RM2024 香港大学ENTERPRIZE战队

上下位机通讯方案开源

PART II 下位机方案

作者: Fallengold (GUO, Zilin); Lawrence Ruan; Zoe Han (Han xinyi)

文档撰写：Fallengold (email: zguobd@connect.ust.hk)

指导/提议：Li Gaoyang, Sora

香港科技大学ENTERPRIZE战队 robomasterhkust@gmail.com

Github链接：https://github.com/hkustenterprize/RM2024-SerialDriver-STM32/tree/main

上位机通讯方案 (PART I) 开源链接： https://github.com/hkustenterprize/RM2024-RosComm

前言

随着Robomaster赛场上竞争一年比一年激烈，技术迭代水平一年比一年迅速，机器人的代码逻辑也一年比一年复杂，在这种形势下，我们对上下位机的通讯带宽与质量必然需要提出更高的要求。在赛季初期，为了提高通讯速率，我们简单在旧有的通讯模块的基础上直接拉高了波特率，然而效果并不理想，总结如下：

- 随着波特率的增大，上位机下发的信息帧往往呈现出离散化与碎片化的趋势。譬如，两个乃至多个信息帧会杂糅在一起下发，而这个过程仅触发过一次空闲帧；有的时候一个信息帧会拆成多个碎片下发给下位机，而这个过程反而会产生多个空闲帧。
- 在通讯速率提高的同时，信号的抗干扰能力也会随之下降，帧传输错误率也会必然提高，如何在正确地处理这些错误帧的同时防止解包越界等异常操作成为了我们新的研究课题。
- 在这个的赛季我们采用了双头哨兵方案， 上下位机交换的信息多而杂，因此，我们需要在驱动的协议层面设计一种兼容性强的框架来支持对不同信息帧的IO管理。
- 通讯模块暴露出的调用接口应该在保证绝对线程安全性的同时，最大化地利用MCU系统资源如DMA、缓冲区内存以及外设等，以支持高性能的信息吞吐。

之后，我们历经一个月的研发与测试，最终完成了新一代上下位机通讯模块。该模块具有如下特点。

- 高速： 经过测试，该通讯模块可以达到双工4M(收发各2M)的稳定传输速率 （CRC错误率 < 0.01%）
- 无流控：不需要额外使能CTS与RTS功能，不仅节省了IO引脚还间接地提升了通讯效率。
- 无锁：通过巧妙地使用原子操作及设计程序流来规避给共享资源上锁，在避免共享资源抢占的同时提升了系统运行的效率。
- 协议层解耦： 通讯层与协议层分离，方便用户快速地添加、删减修改传输接收的信息帧，同时复用通讯层的功能。

在Robomaster 2024的海外赛区及复活赛中，我们在每台机器人上都部署了该通讯方案，并验证了其稳定性。现决定将其开源，以供大家学习点评交流。

该模块需要与上位机模块搭配使用，详情请移步:...

芯片电路

我们将ACT脚引出，可以接入LED灯来方便查看USB的连接状态。电路设计的时候也将CTS和RTS两个流控脚引出，但我们在实际测试的时候发现流控有概率导致芯片死机且对整个通讯性能提升不显著，所以之后代码中就没有使能这两个引脚。更多关于电路板的设计，可以查看ENTERPRIZE战队G4主控板的开源 https://github.com/hkustenterprize/RM2024-MainControlBoard

基本配置

本模块的部署环境为自研的STM32F407VE和STM32G473VE两款开发板，基于队伍工具链GCC ARM GNU工具链。项目文件在STM32CUBEMX中的配置如下。

UART - USB - TTL 参数配置

- 无CTS、RTS流控
- 波特率2M bits/s
- Oversampling = 16
- Single Sample = 1
- 无校验位
- Stop bits = 2
- 数据位8bits

DMA 配置

- TX DMA： 普通模式；bytes对齐
- RX DMA： 环形模式；bytes对齐

一些额外说明

- 如果有条件，尽量要开 *16 bits oversampling* 。官方文档显示，*oversampling*值越低，采样速率越高，但是对时钟的差异敏感越大。考虑到CH343P与G4芯片之间无时钟线同步时钟，采用的异步同步方法，我们理应选用 *16 bits oversampling*。这也与线下调试的实验结果相符合。

- *Single Sample* 可开可不开，开了的话将会降低CRC错误率但是提高外设Noise Error的产生率。我们实测开了的话系统的综合表现会有略微的提升。

模块综述

我们的串口通讯模块基于HAL库封装的DMA IO函数和FreeRTOS嵌入式操作系统提供的内核级别的支持。

```
/**
  * @brief Receive an amount of data in DMA mode till either the expected number
  *        of data is received or an IDLE event occurs.
  * @note  Reception is initiated by this function call. Further progress of reception
is achieved thanks
  *        to DMA services, transferring automatically received data elements in user
reception buffer and
  *        calling registered callbacks at half/end of reception. UART IDLE events are
also used to consider
  *        reception phase as ended. In all cases, callback execution will indicate
number of received data elements.
  * @note  When the UART parity is enabled (PCE = 1), the received data contain
  *        the parity bit (MSB position).
  * @note  When UART parity is not enabled (PCE = 0), and Word Length is configured to 9
bits (M1-M0 = 01),
  *        the received data is handled as a set of uint16_t. In this case, Size must
indicate the number
  *        of uint16_t available through pData.
  * @param huart UART handle.
  * @param pData Pointer to data buffer (uint8_t or uint16_t data elements).
  * @param Size  Amount of data elements (uint8_t or uint16_t) to be received.
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_UARTEx_ReceiveToIdle_DMA(UART_HandleTypeDef *huart, uint8_t
*pData, uint16_t Size);

/**
  * @brief Send an amount of data in DMA mode.
  * @note   When UART parity is not enabled (PCE = 0), and Word Length is configured to
9 bits (M1-M0 = 01),
  *         the sent data is handled as a set of u16. In this case, Size must indicate
the number
  *         of u16 provided through pData.
  * @param huart UART handle.
  * @param pData Pointer to data buffer (u8 or u16 data elements).
  * @param Size  Amount of data elements (u8 or u16) to be sent.
```

```
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart, const uint8_t *pData,
uint16_t Size);
```

然而， 作为一个合格的异步通讯模块封装，直接调用这些HAL库内置的API是不明智的，尤其是在对通讯有较为严苛的要求的情况下。

- **DMA外设的抢占：**假设我们有两个线程，A和B， 两个线程在一个SysTick周期下各调用一次DMA发送函数向上位机发送数据帧。由于我们需要发起DMA请求对指定的一片缓冲区进行读操作，而这个操作需要一定的时间开销，假设线程A先通过DMA发送函数发起请求，DMA开始处理并且搬运线程A指定内存地址的数据， 操作系统若调度至线程B，而线程B此时尝试发起DMA请求而上一轮DMA请求尚未结束，便会造成DMA被抢占的严峻后果。
- **缓冲区资源的抢占：**例如，DMA正在接收来自上位机的数据帧并将其写入RAM的缓冲区中，此时若不对该缓冲区施加一定的保护措施而放任用户进程（CPU）随意地进行访问，便会造成数据帧的污染。

为了弥补原有HAL库层级API对异步支持的不足，在保证对缓冲区的读写安全的同时尽可能提高通讯效率，我们有必要为串口通信模块引入**数据流控制**以及缓冲区的**管理机制**。对此，我们的串口通讯又可以分为TX（发送）与RX（接收）两个部分，两个模块都有一个**环形缓冲区**，各自独立地进行数据流的控制以及资源的调度，而具体实现方式却略有不同。

同时，我们模仿Robomaster裁判系统串口通讯协议设计了模块的串口通讯协议，实现了协议层与驱动层的解**耦**，利于用户定制拓展通讯数据帧，并在驱动层对这些数据帧提供了缓存支持。

模块简图

前置知识（1）：环形缓冲区

> **圆形缓冲区**（circular buffer），也称作**圆形队列**（circular queue），**循环缓冲区**（cyclic buffer），**环形缓冲区**（ring buffer），是一种用于表示一个固定尺寸、头尾相连的缓冲区的数据结构，适合缓存数据流。——维基百科

环形队列具有**FIFO特性**（First Input First Output）且由读（read）、写(write)**双指针**维护的数据结构，广泛运用于各个适用于内存存储/交互的场景，例如，Linux内核中的 `kfifo.c` 的实现便依赖于环形缓冲区。

具体实现机制可简要的表述如下：

- 初始化一片**定长** n 的内存（数组）作为储存空间， 其有效存储空间为 $n-1$ 。初始化读写双指针 i,j 为 -1 ，为了规范化指针操作，我们规定 i 永远指向最后一位**已经写入**的内存地址，而 j 永远指向最后一位**已经读取**的内存地址。
- 用户可以向缓冲区写入大小为 s_1 大小的数据，并且更新*写指针* $i \to (i+s_1) \mod (n)$。注意到我们利用了模运算符来实现了缓冲区“环形”的特性，以下读取操作同理。
- 用户可以向缓冲区读取大小为 s_2 大小的数据，并且更新*读指针* $j \to (j+s_2) \mod (n)$。

- 当 $i = j \mod (n)$ 时候，缓冲区为空。当 $j = i + 1 \mod (n)$ 时， 缓冲区为满。在实际为用户设计读写接口时候需要谨慎地处理以上两种情况。

环形缓冲区提供了一些额外的好处：

- **无锁：**天然对单读单写（SISO）的操作提供了无锁支持。可以想象，只要不违反环形缓冲区的读写规则，读指针永远不会越过写指针造成非法的数据读取，而这个过程是不需要做上锁处理的。
- **环形DMA**: DMA外设支持配置为环形（Circular）模式， 在这种模式下DMA可以自动对一片内存循环读取，恰好与环形缓冲区的设计理念相合。

前置知识（2）：MISO 和 SISO

- SISO（Single Input Single Output）定义了一个系统只有一个输入和输出。
- MISO (Multiple Input Single Output) 定义了一个系统有多个输入和一个输出。
- MIMO (Multiple Input Multiple Output) 定义了一个系统有多个输入和多个输出。

以上概念大多被广泛的运用在控制论上(Control Theory)用来描述系统的特性。 然而，我们也可以意识到，用这些概念描述环形缓冲区乃大有裨益——有的环形缓冲区只需要支持单来源的用户写入，而有的环形缓冲区则可能需要支持多来源的用户输入，而实现这些不同特性的代码可能大相径庭。

协议层

协议层的数据帧依从裁判系统数据帧范式：

名称	数据段大小（bytes）	数据帧偏移量
帧头	3	0
帧头CRC16	2	3
用户自定义数据段	x	5
CRC16	2	5 + x

帧头

```
/**
* @File NewRosCommProtocal
*/

struct FrameHeader
{
    uint8_t sof        = START_BYTE;
    uint8_t dataLen    = 0;
    uint8_t protocolID = 0;
    // uint16_t crc16;
} __attribute__((packed))
```

名称	数据段大小（bytes)	偏移量
起始位	1	0
数据段长度（不含帧头和CRC校验位）	1	1
数据帧ID	1	2
CRC16	2	3

在数据帧末尾有必要进行CRC校验。如果没有这一步，在解包的时候便有几率使用错误的“数据段长度”信息， 从而会造成数组越界等严峻后果。

发送模块

简述

发送模块内蕴一个环形缓冲区，接受来自**多个**用户线程的写入与TX DMA的读取，因而，这个缓冲区是*MISO*的。前文已经叙述， 环形缓冲区天然对*SISO*的操作提供了无锁支持，然而在*MISO*的条件下，不同的用户线程之间对缓冲区的占用存在竞争。设想情况如下—— 线程A正在向缓冲区写入数据， 而此时操作系统的上下文被SysTick触发被转交给了线程B，而恰好线程B亦试图写入数据，由于此时线程A还来不及更新写指针，因而线程B的写入数据将会覆写线程A的写入区域，造成整片缓冲区被污染的不快后果。

解决这个问题的最直接的方法便是为环形缓冲区赋予一个互斥锁（Mutex)，将缓冲区视为一个独占资源，每当一个线程试图访问一个被其他线程占有的缓冲区资源，直到这个缓冲区的独占权被占有的线程释放， 这个线程都会被操作系统阻塞。然而，如果程序中存在大量访问同一个缓冲区的线程，这个过程将会产生大量操作系统的上下文切换，单片机的算力本就紧张，这些额外的性能开销虽然不致命但是足以令人不快。另一个容易想到的办法便是将整个写入的操作声明为原子操作，禁止所有的上下文切换，这虽然可以规避互斥锁所带来的额外的调度开销，然而缺点亦显而易见—— 由于写入内存的操作的用时开销较大，系统便会长时间地处于原子屏蔽的状态中，实时性也会因此降低。

本文提出了另外一种替代方案，通过巧妙地设置一些标志位成功地降低了原子操作的开销，同时亦能规避使用互斥锁。

至于读取部分，由于我们可以认为访问这片缓冲区的DMA通道只有一个，因此不必对这种单读取的情况进行额外的流程控制。每当一次DMA传输结束后且检测到当前的缓冲区内汤有存在待处理数据段，就会在中断中产生一个信号量以提醒发送任务发起下一次DMA请求。

程序解析

如前文所述，管理发送缓冲区的痛点即为在正确地处理**多个**用户的写入同时降低系统的开销（线程阻塞、内存拷贝等）。本模块在继承了经典的环形缓冲区的结构之上，将 *写指针*拆分为 `wIndenTail` *(尾写指针)* 与 `wIndexHead` *头写指针*。 维护算法可以简单描述如下：

状态变量与说明

符号	释义	说明
n	缓冲区大小	MISO缓冲区的大小。
i	读指针	当前等待读取的连续的缓冲区首位地址。在DMA传输完成产生中断信号时候进行更新。DMA开始传输的时候，
j_1	写尾指针	一段或多段正在被写入的连续的缓冲区的结尾。任何新开始的写入操作都会调用`enterWrite()`函数且立刻更新该变量如果某线程写入操作没有完成，而来自另一个线程的写入发生，那后者会根据当前 j_1 作为新的本地写入数据的起始点。
j_2	写头指针	一段或多段正在被写入的连续的缓冲区的开头相对于整片缓冲区的偏移。调用`exitWrite()`的函数表示当前写头指针 j_2 指向的，长度为写入的数据区大小的缓冲区已经被写入完成，所有权被释放。这段数据区的状态就从被写入的状态变为待读取的状态。
N	正在被写入的数据段数量	表示当前缓冲区中有多少个正在处理的写入动作，也表示当前有多少片正在更新的数据写入段。来自任意方的调用`enterWrite()`会为其增加一次计数，相对的， 每次调用`exitWrite()`都会减去一次计数。 $N = 0$ 即表示当前缓冲区此时不被任何线程写入。

算法

1. 用户线程每次写入大小为 s 的数据前，**原子性**地记录当前的写尾指针 $j_1^* = j_1$ ，并立刻更新 $j_1 \to (j_1 + s) \mod (n)$ 。

```
/**
* @File: MISOCircularBuffer.hpp
*/


    /**
```

```
     * @brief Before writing data into the buffer, the caller should call this
function
     * This process will update the tail read index and increment the nested number
by 1
     * @param len: The length of the buffer to be written into the buffer
     * @note The atmoic operation is needed in order to
     */

    uint8_t *enterWrite(uint16_t len)
    {
        uint8_t *ptr;
        ATOMIC_ENTER_CRITICAL();
        {
            ptr = buffer + (wIndexTail + 1) % MISO_BUFFER_SIZE;
            wNestedNum++;
            wIndexTail = (wIndexTail + len) % MISO_BUFFER_SIZE;
        }
        ATOMIC_EXIT_CRITICAL();
        return ptr;
    }
```

2. 以 $(j_1^* + 1) \mod (n)$ 为起点向缓冲区写入数据。

3. 写入结束后，如果当前的嵌套变量 $N = 0$, 那么更新写头指针 $j_2 \to j_1$ 。

```
/**
* @File: MISOCircularBuffer.hpp
*/

    /**
     * @brief After finish writing the data, the caller should called this function
to exit its writing process
     * This process will update the head read index and decrement the nested number
     */

    void exitWrite()
    {
        ATOMIC_ENTER_CRITICAL();
        {
            wNestedNum--;
            if (wNestedNum == 0)
            {
                wIndexHead  = wIndexTail;
                pendingSize = (wIndexHead - rIndex + MISO_BUFFER_SIZE) %
MISO_BUFFER_SIZE;
            }
        }
        ATOMIC_EXIT_CRITICAL();
    }
```

4. 在这些过程中，如果DMA传输完成，依照常规更新读指针 i ；此后如果 $i < j_2$，那么立刻发起DMA请求传输 $(i, j_2]$ 的数据段； 如果 $i = j_2$，此时缓冲区待处理数据大小为 0； 如果 $i > j_2$, 那么考虑数据被截断的情况，我们发起DMA请求传输 $(i, n-1]$ 的数据段。

```
/**
/* @File NewRosComm.cpp
*/


void RosManager::txCpltCallback(UART_HandleTypeDef *handle)
{
    // traceISR_ENTER();
    RosManager *pManager = getManager(handle);
    pManager->txBuffer.exitReadFromISR();
    if (pManager->txBuffer.getNextAvailableReadSizeNoCircular())
    {
        BaseType_t xHigherPriorityTaskWoken = pdFALSE;
        vTaskNotifyGiveFromISR(pManager->txTaskHandle, &xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }
    // traceISR_EXIT();
}

// ...
// ...

void RosManager::txTask(void *pvParameters)
{
    RosManager *pManager = (RosManager *)pvParameters;
    uint8_t *pData;
    uint16_t txSize = 0;

    while (true)
    {
        // Is there any pending message
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        if (not(txSize = pManager->txBuffer.getNextAvailableReadSizeNoCircular()))
        {
#if USE_DEBUG
            pManager->txErrorsCounter[eMISOBufferEmpty]++;
#endif
            continue;
        }
        ATOMIC_ENTER_CRITICAL();
        {
            if ((pData = pManager->txBuffer.enterRead(txSize)) == nullptr)
            {
                pManager->txBuffer.exitRead();
#if USE_DEBUG
                pManager->txErrorsCounter[eMISOBufferMutipleRead]++;
#endif
                continue;
            }
            if (HAL_UART_Transmit_DMA(pManager->huart, pData, txSize) == HAL_OK)
            {
                __HAL_DMA_DISABLE_IT(pManager->huart->hdmatx, DMA_IT_HT);
            }
            else  // Error
            {
                HAL_UART_AbortTransmit(pManager->huart);
                pManager->txBuffer.exitRead();
```

```
#if USE_DEBUG
                pManager->txErrorsCounter[eHALNotOK]++;
#endif
            }
        }
        ATOMIC_EXIT_CRITICAL();
    }
}
```

接收模块

简述

接收模块包含一个环形DMA、一块环形环形缓冲区以及相应的管理机制。不同于发送模块， 接收模块的缓冲区仅接收来自单个环形DMA的写入以及接收模块管理下的读取。 因而，这个模块是 *SISO* 的，除了实现环形缓冲区本身的特性之外不需要针对多输入或者多输出做出额外的处理，在驱动层面的原理相对简单。本模块主要的创新点在于在 **协议层** 上实现了一种高效通用的解包机制，不仅能够正确区分接收的数据包的异常情况，还保证了绝对意义上的0丢包率（即上位机无论： 1. 在什么时候 2. 发什么包 3. 发的包是否破碎还是粘连，都可以正确的处理这些数据包并通知相应的用户线程）。

驱动层配置与设计

- 缓冲区输入：我们需要将RX DMA配制成环形模式，从而自动实现数据的环形写入，规避了重复进行DMA请求的操作开销。

- 在初始化模块的时候，我们需要打开 UART 的IDLE中断、DMA的全满中断以及DMA的半满中断。

```
/*
* @File: NewRosComm.cpp
*/

void RosManager::init()
{
    //...//
    // Start the RX DMA reception event.
    // The hal library will automatically enable the DMA half complete interrupt , DMA
full complete interrupt and UART idle-line interrupt.
    configASSERT(HAL_UARTEx_ReceiveToIdle_DMA(huart, rosRxCircularBuff,
NEW_ROS_RX_BUF_SIZE) == HAL_OK);
    //...//
}
```

当缓冲区写入了一定数据并产生了任意中断事件后，模块内部的接收事件回调函数便会更新环形缓冲区的写指针`RosManager::rxWIndex`，并且产生一个信号量通知守护线程`rxTask()`进行解包，该线程同时负责将有效信息以**回调函数**的方式告知用户线程。

```
/**
 * @File: NewRosComm.cpp
 */


void RosManager::rxCallback(UART_HandleTypeDef *handle, uint16_t size)
{
    // traceISR_ENTER();
    RosManager *pManager                = getManager(handle);
    uint16_t wPtr                       = (size - 1 + NEW_ROS_RX_BUF_SIZE) %
NEW_ROS_RX_BUF_SIZE;
    pManager->rxWIndex                  = wPtr;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveFromISR(pManager->rxTaskHandle, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    // traceISR_EXIT();
}

// .. //

void RosManager::rxTask(void *pvParameters)
{
    RosManager *pManager = (RosManager *)pvParameters;
    EPackageStatus packageState;

    while (true)
    {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        packageState = ePackageComplete;
        while ((pManager->rxSize = pManager->rxBufferPendingBytes()) && packageState !=
ePackageHeaderIncomplete &&
               packageState != ePackagePayLoadIncomplete)
        {
            packageState = pManager->updateAndGetNextFrame();
            if (packageState == ePackageComplete)
            {
                pManager->connected         = true;
                pManager->disConnectCounter = 0;
                uint8_t protocolID          = pManager-
>rxFrameBuffer.frame.header.protocolID;
                uint16_t len                = pManager-
>rxFrameBuffer.frame.header.dataLen;
                for (uint8_t j = 0; j < pManager->frameCallbackCounter; j++)
                {
                    if (pManager->callbackTable[j].id != protocolID)
                        continue;
                    pManager->callbackTable[j].func(pManager-
>rxFrameBuffer.frame.payLoadCRC16, len, pManager->huart);
                }
            }
// Debug
// ...
        }
    }
}
```

协议层解包算法

代码实现

每次调用一次解包函数，都会返回一个解包状态。其声明为如下：

```
/**
/* @File: NewRosComm.hpp
*/

// Package status
// For every call on updateAnGetNextFrame, the function will try to decode a package,
and return the status of the package
enum EPackageStatus
{
    ePackageComplete = 0,       // The next package is complete
    ePackageHeaderIncomplete,   // The header of the next pacakge is incomplete
    ePackagePayLoadIncomplete,  // The payload of the next pacakge is incomplete
    ePackageHeaderCRCError,     // The header of the next package has a CRC error
    ePackageCRCError,           // The next total package has a CRC error
    ePackageStateCount          // Enum variables counter
};
```

每当 `RosManager::TxTask()` 这个线程接收到了一次信号量，就会持续调用解包函数 `RosManager::updateAndGetNextFrame()` 直到缓冲区内数据残缺，在这个过程中持续更新 缓冲区内的读指针 `RosManager::rxRIndex` 以及缓冲区内剩余数据的大小`RosManager::rxSize` 。

```
/**
 * @File NewRosComm.cpp
 */

RosManager::EPackageStatus RosManager::updateAndGetNextFrame()
{
    uint16_t size                                = this->rxSize;
    static constexpr uint16_t CRC16_SIZE         = 2;
    static constexpr uint16_t HEADER_CRC16_SIZE = sizeof(FrameHeader) + CRC16_SIZE;

    // Find the start byte
    uint16_t rPtr = this->rxRIndex;
    while (size > HEADER_CRC16_SIZE)
    {
        rPtr = (rPtr + 1) % NEW_ROS_RX_BUF_SIZE;
        if (rosRxCircularBuff[rPtr] == RosComm::START_BYTE)
        {
            break;
        }
        size--;
    }

    // Verify the completeness of the potential header
    if (size <= HEADER_CRC16_SIZE)
    {
```

```
        this->rxRIndex = (rPtr - 1 + NEW_ROS_RX_BUF_SIZE) % NEW_ROS_RX_BUF_SIZE;  //
START_BYTES - 1
        return ePackageHeaderIncomplete;
    }

    // Copy the buffer into the header
    if (rPtr + HEADER_CRC16_SIZE <= NEW_ROS_RX_BUF_SIZE)
    {
        memcpy(rxFrameBuffer.array, &rosRxCircularBuff[rPtr], HEADER_CRC16_SIZE);
    }
    else
    {
        uint16_t firstPartSize = NEW_ROS_RX_BUF_SIZE - rPtr;
        memcpy(rxFrameBuffer.array, &rosRxCircularBuff[rPtr], firstPartSize);
        memcpy(rxFrameBuffer.array + firstPartSize, rosRxCircularBuff, HEADER_CRC16_SIZE
- firstPartSize);
    }
    size -= HEADER_CRC16_SIZE;
    rPtr = (rPtr + HEADER_CRC16_SIZE) % NEW_ROS_RX_BUF_SIZE;

    // Verify the Header CRC
    if (!Crc::verifyCRC16CheckSum(rxFrameBuffer.array, HEADER_CRC16_SIZE))
    {
        this->rxRIndex = (rPtr - 1 + NEW_ROS_RX_BUF_SIZE) % NEW_ROS_RX_BUF_SIZE;
        return ePackageHeaderCRCError;
    }

    // Read the payload length from the header
    uint16_t payLoadLen = rxFrameBuffer.frame.header.dataLen;
    if (payLoadLen > NEW_ROS_MAX_PACKAGE_SIZE)
    {
        this->rxRIndex = (rPtr - 1 + NEW_ROS_RX_BUF_SIZE) % NEW_ROS_RX_BUF_SIZE;
        return ePackageHeaderCRCError;
    }

    // Verify the completeness of the potential payload
    if (size < payLoadLen + CRC16_SIZE)
    {
        this->rxRIndex = (rPtr - HEADER_CRC16_SIZE - 1 + NEW_ROS_RX_BUF_SIZE) %
NEW_ROS_RX_BUF_SIZE;  // Start bytes
        return ePackagePayLoadIncomplete;
    }

    // Copy the payload and 16CRC into the buffer
    if (rPtr + payLoadLen + CRC16_SIZE <= NEW_ROS_RX_BUF_SIZE)
    {
        memcpy(rxFrameBuffer.frame.payLoadCRC16, &rosRxCircularBuff[rPtr], payLoadLen +
CRC16_SIZE);
    }
    else
    {
        uint16_t firstPartSize = NEW_ROS_RX_BUF_SIZE - rPtr;
        memcpy(rxFrameBuffer.frame.payLoadCRC16, &rosRxCircularBuff[rPtr],
firstPartSize);
        memcpy(rxFrameBuffer.frame.payLoadCRC16 + firstPartSize, rosRxCircularBuff,
payLoadLen + CRC16_SIZE - firstPartSize);
    }
    rPtr = (rPtr + payLoadLen + CRC16_SIZE) % NEW_ROS_RX_BUF_SIZE;
```

```
    // Verify the payload CRC
    if (!Crc::verifyCRC16CheckSum(rxFrameBuffer.array, HEADER_CRC16_SIZE + payLoadLen +
CRC16_SIZE))
    {
        this->rxRIndex = (rPtr - 1 + NEW_ROS_RX_BUF_SIZE) % NEW_ROS_RX_BUF_SIZE;
        return ePackageCRCError;
    }

    // CHEERS! The package is complete
    this->rxRIndex = (rPtr - 1 + NEW_ROS_RX_BUF_SIZE) % NEW_ROS_RX_BUF_SIZE;
    return ePackageComplete;
}
```

错误处理

虽然理论上环形DMA的请求在整个程序的生命周期内只需要调用一次，但是如果系统运行过程中外设产生了Frame Error以及Noise Error等错误，整个DMA传输便会被终止。

因此，我们需要注册一个错误处理回调函数，在发生错误的时候重启DMA。

```
/**
 * @File NewRosComm.cpp
 */

void RosManager::rxErrorCallback(UART_HandleTypeDef *handle)
{
    RosManager *pManager     = getManager(handle);
    BaseType_t interruptMask = pdFALSE;
    interruptMask            = taskENTER_CRITICAL_FROM_ISR();
    {
        HAL_UART_AbortReceive(handle);
        HAL_UARTEx_ReceiveToIdle_DMA(handle, pManager->rosRxCircularBuff,
NEW_ROS_RX_BUF_SIZE);
        pManager->rxRIndex = pManager->rxWIndex = -1;
#if USE_DEBUG
        pManager->rxErrorCounter++;
#endif
    }
    taskEXIT_CRITICAL_FROM_ISR(interruptMask);
}
```

此外，我们也可以多添加一个守护计时器，当检测到长时间没有收到数据帧的时候重启刷新DMA。

```
void RosManager::rxTimerFunc(TimerHandle_t xTimer)
{
    RosManager *pManager = (RosManager *)(pvTimerGetTimerID(xTimer));
    bool lastConnected   = pManager->connected;
    if (++pManager->disConnectCounter > NEW_ROS_RX_TIMEOUT)
    {
        pManager->connected         = false;
        pManager->disConnectCounter = NEW_ROS_RX_TIMEOUT;
    }
```

```
    if (lastConnected && not pManager->connected)
    {
        ATOMIC_ENTER_CRITICAL();
        {
            HAL_UART_AbortReceive(pManager->huart);
            HAL_UARTEx_ReceiveToIdle_DMA(pManager->huart, pManager->rosRxCircularBuff,
NEW_ROS_RX_BUF_SIZE);
            pManager->rxRIndex = pManager->rxWIndex = -1;
        }
        ATOMIC_EXIT_CRITICAL();
    }
}
```

实验结果与分析

为了验证该通讯模块双工通讯的有效性，我们对其进行了超负荷测试。实验结果显示出该模块可以以接近2M的与波特率相近的速率进行准确的信息接收与发送，将丢包情况压缩到了最低。

实验步骤

测试平台

- 开发板：基于香港科技大学ENTERPRIZE的STM32G473VE开发板，MCU主频高达170MHz，详情请见香港科技大学ENTERPRIZE战队的主控板开源报告。
- USB-TTL转换芯片： 板载南京沁恒微电子有限公司生产的CH343P 芯片
- 上位机：Intel NUC i5，接收发送线程运行于ROS2操作系统的节点上

UART - USB - TTL 参数配置

- 无CTS、RTS流控
- 波特率2M bits/s
- Oversampling = 16
- Single Sample = 1
- 无校验位
- Stop bits = 2
- 数据位8bits

测试方法

- 上位机： 在同一个 `serialDriverNode` 里面开启收发双线程：
  - 发送线程： 订阅来自 5 个 用户节点的数据帧，每个节点以 $1000Hz$ 的频率发布两个数据帧 （共有 5 种数据帧，大小分别为 $16, 19, 13, 33, 20$ bytes）。
  - 接收线程： 调用 Linux 底层的 `read()`函数进行串口缓冲区读取 -> 解码数据包 -> 发布数据包（数据段会随机改变）
- 下位机：
  - 发送线程：设置 4 个发送用户线程，调用`NewRosComm` API， 每个线程每毫秒传输 1 个数据包（总共大小 47 bytes，包含实时IMU 数据，数据值会随机变化）
  - 接收线程：模块内部统一管理。

将代码测试代码烧录后，静置 1.5 个小时，之后统计CRC错误率与丢失率。

下位机测试代码

```
#include "FreeRTOS.h"
#include "IMU.hpp"
#include "NewRosComm.hpp"
#include "gpio.h"
#include "main.h"
#include "random"
#include "task.h"
#include "usart.h"

StackType_t uxSendRosMessageStack[4][1024];
StaticTask_t xSendRosMessageTCB[4];
using namespace Core::Communication;
using namespace Core::Drivers;

// RosComm::FrameHeader header1 = {RosComm::START_BYTE,
sizeof(RosComm::SentryGimbalMsg), RosComm::SENTRY_GIMBAL_MSG};
// RosComm::FrameHeader header2 = {RosComm::START_BYTE, sizeof(RosComm::GimbalIMU),
RosComm::GimbalIMU};
// RosComm::FrameHeader header3 = {RosComm::START_BYTE, sizeof(RosComm::ChassisIMU),
RosComm::ChassisIMU};
// RosComm::FrameHeader header4 = {RosComm::START_BYTE, sizeof(RosComm::RefereeState),
RosComm::ROBOT_SURVIVAL_MSG};
// RosComm::FrameHeader header1 = {RosComm::START_BYTE, sizeof(RosComm::GimbalMsg),
RosComm::GIMBAL_MSG};
RosComm::FrameHeader header2 = {RosComm::START_BYTE, sizeof(RosComm::SentryGimbalMsg),
RosComm::TWOCRC_SENTRY_GIMBAL_MSG};
RosComm::SentryGimbalMsg sentryGimbalStatus1;
RosComm::SentryGimbalMsg sentryGimbalStatus2;
RosComm::SentryGimbalMsg sentryGimbalStatus3;
RosComm::SentryGimbalMsg sentryGimbalStatus4;
RosComm::GimbalMsg gimbalImuStatus;
// RosComm::ChassisIMU chassisImuStatus;
// RosComm::RefereeState refereeState;

enum Task
{
```

```
    GIMBAL_MSG = 0,
    SENTRY_GIMBAL_MSG
    // GimbalIMU,
    // CHASSIS_IMU_MSG,
    // ROBOT_SURVIVAL_MSG
};

// RosComm::

void sendRosMessage(void *pvPara)
{
    // HAL_GPIO_WritePin(LED_ACT_GPIO_Port, LED_ACT_Pin, GPIO_PIN_RESET);

    while (true)
    {
        sentryGimbalStatus1.cur_cv_mode  = 1;
        sentryGimbalStatus1.target_color = 2;
        sentryGimbalStatus1.bullet_speed = 1.0f;
        sentryGimbalStatus1.small_q[0]   = 1.0f;
        sentryGimbalStatus1.small_q[1]   = 2.0f;
        sentryGimbalStatus1.small_q[2]   = 2.0f;
        sentryGimbalStatus1.small_q[3]   = 1.0f;
        sentryGimbalStatus1.big_q[0]     = 1.0f;
        sentryGimbalStatus1.big_q[1]     = 2.0f;
        sentryGimbalStatus1.big_q[2]     = 2.0f;
        sentryGimbalStatus1.big_q[3]     = 1.0f;

        RosComm::RosManager::managers[0].transmit(header2, (uint8_t
*)&sentryGimbalStatus1);

        vTaskDelay(1);

        // HAL_GPIO_TogglePin(LED_ACT_GPIO_Port, LED_ACT_Pin);
        // HAL_GPIO_TogglePin(LASER_GPIO_Port, LASER_Pin);
        // vTaskDelay(500);
    }
}

void sendRosMessage2(void *pvPara)
{
    while (true)
    {
        sentryGimbalStatus2.cur_cv_mode  = 1;
        sentryGimbalStatus2.target_color = 2;
        sentryGimbalStatus2.bullet_speed = 1.0f;
        sentryGimbalStatus2.small_q[0]   = IMU::getQuaternion().getW();
        sentryGimbalStatus2.small_q[1]   = IMU::getQuaternion().getY();
        sentryGimbalStatus2.small_q[2]   = IMU::getQuaternion().getZ();
        sentryGimbalStatus2.small_q[3]   = IMU::getQuaternion().getX();
        sentryGimbalStatus2.big_q[0]     = 1.0f;
        sentryGimbalStatus2.big_q[1]     = 2.0f;
        sentryGimbalStatus2.big_q[2]     = 2.0f;
        sentryGimbalStatus2.big_q[3]     = 1.0f;

        RosComm::RosManager::managers[0].transmit(header2, (uint8_t
*)&sentryGimbalStatus2);

        vTaskDelay(1);
```

```
        // HAL_GPIO_TogglePin(LED_ACT_GPIO_Port, LED_ACT_Pin);
        // HAL_GPIO_TogglePin(LASER_GPIO_Port, LASER_Pin);
        // vTaskDelay(500);
    }
}

void sendRosMessage3(void *pvPara)
{
    while (1)
    {
        sentryGimbalStatus3.cur_cv_mode  = 1;
        sentryGimbalStatus3.target_color = 2;
        sentryGimbalStatus3.bullet_speed = 1.0f;
        sentryGimbalStatus3.small_q[0]   = IMU::getQuaternion().getW();
        sentryGimbalStatus3.small_q[1]   = IMU::getQuaternion().getY();
        sentryGimbalStatus3.small_q[2]   = IMU::getQuaternion().getZ();
        sentryGimbalStatus3.small_q[3]   = IMU::getQuaternion().getX();
        sentryGimbalStatus3.big_q[0]     = 1.0f;
        sentryGimbalStatus3.big_q[1]     = 2.0f;
        sentryGimbalStatus3.big_q[2]     = 2.0f;
        sentryGimbalStatus3.big_q[3]     = 1.0f;

        RosComm::RosManager::managers[0].transmit(header2, (uint8_t
*)&sentryGimbalStatus3);

        vTaskDelay(1);
    }
}

void sendRosMessage4(void *pvPara)
{
    while (1)
    {
        sentryGimbalStatus4.cur_cv_mode  = 1;
        sentryGimbalStatus4.target_color = 2;
        sentryGimbalStatus4.bullet_speed = 1.0f;
        sentryGimbalStatus4.small_q[0]   = IMU::getQuaternion().getW();
        sentryGimbalStatus4.small_q[1]   = IMU::getQuaternion().getY();
        sentryGimbalStatus4.small_q[2]   = IMU::getQuaternion().getZ();
        sentryGimbalStatus4.small_q[3]   = IMU::getQuaternion().getX();
        sentryGimbalStatus4.big_q[0]     = 1.0f;
        sentryGimbalStatus4.big_q[1]     = 2.0f;
        sentryGimbalStatus4.big_q[2]     = 2.0f;
        sentryGimbalStatus4.big_q[3]     = 1.0f;

        RosComm::RosManager::managers[0].transmit(header2, (uint8_t
*)&sentryGimbalStatus4);
        RosComm::RosManager::managers[0].transmit(header2, (uint8_t
*)&sentryGimbalStatus4);

        vTaskDelay(1);
    }
}

/**
 * @brief Create user tasks
 */
```

```
void startUserTasks()
{
    IMU::init();

    xTaskCreateStatic(sendRosMessage, "1", 1024, nullptr, 2, uxSendRosMessageStack[0],
&xSendRosMessageTCB[0]);
    xTaskCreateStatic(sendRosMessage2, "2", 1024, nullptr, 5, uxSendRosMessageStack[1],
&xSendRosMessageTCB[1]);
    xTaskCreateStatic(sendRosMessage3, "3", 1024, nullptr, 7, uxSendRosMessageStack[2],
&xSendRosMessageTCB[2]);
    xTaskCreateStatic(sendRosMessage4, "4", 1024, nullptr, 14, uxSendRosMessageStack[3],
&xSendRosMessageTCB[3]);

    RosComm::RosManager::managers[0].init(&huart3);
    RosComm::RosManager::managers[1].init(&huart2);
}
```

测试结果与分析

下位机接收测试

[0]: 总共收到的数据包数量； [1]: 收到帧头不完整的情况次数 ；[2]: 收到数据段不完整的情况；[3]: 帧头CRC错误；[4]: 整个数据帧CRC错误的情况

- 总共接收到 8113356 个数据帧， 其中共有 81 个包发生了CRC错误，计算下来CRC错误率为 0.001%，可以基本忽略不计。此外，在整个过程中外设仅仅产生过一次报错的情况。
- 平均帧接收频率为 $8240Hz$, 每个数据包如前文所述平均（可能估计不准，忘记为每个收到的包的数量作统计了）大小为 $\frac{16+19+33+13+20}{5} = 20.2$ bytes，整个系统有效传输信息量 $> 10 \times 20.2 \times 8240 = 1.65M$ bits/s。

下位机发送测试

上位机接收数据帧统计

- 实际上位机帧接收频率为 $3992Hz$, 基本吻合下位机传输帧的频率。每个数据包大小为 47 bytes, 系统有效传输信息量 $> 3992 \times 47 \times 10 = 1.88M$ bits/s。
- CRC 错误率为 0.035%, 基本可以忽略不计。

下位机开销分析

中断函数在我们的设计中主要起到了一个提供信号量的作用，因此，这一部分开销是可以忽略不计的（ < 5us ）。我们主要关心的是各个任务内部的开销（尤其是涉及到内存拷贝的代码段）。

以下， 我们用 *SystemView* 去观测下位机模块的任务开销。

“ros tx/rx task”是模块内部的守护线程，"1""2""3""4"是用户线程，“Sensor”是IMU驱动守护线程

由以上结果可知，即便在满负荷（单工带宽接近2M）运行的情况下，模块开销依然不高。在考虑SystemView带宽不够会导致观测overflow的情况下， 模块内部开销至多占系统总开销的 5% ，4 个用户线程的开销至多在 10%左右。在赛场上我们的实际使用带宽远远低于最大带宽，实际占用开销可能就只有 5% 左右，可以令人满意。

改进空间

虽然相较于ENTERPRIZE战队之前的通讯模块，本模块在通讯质量与速度方面都有了显著的提升。然而，在赛场上部署调试的过程中，我们依旧发现进步空间，主要为以下两点：

- 没有做延迟测试：虽然我们严格地通过实验验证了本通讯方案的稳定性，但我们在今年CV调试的过程中，偶尔感觉会发现整个系统的延迟变大。为了严格验证是否是本模块的设计仍然有不足，需要与上位机联动进行通讯回环测试。
- 下位机的用户发送API仍然存在两次的内存拷贝，未来可以设计一个开销更低的API同时保证用户使用的简洁性。

我们今年将会继续补全这一部分的测试。

后记

串口通讯模块虽然简单，但细细品茗，亦感玄妙。俗话说，经济基础决定上层建筑，倘若没有铺建一条稳定的通天信息高速公路，众算法大仙在实践之时亦时刻感到挚肘难以称快。在上个赛季，整个模块的开发远未称得上尽善尽美，且本文著笔仓促，如有谬误，敬请原谅并恳请斧正；若有心得之言，也尽可畅快直叙，致电邮箱 zguobd@connect.ust.hk

感谢上个赛季与我并肩作战的战友们，不论是senior还是junior，没有他们的珍贵建议与齐心协作，这个模块不可能在赛场上得以落地；也感谢其他战队宝贵的嵌入式框架开源，我们从中受益良多。

初心高于胜负。