BMAPI SDK

软件开发指南

文档版本	释放日期	作者	变更内容
1.0	2025/09/13	Busmust	初始版本

目录

霸码科技（BUSMUST）的全系列总线分析仪产品均支持BMAPI通用软件编程接口，支持各种主流编程语言，包括但不限于：

- C/C++ (支持Windows/Linux操作系统，支持X86/X64/AARCH64等多个平台)
- Python(支持python-can和udsoncan开源应用库)
- Qt
- C#
- VB.NET
- Labview

简单地说，只要是支持调用dll/so动态库的开发环境，都可以使用BMAPI进行二次开发，从而实现用户自定义的各类总线分析应用层功能。霸码科技提供官方BMAPI SDK，内含动态库以及开发所需的头文件等，并提供了大量例程供您参考。

本文将用十分钟时间带大家了解如何基于BMAPI实现CANFD/LIN等协议的报文收发等功能。

准备工作

在收到总线分析仪硬件之后，开始您的二次开发工作之前，请先进行以下准备工作。

在Busmaster中成功使用分析仪进行报文收发

这一步非常关键，由于二次开发相对复杂易出错且变量因素较多难以排查，强烈建议您在我们验证通过的Busmaster官方环境下先完成初步调试，完成这一步将意味着分析仪与您的总线现场硬件兼容并且可以稳定收发：

1. 登录busmust.com官网，下载并安装最新版本的Busmaster软件，安装该软件的过程中会自动安装设备驱动程序（需要禁用驱动强制驱动签名）。将总线分析仪正确连接至需要通信的总线，然后打开Busmaster，选择已接线的分析仪通道，正确配置波特率、终端电阻、采样点，然后点击Connect
2. 在Message Window中观察接收报文是否稳定，在Transmit Window中尝试发送报文观察是否能够成功发送，如果不能，则需要返回步骤2，检查硬件连接和软件配置，直至能够稳定收发
3. 记录能够稳定工作的波特率、终端电阻、采样点等参数，稍后的二次开发工作将需要把这组正确的参数配置给分析仪硬件。

在BMAPI SDK中成功运行任意官方例程

这一步对不熟悉BMAPI的新客户朋友而言比较重要，由于二次开发相对复杂易出错且变量因素较多难以排查，强烈建议您在SDK的example文件夹中按需选择并成功编译运行至少一个例程，完成这一步将意味着您已成功搭建开发环境，能够直接通过BMAPI接口操作分析仪硬件：

1. 登录busmust.com官网，下载并解压最新版本的BMAPI SDK，注意要整体解压，不能只解压某个例程，否则无法成功编译例程
2. 打开doc/BMAPI.CHM文档，熟悉总体编程模型，今后也可以在这里搜索函数名获取函数的帮助文档（或者直接查阅BMAPI头文件中的注释）
3. 打开example文件夹，熟悉SDK提供的各个例程，选择一个与您的应用场景最接近的例程作为起点，先不对源码做任何修改，直接编译通过并确认能够正常运行（而不是从零开始直接在自己的复杂程序中调用BMAPI）

常见的基础例程包括（查看example文件夹以获得全部例程清单）：

- receive_only：仅演示如何接收报文
- transmit_only：仅演示如何发送报文
- dual_thread_txrx_cpp：演示双线程异步收发报文
- can_analyzer_qt：演示在Qt界面中收发报文（此例程有多种编程语言版本）

常见问题

Q1：Busmaster点击Connect之后收不到任何报文，甚至直接在trace窗口显示BUSOFF错误了。

A1：检查硬件接线是否正确，对端设备是否已经上电并且未处于休眠状态，万用表量取总线终端电阻是否为60欧姆（低速传输时单120欧也可以工作），波特率是否与对端设备匹配（注意CANFD有仲裁段和数据段两个波特率），然后重新插拔分析仪设备后重新连接再试。

Q2：Busmaster能接收报文，但是发送报文后软件提示BUSOFF错误?

A2：与上面的问题类似，但至少说明数字段波特率对了而且对端ECU工作正常，建议进一步检查：

1. 接线稳定可靠，没有过长，较长的连接推荐使用双绞线；
2. 总线终端电阻值符合ISO规范，总线两端各一个120欧电阻，我们的产品可程控开启其中一个120欧电阻；
3. 波特率设置界面的mode不是listen only或者lookback；
4. 数据段波特率您的ECU匹配（车载ECU一般CAN-FD数据段为2000000bps，与仲裁段不同）
5. 采样点位置与您的ECU匹配，点击波特率配置界面的Advanced按钮即可配置分析仪采样点位置（ECU一般是75%-80%，然而有的单片机默认生成的代码在50%，此时需要调整ECU采样点位置）

Q3：能够发送STD和STD+FD报文，但是无法发送STD+FD+BRS报文?

A3：CANFD有仲裁段和数据段两个波特率，当且仅当开启BRS标记时，会同时使用两个波特率，如果我们正确配置了仲裁段波特率和采样点，但是数据段尚未配置好，就会出现这样的现象。另外对于较高的数据段波特率，需要对端ECU设备的CANFD驱动程序正确设置发送延迟补偿（TDC）功能。

Q4：例程编译不通过?

A4：请完整解压缩SDK并预先安装合适的构建工具链（包括gcc），尤其是正确安装libusb（sudo apt-get install libusb-1.0-0-dev）。如果编译仍报错，欢迎将报错截图或者粘贴至霸码科技公众号后台，由技术支持专员帮您分析问题。

Q5：例程编译通过了，但是在linux下无法运行，提示找不到libbmapi64.so。

A5：Linux下有一个环境变量LD_LIBRARY_PATH，请将libbmapi64.so所在的目录添加到这个环境变量中。推荐参考{SDKDIR}/bin/unix64/release/run.sh创建一个自己的启动脚本，使用启动脚本来快速运行程序。

Q6：例程编译通过了，但是在linux下无法打开通道，提示LIBUSB_ERROR_ACCESS错误。

A6：默认情况下，需要sudo权限才能访问usb设备。如需避免sudo，也可以使用“sudo chmod -R 777 /dev/bus/usb/”命令一次性为各个usb设备添加普通用户读写权限。

Q7：我用的是Python，不知道如何配置环境，不知道如何收发报文。

A7：Python环境配置稍微复杂一些，请务必先阅读{SDKDIR}/Python开发必读.txt。我们的SDK已经适配了python-can和udsoncan两个功能强大的第三方通用开源库，这意味着您在通过python操作分析仪时，基本上不需要直接调用BMAPI了，在开发环境搭建好之后，只需要参考互联网上有关python-can和udsoncan的海量公开资料直接进行应用层编程即可。有关python-can和udsoncan的编程问题，已经超出了BMAPI二次开发的范畴，本文不做详细阐述。但如您遇到任何问题，仍可通过任意技术支持渠道联系我们以获得帮助。

报文对象

先来认识一下各个协议的报文对象，我们在收发报文时需要经常操作它们。

BM_DataTypeDef

在BMAPI中，BM_DataTypeDef是代表报文数据的公共数据结构，所有分析仪产品都使用这个抽象对象作为帧头，其内部定义如下：

```
typedef struct
{
  BM_DataHeaderTypeDef header;            /**< data header, see BM_DataHeaderTypeDef for details. */
  uint16_t length;                        /**< length in bytes of the payload byte array (header excluded) */
  uint32_t timestamp;                     /**< 32-bit device local high precision timestamp in microseconds. */
  uint8_t payload[BM_DATA_PAYLOAD_MAX_SIZE];  /**< buffer holding concrete message payload (i.e. a CAN message in BM_CanMessageTypeDef format), followed by an optional tail. */
} BM_DataTypeDef;
```

请关注其中比较关键的几个成员（详细定义请参考BMAPI.CHM）：

1. BM_DataTypeDef.header.type：报文类型，例如：
   a. BM_CAN_FD_DATA：代表BM_DataTypeDef.payload内存储着一帧接收到的CANFD报文
   b. BM_CAN_FD_DATA | BM_ACK_DATA：代表BM_DataTypeDef.payload内存储着一帧发送出去的CANFD报文，即TEF（发送完成事件）
2. BM_DataTypeDef.timestamp：报文32位硬件时间戳（如需获取64位时间戳请调用BM_GetDataPtpTimestamp，而不是获取此成员的值）

BM_CanMessageTypeDef

当(BM_DataTypeDef.header.type & ~BM_ACK_DATA)标明的数据类型是BM_CAN_FD_DATA时，说明BM_DataTypeDef.payload实际上存储着一个BM_CanMessageTypeDef结构体，该结构体内部定义如下：

```
typedef struct
{
  BM_MessageIdTypeDef id;                 /**< CAN message ID, see BM_MessageIdTypeDef for details. */
  union
  {
    BM_TxMessageCtrlTypeDef tx;           /**< TX CAN message control fields, invalid if this is NOT a TX can message. */
    BM_RxMessageCtrlTypeDef rx;           /**< RX CAN message control fields, invalid if this is NOT a RX can message. */
  } ctrl;                                 /**< CAN message control fields, whether TX or RX is taken depends on the message direction. */
  uint8_t payload[64];                    /**< CAN message payload */
} BM_CanMessageTypeDef;
```

请关注其中比较关键的几个成员（详细定义请参考BMAPI.CHM）：

1. BM_CanMessageTypeDef.id.SID：标准帧的CAN ID（扩展帧ID格式则稍复杂，推荐使用BM_GET_CAN_MSG_ID和BM_SET_CAN_MSG_ID两个helper宏）
2. BM_CanMessageTypeDef.ctrl.tx.DLC（或者rx.DLC，两者总是一致的）：报文的DLC，请注意这是CANFD长度码而不是字节数，例如DLC=0xF则代表报文负载长度为64字节而不是15字节
3. 其他报文标记如ctrl.tx.IDE（代表扩展帧），ctrl.tx.FDF（代表CANFD帧），ctrl.tx.BRS（代表使用高速数据段波特率），其中的ctrl.tx.XXX总是和ctrl.rx.XXX完全一致的。

BM_LinMessageTypeDef

当(BM_DataTypeDef.header.type & ~BM_ACK_DATA)标明的数据类型是BM_LIN_DATA时，说明BM_DataTypeDef.payload实际上存储着一个BM_LinMessageTypeDef结构体，该结构体内部定义如下：

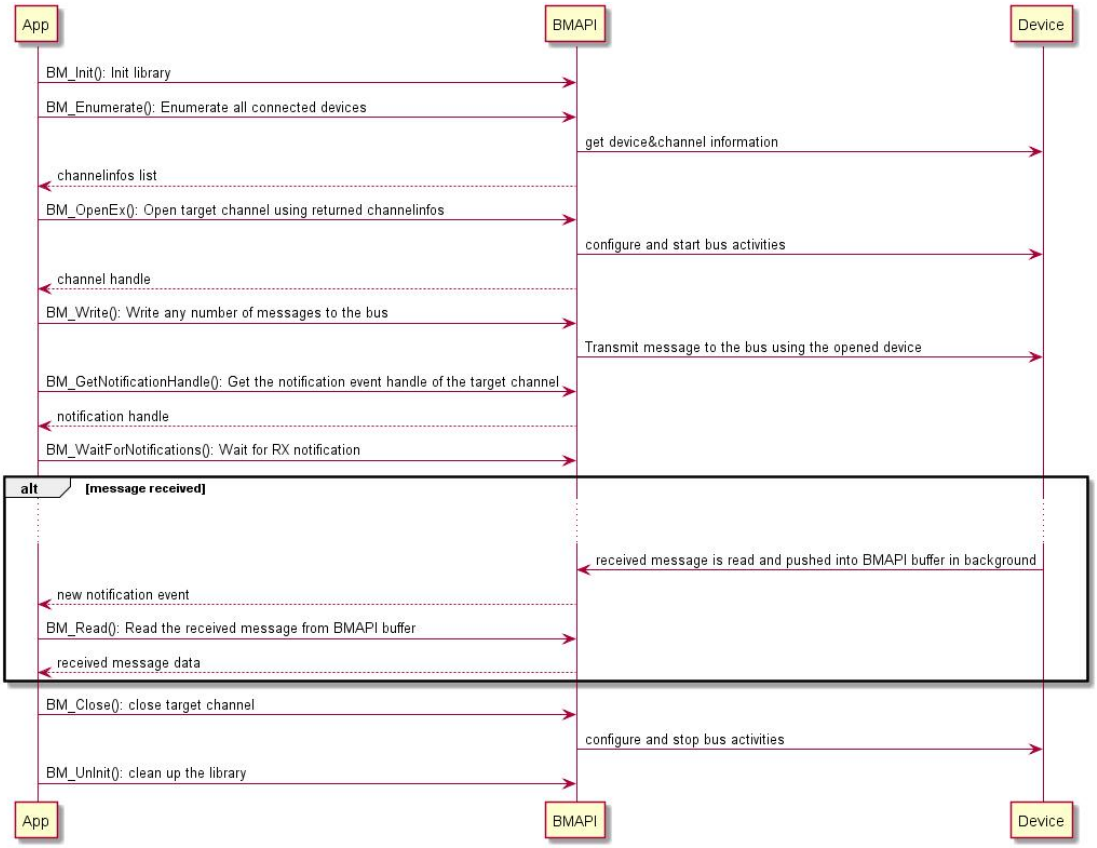
```
typedef struct
{
   uint8_t id;                          /**< LIN message ID */
   uint8_t padding[3];
   union
   {
      BM_LinMessageCtrlTypeDef lin;        /**< LIN message control fields, invalid if this is NOT a LIN message. */
   } ctrl;                              /**< Message control fields. */
   uint8_t payload[8];                     /**< LIN message payload */
} BM_LinMessageTypeDef;
```

请关注其中比较关键的几个成员（详细定义请参考BMAPI.CHM）：

1. BM_LinMessageTypeDef.id：LIN报文的帧ID（取值0-63），而不是PID
2. BM_LinMessageTypeDef.ctrl.lin.DLC：报文的DLC（0-8），同时也是负载长度
3. ctrl.lin.ENHANCED_CHECKSUM：启用增强校验
4. ctrl.lin.TRANSMIT：1代表这是一个写请求，0代表这是主机读请求
5. ctrl.lin.ERRORS：分析仪支持接收错误的LIN帧，ERRORS为错误原因

初始化

恭喜，现在您已准备完毕，可以按照下面的编程模型正式开始二次软件开发工作了。



首先请枚举并发现设备通道、打开通道并完成初始化，以下是伪代码：

```
BM_ChannelInfoTypeDef channelinfos[MAX_CHANNEL_COUNT];
BM_ChannelHandle channels[MAX_CHANNEL_COUNT];
int nchannels = MAX_CHANNEL_COUNT;

/* 初始化BMAPI库，只需要启动程序时运行一次，请勿重复调用 */
BM_Init();

/* 枚举（发现）已连接的通道，并将枚举到的通道信息保存在channelinfos数组中 */
BM_Enumerate(channelinfos, &nchannels);

for (int channelid = 0; channelid < nchannels; channelid++)
{
#ifdef TARGET_CHANNEL_NAME
    // BM_ChannelInfoTypeDef.name内含设备序列号和端口序号，因此具有全局唯一性，
    // 可用于与物理端口进行一一映射，也是多设备连接场景下区分不同设备和端口的主要信息
    if (strcmp(channelinfos[channelid].name, TARGET_CHANNEL_NAME) != 0)
        continue;
#endif

```

```
    /* 逐一打开全部需要使用的通道 */
    BM_BitrateTypeDef bitrate = { 0 };
    bitrate.nbitrate = 500; /* 仲裁段波特率 */
    bitrate.dbitrate = 2000; /* 数据段波特率 */
    bitrate.nsamplepos = 75; /* 仲裁段采样点 */
    bitrate.dsamplepos = 80; /* 数据段采样点 */
    error = BM_OpenEx(
        &channels[channelid], /* 打开成功后这个输出参数将保存打开的通道句柄 */
        &channelinfos[channelid],  /* 这是刚刚枚举到的通道信息 */
        BM_CAN_NORMAL_MODE, /* 端口工作模式，默认为normal */
        BM_TRESISTOR_120, /* 开启/关闭终端电阻 */
        &bitrate,  /* 波特率配置结构体 */
        NULL, 0 /* 硬件滤波器结构体，不需要可以不配置 */
    );
    if (error != BM_ERROR_OK)
    {
        printf(
            "Failed to open %s, error=0x%08x.\n",
            channelinfos[channelid].name, error
        );
    }
}
```

上面的伪代码中，BM_CAN_NORMAL_MODE将分析仪设置为常规模式，分析仪支持若干工作模式：

模式	功能定义
BM_CAN_NORMAL_MODE	CAN常规模式，支持CAN和CANFD
BM_CAN_CLASSIC_MODE	CAN经典模式，不支持CANFD
BM_CAN_LISTEN_ONLY_MODE	仅侦听CANFD总线，无法发送报文，不自动回复ACK，不干扰总线
BM_CAN_CONFIGURATION_MODE	关闭CAN端口
BM_CAN_NORMAL_MODE | BM_CAN_NOACK_MODE	CAN无需响应模式，此模式下发送报文不等待发送完成即可立即返回，也无法判断是否成功发送
BM_CAN_EXTERNAL_LOOPBACK_MODE	总线回环模式，发送到总线的报文自己也能收到
BM_CAN_INTERNAL_LOOPBACK_MODE	自回环模式，自己发送的报文自己会收到，但是总线收不到
BM_LIN_MASTER_MODE	LIN主机模式，支持主机读，主机写，同时支持总线监听
BM_LIN_SLAVE_MODE	LIN从机模式，支持响应主机读请求，同时支持总线监听

接收

BMAPI支持接收多种协议的报文，无论硬件通道是何种类型，我们均可以通过前面提到的BM_Data这个抽象数据类型以及BM_Read等抽象API来读取报文。

轮询接收

轮询（polling）是一种朴素的接收方式， 应用程序只需要以某种方式周期性调用**BM_Read**函数即可持续读取接收到的报文，BM_Read成功读取到一条报文时，会返回BM_ERROR_OK；而如果目前接收缓冲区内已经没有新的报文可以读取，则该函数返回BM_ERROR_QRCVEMPTY（接收缓冲区空）错误码。

请注意BM_ERROR_QRCVEMPTY错误码并不代表一个真正意义上的错误，它只是表明目前暂时没有新的数据需要读取了，您可以稍后再来尝试读取。

为了保证足够的读取速度，请务必在每次定时时间到达时持续循环读取直至读空，而不是每次定时只读取一条报文。

轮询接收的伪代码如下：

```
timeout()
{
    BM_DataTypeDef msg;
    while (BM_Read(channel, &msg) == BM_ERROR_OK)
        process_rx(msg);
}
```

除了BM_Read抽象API以外，针对具体的总线协议，还有一些helper functions，这些函数在内部实现中调用了BM_Read，同时对外提供更具体的更友好的操作接口：

- BM_ReadCanMessage
- BM_ReadLinMessage

具体请查阅BMAPI.CHM。

阻塞批量接收

阻塞批量接收是另外一种朴素的接收方式，应用程序只需要调用**BM_ReadMultiple**函数，指定接收缓冲区、预期接收报文数量以及超时时间，即可自动阻塞后续程序执行。当BM_ReadMultiple成功读取预期数量的报文时，该函数立即返回BM_ERROR_OK（而无需等待超时时间到达），否则将在超时后返回BM_ERROR_BUSTIMEOUT，此时应用程序可以通过指针类型的输出参数来判断实际收到的报文数量。

阻塞式批量接收的伪代码如下：

```
BM_DataTypeDef msgs[MAX_RX_MSG_COUNT];
int n = MAX_RX_MSG_COUNT;
BM_ReadMultiple(channel, msgs, &n, RX_TIMEOUT);
// 函数返回后，n这个输入输出类型的参数将会被变更为实际接收到的报文数量
for (int i = 0; i < n; i++)
    process_rx(msgs[i]);
```

异步通知接收

前面提到的轮询方式非常简单，但是实时性欠佳。考虑一种场景，如果当报文被硬件捕获时，我们的应用程序正好在sleep，则应用程序无法及时处理此报文，只能等到sleep结束，下一次轮询时才能读取并处理。为了提升接收实时性，BMAPI引入了“异步通知”机制。

BMAPI为每个通道（BM_ChannelHandle）提供了一个通知事件（**BM_NotificationHandle**），当这个通道收到报文或者自身报文发送完成时，会得到一次通知事件。我们可以通过**BM_WaitForNotifications**在应用程序中等待这个通知事件，当事件发生时，立即停止等待，读取并开始处理刚刚收到的报文，从而实现一种硬件捕获与软件处理逻辑之间“异步”的接收方式。

异步模式是一种高效的编程模型，实际上，前面提到的BM_ReadMultiple函数以及Busmaster上位机内部都使用了这种异步通知机制。

异步接收的伪代码如下：

```
BM_NotificationHandle notification = NULL;
/* 首先获取通道句柄 */
BM_GetNotification(channel, &notification);
/* 然后开始等待异步通知（支持同时等待多个通知，此处仅等待一个）*/
int rxChannelId = BM_WaitForNotifications(&notification, 1, RX_TIMEOUT);
if (rxChannelId >= 0)
{
    /* 通过BM_Read立即读取通知事件对应的新报文 */
    for (int i = 0; i < openedChannelCount; i++)
        while (BM_Read(channels[i], &msg) == BM_ERROR_OK)
            process_rx(msg);
    /* 无需额外的sleep操作 */
}
```

请注意BM_WaitForNotifications的“粘连”效应：

1. 当报文速率很高时，两次处理通知事件之间已经收到了多条报文，此时一次wait对应多个报文；
2. 当通道数很多时，两次处理通知事件之间已经有多个通道产生事件，但每次仅返回最靠前的有通知事件

的通道编号，此时一次wait对应多个通道。

因此，为了保证所有的报文均能够被及时处理，请务必在每次收到任意通知事件之后，都使用二维循环将所有通道的接收缓冲区全部读空，避免数据累积。

发送

BMAPI支持发送多种协议的报文，无论硬件通道是何种类型，我们均可以通过前面提到的BM_Data这个抽象数据类型以及BM_Write等抽象API来写入报文。

阻塞发送

同步阻塞发送是一种朴素的发送方式， 也就是说，应用程序只需要调用BM_Write函数，指定需要传输的一条报文，以及超时时间，即可自动阻塞后续程序执行。当BM_Write成功将报文发送至物理总线时，该函数立即返回BM_ERROR_OK（而无需等待超时时间到达），否则将在超时后返回BM_ERROR_BUSTIMEOUT。

阻塞式发送的伪代码如下：

```
BM_DataTypeDef msg;
uint32_t timestamp = 0;

/* 初始化需要发送的报文 */
uint8_t payload[64] = { 0x11, 0x22, 0x33, 0x44 };
BM_INIT_CAN_FD_DATA(
    msg,
    0x123/*id*/, 8/*dlc*/,
    0/*ide*/, 0/*fdf*/, 0/*brs*/, 0, 0,
    payload
);


if (BM_Write(channel, &msg, TX_TIMEOUT, &timestamp) == BM_ERROR_OK)
    printf("Message sent to bus @%u\n", timestamp);
```

除了BM_Write抽象API以外，针对具体的总线协议，还有一些helper functions，这些函数在内部实现中调用了BM_Write，同时对外提供更具体的更友好的操作接口：

- BM_WriteCanMessage
- BM_WriteLinMessage

具体请查阅BMAPI.CHM。

常见问题

Q1：调用BM_Write总是发送超时，可能是哪些原因？

A1：超时代表无法发送报文至总线，或者虽然发送至总线了但是我方没有收到自动回应（这是CAN协议的底层机制，由硬件自动完成）。导致这个问题的原因很多，建议先尝试使用经过验证的Busmaster上位机打开同样的端口，使用同样的波特率等配置参数，尝试发送，观察是否发送成功，不成功则修改配置参数。待Busmaster调试通过后再回到开发环境中，使用跟Busmaster一致的配置参数来初始化通道。

Q2：某种原因发送超时之后，就再也无法成功发送了，提示BUSOFF。

A2：同样的建议，首先请使用Busmaster保证能够稳定收发，然后再到二次开发软件里尝试发送。您可以参考前文中的“常见问题”章节来了解如何在Busmaster中排查BUSOFF问题。另外，您可以参考后文中的“故障恢复”章节，了解如何在意外出现BUSOFF故障之后，从故障状态中恢复并继续收发操作。

Q3：BMAPI中的各个函数操作，支持多线程吗？

A3：各个WriteXXX和ReadXXX函数均支持多线程，但是OpenEx和各个SetXXX函数不支持多线程。

阻塞批量发送

在已知将要发送大量报文的情况下，推荐您直接使用阻塞批量发送方式，以降低API调用带来的额外开销，从而显著提高发送速度。通常单帧阻塞发送仅能达到1000fps左右的发送速度，而批量阻塞发送可以达到分析仪设备的理论发送速度极限。

应用程序只需要调用**BM_WriteMultiple**函数，指定发送缓冲区、预期发送报文数量以及超时时间，即可自动阻塞后续程序执行。当BM_WriteMultiple成功发送预期数量的报文时，该函数立即返回BM_ERROR_OK（而无需等待超时时间到达），否则将在超时后返回BM_ERROR_BUSTIMEOUT，此时应用程序可以通过指针类型的输出参数来判断实际成功发送的报文数量。

阻塞式批量发送的伪代码如下：

```
BM_DataTypeDef msgs[MAX_TX_MSG_COUNT];
uint32_t timestamps[MAX_TX_MSG_COUNT];
int n = MAX_TX_MSG_COUNT;
/* 提前准备发送缓冲区里的全部报文内容 */
prepare_tx_msg(msgs, n);
/* 批量发送，函数返回后，n这个输入输出类型的参数将会被变更为实际发送成功的报文数量 */
BM_WriteMultiple(channel, msgs, &n, TX_TIMEOUT, timestamps);
for (int i = 0; i < n; i++)
    printf("TX[%d].TS (on CAN bus) = %u\n", i, timestamps[i]);
```

异步发送

在单次发送大量报文的情况下，阻塞批量发送已经可以达到极高的发送速度了，但是当收发交互较多时，往

往没有机会单次发送大量报文，此时阻塞式本身“等待发送完成”带来的额外开销再次凸显。为了彻底消除阻塞带来的额外开销，BMAPI提供了一种无需阻塞式等待的异步发送模式。

应用程序只需要在调用BM_Write或者BM_WriteMultiple的时候指定**timeout参数为0**，即可自动启用异步发送模式。此时，write函数只负责将数据放入发送缓冲区，然后立即返回，不再等待发送完成，而BMAPI会在能够发送报文的时候自动在内部后台线程中将报文发送至总线。

读取异步发送结果

异步发送模式可以轻松达到分析仪设备的理论发送速度极限，但是对于很多企业级应用场景，应用程序并不能简单地认为发送必然成功，需要获取发送结果，甚至获取在总线上成功发送时刻的时间戳，用于性能分析等。在异步模式下，您可以通过BM_Read接口来读取发送结果（含时间戳）：

1. 当BM_Read输出的BM_DataTypeDef.header.type置位了BM_ACK_DATA标记位，即(BM_DataTypeDef.header.type & BM_ACK_DATA) != 0时，说明读取到的是一个发送完成事件，也就是说这个事件对应的报文已成功发送至目标总线
2. 否则，说明读取到的是分析仪从总线其他节点接收到的一条常规报文。

无论BM_Read读取到的是发送完成事件还是正常接收报文，data中的全部信息（含负载、时间戳等）都是有效的，可以用于显示或者分析。

推荐您使用经典的收发分离双线程编程模型：

1. 首先在主线程中调用BM_OpenEx等函数完成分析仪通道初始化，并创建接收后台线程
2. 接收后台线程执行前文“异步通知接收”章节对应的接收代码，处理送完成事件以及正常接收报文
3. 可以随时在主线程中执行任意异步发送代码（timeout=0）

这种多线程发送、收发分离的编程模型既保证了极限发送速度，又实现了严谨的发送完成事件处理，是BMAPI官方推荐的操作模式，实际上BUSMASTER内部也是这样实现的。其伪代码如下：

```
/* 后台接收线程 */
int RxThread(BM_ChannelHandle channel)
{
    BM_NotificationHandle notification = NULL;
    BM_GetNotification(channel, &notification);
    int rxChannelId = BM_WaitForNotifications(&notification, 1, RX_TIMEOUT);
    if (rxChannelId >= 0)
    {
        for (int i = 0; i < openedChannelCount; i++)
            while (BM_Read(channels[i], &msg) == BM_ERROR_OK)
```

```
            if (msg.header.type & BM_ACK_DATA)
                process_tx_complete_event(msg); /* 处理发送完成事件 */
            else
                process_rx(msg); /* 处理标准接收报文 */
    }
}

/* 主线程 */
int main(void)
{
    /* 首先完成通道初始化 */
    BM_ChannelHandle channel = NULL;
    BM_OpenEx(&channel, ...);

    /* 然后创建后台接收线程（负责处理标准接收报文和发送完成事件），将通道句柄传递给后台线程 */
    CreateThread(RxThread, channel);

    /* 此后可以随时异步发送单帧或多帧报文 */
    BM_Write(channel, &data, 0/* 超时时间为0代表异步发送 */, NULL);
}
```

忽略异步发送结果

对于一些简单的测试场景，我们并不关注报文成功发送的时刻，甚至不关注报文是否发送成功。此时可以通过BM_CAN_NOACK_MODE这个特殊模式标志位来禁用发送完成事件，在异步发送提高发送速度的同时，避免异步的“发送完成事件”占用接收缓冲区，降低代码复杂度，伪代码如下：

```
/* 禁用发送完成事件 */
BM_SetCanMode(channel, BM_CAN_NORMAL_MODE | BM_CAN_NOACK_MODE);
Sleep(10);

/* 现在可以简单地异步发送了，此时任何异步发送都是立即返回并且不检查是否发送成功的 */
BM_Write(channel, &data, 0/*异步发送*/, NULL);
```

请注意，如果在您的设计中不准备使用BM_CAN_NOACK_MODE这个模式，请务必及时通过BM_Read将发送完成事件全部取出，否则当发送完成事件堆积并占满接收缓冲区的时候，BMAPI将无法继续收发报文。

ISOTP发送

对于汽车行业常用的UDS诊断相关操作，BMAPI提供了底层ISOTP流控支持。您可以通过BM_WriteIsotp和BM_ReadIsotp两个API，直接传入诊断报文内容缓冲区和长度，即可方便地收发诊断报文，而无需关注诊断

协议底层的拆包、流控和硬件收发操作。

请注意，使用这两个函数需要提供一个BM_IsotpConfigTypeDef类型的配置结构体，用于指示本次诊断会话的各项参数：

```
typedef struct
{
    uint8_t version;                    /**< Currently must be set to 0x01                                          */
uint8_t mode;
/**< See BM_IsotpModeTypeDef for details, Default mode is normal (non-extended-addressing) UDS client(tester) */
    struct
    {
        uint16_t a;                     /**< A timeout in milliseconds: =N_As if writing as tester or reading as ECU, otherwise =N_Ar */
        uint16_t b;                     /**< B timeout in milliseconds: =N_Bs if writing as tester or reading as ECU, otherwise =N_Br */
        uint16_t c;                     /**< C timeout in milliseconds: =N_Cs if writing as tester or reading as ECU, otherwise =N_Cr */
    } testerTimeout;
    struct
    {
        uint16_t a;                     /**< A timeout in milliseconds: =N_Ar if writing as tester or reading as ECU, otherwise =N_As */
        uint16_t b;                     /**< B timeout in milliseconds: =N_Br if writing as tester or reading as ECU, otherwise =N_Bs */
        uint16_t c;                     /**< C timeout in milliseconds: =N_Cr if writing as tester or reading as ECU, otherwise =N_Cs */
    } ecuTimeout;
    struct
    {
        uint8_t stmin;
/**< STmin raw value (0x00-0x7F or 0xF1-0xF9) if Busmust device is acting as UDS server. Set as 0 if acting as UDS client(normal case). */
        uint8_t blockSize;
/**< Blocksize if can card is acting as UDS server, 0 means no further FC is needed. Set as 0 if acting as UDS client(normal case). */
        uint8_t fcFrameLength;              /**< Flow control frame length in bytes                                     */
        uint8_t hardwareIsotpDisabled;
 /**< Disable BM_WriteIsotp to use 3rd generation's Hardware ISOTP support. CAUTION!!! 0=ENABLED (by default), 1=DISABLED */
    } flowcontrol;
    uint8_t extendedAddress;                /**< UDS Address in Extended Addressing mode */
    uint8_t paddingEnabled;                 /**< Enable padding for unused payload bytes */
    uint8_t paddingValue;                   /**< Padding byte value (i.e. 0xCC) for unused payload bytes */
uint8_t longPduEnabled;
/**< Enable long PDU (only if CAN message DLC>8 and (CAN_DL>8 or FF_DL>4095)), otherwise BM_WriteIsotp returns an error on long write request */
uint8_t functionalAddressingEnabled;
/**< Enable BM_ReadIsotp() to handle functional addressing UDS requests, currently only 0x7DF is supported            */
    uint8_t padding[1];
BM_IsotpCallbackHandle callbackFunc;
/**< Callback function when any progress is made, used typically by GUI to show progress bar                       */
uintptr_t callbackUserarg;
/**< Callback userarg when any progress is made, used typically by GUI to show progress bar                         */
BM_DataTypeDef testerDataTemplate;
/**< All tester messages will be formatted/checked using this template, configure CAN message ID and IDE/FDF flags here  */
```

```
42  BM_DataTypeDef ecuDataTemplate;
43  /**< All ECU messages will be formatted/checked using this template, configure CAN message ID and IDE/FDF flags here    */
44  } BM_IsotpConfigTypeDef;
```

请关注其中比较关键的几个成员（详细定义请参考BMAPI.CHM）：

1. BM_IsotpConfigTypeDef.ecuDataTemplate.SID：诊断会话中，ECU角色所使用的报文ID
2. BM_IsotpConfigTypeDef.testerDataTemplate.SID：诊断会话中，诊断仪角色所使用的报文ID
3. BM_IsotpConfigTypeDef.mode: 工作模式，当您使用BMAPI来实现诊断仪（例如进行固件升级）时，请设置为BM_ISOTP_NORMAL_TESTER；当您使用BMAPI来模拟ECU时，请设置为BM_ISOTP_NORMAL_ECU
4. BM_IsotpConfigTypeDef.ecuTimeout.b：当您使用BMAPI来实现诊断仪时，在诊断仪发出FF帧之后，需要等待ECU的FC帧响应，这个超时参数以毫秒为单位指定了该超时时间

您在需要进行大量UDS通讯时，可以尝试使用BM_WriteIsotp来替代BM_Write和BM_WriteMultiple等基础API，以简化程序设计（无需人工流控），同时获得更好的发送性能。

值得一提的是，BUSMUST第三代分析仪的固件内置了ISOTP（ISO15765）协议的流控机制，可以在收到对端的流控帧（FC）后极短的延迟内（微秒级别）立即启动连续帧（CF）的数据发送，并且在多个连续帧之间可以达到背靠背传输，这样ISOTP传输过程中的空闲等待时间将显著缩短。而且，CAN卡固件可保证“硬实时”，从而避免由于Windows等操作系统的随机卡顿导致的偶发超时错误。如需禁用该功能，可以将BM_IsotpConfigTypeDef.flowcontrol.hardwareIsotpDisabled设置为true。

使用ISOTP API进行UDS诊断操作的伪代码如下：

```
1  /* 准备UDS请求和响应缓冲区，请注意该缓冲区无需添加ISOTP流控头 */
2  uint8_t request[] = {0x2e, 0x01, 0x80, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66};
3  uint8_t response[4096];
4  uint32_t len = sizeof(response);
5  /* 准备ISOTP配置 */
6  BM_IsotpConfigTypeDef isotp = { 0 };
7  isotp.version = 1;
8  isotp.mode = BM_ISOTP_NORMAL_TESTER; /* Acting as a UDS Tester to download data to ECU */
9  BM_INIT_CAN_FD_DATA(isotp.testerDataTemplate, TESTER_MSG_ID, 8/*dlc*/, 0/*ide*/, 0/*fdf*/, 0/*brs*/, 0, 0,
   NULL);
10 BM_INIT_CAN_FD_DATA(isotp.ecuDataTemplate, ECU_MSG_ID, 8/*dlc*/, 0/*ide*/, 0/*fdf*/, 0/*brs*/, 0, 0, NULL);
11 isotp.paddingEnabled = 1;
12 isotp.paddingValue = 0xCCU;
13 isotp.ecuTimeout.b = 200; /* timeout for ECU to respond */
14 isotp.flowcontrol.hardwareIsotpDisabled = 0; /* Enable HW ISOTP (if available) */
15 /* 发送UDS请求 */
16 BM_WriteIsotp(channel, request, sizeof(request), TX_TIMEOUT, &isotp);
```

```
/* 接收UDS响应 */
if (BM_ReadIsotp(channel, response, &len, RX_TIMEOUT, &isotp) == BM_ERROR_OK)
{
    for (int i = 0; i < len; i++)
        printf("RESPONSE[%d] = %02x\n", i, response[i]);
}

```

定时任务发送

在很多应用场景下，需要周期性发送多个ID的报文，并对报文的时间槽（time slot）控制精度有较高的要求，例如CANFD协议的节点仿真，或者LIN协议的主机调度表等。由于电脑主机的Windows/Linux大型操作系统无法保证硬实时，偶发的卡顿或者超时问题往往比较难以解决。

针对这种情况，霸码科技的第三代分析仪硬件提供了高达64个定时发送任务，您可以通过BM_SetTxTasks来配置这些发送任务，由硬件自动执行，以保证1ms以内的时间调度精度。

在BMAPI中，每一个发送任务由一个BM_TxTaskTypeDef结构体来定义，结构体内部大体如下：

```
typedef struct
{
  uint8_t  type;                    /**< Type ID of the TX task, see BM_TxTaskTypeTypeDef for details. */
  union
  {
    uint8_t version;                /**< Version of BM_TxTaskTypeDef, set to 1 for BMAPI1.x. */
    uint8_t unused;                 /**< For backward-compatibility only */
  };
  uint8_t  flags;                   /**< CAN message control Flags, see BM_MessageFlagsTypeDef for details. */
  struct
  {
    uint8_t length : 7;             /**< Length of payload in unit given by 'lengthunit' (not DLC) */
    uint8_t lengthunit : 1;         /**< Unit of length, 0=1B, 1=128B, default as 0, that is, length in bytes */
                                /**< Note that not payload buffer might not be 100% used,
                                   user would need to check specific data header in payload for further information,
                                   i.e. ((BM_CanfdDataTypeDef*)txtask.payload)->ctrl.tx.DLC */
  };
  uint16_t delay;                   /**< Delay within tx cycle, that is, offset of tx timing slot within tx cycle (given by 'cycle' field)*/
                                /**< i.e. If cycle=50 and delay=10, this txtask will be executed at 10, 60, 110, 160, etc. */
  uint16_t cycle;                   /**< ms delay between rounds */
  uint16_t nrounds;                 /**< num of cycles, nrounds=0xFFFFU indicates INFINITE */
  uint16_t nmessages;               /**< messages per round, default as 1 message/cycle */

  union
  {
```

```
26      uint32_t id;                    /**< Generic ID field, normally you would need to set specific ID structure instead (e.g. CAN.SID). */
27      struct
28      {
29          uint32_t SID : 11;          /**< CAN Standard ID */
30          uint32_t EID : 18;          /**< CAN Extended ID */
31          uint32_t SID11 : 1;         /**< Reserved */
32          uint32_t unimplemented1 : 2;    /**< Reserved */
33      } can;
34      struct
35      {
36          uint8_t ID;                 /**< LIN message id, 0~63 */
37          uint8_t CHECKSUM;           /**< LIN manual checksum value, only valid if (flags &
    BM_LIN_MESSAGE_FLAGS_USER_CHECKSUM) != 0 */
38          uint16_t reserved;
39      } lin;
40  };
41
42  /* 篇幅限制，此处省略，每一个发送任务支持多种测试花样，并可在此处控制测试花样的参数*/
43  /* 例如可以通过incdata类型的测试花样来模拟E2E的alive counter */
44  ... pattern ...
45
46  /* 目前发送任务尚不支持完整的E2E功能，尤其是checksum功能，但预留了相关结构体成员，此处省略 */
47  ... e2e ...
48  uint8_t  payload[64];               /**< Default payload data, note this is also the template payload of the unchanged part in a volatile
    TX task */
49 } BM_TxTaskTypeDef;
```

请关注其中比较关键的几个成员（详细定义请参考BMAPI.CHM）：

1. BM_TxTaskTypeDef.type：当前发送任务对应的类型，例如：
   a. BM_TXTASK_FIXED：重复发送完全固定的ID和负载内容
   b. BM_TXTASK_INCDATA：固定报文ID，但负载内容按照指定的bit位置和宽度循环递增，可用于丢包检查，甚至用于模拟E2E的alive counter
   c. BM_TXTASK_INCID：固定报文负载内容，扫描指定的报文ID范围
2. BM_TxTaskTypeDef.id：指定了当前发送任务对应的报文ID（CANFD和LIN的id格式不同）
3. BM_TxTaskTypeDef.cycle：报文发送周期，单位为ms，周期控制精度优于1ms（硬实时保证）
4. BM_TxTaskTypeDef.delay：报文发送时间槽延迟，也就是当前报文ID在调度表周期内的槽延迟，例如某个发送任务的cycle=50，delay=5，则该报文会分别在第5ms，55ms，105ms，155ms，...依次被激活发送一次，可用于精确模拟LIN调度表等。

使用定时任务API进行报文发送的伪代码如下：

```
BM_TxTaskTypeDef txtasks[64] = { 0 };
for (int i = 0; i < scheduleTable.length; i++)
{
    txtasks[i].type = BM_TXTASK_FIXED;
    txtasks[i].cycle = scheduleTable.cycle; /* 调度周期/报文发送周期 */
    txtasks[i].delay = scheduleTable.items[i].delay; /* 调度周期内的帧延迟 */
    txtasks[i].nrounds = 0xFFFFU; /* 无限次周期发送，永不停止 */
    txtasks[i].nmessages = 1; /* 每次定时时间到达时，发送1条 */
    txtasks[i].id = scheduleTable.items[i].id;
    txtasks[i].length = scheduleTable.items[i].len; /* 报文负载长度，单位是字节 */
    txtasks[i].flags = 0; /* 可根据协议为CANFD或者LIN，指定额外的选项，例如指定CANFD.BRS */
    memset(txtasks[i].payload, 0, sizeof(txtasks[i].payload)); /* 按需发送报文负载内容 */
}
if (BM_SetTxTasks(channel, txtasks, scheduleTable.length) != BM_ERROR_OK)
    printf("Failed to configure tx tasks.\n");

/* 此时已经开始自动周期性发送各个报文 */
Sleep(1000);

/* 通过写入全空的任务列表来结束自动发送 */
BM_TxTaskTypeDef dummy[64] = { 0 };
BM_SetTxTasks(channel, dummy, 64);
```

定时任务发送可以使用极低的CPU占用率做到满载发送速率，同时保持1ms的发送时间精度，推荐在需要周期性发送报文的场景中使用。

离线（脱机）自动发送

BUSMUST第三代分析仪设备支持脱机自动执行定时发送任务，无需连接上位机。如需使用此功能，您只要在上述伪代码的BM_SetTxTasks之后，调用以下代码将各项相关配置存储在非易失内存：

```
BM_SaveConfig(channel,
    BM_CAN_MODE |
    BM_CAN_BITRATE |
    BM_CAN_TERMINAL_RESISTOR |
    BM_CAN_TXTASK_TABLE
);
```

完成上述步骤后，重启分析仪硬件，即可自动应用离线配置并进入离线发送模式。

序列发送

尽管前文中提到的定时任务支持测试花样（pattern）控制，目前却仍然无法满足AUTOSAR中完整的E2E要

求，因为E2E往往要求动态校验和（checksum），而校验和很多时候是OEM私有算法，不方便被预置于分析仪固件中。

为了在最大化利用硬件的自动发送能力的同时支持E2E，BUSMUST第三代分析仪设备在原有的单帧发送和定时发送任务基础上，又添加了一种全新的“序列播放”发送功能。您可以提前创建或者加载一个报文序列，并将该序列下载至CAN分析仪的序列播放缓冲区中，然后每次只需调用BM_SetReplay接口，即可启动一次序列播放。或者您可以配置为周期播放，则硬件会为您全自动循环播放缓冲区中的报文。相关伪代码如下：

```
BM_DeviceHandle device;
BM_GetDevice(channel, &device);

/* 提前准备发送缓冲区里的全部报文内容 */
BM_DataTypeDef msgs[MAX_TX_MSG_COUNT] = { 0 };
int n = MAX_TX_MSG_COUNT;
/* 提前准备发送缓冲区里的全部报文内容 */
prepare_tx_msg(msgs, n);
/* 将写入目标缓冲区临时切换为序列发送缓冲区REPLAYQ_BUFFER */
BM_SetBuffer(device, BM_WRITE_BUFFER, BM_REPLAYQ_BUFFER);
/* 批量将报文内容下载至序列发送缓冲区REPLAYQ_BUFFER，但不会将这些内容立即发送至总线 */
BM_WriteMultiple(channel, msgs, &n, TX_TIMEOUT, NULL);
/* 写入REPLAYQ_BUFFER完成后记得切换回默认模式 */
BM_SetBuffer(device, BM_WRITE_BUFFER, BM_DEFAULT_BUFFER);

/* 配置并启动全自动序列播放 */
BM_ReplayConfigTypeDef replay = { 0 };
replay.version = 1;
replay.mode = BM_STORAGE_ALWAYS_ON; /* 调用BM_SetReplay后立即播放序列 */
strcpy(replay.path.format, "<RAMBUF>");
replay.channels = 0xFFFFU; /* 播放序列中任意通道的报文 */
replay.direction = 0x3U; /* 播放序列中任意方向的报文 */
replay.cyclic = 1; /* 设置为1使能循环播放此序列，否则单次播放此序列 */
BM_SetReplay(device, &replay); /* 配置序列播放，并立即播放一次已下载的序列 */

/* 此时已经开始自动周期性发送各个报文 */
Sleep(1000);

/* 将模式设置为DISABLED可以关闭自动序列播放 */
replay.mode = BM_STORAGE_DISABLED;
BM_SetReplay(device, &replay);
```

离线（脱机）自动发送

BUSMUST第三代分析仪设备支持脱机自动发送已下载的报文序列，无需连接上位机。如需使用此功能，您

只要在上述伪代码的基础上调整两处：

1. 设置下载目标缓冲区时，不要使用RAM中的BM_REPLAYQ_BUFFER，而是使用非易失内存中的BM_REPLAYFILE_BUFFER
2. 下发序列发送配置时，序列缓冲区路径名不要使用<RAMBUF>，而是使用0000.BBD

最后，在BM_SetReplay之后，调用以下代码将各项相关配置存储在非易失内存：

```
BM_SaveConfig(channel,
    BM_CAN_MODE |
    BM_CAN_BITRATE |
    BM_CAN_TERMINAL_RESISTOR |
    BM_CAN_REPLAY_CONFIG
);
```

完成上述步骤后，重启分析仪硬件，即可自动应用离线配置并进入离线发送模式。

发送方法一览表

下表对比了前文中介绍的各个发送方法，请根据实际场景的需求选择最合适的发送方法。

方法	开发难度	发送速度	时间精度	通用性	功能特色
阻塞发送	简单	低	低	全系列支持	
阻塞批量发送	简单	高	低	全系列支持	
异步发送	复杂	极高	中	全系列支持	
ISOTP发送	一般	极高	N/A	仅第三代支持	支持硬件ISOTP加速
定时任务发送	一般	极高	高	第三代支持64个，前两代只支持1个	支持离线发送
序列发送	一般	极高	高	仅第三代支持	支持离线发送

无缓冲收发

在总线分析仪内部，总线收发的报文先被临时放置于缓冲区中，然后集中通过USB传输，这种实现逻辑避免了USB总线上大量短报文高频传输导致过高的overhead（额外开销）。但是，在某些极端情况下，缓存并集中传输导致的微小延迟也是不可接受的，此时又该如何处理呢?

为了进一步减小临时将报文放置于缓冲区并等待集中传输所产生的接收延迟，BMAPI提供了一种无缓冲模式。在无缓冲模式下，可以禁用BMAPI内部以及分析仪硬件的报文缓冲区，做到即来即传输，再配合异步通知机制，可以将接收延迟降低到极致。

可以通过下面的函数调用来进入无缓冲模式：

```
BM_DeviceHandle device;
BM_GetDevice(channel, &device);
BM_SetBuffer(device, BM_READ_BUFFER, BM_NO_BUFFER); /* 禁用硬件端报文缓冲区 */
BM_SetBuffer(device, BM_WRITE_BUFFER, BM_NO_BUFFER); /* 禁用主机端报文缓冲区 */
```

请注意:

1. 对于多通道设备，无缓冲模式将同时对所有通道生效，无法针对某个通道单独开启或关闭无缓冲模式，因此请确保在操作期间，任何通道都没有在收发报文
2. 无缓冲模式旨在实现极低的延迟，因此只支持异步实现，也就是说，在没有其他特殊配置的情况下（例如NO_ACK），发送报文时将立即返回，发送结果（包含成功发送到总线时刻的时间戳）将通过BM_Read返回主机

另外，无缓冲模式虽然降低了延迟，但是由于小报文快传输导致过高的overhead（额外开销），也导致降低了总吞吐量。因此当您不再需要执行一些要求极低延迟的操作之后，建议执行下面的伪代码退出无缓冲模式：

```
BM_SetBuffer(device, BM_READ_BUFFER, BM_DEFAULT_BUFFER);
BM_SetBuffer(device, BM_WRITE_BUFFER, BM_DEFAULT_BUFFER);
```

应用案例

某测试团队考虑在电脑主机端通过BUSMUST的CANFD总线分析仪模拟一个ECU的诊断响应，这要求模拟器能够快速地响应UDS底层的ISOTP请求。例如，当这个模拟程序收到10 14 2E 01 80 12 34 56时，需要以最小延迟回应30 00 00流控帧。

此时可以使用以下伪代码来实现极低延迟的收发，并测量实际响应延迟：

```
BM_DeviceHandle device;
BM_GetDevice(channel, &device);
BM_SetBuffer(device, BM_READ_BUFFER, BM_NO_BUFFER); /* 禁用硬件端报文缓冲区 */
BM_SetBuffer(device, BM_WRITE_BUFFER, BM_NO_BUFFER); /* 禁用主机端报文缓冲区 */

BM_WaitForNotifications(&notification, 1, RX_TIMEOUT); /* Wait for ISOTP FF */
BM_Read(channel, &ffmsg); /* Read ISOTP FF */
BM_Write(channel, &fcmsg, -1/* Timeout value ignored*/, NULL); /* Write ISOTP FC */
BM_WaitForNotifications(&notification, 1, TX_TIMEOUT); /* Read ISOTP FC Write Compete Event */
BM_Read(channel, &fcmsg); /* Read ISOTP FC Transmit Complete Event (with timestamp) */

uint64_t ffts = 0;
uint64_t fcts = 0;
BM_GetDataPtpTimestamp(channel, &ffmsg, &ffts);
BM_GetDataPtpTimestamp(channel, &fcmsg, &fcts);
printf("%u ns elapsed from FF to FC.\n", (uint32_t)(fcts - ffts));
```

使用500kbps的波特率配置，分别使用无缓冲模式和默认缓冲模式进行上述测试，典型延迟值分别为300us和1.2ms，性能提升明显。

时间同步

BUSMUST全系列分析仪均支持32位硬件时间戳，单位为微秒，可用于精确分析报文相对时间问题，例如两条报文之间的时间间隔，某个ID的报文周期等。无论是发送还是接收的报文，您只需要在成功发送或者接收之后，获取这条报文对应的BM_DataTypeDef.timestamp即可。32位硬件时间戳是分析仪的基础功能，默认即开启，无需额外的配置，后文不再赘述。

除此之外，BUSMUST第三代分析仪还支持PTP（精确时间协议）授时，可跨分析仪设备进行时间同步！经过PTP授时之后，每条收发的报文都可以获取到一个硬件自动记录的64位永不翻转PTP纳秒时间戳，可用于对报文时间戳精度要求较高的场景。

PTP时间戳相较于设备本地时间戳的优势在于，其中包含了绝对的年月日时分秒纳秒信息。永不翻转的含义在于，在可预见的设备使用寿命期间，该时间戳不会出现翻转或归零的问题。

设备授时

PTP功能默认是关闭的，如需启用PTP，请在打开对应的通道之后调用BM_SetPtpMode，并在成功打开所有通道之后，调用BM_SyncPtpTimes来同步主机以及所有这些通道之间的PTP时间，伪代码如下：

```
BM_ChannelInfoTypeDef channelinfos[MAX_CHANNEL_COUNT];
BM_ChannelHandle channels[MAX_CHANNEL_COUNT];
int n = MAX_CHANNEL_COUNT;
BM_Enumerate(channelinfos, &n);
for (int i = 0; i < n; i++)
{
    /* 首先打开通道，请参考前文初始化章节 */
    BM_OpenEx(&channels[i], ...);
    /* 然后开启PTP同步 */
    BM_SetPtpMode(channels[i], BM_PTP_INPUT_USB_SOF);
}
/* 最后将所有通道与主机一起进行一次授时，将主机的绝对PTP时间下发给各个通道 */
if (BM_SyncPtpTimes(channels, n) != BM_ERROR_OK)
    printf("Failed to sync PTP timestamps with host.\n");
/* 对于第三代设备，授时后无需再次同步，硬件会自动时时刻刻保持同步 */
```

获取时间戳

BMAPI提供了多个用于获取时间的API，请按需使用：

- BM_GetHostPtpTime()：获取电脑主机的实时PTP时间，格式为从1970-1-1开始的64位纳秒
- BM_GetPtpTime(channel): 获取分析仪设备的实时PTP时间，格式为从1970-1-1开始的64位纳秒

- BM_GetDataPtpTime(channel): 获取报文捕获时刻的历史PTP时间，格式为从1970-1-1开始的64位纳秒（请注意这个API在BMAPI SDK 1.13.0旧版本返回的是微秒，建议更新SDK以统一为纳秒）
- BM_GetTimestamp(channel)：获取分析仪设备的实时本地时间，格式为从上电开始的32位微秒
- data.timestamp：获取报文捕获时刻的历史本地时间戳，格式为从上电开始的32位微秒

例如，您可以通过下面的伪代码，方便地获取某条发送或者接收的报文对应的PTP时间，即该报文出现在总线上的时刻所对应的历史年月日时分秒纳秒：

```
uint64_t ptpns = 0;
BM_Read(channel, &data); /* 首先以任何方式读取到一条发送或者接收的报文记录 */
BM_GetDataPtpTimestamp(channel, &data, &ptpns); /* 然后获取其时间戳 */

/* 以下使用C语言运行时库函数来演示时间戳的格式化 */
time_t s = (time_t)(ptpns / 1000000000ULL);
uint32_t ns = (uint32_t)(ptpns % 1000000000ULL);
struct tm* t = localtime(&t);
printf(
    "Data PTP timestamp: %04d-%02d-%02d %02d:%02d:%02d.%09u",
    t->tm_year + 1900, // 年份（如2025）
    t->tm_mon + 1,     // 月份（1-12）
    t->tm_mday,        // 日期（1-31）
    t->tm_hour,        // 小时（0-23）
    t->tm_min,         // 分钟（0-59）
    t->tm_sec,         // 秒（0-59）
    ns                 // 纳秒
);
```

请注意，由于捕获报文并通过USB传输所花费的时间不为零，因此无论您在捕获报文之前还是在捕获报文之后调用BM_GetPtpTime，所获取到的时间戳都会与BM_GetDataPtpTime数值稍有偏差，这是合理的现象。如果您需要验证PTP同步精度，请将多个通道或者多个设备接入同一条目标总线并接收同一条物理报文，然后比较这些通道或设备接收该条报文对应的时间戳。

故障处理

很多企业级应用场景都要求无人值守，因此对解决方案整体的鲁棒性要求极高。而总线分析仪作为一种直接接触总线的设备，难以避免地会受到各种总线现场的异常情况的干扰而进入故障状态，例如：

1. DB9端子接触不良、对端ECU设备意外掉电等，都可能会导致正在发送CANFD报文的通道进入CANFD规范约定的BUSOFF类似的状态
2. USB线松动，主机USB端口供电不足等，都可能会导致分析仪USB通讯断开，甚至设备从电脑上消失
3. 尽管经过大量验证，但分析仪固件或者BMAPI库作为程序软件难免疏漏，不排除有部分软件缺陷会导致收发中断等异常
4. 二次开发过程中，一些错误的编程模式和API调用顺序，也会导致BMAPI运行报错，例如一些常见的未初始化、空句柄错误

为了提高应用程序的稳定性，确保能够及时检测到上述错误，并尽量实现从上述错误中恢复，推荐您在量产应用程序中加入故障检测和自动恢复逻辑。这样万一系统因某种非预期的原因进入异常状态，也能快速从故障中恢复，避免因长时间服务中断而需要人工介入恢复。

推荐您将下面的recoverFromError示例代码集成至您的工程中，通过以下两个条件触发执行：

1. 定时执行recoverFromError函数以扫描您的系统是否发生故障，此时port传入-1代表全部扫描
2. 当BM_Write等写入API返回错误时，说明系统已经发生严重问题，应该立即调用recoverFromError尝试强制修复问题，此时port传入出错的通道号，代表仅强制修复该通道

```
void recoverFromError(int port /* = -1*/)
{
#ifdef QT_VERSION_MAJOR
#define DEMO_PRINT qWarning
#elif defined(_WIN32)
#define DEMO_PRINT OutputDebugString
#else
#define DEMO_PRINT printf
#endif

    // Note: It takes some time to run BM_Close and BM_OpenEx,
    // and channel handles will be invalidated when recovering from error,
    // make sure no other threads are using channel handles (e.g. calling BM_Write) during the recovery,
    // or, use single-threaded design, just like this demo.
    for (int i = 0; i < openedChannelCount; i++)
    {
        uint64_t currentTs = BM_GetHostPtpTime();
        uint64_t elapsed = currentTs - channelRecoveryConfigs[i].previousRecoveryTs;
```

```
19      if (elapsed >= 1000000000ULL)
20      {
21          BM_CanStatusInfoTypeDef canStatus;
22          BM_StatusTypeDef status = BM_GetStatus(channels[i], &canStatus);
23          if (status & BM_ERROR_INITIALIZE)
24          {
25              // BM_Init() is not called yet.
26              // Read our SDK documentation for details.
27              DEMO_PRINT("BM_ERROR_INITIALIZE\n");
28          }
29          else if (status & BM_ERROR_ILLPARAMVAL)
30          {
31              // Channel handle is invalid.
32              // Maybe it's not opened yet (using BM_OpenEx) or already closed?
33              DEMO_PRINT("BM_ERROR_ILLPARAMVAL\n");
34              BM_OpenEx(&channels[i], &channelRecoveryConfigs[i].info, channelRecoveryConfigs[i].mode, channelRecoveryConfigs[i].tres,
    &channelRecoveryConfigs[i].bitrate, &channelRecoveryConfigs[i].rxfilters[0], 1);
35              channelRecoveryConfigs[i].previousRecoveryTs = currentTs;
36          }
37          else if (status & BM_ERROR_ILLOPERATION)
38          {
39              // USB Device operation failed.
40              // Maybe the device is unplugged from host PC?
41
42              DEMO_PRINT("BM_ERROR_ILLOPERATION\n");
43              // Close all channels in the same device.
44              uint64_t recoveredChannelMask = 0;
45              for (int j = i; j < openedChannelCount; j++)
46              {
47                  BM_ChannelInfoTypeDef siblingInfo;
48                  BM_GetChannelInfo(channels[j], &siblingInfo);
49                  if (memcmp(channelRecoveryConfigs[i].info.sn, siblingInfo.sn, sizeof(siblingInfo.sn)) == 0 &&
50                      memcmp(channelRecoveryConfigs[i].info.uid, siblingInfo.uid, sizeof(siblingInfo.uid)) == 0)
51                  {
52                      BM_Close(channels[j]);
53                      channels[j] = NULL;
54                      recoveredChannelMask |= 1ULL << j;
55                  }
56              }
57              // Try reset device and remove device from opened device list kept by bmapi.
58              //BM_ResetDevice(channels[i]);
59              for (int j = i; j < openedChannelCount; j++)
60              {
61                  if (recoveredChannelMask & (1ULL << j))
62                  {
63                      BM_OpenEx(&channels[j], &channelRecoveryConfigs[j].info, channelRecoveryConfigs[j].mode,
    channelRecoveryConfigs[j].tres, &channelRecoveryConfigs[j].bitrate, &channelRecoveryConfigs[j].rxfilters[0], 1);
```

```
                channelRecoveryConfigs[j].previousRecoveryTs = currentTs;
            }
        }
    }
    else if ((status & BM_ERROR_BUSOFF) || (status & BM_ERROR_BUSPASSIVE) ||
             (status == BM_ERROR_OK && (canStatus.TXBO || canStatus.TXBP)))
    {
        // BUSOFF
        // Maybe the remote CAN device is disconnected,
        // or you might want to check your bitrate, sample-position, tres configuration.
        if (i == port)
        {
            DEMO_PRINT("BUSOFF RECOVERY\n");
            // Sometimes BUSOFF might be reported from a good channel.
            // We only perform BUSOFF recovery on the channels that are reported to have tx problems (e.g. tx timeout).
            BM_CanModeTypeDef oldmode;
            BM_GetCanMode(channels[i], &oldmode);
            BM_SetCanMode(channels[i], BM_CAN_INTERNAL_LOOPBACK_MODE);
            static BM_CanMessageTypeDef dummyMessages[256] = { 0 };
            uint32_t nmessages = (uint32_t)(sizeof(dummyMessages) / sizeof(dummyMessages[0]));
            for (uint32_t k = 0; k < nmessages; k++)
            {
                dummyMessages[k].ctrl.tx.DLC = 1;
            }
            BM_WriteMultipleCanMessage(channels[i], dummyMessages, &nmessages, NULL, 1000, NULL);
            BM_SetCanMode(channels[i], oldmode);
            BM_ClearBuffer(channels[i]);
            channelRecoveryConfigs[i].previousRecoveryTs = currentTs;
        }
    }
  }
}
```

值得一提的是，上面的示例代码中，从BUSOFF状态恢复使用了一种较为温和的符合CAN规范标准定义的实现：当发生BUSOFF类似错误时，说明TEC（发送错误计数器）已经不低于128，我们可以先将通道设置为INTERNAL LOOPBACK，从而保证接下来必然发送成功且不会干扰总线，然后快速发送大量报文让TEC自然降低为0，从而退出BUSOFF状态，然后再回到之前的工作模式即可。

技术支持

软件开发总是一个复杂易出错的过程，您在使用BMAPI进行二次软件开发的过程中，如遇到技术问题，欢迎通过以下渠道获取官方技术支持：

- 公众号后台私信留言，如未能及时解决问题也可在留言后点击弹出的“咨询客服”按钮直接与企业客服实时1对1沟通
- 如您的问题包含日志或者源码包，请发邮件至support@busmust.com，并描述清楚问题现象
- 关于产品型号、功能支持情况等通用基础技术问题，也可咨询淘宝旗舰店旺旺客服