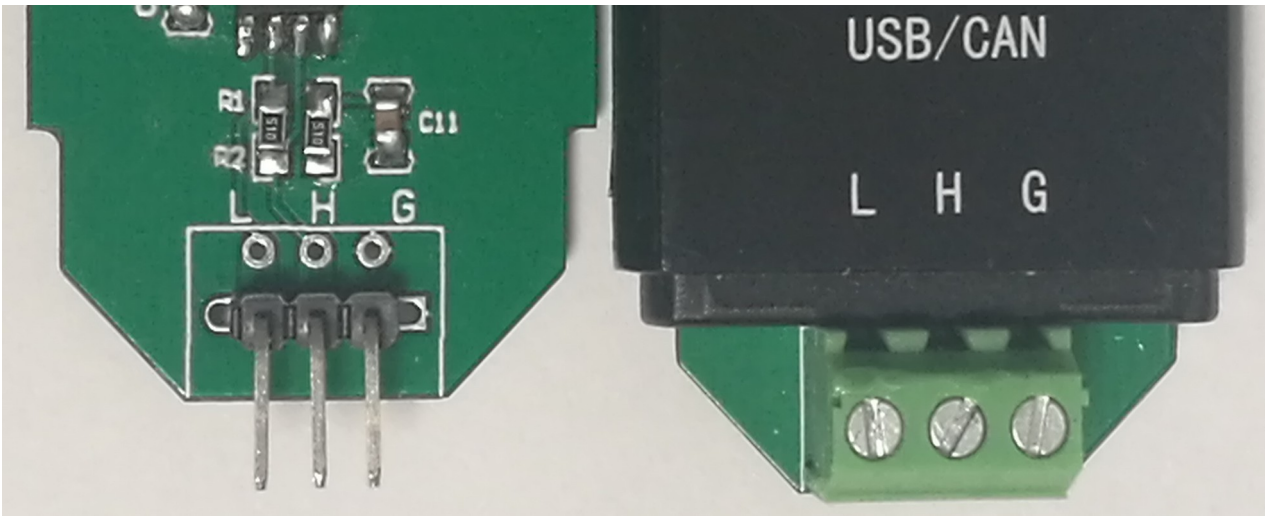


图 4 CAN 端口接线端子

4. 转换器种类切换

1）当USB 端口悬空时，主、从SPI 接口进行点对点连接，由主SPI 端向从SPI 端进行3.3V 供电，本模块将自动工作在CANFD 转SPI 模式下。

2）当SPI 端口悬空，USB 端口插入PC 机，模块会自动切换为CAN(FD)转USB。

3）当SPI 端口悬空，同时也不接入USB 端口，设备复用ISP 端口的3.3V 和GND 供电，复用ISP 端口的TX,RX,GND 做为TTL 串行通讯接口，设备自动配置为CAN(FD)转TTL 串口。

设备上电后，CAN(FD)端口默认为“静听状态”，CAN 速率为1000K，FD 速率为2000K。用户需要修改这个配置，以使其与当前应用CAN 环境相匹配。

5. 验证测试

为了给用户一个更加直观的使用体验，我们使用相同的硬件模块，按 SPI 收发协议开发了一个 SPI 转 USB 透传模块，这样只要将透传模块的 SPI 端口（主）与 CANFD 转 SPI 模块的 SPI 端口（从）进行点对点对接。然后，再将透传模块的 USB 端插入 PC 机，用户就可以在 PC 机上直观的测试 CANFD 转 SPI 模块了。见图 5

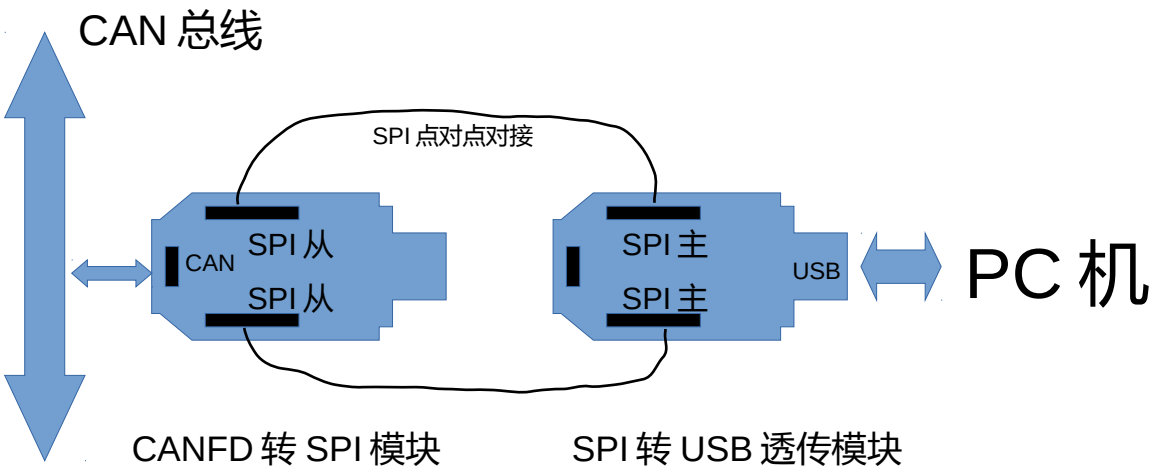


图 5 SPI 透传转换测试连接图

用户可以通过 PC 机上的串口调试工具，直接操控 CANFD 转 SPI 模块上的收发数据流，就如同我们直接使用 CANFD 转 USB 模块一样。

由于是透传工具，PC 机串口调试工具上的收发数据流会

原样传递到两个模块的SPI 端口。

用户在使用单片机或微处理器扩展CANFD 端口时，其SPI 端口收发的数据流与使用透传模块在串口调试工具中所显示的是一致的。

为了给读者一个比较好的视觉体验，本说明书的后续部分，对通讯协议的解析阐述中，将使用串口调试工具端的数据流进行描述。这些数据流实为用户使用SPI 扩展CANFD 端口时，SPI 端口的数据流。

6. CAN 端口参数配置

设备上电后，主 SPI 端口需要向 CAN(FD)转 SPI 模块发送 CAN 端口配置指令，以使其 CAN 端口参数与当前应用环境相匹配。

实际上，在 CAN(FD)转 SPI 设备内部分配了一个固有的寄存器空间来存放其配置参数，用户可以通过修改该寄存器中的内容，完成配置参数和设置工作状态。表 1 为该寄存器中相关参数的分配状况。

表 1 固有的配置参数寄存器空间的分配

序号	寄存器地址	字节数	参数描述
1	FF00	1	当该值置 1 时，启动配置，然后自动归 0.
2	FF01	1	选择工作模块： 0:为普通调试模块 1:为普通工作模块 2:为 TTCANopen 调试模块 3:为 TTCANopen 工作模块 4:为 TTCANopen 心跳模块
3	FF02	1	配置工作状态： 0:为 CAN 状态 1:为静听态 2:为 CANFD 态

4	FF03	1	配置 CAN 和 CANFD 速率： 低半字节配置 CAN 速率 1～9 分别对应 CAN 速率的 10、20、50、100、125、250 、500、800、1000 kbps 高半字节配置 CANFD 的倍率 1～A 分别对应 CANFD 为 CAN 速率的 1～10 倍
5	FF04	1	设置两帧发送间隔（微妙） 高半字节为发送间隔首数据， 低半字节为 0 的个数。 注意：当该字节被设置为 0xAA 时 将由模块根据通讯速率自动 生成发送间隔
6	FF05	1	选择配置采样点 低半字节配置 CAN 采样点 高半字节配置 FD 采样点 0:为自动配置采样点 1:为 75%左右 2:为 80%左右 3:为 85%左右 4:同 v3.2 版
7	FF06～FF0F	2	保留

为了能够设置和修改该寄存器内容，设计了专用配置指令格式如下：（十六进制 HEX 发送）

表 2 设置配置参数寄存器指令格式：

帧头	帧标识	寄存器地址	字节数	数据内容	帧尾
AA	66	FF00 至 FF0F	01 至 10	xx ～ xx	DD

用户配置完参数后，需将 0xFF00 寄存器置 1，新配置方才生效，生效后该寄存器内容自动归零。

配置参数指令举例：

例 1；修改 CAN 发送速率为 1000000bps

AA 66 FF03 01 19 DD

AA 66 FF00 01 01 DD

例 2:修改 CAN 工作状态为 CANFD

AA 66 FF02 01 02 DD

AA 66 FF00 01 01 DD

当然用户也可以用一条指令将配置参数一次性完成（推荐）。

例 3:一次配置全部参数，包括配置立即生效

AA 66 FF00 06 01 01 02 59 AA 00 DD

该指令配置为“普通工作模块”、“CANFD 态”、“CAN 速率 1000kbps”、“CAN FD 速率 5000kbps”、“自动配置发送间隔”、“自动配置采样点”配置立即生效。

当配置成功后，主 SPI 端口会收到配置成功的信息（ASCII），其中包括配置帮助信息和当前 CAN 端口工作状态，使用透传模块可以在串口接收窗口观察到这些信息见图 6.

用户在配置模块参数时，需要注意可配置参数的取值范围以其适用性，如：通讯速率除了受主控 MCU 芯片的限制，同时也受 CAN 接口芯片指标的限制，以及终端匹配电阻的连接形式的影响。用户在使用模块时应当按照所用模块的硬件配置和应用环境，进行合理设置。 表 3 为部分主控芯片和接口芯片 CAN 通讯速率相关参数：

表 3 模块所用各芯片的 CAN 通讯速率范围

芯片代号	CAN 速率 kbps	FD 速率 kbps
GD32E103	10 至 1000	10 至 6000
STM32H750	10 至 1000	10 至 5000
TJA1057	40 至 1000	40 至 5000
ADM3057	10 至 1000	10 至 12000
QBD1044	10 至 1000	10 至 6000

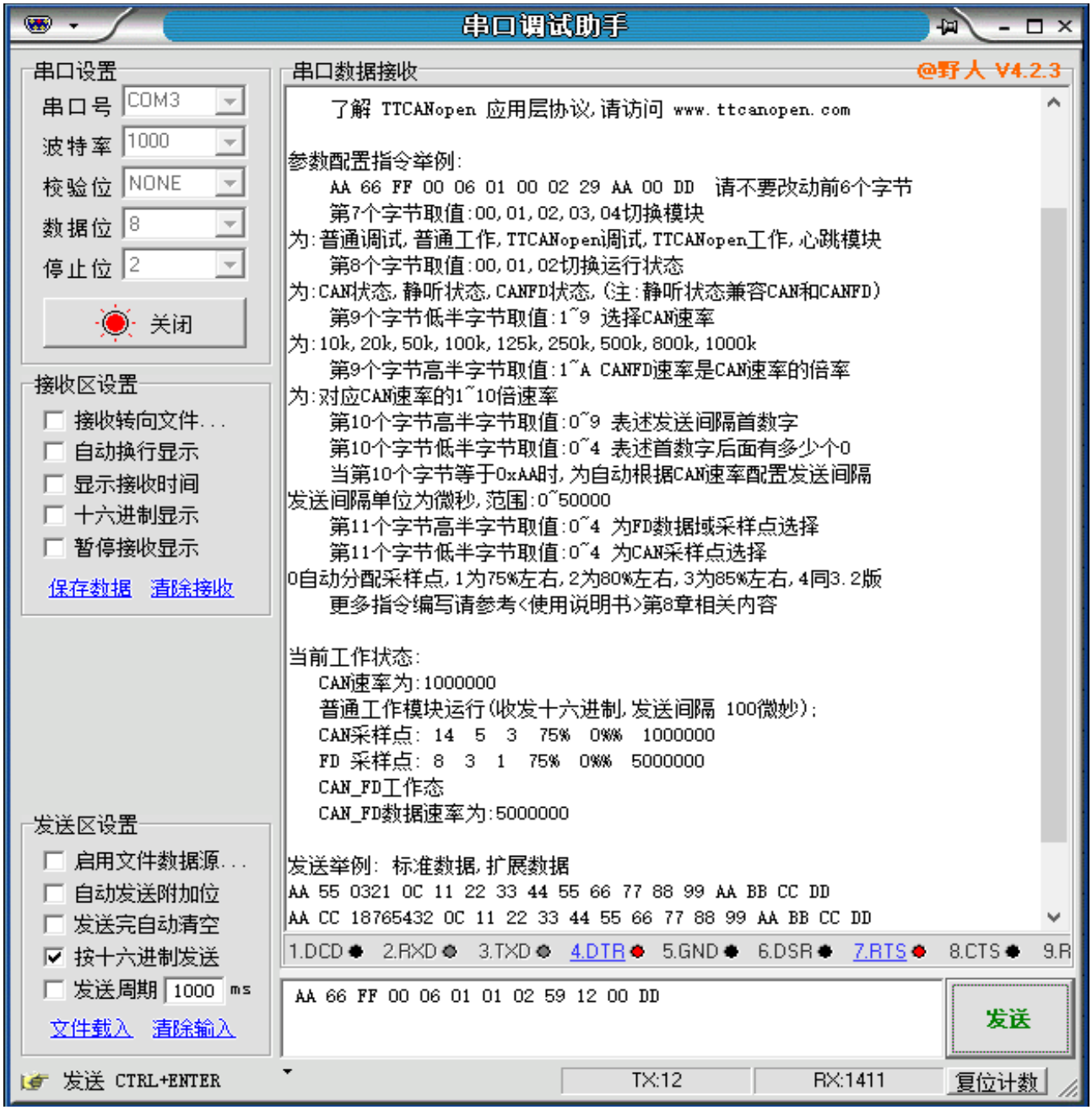


图 6 使用配置指令成功显示反馈信息举例

7. 普通调试模块

用户可以通过上节的配置指令，将设备配置为普通调试模块。

用户可以将模块配置为CAN状态或CANFD状态，以使其与被调试的CAN网络状态相匹配，当用户只需对原网络进行监听时，也可将模块配置为静听态，这样对原网络的影响最小，并能同时静听原网络中的CAN指令和CANFD指令。

为了方便用户观察原网络中指令的运行情况，普通调试模块输出为字符串格式，每帧指令独占一行，用户可使用SPI透传模块从串口调试工具中直接进行调试观察。

普通调试模块支持4种指令帧的接收，其格式如下：

1）接收标准遥控帧指令格式：

帧计数	接收时标	帧标识	ID字	DLC	帧标识
00000000至FFFFFFFF	时:分:秒:毫秒.微妙	<	0000至07EF	0至8	>

2）接收标准数据帧指令格式：

帧计数	接收时标	帧标识	ID 字	DLC	帧标识	数据场 DLC 个字节
00000000 至 FFFFFFFF	时:分:秒:毫秒.微妙	[	0000 至 07EF	0 至 8	]	xx ～ xx

3）接收扩展遥控帧指令格式：

帧计数	接收时标	帧标识	ID 字	DLC	帧标识
00000000 至 FFFFFFFF	时:分:秒:毫秒.微妙	<	00000000 至 1FBFFFFF	0 至 8	>

4）接收扩展数据帧指令格式：

帧计数	接收时标	帧标识	ID 字	DLC	帧标识	数据场 DLC 个字节
00000000 至 FFFFFFFF	时:分:秒:毫秒.微妙	[	00000000 至 1FBFFFFF	0 至 8	]	xx ～ xx

注：上述格式是按普通 CAN 状态表述的，CANFD 状态不支持遥控帧，CANFD 指令中 DLC 字段和其所对应的数据场长度，遵守 CANFD 协议相关规定。DLC 字段用 10 进制显示。

普通调试模块接收指令显示举例：

1）标准遥控帧：

00000001 00:03:07:671.200 < 0123 8 >

2）标准数据帧：

00000002 00:03:07:671.346 [0321 4] 11 22 33 44

3）扩展遥控帧：

00000003 00:03:07:671.443 < 12345678 6 >

4）扩展数据帧：

00000004 00:03:07:671.605 [18765432 3] 11 22 33

上例中:（举例:扩展数据帧）

第一个字符串“00000004”：

为接收帧计数，计数范围 0～0xFFFFFFFF;

第二个字符串“00:03:07:671.605”：

为接收时标，时：分：秒：毫秒.微妙，分辨率为 1 微秒;

第三个[扩起来]的字符串“[18765432 3]”：

为帧 ID 和 DLC，(ID 为 32 位十六进制数,低 29 位有效。DLC 为十进制数，CAN 状态为 0 至 8，CANFD 状态为 0 至 64，并遵守 CANFD 协议相关规定)；<尖括号>则表述遥控帧;

（注：对于标准帧，ID 为 16 位十六进制数，低 11 位有效）

第四个字符串“11 22 33 ”：

为数据帧中 DLC 个字节数据内容。

图 7 为通过透传模块在串口调试助手观察到的四种帧举例：标准遥控帧、标准数据帧、扩展遥控帧和扩展数据帧。

图 7 普通调试模块接收显示举例

普通调试模块支持4种指令帧的发送，并对发送帧进行“回显”，发送指令使用十六进制方式，其格式如下：

1）发送标准遥控帧指令格式：

帧头	帧标识	ID字	DLC	帧尾
AA	77	0000 至 07EF	00 至 08	DD

2）发送标准数据帧指令格式：

帧头	帧标识	ID字	DLC	数据场DLC个字节	帧尾
AA	55	0000 至 07EF	00 至 08	xx ～ xx	DD

3）发送扩展遥控帧指令格式：

帧头	帧标识	ID字	DLC	帧尾
AA	FF	00000000 至 1FBFFFFF	00 至 08	DD

4）发送扩展数据帧指令格式：

帧头	帧标识	ID字	DLC	数据场DLC个字节	帧尾
AA	CC	00000000 至 1FBFFFFF	00 至 08	xx ～ xx	DD

注：CANFD 不支持遥控帧，CANFD 指令中 DLC 字段和其所对应的数据场长度，遵守 CANFD 协议相关规定。

普通调试模块发送指令举例（十六进制方式）：

1）发送标准遥控帧：

AA 77 0123 08 DD

2）发送标准数据帧：

AA 55 0321 08 11 22 33 44 55 66 77 88 DD

3）发送扩展遥控帧：

AA FF 12345678 06 DD

4）发送扩展数据帧：

AA CC 18765432 08 11 22 33 44 55 66 77 88 DD

图 8 为在经过透传模块后，串口调试助手中普通调试模块发送指令和回显示例：标准遥控帧、标准数据帧、扩展遥控帧和扩展数据帧。

图 8 普通调试模块发送指令和回显示例

8. 普通工作模块

用户可以通过配置指令，将设备配置为普通工作模块。

用户可以将模块配置为CAN状态或CANFD状态，以使其与用户当前CAN网络状态保持一致。

为了提高指令效率，普通工作模块输入输出皆使用16进制格式，帧与帧之间使用帧头帧尾标识。

普通工作模块支持4种指令帧的输入输出，其中，发令格式与普通调试模块相同，但普通工作模块不“回显”发送指令。

普通工作模块接收指令格式如下：

1）接收标准遥控帧指令格式：

帧头	接收时标	帧标识	ID字	DLC	帧尾
BB	tt tt tt tt	77	0000 至 07EF	00 至 08	EE

2）接收标准数据帧指令格式：

帧头	接收时标	帧标识	ID 字	DLC	数据场 DLC 个字节	帧尾
BB	tt tt tt tt	55	0000 至 07EF	00 至 08	xx ～ xx	EE

3）接收扩展遥控帧指令格式：

帧头	接收时标	帧标识	ID 字	DLC	帧尾
BB	tt tt tt tt	FF	00000000 至 1FBFFFFF	00 至 08	EE

4）接收扩展数据帧指令格式：

帧头	接收时标	帧标识	ID 字	DLC	数据场 DLC 个字节	帧尾
BB	tt tt tt tt	CC	00000000 至 1FBFFFFF	00 至 08	xx ～ xx	EE

注 1：tt tt tt tt 为 32 位接收时间计数（计数单位为 0.1ms），其低字节在前，高字节在后；而 ID 字是高字节在前，低字节在后，解帧时需特别留意。

注 2：CANFD 不支持遥控帧，CANFD 指令中 DLC 字段和其所对应的数据场长度，遵守 CANFD 协议相关规定。

普通工作模块接收指令举例：

1）标准遥控帧：

BB B5 3C 02 00 77 01 23 08 EE

2）标准数据帧：

BB B6 3C 02 00 55 03 21 08 11 22 33 44 55 66 77 88 EE

3）扩展遥控帧：

BB B7 3C 02 00 FF 12 34 56 78 06 EE

4）扩展数据帧：

BB B9 3C 02 00 CC 18 76 54 32 08 11 22 33 44 55 66 77 88 EE

图 9 为普通工作模块，通过透传模块指令接收示例，分别为：标准遥控帧、标准数据帧、扩展遥控帧和扩展数据帧，在这里每帧指令并没有独立成行，而是用帧头 BB 和帧尾 EE 进行分割的。

图 9 普通工作模块接收指令示例

9. SPI 接口与通讯协议

CANFD 转 SPI 模块的 SPI 接口接点与 WIZ820io 相一致。但其修改了 RST 和 INT 两个接点的用途和含义。下面对各接点用途按排列顺序进行逐一进行描述。（I／O 按 SPI 从设备端进行描述）

图 10　CANFD 转 SPI 接口

参照图 10，其左边 6p 插座自上而下，各接点标识分别为：MI,RST,NC,3V,3V,GND。其右边 6p 插座自上而下，各接点标识为：INT,CS,SCK,MO,GND,GND。

MI -- 输出，SPI 数据的主入从出接点。

RST-- 输出，接收握手信号。★★★

NC -- 空

3V -- 输入，设备 3.3V 电源接点

GND-- 地接

INT-- 输入，指令握手信号。★★★

CS -- 输入，SPI 片选信号。

SCK-- 输入，SPI 时钟信号。

MO - 输入，SPI 数据的主出从入接点。

★★★关于接收握手信号（RST）的说明：本 CANFD 转 SPI 模块，是由单片机实现的，它在很大程度上简化了主 SPI 对 CANFD 总线的配置，使模块非常易用上手。但当单片机做为 SPI 从设备时会偶发出现发送覆盖现象。这是因为从 SPI 单片机大多是以中断方式接收主 SPI 数据的，每传输一个字节产生一次中断处理，由于从 SPI 单片机中有多重中断需要响应，其中有些中断比 SPI 接收中断更为重要和耗时，比如计时中断和 CAN 总线中断等，这就致使单片机并不能够保证总是在第一时间响应 SPI 接收中断，由于标准的 SPI 通讯协议是没有握手信号的，这就可能致使主 SPI 端，在从端还未处理上次数据时，就已经发送下一个数据了，产生数据覆盖错

误。为此设计了 SPI 接收握手信号（RST），从 SPI 在接收中断处理完成后，将该信号电平反转，通知主 SPI 可以发送下一个字节数据了，从而有效的避免了传输数据覆盖错误的出现。

★★★关于指令握手信号（INT）的说明：由于从 SPI 是通过中断接收数据的，如无特殊协议标识，从 SPI 是无法辨别在接收字节序列中，指令字的起始，从而无法正确解读指令和指令所携带的数据。为此设计了 SPI 指令握手信号（INT），当主 SPI 发送指令字的第一个字节时，预先将此接点电平拉低，此字节成功发送后，再将其置高。从 SPI 便可通过该接点电平分辨指令的起始。

通过增加两个握手信号，大大提高了 SPI 通讯的可靠性，同时，通过严格的测试验证，其系统效率并未见降低。我们使用一个 CANFD 转 SPI 和一个 SPI 转 USB 透传模块进行级联，构成一个 CANFD 转 USB，在 5M CANFD 满负荷的通讯条件下，其与单一的 CANFD 转 USB 模块比对，性能并无差别。

SPI 通讯协议：

无论是读数据还是写数据，通讯都是由主 SPI 单片机发起；从 SPI 单片机不会主动向主 SPI 单片机发送任何信息。

写指令格式（HEX）：

指令域 （两字节）	发送数据域 （n 个字节，n≤250）
41 42	xx xx xx xx xx xx ……

指令传输说明：

1）主 SPI 向从 SPI 启动写指令前，需将片选信号（CS）置低直至整条指令和数据发送结束，片选信号（CS）置高。

2）主 SPI 每发送一个字节前，先要读取接收握手信号（RST）的当前状态，字节发送后，再次持续读取接收握手信号（RST），直至信号发生反转，再发送下一个字节。

3）主 SPI 在发送指令域第一个字节（0x41）前，需将指令握手信号（INT）置低，等第一个字节成功发送后，再将指令握手信号（INT）置高。

4）当主 SPI 发送指令域第一个字节（0x41）时，从 SPI 会返回一个随机无效字节，当主 SPI 发送指令域第二个字节

（0x42）时，从 SPI 会返回一个设备状态信息字节，如果该字节内容是 0x61，说明指令被正确接收；如果该字节内容是 0x58，则设备的存储缓冲区以满，需停止指令数据的发送传输；如果该字节内容是 0xFE，则提示发送指令出错，需停止发送。

5）当主 SPI 发送数据域字节时，从 SPI 会返回上次接收到的数据。

6）发送函数举例（GD32）：

```
uint32_t Send_buf_data_to_spi(unsigned char *buf, uint32_t n)  //注: n <= 250
{
    unsigned char  j;                   //暂存从SPI返回的数据字节
    uint32_t size;                      //要发送的字节数
    FlagStatus  pin2_s,  pin2_r;        //记录接收握手信号RST的状态

    gpio_bit_reset(GPIOA, GPIO_PIN_4);  //CS片选置低, 启动发送进程
    for(int i=0; i<18; i++)   { ; }     //这句可以去掉,用于等待CS稳定被从机识别
    size = n;

    //发送指令第一个字节
    gpio_bit_reset(GPIOA, GPIO_PIN_3); //设置指令握手信号INT为低电平,表示当前传输的是命令首字节
    pin2_s = gpio_input_bit_get(GPIOA,GPIO_PIN_2);  //发送前获取接收握手信号RST的状态
    while (RESET == (SPI_STAT(SPI0) & SPI_FLAG_TBE))  { ; }  //查询发送寄存器标志
    SPI_DATA(SPI0) = (uint32_t)'A';   //向指定SPI端口发送指令第一个字节  'A'
    while(RESET == (SPI_STAT(SPI0) & SPI_FLAG_RBNE))  { ; } //查询接收数据标识
    j = SPI_DATA(SPI0);  //获取从SPI返回的字节
    for(uint32_t i=0;  i<10000000; i++)  //持续读取接收握手信号RST的状态,直至发生反转
    {
       pin2_r = gpio_input_bit_get(GPIOA,GPIO_PIN_2);  //发送数据后再次获取pin2管脚的状态
       if( pin2_r != pin2_s)  {  break; }
    }
    gpio_bit_set(GPIOA, GPIO_PIN_3); //恢复指令握手信号INT为高电平,表示当前传输的命令首字节结束

    //发送指令第二个字节
    pin2_s = gpio_input_bit_get(GPIOA,GPIO_PIN_2);  //发送前获取接收握手信号RST的状态
    while (RESET == (SPI_STAT(SPI0) & SPI_FLAG_TBE))  { ; }  //查询发送寄存器标志
    SPI_DATA(SPI0) = (uint32_t)'B';    //向指定SPI端口发送指令第二个字节  'B'
    while(RESET == (SPI_STAT(SPI0) & SPI_FLAG_RBNE))  { ; } //查询接收数据标识
    j = SPI_DATA(SPI0);  //获取从SPI返回的字节
    if(j == 'X')  {  size = 0;  }   //从机接收寄存器缓冲区满
    else if(j == 0xFE) { size = 0;  print("\r\n --- 发送指令出错 ---!", 24);}
    for(uint32_t i=0;  i<10000000; i++)  //持续读取接收握手信号RST的状态,直至发生反转
    {
       pin2_r = gpio_input_bit_get(GPIOA,GPIO_PIN_2);
       if( pin2_r != pin2_s)  {   break;  }
    }
```

```
    //发送缓冲区的n个数据
    for(uint32_t i=0; i<size; i++ )
    {
      pin2_s = gpio_input_bit_get(GPIOA,GPIO_PIN_2);  //发送前获取接收握手信号RST的状态
      while (RESET == (SPI_STAT(SPI0) & SPI_FLAG_TBE))  { ; }  //查询发送寄存器标志
      SPI_DATA(SPI0) = (uint32_t)*buf++;   //向指定SPI端口发送数据
      while(RESET == (SPI_STAT(SPI0) & SPI_FLAG_RBNE))  { ; } //查询接收数据标识
      j = SPI_DATA(SPI0);   //获取从SPI返回的字节
      for(uint32_t i=0;  i<10000000; i++)  //持续读取接收握手信号RST的状态,直至发生反转
      {
         pin2_r = gpio_input_bit_get(GPIOA,GPIO_PIN_2);
         if( pin2_r != pin2_s)  {   break;  }
      }
    }

    for(int i=0; i<18; i++)   { ; }   //这句可以去掉,用于CS信号结束延迟
    gpio_bit_set(GPIOA, GPIO_PIN_4);  //片选CS置高, 发送过程结束
    for(int i=0; i<18; i++)   { ; }   //这句可以去掉

    return size;
}
```

读指令格式（HEX）：

指令域 （两字节）	发送数据域 （发送 n 个亚元，获取从 SPI 返回对数据）
43 44	00 00 00 00 00 00 ……

指令传输说明：

1）主 SPI 向从 SPI 启动读指令前，需将片选信号（CS）置低直至整条指令和数据发送结束，片选信号（CS）置高。

2）主 SPI 每发送一个字节前，先要读取并保存接收握手信号（RST）的当前状态，字节发送后，再次持续读取接收握手信号（RST），直至信号发生反转，再发送下一个字节。

3）主 SPI 在发送指令域第一个字节（0x43）前，需将指令握手信号（INT）置低，等其成功发送后，再将指令握手信

号（INT）置高。

4）当主 SPI 发送指令域第一个字节（0x43）时，从 SPI 会返回一个随机无效字节，当主 SPI 发送指令域第二个字节（0x44）时，从 SPI 会返回一个字节特征字，其值为 n，当 n≤250 时，表明从 SPI 单片机有 n 个字节等待传输；当 n=251 时，等待传输的是 500 个字节；当 n=252 时，等待传输的是 1000 个字节；当 n=253 或 254 时，表示指令出错。

5）主 SPI 需要发送 n 个字节的 0x00 亚元，从而获取从 SPI 返回的需读取数据。

```
uint32_t get_dat_form_spi_to_buf(unsigned char *R_buf)
{
  unsigned char  j;                 //暂存从SPI返回的数据字节
  uint32_t  sizeG=0, nn=0;          //接收到写入R_buf的字节数
  FlagStatus  pin2_s,  pin2_r;      //记录接收握手信号RST的状态

  gpio_bit_reset(GPIOA, GPIO_PIN_4);  //CS片选置低, 启动发送进程
  //for(int i=0; i<18; i++)   { ; }   //这句可以去掉,用于等待CS稳定被从机识别

  //发送指令第一个字节
  gpio_bit_reset(GPIOA, GPIO_PIN_3);  //设置指令握手信号INT为低电平,表示当前传输的是命令首字节
  pin2_s = gpio_input_bit_get(GPIOA,GPIO_PIN_2);  //发送前获取接收握手信号RST的状态
  while (RESET == (SPI_STAT(SPI0) & SPI_FLAG_TBE))  { ; }  //查询发送寄存器标志
  SPI_DATA(SPI0) = (uint32_t)'C';   //向指定SPI端口发送指令第一个字节  'C'
  while(RESET == (SPI_STAT(SPI0) & SPI_FLAG_RBNE))  { ; } //查询接收数据标识
  j = SPI_DATA(SPI0);  //获取从SPI返回的字节
  for(uint32_t i=0;  i<10000000; i++)  //持续读取接收握手信号RST的状态,直至发生反转
  {
   pin2_r = gpio_input_bit_get(GPIOA,GPIO_PIN_2);
   if( pin2_r != pin2_s)  {  break; }
  }
  gpio_bit_set(GPIOA, GPIO_PIN_3);  //恢复指令握手信号INT为高电平,表示当前传输的命令首字节结束

  //发送指令第二个字节
  pin2_s = gpio_input_bit_get(GPIOA,GPIO_PIN_2);  //发送前获取接收握手信号RST的状态
  while (RESET == (SPI_STAT(SPI0) & SPI_FLAG_TBE))  { ; }  //查询发送寄存器标志
  SPI_DATA(SPI0) = (uint32_t)'D';    //向指定SPI端口发送指令第二个字节  'D'
  while(RESET == (SPI_STAT(SPI0) & SPI_FLAG_RBNE))  { ; } //查询接收数据标识
  nn = SPI_DATA(SPI0);  //获取从SPI返回的字节,
  for(uint32_t i=0;  i<10000000; i++)   //持续读取接收握手信号RST的状态,直至发生反转
  {
     pin2_r = gpio_input_bit_get(GPIOA,GPIO_PIN_2);  //发送数据后再次获取pin2管脚的状态
     if( pin2_r != pin2_s)  {   break;  }
  }

  if((nn == 254) || (nn == 253) ) { sizeG = 0;  print("\r\n  ---- 接收指令出错 --- !!!", 28); }
  else if(nn == 252)    { sizeG = 1000; }
  else if(nn == 251)    { sizeG = 500; }
  else if(nn == 250)    { sizeG = 250; }
  else                  { sizeG = nn; }
```

```
  //接收n个从机数据写入R_buf
  for(int i=0; i<sizeG; i++ )
  {
    pin2_s = gpio_input_bit_get(GPIOA,GPIO_PIN_2);  //发送前获取接收握手信号RST的状态
    while (RESET == (SPI_STAT(SPI0) & SPI_FLAG_TBE))  { ; }  //查询发送寄存器标志
    SPI_DATA(SPI0) = (uint32_t)(0x00);   //向指定SPI端口发送数据亚元
    while(RESET == (SPI_STAT(SPI0) & SPI_FLAG_RBNE))  { ; }  //查询接收数据标识
    j = SPI_DATA(SPI0);   //获取从SPI返回的字节,
    *R_buf++ = j;         //将返回数据写入R_buf
    for(uint32_t i=0;  i<10000000; i++)   //持续读取接收握手信号RST的状态,直至发生反转
    {
       pin2_r = gpio_input_bit_get(GPIOA,GPIO_PIN_2);
       if( pin2_r != pin2_s)  {   break;  }
    }
  }

  //for(int i=0; i<18; i++)   { ; }   //这句可以去掉,
  gpio_bit_set(GPIOA, GPIO_PIN_4);    //片选CS置高, 接收过程结束
  //for(int i=0; i<18; i++)   { ; }   //这句可以去掉,

  return sizeG;
}
```

在 CANFD 转 SPI 模块设计有两个循环缓冲区，一个 4K 的循环缓冲区，接收来自主 SPI 的数据，单片机对该区数据进行解析组帧后，发送到 CAN 总线上去；一个 16K 的循环缓冲，接收来自 CAN 总线的数据，等待主 SPI 进行读取。主 SPI 必须在 16K 循环缓冲区填满之前读取数据。

在设备上电后，主 SPI 单片机需对 SPI 接口进行初始化，才能正常工作，下面以 GD32 单片机为例，给出主 SPI 初始化代码。

```
void SPI_Configuration(void)
{
    //  PA2 ---- RST  接收握手
    //  PA3 ---- INT  指令握手
    //  PA4 ---- CS   片选信号
    //  PA5 ---- SCK  时中藕庞
    //  PA6 ---- MI   主入从出
    //  PA7 ---- MO   主出从入

    spi_parameter_struct spi_init_struct;
    rcu_periph_clock_enable(RCU_GPIOA);
    rcu_periph_clock_enable(RCU_SPI0);

    gpio_init(GPIOA, GPIO_MODE_AF_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_5 | GPIO_PIN_7 );
    gpio_bit_set(GPIOA, GPIO_PIN_5);
    gpio_bit_set(GPIOA, GPIO_PIN_7);
    gpio_init(GPIOA, GPIO_MODE_IN_FLOATING, GPIO_OSPEED_50MHZ, GPIO_PIN_6);
    gpio_bit_set(GPIOA, GPIO_PIN_6);
    gpio_init(GPIOA, GPIO_MODE_OUT_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_4);
    gpio_bit_set(GPIOA, GPIO_PIN_4);

    spi_init_struct.trans_mode           = SPI_TRANSMODE_FULLDUPLEX;
    spi_init_struct.device_mode          = SPI_MASTER;;
    spi_init_struct.frame_size           = SPI_FRAMESIZE_8BIT;;
    spi_init_struct.clock_polarity_phase = SPI_CK_PL_LOW_PH_1EDGE;
    spi_init_struct.nss                  = SPI_NSS_SOFT;
    spi_init_struct.prescale             = SPI_PSC_8 ;
    spi_init_struct.endian               = SPI_ENDIAN_MSB;

    spi_init(SPI0, &spi_init_struct);

    spi_crc_polynomial_set(SPI0,7);
    spi_enable(SPI0);

    gpio_init(GPIOA, GPIO_MODE_IN_FLOATING, GPIO_OSPEED_50MHZ, GPIO_PIN_2);
    gpio_init(GPIOA, GPIO_MODE_OUT_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_3);
    gpio_bit_set(GPIOA, GPIO_PIN_3);
}
```

说明书以下部分与 TTCANopen 协议有关，有需求的用户可以继续阅读。

本品为业界首个支持 TTCANopen 应用层协议的 CAN(FD)转 SPI 模块，用户可以访问 www.ttcanopen.com 网站获得更多的 TTCANopen 相关知识。

TTCANopen 是由国人打造的一款拥有自主知识产权的 CAN 应用层协议，与以往的应用层协议不同， TTCANopen 实际上是一个开放的协议框架。它是由众多的“基本子协议”和“应用子协议”构成，用户可以根据自己的专业应用特点，去选择“应用子协议”;也可以使用“基本子协议”去生成新的“应用子协议”; TTCANopen 协议在应用层界面直接兼容最新的 CAN_FD。

TTCANopen 应用层协议以连续寄存器空间为信息载体，使用了两个不同的 CAN 标识符划分的方式分别传递过程变量和非过程变量，巧妙的解决了寄存器空间分层管理与其指令优先级分配不平衡的矛盾。

TTCANopen 还特别规范了设备内部多条指令的发送原则，这是其它应用层协议所不具备的。

TTCANopen 引入了系统心跳的概念，用以同步总线上的设备，使系统拥有统一的时钟，并使用时间相位矩阵取代了传统的指令查询矩阵，提高了总线带宽的利用率，大大的减小了浪涌出现的概率，使应用系统实现了由传统的指令驱动到现代的时间驱动的跨越。

TTCANopen 应用层协议对 CAN 标识符的划分及其物理含义的分配，如:优先级段、寄存器基地址段、设备编号段、功能码段都具有明确的应用层协议指令含义，据此设计的应用层协议具有很好的直读性，易于解析，大大降低了学习入门的难度。是一款非常适合推广和普及的应用层协议。

另外，TTCANopen 有着类 modbus 特性，使其能够非常容易的和市面上大多数组态软件完美对接。为用户构建应用系统提供了极大的便利。

10. TTCANopen 调试模块

用户可以通过配置指令，将设备配置为 TTCANopen 调试模块。

用户可以将模块配置为 CAN 状态或 CANFD 状态，以使其与被调试的 TTCANopen 网络状态相匹配，当用户只需对原网络进行监听时，也可将模块配置为静听态，这样对原网络的影响最小，并能同时静听原网络中的 CAN 指令和 CANFD 指令。

为了方便用户观察原网络中指令的运行情况，TTCANopen 调试模块输出为字符串格式，每帧指令独占一行，用户可以通过透传模块在串口调试助手中进行调试观察。

与普通模块不同，TTCANopen 模块将 29 位 ID 字拆分为四个具有实际物理含义的子段，分别为：优先级段、寄存器基地址段、设备编号段和功能码段，使其能够贴近实际应用构成一套完整的应用层指令系统和管理系统。

TTCANopen 调试模块支持 4 种 TTCANopen 指令帧的接收，其格式如下：

1）TTCANopen 扩展数据帧指令格式：（在用）

帧计数	接收时标	帧标识	优先级	寄存器基地址	设备地址	指令	DLC	帧标识
00000000至FFFFFFFF	时:分:秒:毫秒.微妙	[	0或1	0000至FBFF	00至7F	00至1F	00至08	]

数据场 DLC 个字节
xx ～ xx

2）TTCANopen 扩展遥控帧指令格式：（未用）

帧计数	接收时标	帧标识	优先级	寄存器基地址	设备地址	指令	DLC	帧标识
00000000至FFFFFFFF	时:分:秒:毫秒.微妙	<	0或1	0000至FBFF	00至7F	00至1F	00至08	>

3）miniTTCANopen 标准数据帧指令格式：（未用）

帧计数	接收时标	帧标识	寄存器地址	段地址	DLC	帧标识	数据场 DLC 字节
00000000至FFFFFFFF	时:分:秒:毫秒.微妙	[	00至FD	00至07	00至08	]	xx ～ xx

4）miniTTCANopen 标准遥控帧指令格式：（未用）

帧计数	接收时标	帧标识	寄存器地址	段地址	DLC	帧标识
00000000至FFFFFFFF	时:分:秒:毫秒.微妙	<	00至FD	00至07	00至08	>

注 1：上述格式是按普通 CAN 状态表述的，CANFD 状态不支持遥控帧，CANFD 指令中 DLC 字段和其所对应的数据场长度，遵守 CANFD 协议相关规定。DLC 字段用 10 进制显示。

注 2：对于标准帧，支持 miniTTCANopen 协议，尚未推出。

注 3：TTCANopen 应用层协议只使用了 29 位扩展数据帧。

TTCANopen 调试模块接收帧举例：

1）TTCANopen 扩展数据帧：（在用）

00000012 23:18:56:566.211 [1 7100 32 19 2] 11 22

2）TTCANopen 扩展遥控帧：（注：未启用）

00000013 23:18:56:566.623 < 1 7100 32 19 8 >

3）miniTTCANopen标准数据帧：（注：未启用）

00000014 23:18:56:567.135 [35 3 4] 11 22 33 44

4）niniTTCANopen标准遥控帧：（注：未启用）

00000015 23:18:56:567.376 < 35 3 8 >

上例中:（TTCANopen扩展数据帧）

第一个字符串“00000012”：

为接收帧计数，计数范围 0～0xFFFFFFFF;

第二个字符串“23:18:56:566.211”：

为接收时标，分辨率为1微秒;

第三个[扩起来]的字符串“[1 7100 32 19 2]”：

分别为 1位优先级、16位寄存器基地址、7位设备地址、5位指令功能码和DLC;　（DLC为十进制数，CAN状态其为0至8，CAN_FD状态为0至64，并遵守CAN_FD协议相关规定）<尖括号>则表述遥控帧;

第四个字符串“11 22”：

为数据帧中DLC个字节数据内容。

图 11 为通过透传模块在串口调试助手接收窗口观察到的四种帧举例，分别为：miniTTCANopen 标准遥控帧、miniTTCANopen 标准数据帧、TTCANopen 扩展遥控帧和 TTCANopen 扩展数据帧。

图 11 TTCANopen 调试模块接收显示举例

TTCANopen 调试模块支持 4 种指令帧的发送，并对发送帧进行“回显”，发送指令使用十六进制方式，其格式如下：

1）发送 TTCANopen 扩展数据帧指令格式：（在用）

帧头	帧标识	优先级	寄存器基地址	设备地址	指令	DLC	数据场 DLC 个字节	帧尾
AA	CC	00 或 01	0000 至 FBFF	00 至 7F	00 至 1F	00 至 08	xx ～ xx	DD

2）发送 TTCANopen 扩展遥控帧指令格式：（未用）

帧头	帧标识	优先级	寄存器基地址	设备地址	指令	DLC	帧尾
AA	FF	00 或 01	0000 至 FBFF	00 至 7F	00 至 1F	00 至 08	DD

3）发送 miniTTCANopen 标准数据帧指令格式：（未用）

帧头	帧标识	寄存器地址	段地址	DLC	数据场 DLC 个字节	帧尾
AA	55	00 至 FD	00 至 07	00 至 08	xx ～ xx	DD

4）发送miniTTCANopen标准遥控帧指令格式：（未用）

帧头	帧标识	寄存器地址	段地址	DLC	帧尾
AA	77	00 至 FD	00 至 07	00 至 08	DD

注1：寄存器基地址为16位十六进制数，高地址在前，低地址在后。支持标准帧的miniTTCANopen协议格式尚未推出。

注2：CANFD不支持遥控帧，CANFD指令中DLC字段和其所对应的数据场长度，遵守CANFD协议相关规定。

TTCANopen发送指令举例：

1）扩展数据帧：（注：TTCANopen协议只使用扩展数据帧）

AA CC 01 7100 32 19 08 11 22 33 44 55 66 77 88 DD

2）扩展遥控帧：（注：未启用）

AA FF 01 7100 32 19 08 DD

3）标准数据帧：（注：未启用）

AA 55 35 03 08 11 22 33 44 55 66 77 88 DD

4）标准遥控帧：（注：未启用）

AA 77 35 03 08 DD

图 12 为通过透传模块串口调试助手中 TTCANopen 调试模块发送指令和回显示例，分别为：miniTTCANopen 标准遥控帧、miniTTCANopen 标准数据帧、TTCANopen 扩展遥控帧和 TTCANopen 扩展数据帧。

图 12 TTCANopen 调试模块发送指令和回显示例

11. TTCANopen 工作模块

用户可以通过配置指令，将设备配置为TTCANopen 工作模块。

用户可以将模块配置为CAN 状态或CANFD 状态，以使其与用户当前CAN 网络状态保持一致。

为了提高指令效率，TTCANopen 工作模块输入输出皆使用16 进制格式，帧与帧之间使用帧头帧尾标识。

TTCANopen 工作模块支持4 种指令帧的输入输出，其中，发令格式与TTCANopen 调试模块相同，但TTCANopen 工作模块不“回显”发送指令。

TTCANopen 工作模块接收格式如下：

1）接收TTCANopen 扩展数据帧指令格式：（在用）

帧头	接收时标	帧标识	优先级	寄存器基地址	设备地址	指令	DLC	数据场 DLC 个字节	帧尾
BB	tt tt tt tt	CC	00 或 01	0000 至 FBFF	00 至 7F	00 至 1F	00 至 08	xx ～ xx	EE

2）接收TTCANopen扩展遥控帧指令格式：（未用）

帧头	接收时标	帧标识	优先级	寄存器基地址	设备地址	指令	DLC	帧尾
BB	tt tt tt tt	FF	00或01	0000至FBFF	00至3F	00至1F	00至08	EE

3）接收miniTTCANopen标准数据帧指令格式：（未用）

帧头	接收时标	帧标识	寄存器地址	段地址	DLC	数据场DLC个字节	帧尾
BB	tt tt tt tt	55	00至FD	00至07	00至08	xx ～ xx	EE

4）接收miniTTCANopen标准遥控帧指令格式：（未用）

帧头	接收时标	帧标识	寄存器地址	段地址	DLC	帧尾
BB	tt tt tt tt	77	00至FD	00至07	00至08	EE

注1：tt tt tt tt为32位接收时间计数（0.1ms），其低字节在前，高字节在后；而ID字是高字节在前，低字节在

后，解帧时需特别留意。

注 2：CANFD 不支持遥控帧，CANFD 指令中 DLC 字段和其所对应的数据场长度，遵守 CANFD 协议相关规定。

在串口调试工具中接收指令举例：

1）TTCANopen 扩展数据帧：（在用）

BB 15 23 00 00 CC 01 71 00 32 19 03 11 22 33 EE

2）TTCANopen 扩展遥控帧：（注：未启用）

BB 16 23 00 00 FF 01 71 00 32 19 08 EE

3）miniTTCANopen 标准数据帧：（注：未启用）

BB 18 23 00 00 55 35 03 06 11 22 33 44 55 66 EE

4）miniTTCANopen 标准遥控帧：（注：未启用）

BB 19 23 00 00 77 35 03 08 EE

图 13 为通过透传模块在串口中，TTCANopen 工作模块，指令接收示例，在这里每帧指令并没有独立成行，而是用帧头 BB 和帧尾 EE 进行分割的。

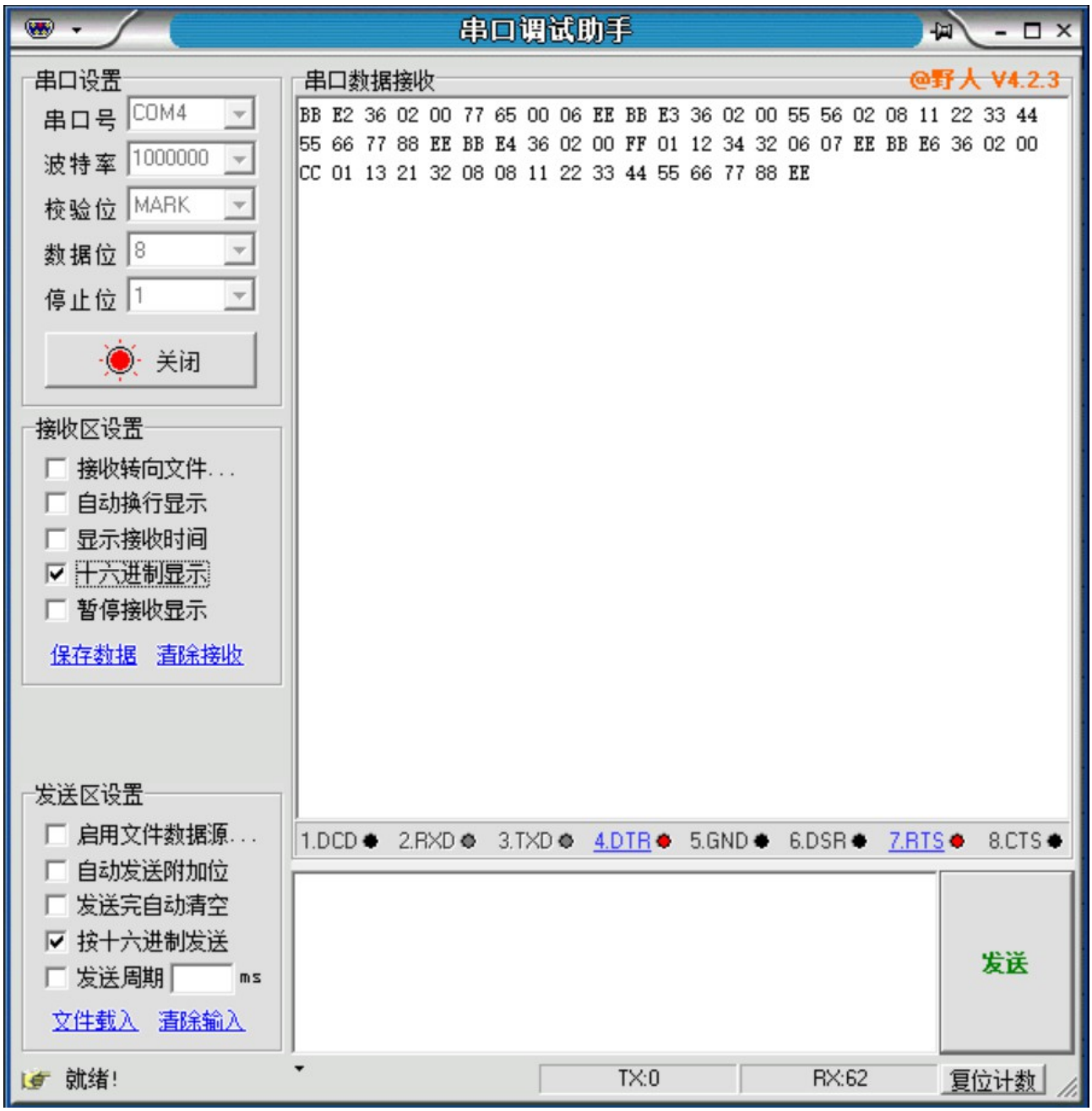


图 13 TTCANopen 工作模块接收指令演示

12. 模块 TTCANopen 特性（只针对 TTCANopen 状态）

1）模块遵从“TTCANopen 应用层协议 CAN 标识的分配方法”，见附录 1；

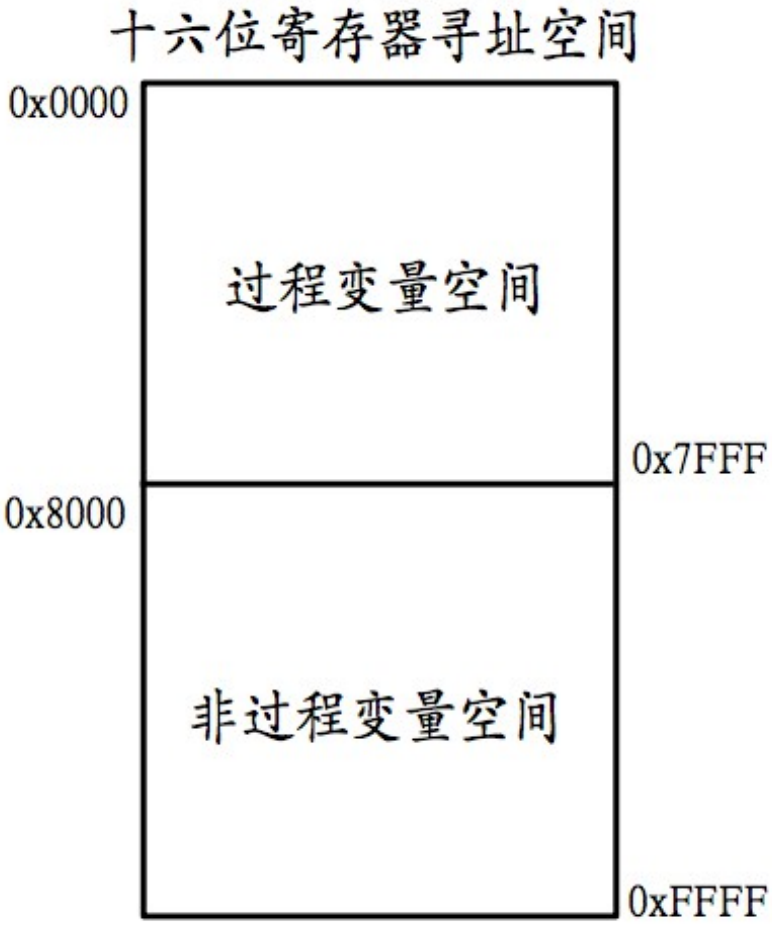
2）模块遵从“TTCANopen 应用层协议设备内部多条指令的发送规则”，见附录 2；

3）模块接受系统心跳，并与之同步，图 14 为 TTCANopen 调试模块的时间同步过程。

图 14 TTCANopen 调试模块与系统时间同步

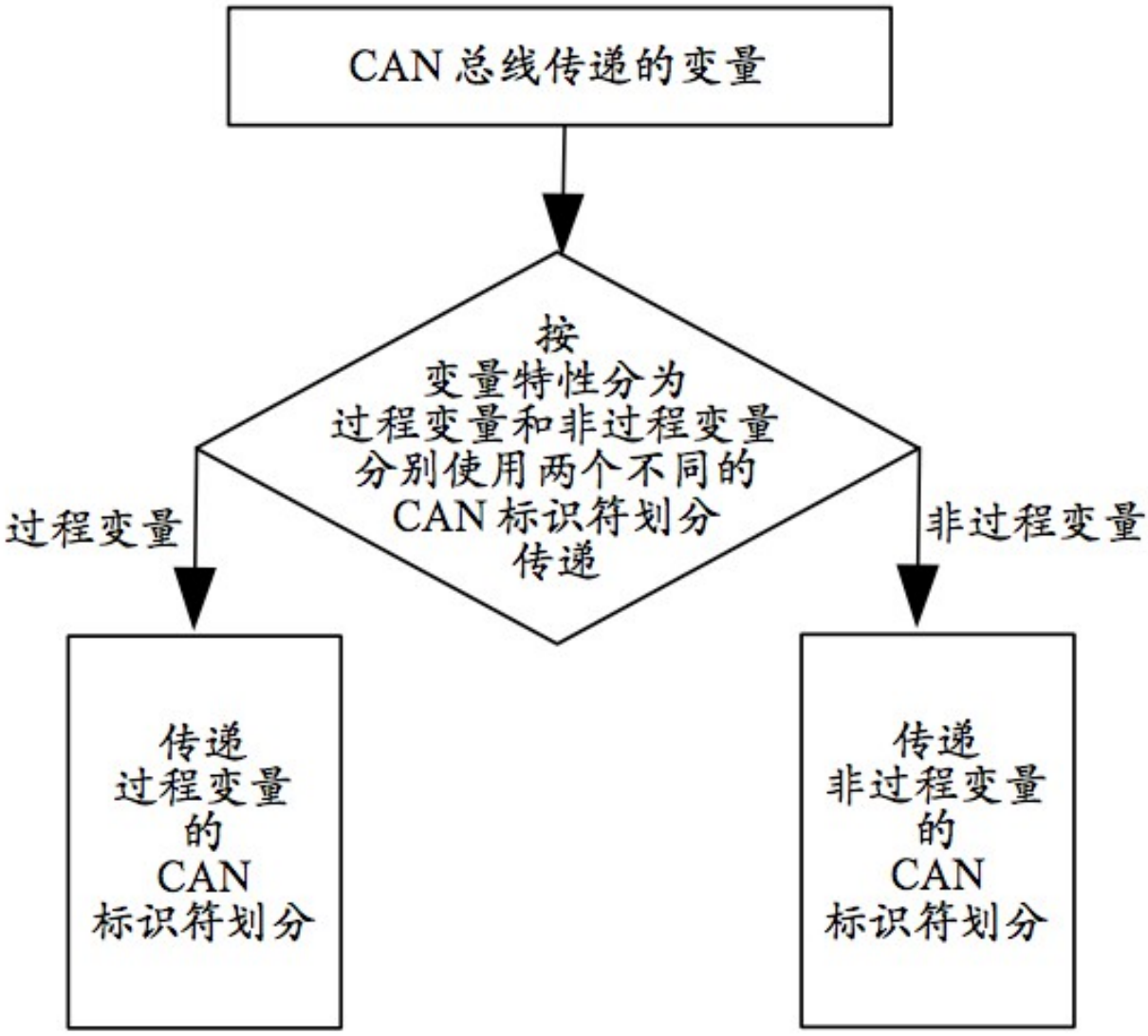
附录 1:　　TTCANopen 应用层协议 CAN 标识的分配方法

TTCANopen 应用层协议以连续寄存器空间为信息载体，并将其十六位寄存器寻址空间平分为两部分，见附图 1 所示。其中：0x0000～0x7FFF 用于存放、传递过程变量；0x8000～0xFFF 用于存放、传递非过程变量。



附图 1 寄存器空间的划分

TTCANopen 应用层协议根据CAN总线传递变量特性的不同，即过程变量或非过程变量，使用两个不同的CAN标识符划分分别传递上述两种变量，见附图2所示，从而为两种变量提供了不同的总线竞争途径，并为应用层协议实施生产者-消费者模型和主机-从机模型提供了明确的对象指向。



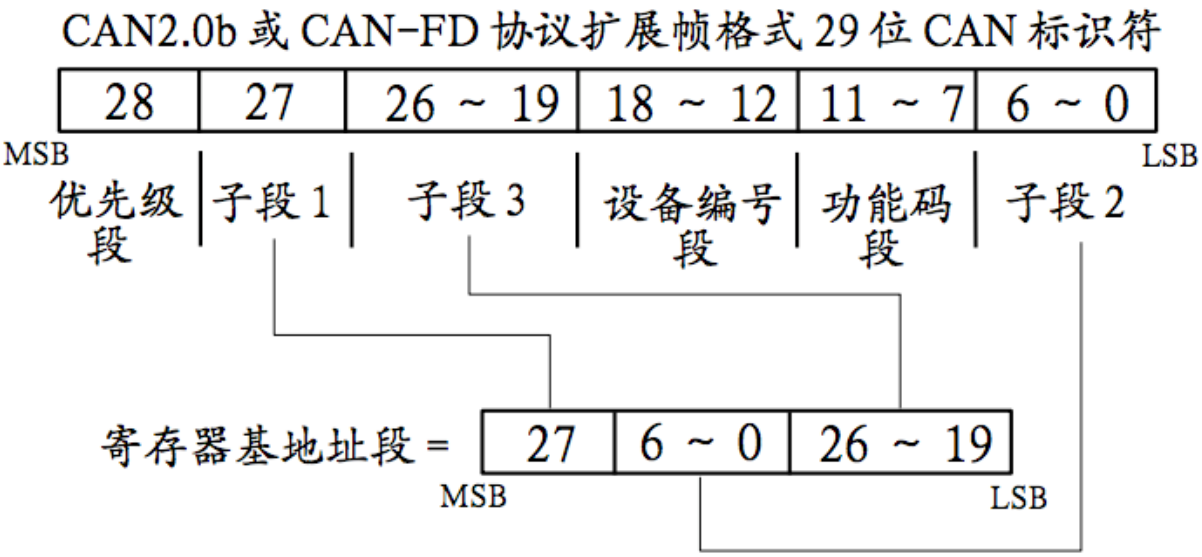
附图2 使用不同的CAN标识符划分传递两种不同的变量

传递过程变量的CAN标识符划分，采用了在CAN标识符中抽取多个位片段，并重新排列拼接组合的方式，构成应用层传递过程变量的寄存器基地址段，从而可以在过程变量寄存器基地址空间构成多个等价竞争层，很好的解决了变量分层管理与总线竞争分配不平衡的矛盾。

传递过程变量的CAN标识符划分，见附图3所示，其将CAN2.0b或CAN_FD协议扩展帧格式中29位CAN标识符划分为六段，即:优先级段、子段1、子段3、设备编号段、功能码段和子段2，其中:子段1、子段2和子段3排列拼接组合构成寄存器基地址段，各段组成说明如下:

优先级段:　由CAN标识符的最高位第28位组成;
子段1:　由CAN标识符的第27位组成;
子段3:　由CAN标识符的第26～19位组成;
设备编号段:由CAN标识符的第18～12位组成;
功能码段:　由CAN标识符的第11～7位组成;
子段 2:　由CAN标识符的第6～0位组成;

寄存器基地址段:由子段1、子段2和子段3排列拼接组合而成，即:由CAN标识符的第27位、6～0 位和 26～19位组成十六位的寄存器基地址段。



附图 3 传递过程变量的 CAN 标识符划分

当 CAN 标识符的第 27 位(即:寄存器基地址段的最高位)为 0 时，寄存器基地址空间指向过程变量空间(即:0x0000～0x7FFF)，由于十六位寄存器基地址段的低八位(26～19 位)处于 CAN 标识符的较高端 MSB，而其次高七位(6～0 位)处于 CAN 标识符的低端 LSB，由此过程变量寄存器空间变量的竞争能力由其十六位寄存器基地址段的低八位决定，从而在过程变量寄存器地址空间形成了 128 个等价竞争层，每层包含 256 个地址空间，在每层相同位置上的过程变量具有相同的总线竞争能力，在同层中，低地址变量的竞争能力优于高地址变量。

传递非过程变量的 CAN 标识符划分，见附图 4 所示，其将 CAN2.0b 或 CAN_FD 协议扩展帧格式中 29 位 CAN 标识符划分

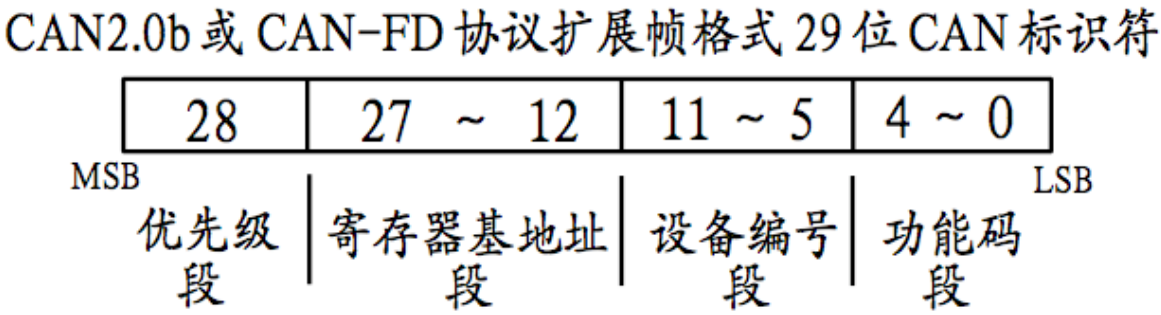
为四段，即:优先级段、寄存器基地址段、设备编号段和功能码段，各段组成说明如下:

优先级段:　　由 CAN 标识符的最高位第 28 位组成;

寄存器基地址段:由 CAN 标识符第 27～12 位组成;

设备编号段:　　由 CAN 标识符的第 11～5 位组成;

功能码段:　　由 CAN 标识符的第 4～0 位组成;



附图 4 传递非过程变量的 CAN 标识符划分

一般非过程变量的实时性要求不高，在应用层通常适用主机-从机模型，主机和从机间采用问答方式通讯，变量分段管理与其总线竞争能力没有突出的矛盾，因此从 CAN 标识符的高端直接截取十六位(27～12 位)构成寄存器基地址段，当 CAN 标识符的第 27 位(即: 寄存器基地址段的最高位)为 1 时，寄存器基地址空间指向非过程变量空间(即:0x8000～0xFFFF)，

非过程变量的总线竞争能力整体低于过程变量，这是应用层所期望的，由于CAN标识符的高七位不能连续出现高电平1，当优先级位(28位)为1时，非过程变量实际有效的地址空间为0x8000～0xFBFF，也就是在0xF000段的尾部集中有1024个地址空间(0xFC00～0xFFFF)不能使用，如果对非过程变量采用和过程变量一样的寄存器拼接组合的方法构成寄存器基地址段，势必将这1024个不可用地址空间分散到各等价层中，这对非过程变量寄存器空间的使用和管理无益，因此应用层对过程变量和非过程变量分别采用不同的CAN标识符划分进行总线传递是非常必要的。

TTCANopen应用层协议对CAN标识符的划分及其物理含义的分配，如:优先级段、寄存器基地址段、设备编号段、功能码段都具有明确的应用层协议指令含义，据此设计的应用层协议具有很好的直读性和开放性，使应用层协议易于解析，便于用户在框架内构建适合其使用环境的应用层子协议。

TTCANopen应用层协议以连续寄存器空间为信息载体，其应用层协议指令是对该连续寄存器空间的读写访问，由此使其能够天然的兼容CAN-FD协议对数据场长度的扩充。

附录 2:　　TTCANopen 应用层协议
设备内部多条指令的发送规则

通常 CAN 通讯协议只规范了多个设备在总线上同时发送指令时的竞争顺序，而对于单个设备内部产生的多条指令的发送顺序却很少涉及。

在纯主从模式下，设备内部产生多条指令的可能性并不大，相关问题很难暴露，而当系统发展演绎到全触发方式后情况就不同了，一个设备可能具有多个触发变量，它们可以是周期触发、变化触发、越界触发等等，而总线的发送速率是有限的，且同时还存在其他设备对总线的竞争，因此在同一设备中就有可能积累多条指令等待发送。

通常设备会默认以“先生优势”规则处理这些指令，即先产生的指令优先发送，这样处理的最大好处是指令动作先后逻辑规范，这对那些具有先后关联的系列指令尤为重要，但事物总是两方面的，这种“先生优势”规则，会延迟和阻碍后产生的高优先级指令的发送，如：告警指令，尤其是当总线上存在竞争时，情况尤为严重，如:当设备指令发送端口被一条先生成的低优先级指令占据时，其在总线上的竞争处于劣势，从而使该设备中后产生的告警指令无法参与竞争而

被推迟发送，这种延迟有时可能是致命的。 因此我们需要重新规范设备内部多条指令的发送规则，最粗暴简单的方法就是对这多条指令进行优先级排队，将 ID 字小的指令排在前面，每当设备产生一条新指令时都要重新进行一次指令排序(注意:这种重新排序应当包括那条已经处在发送端口的未发指令)，这样设备便具有了最优的发送竞争力，但同时可能会破坏前面所说的某些指令的先后逻辑关联，可能使系统运转出现偏差。

为此 TTCANopen 应用层协议做出如下规范，当一设备同时具有多条指令等待发送时：

1)优先级位为 0 的指令优先于优先级位为 1 的指令发送;

2)优先级位相同的指令按其产生的先后顺序发送。

虽然只有短短的两条规定，但它却完成了其它应用层协议很难规范完成的内容，而这对于 TTCANopen 协议来说却是自然天成的，它的第一条规定确保了低优先级(1)的指令不能阻碍高优先级(0)指令的发送，使设备中诸如告警指令得以顺利抢先通过总线发送；第二条规定保障了具有先后逻辑关系的系列指令的顺利完成，因为我们一般不会将这些相关指令定义在不同的优先级位上。

更多内容请关注：www.ttcanopen.com

免责声明

本着对用户负责的精神，说明书尽可能翔实正确的对产品功能和指标进行描述，但不能保证没有遗漏和错误。另外，随着产品不断升级和完善，说明书内容也具有一定的时效性和适用性，可能会在没有特别通知的情况下，进行修改和补充，请用户及时关注 www.ttcanopen.com 网站中相关信息。感谢您的包容与支持。

补充说明：

1，GD 公司现已经将带 CANFD 的 GD32E103 系列，更名为 GD32C103 系列。