PIXHAWk MAVLINK 消息自定义

对于 PIXHAWK 这个系统来讲，MAVLINK 是个小型的数据通信协议，负责地面站和飞控本身的数据交互和地面站向飞控发送数据指令。前面的文章已经讲过 MAVLINK 这个数据格式解析的分析，这里不做讲解。因为我们在使用基于这套飞控开发的过程中，会有很多自定义的数据要求，比如我们添加一个新的传感器(在飞控中添加一个自定义传感器，具体请看相关章节)，我们会把新的传感器数据发送回来和地面站交互，这时候就会涉及到自定义一个 MAVLINK 的消息包，自定义消息包的数据，发送给地面站来解析显示。对于 MAVLINK 这套系统，有对应的工具，来自动生成 MAVLINK 的库函数，只要约定好消息包的结构体，我们就可以用相应的 MAVLINK 工具，来生成 MAVLINK 库的头文件和功能接口函数。

使用 MAVLINK GENERATOR(mavlink 生成器) 基于 python 的，下载即可。

1 生成自定义 MAVLINK 消息

Firmware\mavlink\include\mavlink\v1.0\message_definitions 在源码的 mavlink 库里面可以看到定义了很多.XML 文件，这些.XML 文件就是需要自己定义的文件，这个文件的写法可以看到：

```
<message id="0" name="HEARTBEAT">
  <description>The heartbeat message shows that a system is present and responding. The type of the MAV and Autopilot hardware allow the receiving system to treat further messages from this system appropriate (e.g. by laying out the user interface based on the autopilot).</description>
```

<field type="uint8_t" name="type">Type of the MAV (quadrotor, helicopter, etc., up to 15 types, defined in MAV_TYPE ENUM)</field>

<field type="uint8_t" name="autopilot">Autopilot type / class. defined in MAV_CLASS ENUM</field>

<field type="uint8_t" name="base_mode">System mode bitfield, see MAV_MODE_FLAGS ENUM in mavlink/include/mavlink_types.h</field>

<field type="uint32_t" name="custom_mode">Navigation mode bitfield, see MAV_AUTOPILOT_CUSTOM_MODE ENUM for some examples. This field is autopilot-specific.</field>

<field type="uint8_t" name="system_status">System status flag, see MAV_STATUS ENUM</field>

<field type="uint8_t_mavlink_version" name="mavlink_version">MAVLink version</field>

</message>

这个 xml 文件定义了 mavlink 消息 ID 号，消息结构体数据类型，上面是一个系统自定义心跳包的消息自定义。要添加自己的 mavlink 消息，也要按照这种规范写。主要是消息名字，消息 ID 和数据结构体，写好之后用前面那个 mavlink generator(mavlink 消息生成器)，就可以生成自定义的消息包了。自动生成一些 mavlink 的消息发送等函数，我们可以看下这个心跳包的函数有哪些：

static inline uint16_t mavlink_msg_heartbeat_pack

static inline uint16_t mavlink_msg_heartbeat_pack_chan

static inline uint16_t mavlink_msg_heartbeat_encode//消息编码函数

static inline uint16_t mavlink_msg_heartbeat_encode_chan

static inline void mavlink_msg_heartbeat_send//消息发送函数

static inline void mavlink_msg_heartbeat_send_struct

static inline void mavlink_msg_heartbeat_send_buf

static inline uint8_t mavlink_msg_heartbeat_get_type

static inline uint8_t mavlink_msg_heartbeat_get_autopilot

static inline uint8_t mavlink_msg_heartbeat_get_base_mode

重要的就是两个消息发送和消息编码函数，在具体的调用函数里面就是调用这个消费发送函数， 把 mavlink 的消息发送出去，这些都是,mavlink 函数自动生成的函数，我们在用的时候调用即可。具体目录为~/src/Firmware/mavlink/include/mavlink/v1.0/common/mavlink_msg_heartbeat.h

下面看看这个消息生成软件是怎么使用的。

http://dev.px4.io/custom-mavlink-message.html 这个网站可以看到详细的资料，包括 mavlink generator 的使用。

生成完之后，我们可以看到对应的头文件自动生成。把这些头文件复制到源码里面，就可以调用了。

注意在写 xml 文件时要严格注意格式，不能有多余空格等，否则编译会报错。

2 使用自定义好的 MAVLINK 消息

在源码 mavlink 的文件夹里面，我们可以找到 mavlink_messages.cpp 这个 cpp 的源码。我们可以在这个代码里面看到所有的消息包的实现：(下面是心跳包的数据流类，我们要自定义的就是仿照他的格式来写)

```
class MavlinkStreamHeartbeat : public MavlinkStream
{
public:
    const char *get_name() const
    {
        return MavlinkStreamHeartbeat::get_name_static();
    }
    static const char *get_name_static()
    {
        return "HEARTBEAT";
    }
    static uint8_t get_id_static()
    {
        return MAVLINK_MSG_ID_HEARTBEAT;
    }
    uint8_t get_id()
```

```
	{
		return get_id_static();
	}
	static MavlinkStream *new_instance(Mavlink *mavlink)
	{
		return new MavlinkStreamHeartbeat(mavlink);
	}
	unsigned get_size()
	{
		return MAVLINK_MSG_ID_HEARTBEAT_LEN + MAVLINK_NUM_NON_PAYLOAD_BYTES;
	}
	bool const_rate() {
		return true;
	}
private:
	MavlinkOrbSubscription *_status_sub;
	/* do not allow top copying this class */
	MavlinkStreamHeartbeat(MavlinkStreamHeartbeat &);
	MavlinkStreamHeartbeat& operator = (const MavlinkStreamHeartbeat &);
protected:
	explicit MavlinkStreamHeartbeat(Mavlink *mavlink) : MavlinkStream(mavlink),
```

```
        _status_sub(_mavlink->add_orb_subscription(ORB_ID(vehicle_status)))
    {}
    void send(const hrt_abstime t)
    {
        struct vehicle_status_s status;
        /* always send the heartbeat, independent of the update status of the topics */
        if (!_status_sub->update(&status)) {
            /* if topic update failed fill it with defaults */
            memset(&status, 0, sizeof(status));
        }
        uint8_t base_mode = 0;
        uint32_t custom_mode = 0;
        uint8_t system_status = 0;
        get_mavlink_mode_state(&status, &system_status, &base_mode, &custom_mode);
        mavlink_msg_heartbeat_send(_mavlink->get_channel(), _mavlink->get_system_type(),
MAV_AUTOPILOT_PX4,
            base_mode, custom_mode, system_status);//调用消息发送函数，完成消息发
送
    }
};
```

这就是心跳包的 mavlink 功能实现。这个源码文件里面有很多这样的类，它们都继承了 public MavlinkStream 这个基类，然后在基类的基础上实现了

```
virtual const char *get_name() const = 0;

virtual uint8_t get_id() = 0;

virtual const char *get_name() const = 0;

virtual bool const_rate() { return false; }

virtual unsigned get_size() = 0;

virtual unsigned get_size_avg() { return get_size(); }

virtual void send(const hrt_abstime t) = 0;
```

这几个虚函数 get_id()是得到消息 ID，get_name()是得到消息名字，这些消息 ID 和消息名字是在前面 xml 文件里面自己定义好的，通过 mavlink generator 可以自动生成这些消息 ID 和消息名字的头文件定义。比如这个心跳包消息

```
static const char *get_name_static()

{

    return "HEARTBEAT";

}

static uint8_t get_id_static()

{

    return MAVLINK_MSG_ID_HEARTBEAT;

}
```

这个 return MAVLINK_MSG_ID_HEARTBEAT;就是消息包 ID。因为这个 static uint8_t get_id_static()是无符号 8 位 int 型，所以 mavlink 支持的消息的数量是最多 255 条消息，其中官方定义已经定义了很多消息包，比如心跳包，姿态包等等。如果我们要自定义消息 ID，我们要找到系统没有使用的消息 ID，而不能和原有的消息 ID 重复。

我们可以按照心跳包的消息格式，我们来在 mavlink_messages.cpp 写一个自己定义的消息，要在头文件里面添加好我们用 mavlink generator 生成的 mavlink 头文件，里面定义了消息 ID 的编号和这个消息的结构体的函数。

其中还有一个重要的函数 mavlink_msg_heartbeat_send 这个函数就是最终的发送 mavlink 数据包的接口，而这个函数是用 mavlink generator(mavlink 消息生成器)自动生成好的。封装好了串口的数据发送，我们直接拿来用即可。具体的 mavlink_msg_heartbeat_send 实现我们可以在生成好的函数里面可以看到，这里不做叙述。

最终我们要追加一个我们自定义的 mavlink 数据流类到 StreamListItem 数据流链表里面，这个数据流链表在 MAVLINK_messages.cpp 里面，在 MAVLINK 模块的主函数中会不断轮询这个 StreamListItem 数据流链表，并且执行链表里面的数据发送函数来实现 MAVLINK 消息的不断发送，我们添加了自己定义的函数，那么我们就要把我们自己定义的类也添加到这个链表里面。

```
const StreamListItem *streams_list[] = {
    new StreamListItem(&MavlinkStreamHeartbeat::new_instance, &MavlinkStreamHeartbeat::get_name_static, &MavlinkStreamHeartbeat::get_ic
    new StreamListItem(&MavlinkStreamStatustext::new_instance, &MavlinkStreamStatustext::get_name_static, &MavlinkStreamStatustext::get
    new StreamListItem(&MavlinkStreamCommandLong::new_instance, &MavlinkStreamCommandLong::get_name_static, &MavlinkStreamCommandLong::
    new StreamListItem(&MavlinkStreamSysStatus::new_instance, &MavlinkStreamSysStatus::get_name_static, &MavlinkStreamSysStatus::get_ic
    new StreamListItem(&MavlinkStreamHighresIMU::new_instance, &MavlinkStreamHighresIMU::get_name_static, &MavlinkStreamHighresIMU::get
    new StreamListItem(&MavlinkStreamAttitude::new_instance, &MavlinkStreamAttitude::get_name_static, &MavlinkStreamAttitude::get_id_st
    new StreamListItem(&MavlinkStreamAttitudeQuaternion::new_instance, &MavlinkStreamAttitudeQuaternion::get_name_static, &MavlinkStrea
    new StreamListItem(&MavlinkStreamVFRHUD::new_instance, &MavlinkStreamVFRHUD::get_name_static, &MavlinkStreamVFRHUD::get_id_static),
    new StreamListItem(&MavlinkStreamGPSRawInt::new_instance, &MavlinkStreamGPSRawInt::get_name_static, &MavlinkStreamGPSRawInt::get_ic
    new StreamListItem(&MavlinkStreamSystemTime::new_instance, &MavlinkStreamSystemTime::get_name_static, &MavlinkStreamSystemTime::get
    new StreamListItem(&MavlinkStreamTimesync::new_instance, &MavlinkStreamTimesync::get_name_static, &MavlinkStreamTimesync::get_id_st
    new StreamListItem(&MavlinkStreamGlobalPositionInt::new_instance, &MavlinkStreamGlobalPositionInt::get_name_static, &MavlinkStream
    new StreamListItem(&MavlinkStreamLocalPositionNED::new_instance, &MavlinkStreamLocalPositionNED::get_name_static, &MavlinkStreamLoc
    new StreamListItem(&MavlinkStreamVisionPositionNED::new_instance, &MavlinkStreamVisionPositionNED::get_name_static, &MavlinkStream\
    new StreamListItem(&MavlinkStreamLocalPositionNEDCOV::new_instance, &MavlinkStreamLocalPositionNEDCOV::get_name_static, &MavlinkStr
    new StreamListItem(&MavlinkStreamEstimatorStatus::new_instance, &MavlinkStreamEstimatorStatus::get_name_static, &MavlinkStreamEstin
    new StreamListItem(&MavlinkStreamAttPosMocap::new_instance, &MavlinkStreamAttPosMocap::get_name_static, &MavlinkStreamAttPosMocap::
    new StreamListItem(&MavlinkStreamHomePosition::new_instance, &MavlinkStreamHomePosition::get_name_static, &MavlinkStreamHomePositic
    new StreamListItem(&MavlinkStreamServoOutputRaw<0>::new_instance, &MavlinkStreamServoOutputRaw<0>::get_name_static, &MavlinkStreams
    new StreamListItem(&MavlinkStreamServoOutputRaw<1>::new_instance, &MavlinkStreamServoOutputRaw<1>::get_name_static, &MavlinkStreams
    new StreamListItem(&MavlinkStreamServoOutputRaw<2>::new_instance, &MavlinkStreamServoOutputRaw<2>::get_name_static, &MavlinkStreams
    new StreamListItem(&MavlinkStreamServoOutputRaw<3>::new_instance, &MavlinkStreamServoOutputRaw<3>::get_name_static, &MavlinkStreams
    new StreamListItem(&MavlinkStreamHILControls::new_instance, &MavlinkStreamHILControls::get_name_static, &MavlinkStreamHILControls::
    new StreamListItem(&MavlinkStreamPositionTargetGlobalInt::new_instance, &MavlinkStreamPositionTargetGlobalInt::get_name_static, &Ma
    new StreamListItem(&MavlinkStreamLocalPositionSetpoint::new_instance, &MavlinkStreamLocalPositionSetpoint::get_name_static, &Mavlir
    new StreamListItem(&MavlinkStreamAttitudeTarget::new_instance, &MavlinkStreamAttitudeTarget::get_name_static, &MavlinkStreamAttituc
    new StreamListItem(&MavlinkStreamRCChannels::new_instance, &MavlinkStreamRCChannels::get_name_static, &MavlinkStreamRCChannels::get
```

这就是消息添加处。

这个链表会在 Mavlink_main.cpp 里面循环执行链表里面的函数

```
/* update streams */
        MavlinkStream *stream;
LL_FOREACH(_streams, stream)
{
    stream->update(t);//循环发送 MAVLINK 消息包
}
```

来更新串口 mavlink 数据流。

注意看到这是个 update 函数；这个 update 实际是在调用 send 函数，这些 send 函数就是发送串口 mavlink 数据流。

```
StreamListItem *streams_list[] = {...
new StreamListItem(&MavlinkStreamCaTrajectory::new_instance,
&MavlinkStreamCaTrajectory::get_name_static),
nullptr
};
```

我们自定义好的 MAVLINK 消息，就可以在地面站里面显示了，前提是地面站的 mavlink 包能够解析这个你自定义的数据，一般来讲通过 MAVLINK generator 生成器生成的 mavlink 库，和地面站的 mavlink 库是同一套库，就没问题可以保证解析到。比较偷懒的做法是把自己要发生的消息，通过 mavlink 原有的包发送出去，地面站解析这个原有的包即可。比如用 debug　MAVLINk 消息包。