# PX4 Pixhawk程序研究笔记

## 编译环境建立

参考链接：<http://pixhawk.org/dev/toolchain_installation_win>

1、首先确保电脑安装了Java运行环境。

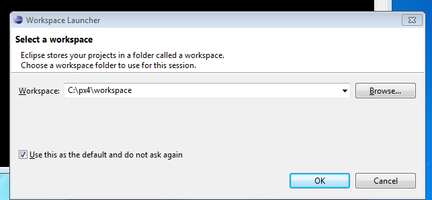
2、下载并安装PX4 Toolchain，链接：[http://pixhawk.org/firmware/downloads#px4\_arm\_toolchain](http://pixhawk.org/firmware/downloads%23px4_arm_toolchain)

3、在开始菜单中选择：PX4 Toolchain -> PX4 Software download来获取一个初始软件设置。它会在安装路径下（默认为C:\px4）下载如下文件夹：

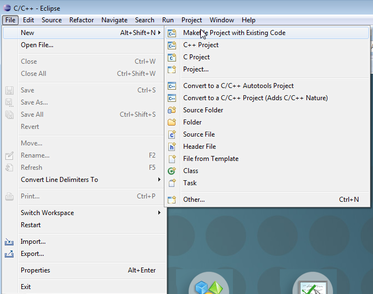
* px4
  + Firmware – PX4 firmware (for all modules), includes MAVLink
    - NuttX – The NuttX Real Time Operating System (RTOS)
  + libopencm3 – Optional: Open Source Cortex Mx library, used only in the bootloaders
  + Bootloader – Optional: Bootloaders, does normally not need to be touched

4、配置Eclipse，开始菜单 -> 所有程序 –> PX4 Toolchain -> PX4 Eclipse

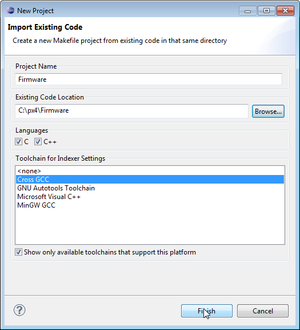
默认的workspace路径是刚好正确的：

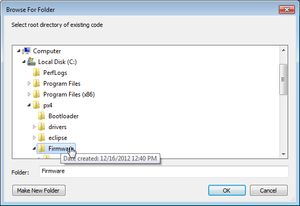


New → Makefile Project with Existing Code：

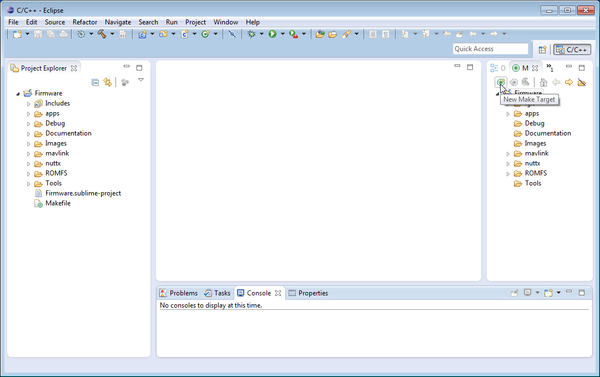


选择Cross GCC，并指定文件夹位置为：“c:\px4\Firmware”。





打开右边的 “Make Target” 并点击 “New Make Target”：



你应当创建如下Targets:

* archives – builds the NuttX OS
* all – builds the autopilot software (depends on archives)
* distclean – cleans everything, including the NuttX build
* clean – cleans only the application (autopilot) part
* upload px4fmu-v1\_default – uploads to PX4FMU v1.x boards
* upload px4fmu-v2\_default – uploads to PX4FMU v2.x boards

**编译方法：**

参考链接：<http://pixhawk.org/dev/flash_px4fmu_win>

1、双击distclean；

2、双击archives；

3、双击all；

4、双击upload px4fmu-v1\_default（PX4）或upload px4fmu-v2\_default（Pixhawk）来上传固件。

注意：只有在Nuttx更新或者改变时才需要进行”distclean“和”archives“。如果你只是编辑了PX4的程序，最便捷的办法是直接运行upload px4fmu-v1\_default（PX4）或upload px4fmu-v2\_default（Pixhawk）来编译并上传固件。

## Eclipse使用技巧

1、选中一个函数，鼠标不动，0.5秒后会弹出一个悬浮框显示该函数的定义，双击该框，会出现滑动条，这时可以使用这个框看此函数的全部定义。以上操作可以由“F2”键代替。

2、选中一个函数，按“F3”直接跳转到该函数的定义处，而不是通过一个悬浮框显示。

## 板载软件结构

PX4自动驾驶仪软件可分为三大部分：实时操作系统、中间件和飞行控制栈。

1. [NuttX](http://pixhawk.org/dev/nuttx/start)实时操作系统

提供POSIX-style的用户操作环境(如*printf(), pthreads,/dev/ttyS1,open(),write(),poll(),ioctl()*)，进行底层的任务调度。

2. PX4中间件

PX4中间件运行于操作系统之上，提供设备驱动和一个微对象请求代理（micro object request broker，uORB)用于驾驶仪上运行的单个任务之间的异步通信。

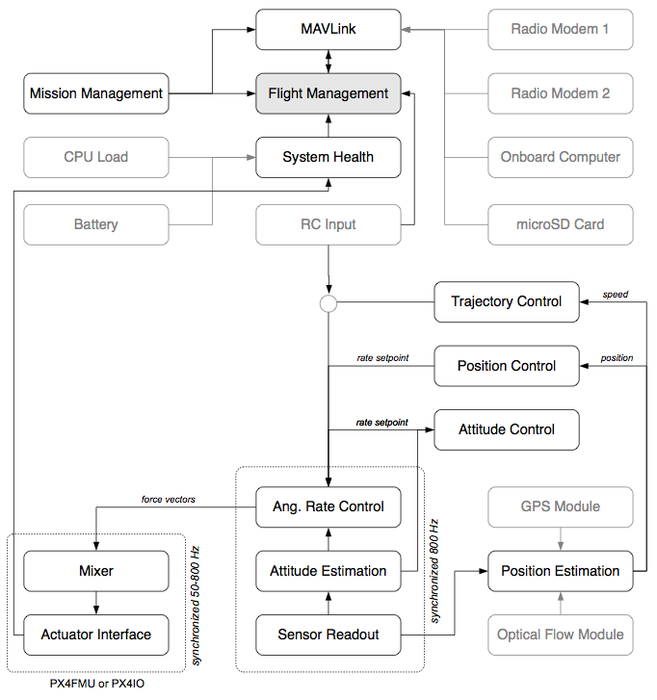
3.PX4飞行控制栈

飞行控制栈可以使用PX4的控制软件栈，也可以使用其他的控制软件，如[APM:Plane、APM:Copter](http://pixhawk.org/firmware/apm)，但必须运行于PX4中间件之上。

PX4飞行控制栈遵循BSD协议，可实现多旋翼和固定翼完全自主的航路点飞行。采用了一套通用的基础代码和通用的飞行管理代码，提供了一种灵活的、结构化的方法，可以用相同的航路点和安全状态机来运行不同的固定翼控制器或旋翼机控制器。

其板载程序结构图如下：

参考链接： <http://pixhawk.org/dev/architecture>



上图中每个框表示一个概念上的任务（task） 。图中不是所有模块都是默认使能的，一些模块是冗余的，比如当姿态控制（attitude control）活动时，位置控制（position control）是不活动的。浅灰色的框表示作为主模块（main blocks）接口的关键外设。图中许多模块被作为单独的任务（tasks）来完成的，不同任务间通过“inter process communication”来通信。

## 源程序文件说明

本章列举源程序中各个文件夹、各个C、CPP文件以及头文件的作用。

* src/lib/geo/geo.c该文件定义的所有与地球坐标系相关的函数（geo: geodesic，测地学的）。提供了与经纬度、地图坐标、坐标系翻转等相关的函数。
* src/lib/launchdetection中包含了自动降落相关的程序。
* src/modules/commonder文件夹包含了所有的与地面站相关的命令：commonder.cpp为主要程序，同时该文件夹中还包含了加速度计校准、空速计校准、磁罗盘校准、遥控器校准等程序。
* src/modules/uORB中包含了与uORB相关的程序。
* src/modules/px4iofirmware中包含了STM32F103那个单片机的源程序，它编译后的结果将作为ROM存储在FMU（STM32F427）单片机的固件中，位于程序文件系统的etc/extras/px4io-v2\_default.bin中。
* src/modules/dataman中包含了与数据管理相关的函数。
* src/systemcmds/param包含了与系统参数相关的程序，这些参数包含机架类型、各种PID以及各种设置等参数。
* src/ROMFS/px4fmu\_common/init.d包含了系统其中的各种脚本，其中最下面的“rcS”为主脚本，系统流程启动以它为准，同时它内部还会不断调用其他子脚本（如rc.sensors脚本，对应各种传感器，其调用命令为：sh /etc/init.d/rc.sensors）。通读“rcS”脚本文件，即可明白PX4的启动和运行流程

## 进程间通信（uORB）

参考链接：<http://blog.arm.so/docs/183-0503.html>

uORB是Pixhawk系统中非常重要且关键的一个模块，它肩负了整个系统的数据传输任务，所有的传感器数据、GPS、PPM信号等都要从芯片获取后通过uORB进行传输到各个模块进行计算处理。

**uORB 的架构简述：**uORB全称为micro object request broker (uORB),即“微对象请求代理器”，实际上uORB是一套跨进程的IPC通讯模块。在Pixhawk中， 所有的功能被独立以进程模块为单位进行实现并工作。而进程间的数据交互就由为重要，必须要能够符合实时、有序的特点。

Pixhawk 使用NuttX实时ARM系统， 而uORB对于NuttX而言，它仅仅是一个普通的文件设备对象，这个设备支持Open、Close、Read、Write、Ioctl以及Poll机制。 通过这些接口的实现，uORB提供了一套“点对多”的跨进程广播通讯机制， “点”指的是通讯消息的“源”，“多”指的是一个源可以有多个用户来接收、处理。而“源”与“用户”的关系在于，源不需要去考虑用户是否可以收到某条被广播的消息或什么时候收到这条消息。它只需要单纯的把要广播的数据推送到uORB的消息“总线”上。对于用户而言，源推送了多少次的消息也不重要，重要的是取回最新的这条消息。

uORB实际上是多个进程打开同一个设备文件，进程间通过此文件节点进行数据交互和共享。

**uORB 的系统实现：**

uORB的实现位于固件源码的src/modules/uORB/uORB.cpp文件，它通过重载CDev基类来组织一个uORB的设备实例。并且完成Read/Write等功能的重载。uORB 的入口点是uorb\_main函数，在这里它检查uORB的启动参数来完成对应的功能，uORB支持start/test/status这3条启动参数，在Pixhawk的rcS启动脚本中，使用start参数来进行初始化，其他2个参数分别用来进行uORB功能的自检和列出uORB的当前状态。

在rcS中使用start参数启动uORB后，uORB会创建并初始化它的设备实例， 其中的实现大部分都在CDev基类完成。这个过程类似于Linux设备驱动中的Probe函数，或者Windows 内核的DriverEntry，通过init调用完成设备的创建，节点注册以及派遣例程的设置等。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 下面是官网资料 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

参考链接：<http://pixhawk.org/dev/shared_object_communication>

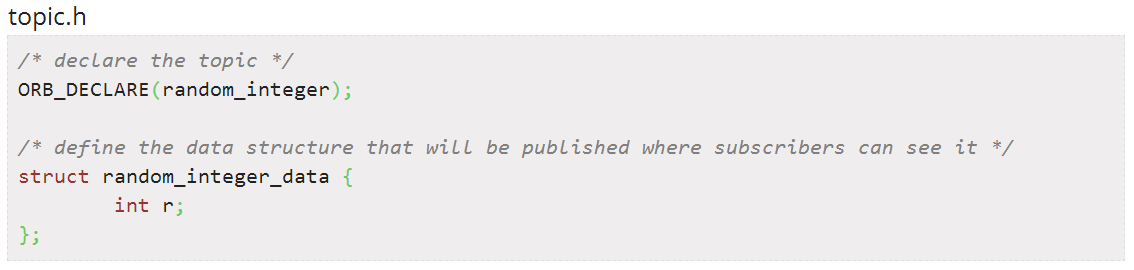
进程（process）/程序（application）间通信（如将传感器信息从传感器app传送到姿态滤波app）是PX4程序结构的核心部分。进程（process，在此处被称作nodes）通过被命名的总线（buses，在此处被称作topic）交换信息。在PX4中，一个topic只包含一种信息类型，比如，vehicle\_attitude 这个topic将一个包含姿态结构体（roll、pitch、yaw）的信息传送出去。Nodes可以在bus/topic上publish（发布）一个信息（即发送数据），也可以向一个bus/topic subscribe（订阅）信息（即接收数据）。它们（Nodes）并不知道它们在跟谁通信。一个topic可以面向多个publishers（发布者）和多个subscribers（订阅者）。这种方式可以避免死锁问题，在机器人中很常见。为达到有效率，在bus/topic中，永远只有一个信息被传送，没有保持队列之说（即新来的信息会覆盖之前的信息，不存在有一串信息排队的情况）。

这个发布/订阅 （publisher / subscriber）机制是通过微对象请求代理（micro object request broker,简称uORB）来实现的。

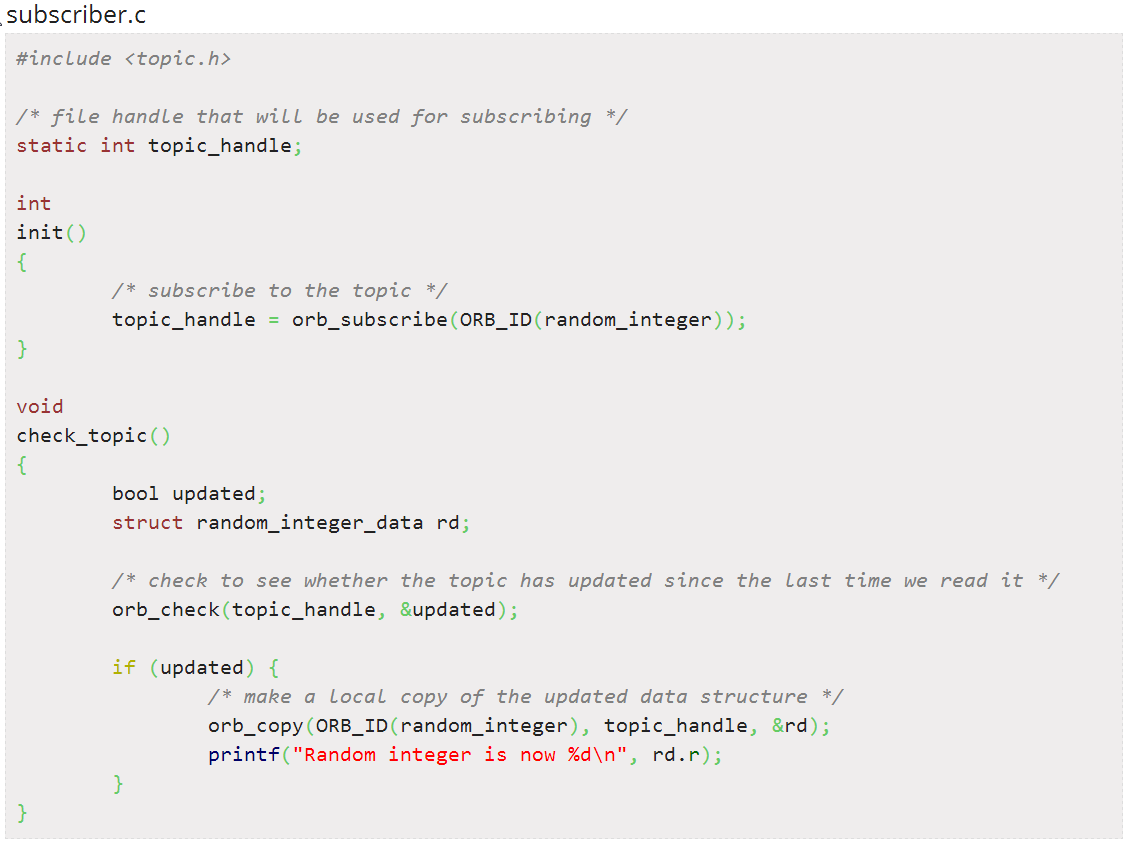
* 系统已存的topics通过Doxygen工具自动生成了文档，其链接为：

<https://pixhawk.ethz.ch/docs/group__topics.html>

下面是关于publisher / subscriber的一个简单的例子，这个publisher（发布者）advertises（通告）一个名叫random\_integer的topic，在这个topic中更新入随机数。subscriber（订阅者）检查并打印出更新值。







* **发布（Publishing）**

发布（Publishing）包含三个独立但是相关的过程：定义（defining ）一个topic，通告（advertising ）这个topic，发布（publishing ）更新。

1. **定义一个topic（Defining a Topic）**

系统已经定义了许多标准的topic来为模块间提供通信接口。如果一个发布者（Publisher）想要使用这些topic及其相关数据结构，不需要做额外的工作。

**用户topic**：要定义一个用户topic，发布者（publisher）需要创建一个对订阅者（subscriber）可见的头文件（header file）（可以参考上面的topic.h文件）。这个头文件（header file）中必须包含如下内容：

* 以这个topic的名字为参数的ORB\_DECLARE()宏的一个实例。
* 一个描述将要publish（发布）的数据结构的结构体定义。

Topic名字必须是有意义的；PX4的惯例是 “类别\_name”；比如，原始传感器数据通过sensors\_raw这个topic来发布。

除了头文件（header file），发布者（publisher）还需在源文件（source file）中添加一个 ORB\_DEFINE()宏的实例，这个实例将会在固件被bulid的时候被编译和链接（参考上面的publisher.c文件）。这个宏创建了一个数据结构体，这个结构体将由ORB用来唯一地识别一个topic的身份。

如果一个topic是由一个软件组件发布的，并且是可选（optional）的，可能不会在固件中出现，那么头文件中可以代替使用ORB\_DECLARE\_OPTIONAL（）宏。这种方法声明的topic可需要专门的句柄（handling），但是下面将会讲到，订阅者在订阅这类topic时有一些额外需要考虑的地方。

1. **通告一个topic（Advertising a Topic）**

在数据可以被发布（publish）到一个topic之前，这个topic必须被通告（advertise）一下。使用的是orb\_advertise（）函数，这个函数也将初始化数据发布到了topic中。这个函数的原型如下：

*/\*\**

*\* Advertise as the publisher of a topic.*

*\**

*\* This performs the initial advertisement of a topic; it creates the topic*

*\* node in /obj if required and writes the initial data.*

*\**

*\* @param meta The uORB metadata (usually from the ORB\_ID() macro)*

*\* for the topic.*

*\* @param data A pointer to the initial data to be published.*

*\* For topics published by interrupt handlers, the advertisement*

*\* must be performed from non-interrupt context.*

*\* @return ERROR on error, otherwise returns a handle*

*\* that can be used to publish to the topic.*

*\* If the topic in question is not known (due to an*

*\* ORB\_DEFINE\_OPTIONAL with no corresponding ORB\_DECLARE)*

*\* this function will return -1 and set errno to ENOENT.*

*\*/*

**extern** int orb\_advertise(const struct orb\_metadata \*meta, const void \*data);

meta变量是指向由“ORB\_DEFINE（）”宏定义的数据的一个指针。通常一个是使用“ORB\_ID（）”宏来提供的，这个宏起到了一个将topic name转换为topic原数据结构体name的作用。

注意，由于更新可以在中断中被发布，通告（advertise）一个topic必须在正常进程中进行。

**多个发布者（Multiple Publishers）:**一次只能有一个发布者来将一个

topic通告（advertise），但是topic handle（句柄） （句柄是一个file descriptor,可以直接传递给close（）函数）可以由一个publisher（发布者）关闭，然后由另一个发布者通告（advertise）。

1. **发布更新（Publishing Updates）**

一个topic被通告（advertise）后，一个由通告函数返回的handle（句柄）可以用来向topic中发布更行。

*/\*\**

*\* Publish new data to a topic.*

*\**

*\* The data is atomically published to the topic and any waiting subscribers*

*\* will be notified. Subscribers that are not waiting can check the topic*

*\* for updates using orb\_check and/or orb\_stat.*

*\**

*\* @handle The handle returned from orb\_advertise.*

*\* @param meta The uORB metadata (usually from the ORB() macro)*

*\* for the topic.*

*\* @param data A pointer to the data to be published.*

*\* @return OK on success, ERROR otherwise with errno set accordingly.*

*\*/*

**extern** int orb\_publish(const struct orb\_metadata \*meta, int handle, const void \*data);

注意ORB不缓存多个更新，所以当一个订阅者检查一个topic的时候，它只能看到最新的更新。

* **订阅（Subscribing）**

订阅（subscribling）一个topic需要如下步骤：

* 一个ORB\_DEFINE() 或者 ORB\_DEFINE\_OPTIONAL()宏（比如在由订阅者包含的一个头文件中）
* 一个发布给topic的数据结构体的定义（通常也是在那个头文件中）

满足上述条件后，使用如下函数来订阅一个topic:

*/\*\**

*\* Subscribe to a topic.*

*\**

*\* The returned value is a file descriptor that can be passed to poll()*

*\* in order to wait for updates to a topic, as well as orb\_read,*

*\* orb\_check and orb\_stat.*

*\**

*\* Subscription will succeed even if the topic has not been advertised;*

*\* in this case the topic will have a timestamp of zero, it will never*

*\* signal a poll() event, checking will always return false and it cannot*

*\* be copied. When the topic is subsequently advertised, poll, check,*

*\* stat and copy calls will react to the initial publication that is*

*\* performed as part of the advertisement.*

*\**

*\* Subscription will fail if the topic is not known to the system, i.e.*

*\* there is nothing in the system that has defined the topic and thus it*

*\* can never be published.*

*\**

*\* @param meta The uORB metadata (usually from the ORB\_ID() macro)*

*\* for the topic.*

*\* @return ERROR on error, otherwise returns a handle*

*\* that can be used to read and check the topic for updates.*

*\* If the topic in question is not known (due to an*

*\* ORB\_DEFINE\_OPTIONAL with no corresponding ORB\_DECLARE)*

*\* this function will return -1 and set errno to ENOENT.*

*\*/*

**extern** int orb\_subscribe(const struct orb\_metadata \*meta);

如果一个optional topic没有被提供，那么对它的订阅将失败，但是别的订阅依然会成功，并且创建这个topic即使它还没被通告。这很大程度低降低了系统启动顺序的安排难度。

一个任务中订阅者的数量没有上限。

取消订阅一个topic，使用如下函数：

*/\*\**

*\* Unsubscribe from a topic.*

*\**

*\* @param handle A handle returned from orb\_subscribe.*

*\* @return OK on success, ERROR otherwise with errno set accordingly.*

*\*/*

**extern** int orb\_unsubscribe(int handle);

**从topic中复制出数据**：订阅者并不直接调用ORB中存储的数据，也不直接与其他订阅者共享它，而是将其从ORB中复制到一个临时缓存中。这个复制过程避免了死锁问题，并且使得发布和订阅函数很简单。并且也允许订阅者直接修改数据（如果需要的话）而不影响其他订阅者。

当一个订阅者想复制一份最新的副本时，使用如下函数：

*/\*\**

*\* Fetch data from a topic.*

*\**

*\* @param meta The uORB metadata (usually from the ORB() macro)*

*\* for the topic.*

*\* @param handle A handle returned from orb\_subscribe.*

*\* @param buffer Pointer to the buffer receiving the data.*

*\* @return OK on success, ERROR otherwise with errno set accordingly.*

*\*/*

**extern** int orb\_copy(const struct orb\_metadata \*meta, int handle, void \*buffer);

**检查更新**：订阅者可以使用如下函数来检查从它上次订阅后，topic是否被再次发布。

*/\*\**

*\* Check whether a topic has been published to since the last orb\_copy.*

*\**

*\* This check can be used to determine whether to copy from the topic when*

*\* not using poll(), or to avoid the overhead of calling poll() when the*

*\* topic is likely to have updated.*

*\**

*\* Updates are tracked on a per-handle basis; this call will continue to*

*\* return true until orb\_copy is called using the same handle. This interface*

*\* should be preferred over calling orb\_stat due to the race window between*

*\* stat and copy that can lead to missed updates.*

*\**

*\* @param handle A handle returned from orb\_subscribe.*

*\* @param updated Set to true if the topic has been published since the*

*\* last time it was copied using this handle.*

*\* @return OK if the check was successful, ERROR otherwise with*

*\* errno set accordingly.*

*\*/*

**extern** int orb\_check(int handle, bool \*updated);

当一个topic在它被通告（advertise）之前被发布，这个函数将返回没有更新直到它被通告。

**发布时间戳**：订阅者可以使用如下函数检查最新的发布发生的时间：

*/\*\**

*\* Return the last time that the topic was published.*

*\**

*\* @param handle A handle returned from orb\_subscribe.*

*\* @param time Returns the time that the topic was published, or zero if it has*

*\* never been published/advertised.*

*\* @return OK on success, ERROR otherwise with errno set accordingly.*

*\*/*

**extern** int orb\_stat(int handle, uint64\_t \*time);

调用这个函数前要额外小心，因为无法保证这个topic不会在这个函数被调用后立即被发布。

**等待更新**：一个将发布者发出的信息作为信息来源的订阅者（subscriber）可以同时等待多个发布者的发布。这个和等待一个file descriptor一样，也是使用poll()函数（这是一个标准通用函数,并非PX4独有，可以在百度上搜索其用法）这个方法可行，是因为实际上订阅也是一个file descriptor。

下面的例子展示了一个等待三个发布者的订阅者的情况。如果一秒钟内没有更行出现，则一个timeout 计数器将被更新并发布出去。

**color\_counter.h**

ORB\_DECLARE(color\_red);

ORB\_DECLARE(color\_green);

ORB\_DECLARE(color\_blue);

ORB\_DECLARE(color\_timeouts);

*/\* structure published to color\_red, color\_green, color\_blue and color\_timeouts \*/*

struct color\_update

{

int number;

};

**color\_counter.c**

*#include <poll.h>*

ORB\_DEFINE(color\_timeouts, struct color\_update);

void

subscriber(void)

{

int sub\_red, sub\_green, sub\_blue;

int pub\_timeouts;

int timeouts = 0;

struct color\_update cu;

*/\* subscribe to color topics \*/*

sub\_red = orb\_subscribe(ORB\_ID(color\_red));

sub\_green = orb\_subscribe(ORB\_ID(color\_green));

sub\_blue = orb\_subscribe(ORB\_ID(color\_blue));

*/\* advertise the timeout topic \*/*

cu.number = 0;

pub\_timeouts = orb\_advertise(ORB\_ID(color\_timeouts), &cu);

*/\* loop waiting for updates \*/*

for (;;) {

*/\* wait for updates or a 1-second timeout \*/*

struct pollfd fds[3] = {

{ .fd = sub\_red, .events = POLLIN },

{ .fd = sub\_green, .events = POLLIN },

{ .fd = sub\_blue, .events = POLLIN }

};

int ret = poll(fds, 3, 1000);

*/\* check for a timeout \*/*

if (ret == 0) {

puts("timeout");

cu.number = ++timeouts;

orb\_publish(ORB\_ID(color\_timeouts), pub\_timeouts, &cu);

*/\* check for color updates \*/*

} else {

if (fds[0].revents & POLLIN) {

orb\_copy(ORB\_ID(color\_red), sub\_red, &cu);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("red is now %d\n", cu.number);

}

if (fds[1].revents & POLLIN) {

orb\_copy(ORB\_ID(color\_green), sub\_green, &cu);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("green is now %d\n", cu.number);

}

if (fds[2].revents & POLLIN) {

orb\_copy(ORB\_ID(color\_blue), sub\_blue, &cu);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("blue is now %d\n", cu.number);

}

}

}

}

**限制更新速率**：一个订阅者可能想要限制它们接收的topic的更新速率，这个可以通过下面的函数实现：

*/\*\**

*\* Set the minimum interval between which updates are seen for a subscription.*

*\**

*\* If this interval is set, the subscriber will not see more than one update*

*\* within the period.*

*\**

*\* Specifically, the first time an update is reported to the subscriber a timer*

*\* is started. The update will continue to be reported via poll and orb\_check, but*

*\* once fetched via orb\_copy another update will not be reported until the timer*

*\* expires.*

*\**

*\* This feature can be used to pace a subscriber that is watching a topic that*

*\* would otherwise update too quickly.*

*\**

*\* @param handle A handle returned from orb\_subscribe.*

*\* @param interval An interval period in milliseconds.*

*\* @return OK on success, ERROR otherwise with ERRNO set accordingly.*

*\*/*

**extern** int orb\_set\_interval(int handle, unsigned interval);

速率限制是针对某一个特定的订阅者的，单个topic可以对多个订阅者有多种限制速率。

## 添加一个uORB topic和mavlink解析程序

参考链接：<http://pixhawk.org/dev/add_uorb_topic>

**步骤一：添加一个uORB topic**

在src/modules/uORB/topics文件夹下添加一个新的名为“ca\_trajectory\_msg.h”的头文件，其内容如下：

*#ifndef TOPIC\_CA\_TRAJECTORY\_MSG\_H*

*#define TOPIC\_CA\_TRAJECTORY\_MSG\_H*

*#include <stdint.h>*

*#include "../uORB.h"*

*/\*\* global 'actuator output is live' control. \*/*

struct ca\_traj\_struct\_s {

uint64\_t timestamp;

uint64\_t time\_start\_usec; *///< starting time of the trajectory.*

uint64\_t time\_stop\_usec; *///< stopping time of the trajectory.*

float coefficients[28]; *///< coefficients of the polynomial trajectory.*

uint16\_t seq\_id; *///< sequence id of the sent trajectory piece.*

};

ORB\_DECLARE(ca\_trajectory\_msg);

*#endif*

编辑src/modules/uORB/objects\_common.cpp文件并添加如下内容：

*#include "topics/ca\_trajectory\_msg.h"*

ORB\_DEFINE(ca\_trajectory\_msg, struct ca\_traj\_struct\_s);

**步骤二：添加一个mavlink解析程序**

这将会将一个输入的mavlink消息解析并传入uORB topic中。假设mavlink消息和uORB topic有着相同的结构。（注意：下文列出的示例程序中，行首出现“+”表示在原程序中添加该行，“-”表示删除原程序中该行）

编辑src/modules/mavlink/mavlink\_bridge\_header.h

-*#include <v1.0/common/mavlink.h>*

+*#include <v1.0/ca\_pixhawk/mavlink.h>*

在src/modules/mavlink/mavlink\_messages.cpp中添加如下行：

+*#include <uORB/topics/ca\_trajectory\_msg.h>*

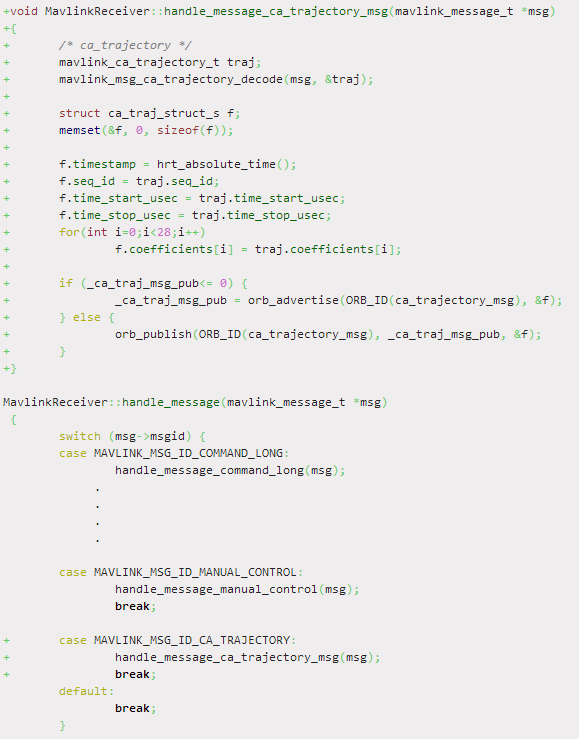
在src/modules/mavlink/mavlink\_receiver.h中添加如下行：

+*#include <uORB/topics/ca\_trajectory\_msg.h>*

+void handle\_message\_ca\_trajectory\_msg(mavlink\_message\_t \*msg);

+orb\_advert\_t \_ca\_traj\_msg\_pub;

在src/modules/mavlink/mavlink\_receiver.cpp中添加下列函数：



## 系统启动控制

参考链接：<http://pixhawk.org/dev/system_startup>

PX4的启动是由位于“/etc/init.d/rcS”下的启动脚本（startup script）来控制的，这个脚本位于被编译到固件中的ROM文件系统中。这个脚本检测可用的硬件，加载硬件驱动，并且根据你的设置启动系统正常运行所需的所有app（任务软件，包括位置和姿态估计，位置和姿态控制，遥测等）。

所有属于自动启动程序的脚本文件可以在固件源程序的如下目录中找到：ROMFS/px4fmu\_common/init.d .

这个rcS脚本执行如下步骤：

1. 挂载SD卡；
2. 如果SD中有etc/rc.txt文件，则按照该启动脚本文件进行启动；
3. 如果SD卡中没有etc/rc.txt文件，下面的自动启动程序将会被执行：
4. 如果SYS\_AUTOSTART变量被设置为一个有效的配置，预定义的模型配置（predefined model configuration）之一将会被加载。模型设置包含一系列的环境变量，后面将会详细列出。
5. 如果SD卡中有etc/config.txt文件，它将被执行。这个文件允许对部分或者整个预定义的模型配置参数进行覆盖。
6. 实际的启动app将会根据模型配置定义的环境变量被执行。
7. 如果SD卡中etc/extras.txt文件，它将被执行。这个文件允许在自动启动程序被执行完成后启动用户的app（如遥测）。

注意：在SD卡中创建etc/rc/txt文件将会使得被编译在固件内部的启动文件不再执行。由于创建这个文件后你必须亲自启动所有的基本系统进程，所以除非你是知道自己在做什么的高级用户，否则请不要创建这个文件。在大多数情况下，你只需要通过创建etc/config.txt和（或者）etc/extras.txt文件来改变自动启动程序（这样就不需要担心会影响到系统必备程序的启动）。

注意：启动脚本文件中每行最多允许有80个字符。任何超过80个字符的行都有可能造成启动出错。

预编译模型配置（Predefined Model Configurations）:下面是针对一些可以买到的ARF/RTF模型的预定义的模型配置。要使用这些模型配置中的一个，只需要设置环境变量SYS\_AUTOSTAET = <AUTOSTART\_ID>。如果要将与机架相关的参数（如姿态控制参数：MC\_XXX）设置为该模型配置默认的参数，将参数SYS\_AUTOCONFIG = 1。保存参数并重新启动飞控。这些配置参数将生效，然后SYS\_AUTOCONFIG将自动变回0。选择的模型配置可以通过etc/config.txt来进行覆盖。

下表是Autostart ID与模型的对应关系：

|  |  |
| --- | --- |
| **Autostart ID** | **Model** |
| 1001 | [HIL](http://pixhawk.org/users/hil) Quadcopter X |
| 1003 | [HIL](http://pixhawk.org/users/hil) Quadcopter + |
| 3031 | [Phantom FPV Flying Wing](http://pixhawk.org/platforms/planes/phantom_fpv_flying_wing) |
| 3033 | Wing Wing (aka Z-84) Flying Wing |
| 3034 | FX-79 Buffalo Flying Wing |
| 4008 | [AR.Drone mechanical frame](http://pixhawk.org/platforms/multicopters/ar_drone) |
| 4010 | [DJI Flamewheel F330 Quadcopter](http://pixhawk.org/platforms/multicopters/dji_flamewheel_330) |
| 4011 | [DJI Flamewheel F450 Quadcopter](http://pixhawk.org/platforms/multicopters/dji_flamewheel_330) |
| 10015 | [Team Blacksheep Discovery Quadcopter](http://pixhawk.org/platforms/multicopters/tbs_discovery) |
| 10016 | 3DR Iris Quadcopter |

**配置变量（Configuration Variables）:**下表中列出的变量定义哪个app会被启动，哪个硬件将被使用，以及一些别的参数。它们可以通过SD卡中的etc/config.txt文件进行设置。如果SYS\_AUTOSTART变量被设置为一个有效的配置，etc/config.txt可以用来调整这些配置，比如设置PWM范围。

|  |  |  |
| --- | --- | --- |
| **Environment variable** | **Possible values (default is underlined)** | **Description** |
| VEHICLE\_TYPE | mc = multicopter, fw = fixed wing, none = non-flying configuration without controllers running | Type of vehicle |
| MIXER | For multicopters: FMU\_quad\_x,FMU\_quad\_+,FMU\_quad\_w,FMU\_hex\_x,FMU\_hex\_+, … | Mixer to use. See section 'Output mixers' below. |
| For fixed wings: FMU\_AERT, FMU\_AET,FMU\_Q |
| USE\_IO | yes, no | If the PX4IO should be used. Set to no if using standalone PX4FMUv1 setup. |
| OUTPUT\_MODE | io, fmu, mkblctrl | Control output mode, default value depends on USE\_IO variable. |
| PWM\_OUTPUTS | List of channels, e.g. 1234, default is none | Channels the following PWM parameters get applied to. Please note that certain channels are grouped and need to be changed together as documented [here](http://pixhawk.org/firmware/apps/pwm)! |
| PWM\_RATE | Frequency in Hz, default is 50 | PWM output rate. To improve ESC latency on multicopter setups, it's recommended to use 400 Hz if ESCs support it. |
| PWM\_DISARMED | Signal length in microseconds, default is 0 | PWM output value for disarmed state. 0 (default value) means no PWM signal, it's safe, but many ESC models will beep “no signal” after disarm, so it can be set to lower limit of PWM range |
| PWM\_MIN | Signal length in microseconds, default is 1000 | PWM value for minimum throttle. Multicopter props **MUST** spin at low idle speed; this is necessary to prevent full stop of props during flight and warn that the system is armed! |
| PWM\_MAX | Signal length in microseconds, default is 2000 | PWM value for maximum throttle. |
| FMU\_MODE | pwm, gpio For FMUv1 also: serial, gpio\_serial,pwm\_serial, pwm\_gpio | FMU output mode, e.g. set to serial to get debug console on PX4FMUv1 USART2. |
| HIL | yes, no | HIL (Hardware In the Loop) mode, this will force OUTPUT\_MODE to fake HIL output, disable logging and GPS |

**输出混控（Output Mixers）:**输出混控定义了控制器的输出如何映射为马达和舵机控制输出。所有被编译在固件中的混控文件位于ROM文件系统的/etc/mixers文件夹中。

可以使用一个自己修改的混控。只需要将你自己写的混控文件（mixer file）放到SD卡的etc/mixers文件夹下即可。比如，使用用户自己设置的混控文件来控制一架已经被定义为MIXER = FMU\_Q的固定翼飞机，那么SD卡中的混控文件必须被命名为FMU\_Q.mix。之后SD卡中的这个文件将会覆盖掉被编译到固件中的那个混控文件。

**用户模型配置（Custom Model）例子:**

配置一个如下型号的四轴：“+”型、电调接PX4IO板，PWM范围为1100~1900，那么config.txt文件包含如下内容：

# Generic Quadcopter +

set VEHICLE\_TYPE mc

set MIXER FMU\_quad\_+

# PX4IO PWM output will be used by default

set PWM\_OUTPUTS 1234 # Change parameters for the first 4 outputs

set PWM\_RATE 400 # Set PWM rate to 400 Hz for better performance

set PWM\_DISARMED 900 # Motors should stop at this PWM value

set PWM\_MIN 1100 # Motors should spin at low, idle speed at this PWM value

set PWM\_MAX 1900 # Motors should spin at max speed at this PWM value

## 系统启动控制调试（Start Script Debugging）

参考链接：<http://pixhawk.org/dev/start_script_debugging>

PX4的软件是在不断进化的，因此有时系统启动方式需要进行修改。我们做到使得这些更改最小。如果启动脚本不执行，系统将直接进入调试（debug）状态。

自动启动脚本是由开源社区共同维护的，但是，如果的microSD卡中有自定义的启动脚本，你需要在每次软件进化后更新它们。

**基本调试技巧：**

固件中有名为“test”的测试命令，它可以用来帮助列出所有测试项目，并且很有利于隔离问题。比如test sensor用来测试传感器。单独的驱动（driver）也有测试，大多数情况下是一个状态指令（status command）（比如：lsm303d status）。

**microSD中的用户脚本**

* 如果你想要添加单独的app，使用EXTRAS\_FILE，并将其放入SD卡中的/etc/extras.txt文件中。它能覆盖掉原有平台参数。
* 如果你确实需要进行一个完全的自定义启动，在SD卡的/etc/rc.txt中放置RC\_FILE。只有在你知道自己在干什么的情况下才可以做。

**调试（Debugging）**

下面的调试例子是针对rc.txt的，但是它也适用于其他类似的文件。

1. 如果SD卡中有/etc/rc.txt，将它重命名为/etc/rc\_test.txt；
2. 将SD卡放回飞控板并启动；
3. 使用USB连接终端；
4. 手动执行测试脚本：sh /fs/microsd/etc/rc\_test.txt；
5. 这一步是关键：观察脚本的启动，找到它是在哪里运行失败并停止的。检查失败处相关代码的语法错误和逻辑错误。修改错误，重新运行直至成功。
6. 小技巧：为了发现是哪一行错了，在启动脚本中添加一些echo命令来将程序运行状态输出，从而可以便捷地发现是哪里出了问题。

* echo “here1”
* echo “here2”
* 等等

## 混控（MIX）

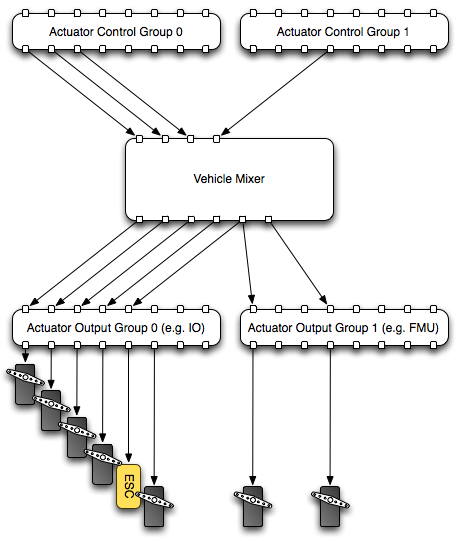
参考链接：<http://pixhawk.org/dev/mixing>

**执行器控制组（Actuator Control Groups）：**一个执行器控制组是由8个涉及相同类型动作的输出组成，比如机身、任务负载或者增稳云台。它由一个执行器（actuator）发出并输入到混控中。不同的的控制组可以由不同的进程发出。

**混控（Mixers）:** 混控是一套独立的定标器（用来归一化数值） 、映射器组成的，它读入控制输入值，输出执行器需要的最终输出值。

**执行器输出组（Actuator Output Groups）：**一个执行器输出组是由8个输出组成，这8个输出属于同一个物理输出设备，如PX4FMU板或者PX4IO板。它代表混控的输出。

三者逻辑关系如下图所示：



**控制组：**

* 控制组0（飞行控制）

0: roll (-1..1)

1: pitch (-1..1)

2: yaw (-1..1)

3: throttle (0..1)

4: flaps (-1..1)

5: spoilers (-1..1)

6: airbrakes (-1..1)

7: reserved (-1..1)

* 控制组1（机身）

0: aux0 (landing gear, -1..1)

1: aux1 (parachute, -1..1)

2: aux2

3: aux3

4: aux4

5: aux5

6: aux6

7: aux7

* 控制组2（任务负载）

0: gimbal roll

1: gimbal pitch

2: gimbal yaw

3: gimbal shutter

4: reserved

5: reserved

6: reserved

7: reserved

* 控制组3（手动信号直通）

0: RC roll

1: RC pitch

2: RC yaw

3: RC throttle

4: RC mode switch

5: RC aux1

6: RC aux2

7: RC aux3

**名词解释：**

**混控器（Mixer）:** 一个根据预先设定好的规则和参数将一系列输入整合为一系列（或一个）输出的模块。

**定标器（Scaler）：**一个将单一输入按照参数生成为单一输出的算术模块（通常的作用是将输入值归一化或者限制为一个范围之内）。

**输入（Input）：**输入给混控器的数字值，范围为-1.0到1.0 。它可以是一个飞行器控制量，如“滚转（roll）”，或者别的数字。

**输出（Output）:**混控器的输出值。

**基本混控（Mixing Basics）：**一个模块在它准备生产一组新的输出时调用混控或者混控组。比如：当它发现它所监控的输入发生变化时，它将调用混控。混控器将接收输入值，将它们根据混控定义进行归一化和混合。之后模块就可以输出数值了。比如：产生舵机控制量，或者将输出的数据发送给别的模块使用。

**缩放（Scaling）：**混控器的输入经常需要进行缩放来适应不同输入的有效位。根据不同的环境，不同的缩放规则会被使用。

**简单缩放（Simple Scaling）：**将输入值简单地与一个固定的系数相乘。这种方法在输入的基准值已知的情况或者输入值不需要修改的情况下使用。

**线性缩放（Linear Scaling）：**这种缩放方法可以进行两侧不均匀缩放，也可以将输出进行偏移或者收紧。它的输入有：

* 负侧缩放系数
* 正侧缩放系数
* 偏移值
* 输出下限
* 输出上限

它的工作流程如下：

if (input < 0)

output = (input \* NEGATIVE\_SCALE) + OFFSET

else

output = (input \* POSITIVE\_SCALE) + OFFSET

if (output < LOWER\_LIMIT)

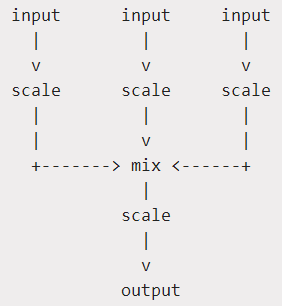
output = LOWER\_LIMIT

if (output > UPPER\_LIMIT)

output = UPPER\_LIMIT

**混控（Mixers）：**不同的混控有不同的形式。

**简单混控（Simple Mixer）：**顾名思义，简单混控读取一个或者多个输入，将其进行缩放，将缩放后的值相加，将相加后的值缩放，然后输出。



简单混控的的参数有：

* 输入组
* 每个输入的缩放参数
* 输出的缩放参数

输入值和输出值可以是任何浮点数，但是通常会是-1.0到1.0之间（使用线性缩放）。

**多旋翼混控（Multirotor Mixer）：**此混控是专门用来为多旋翼的混合飞行控制（滚转roll，俯仰pitch，航向yaw，油门thrust）设计的用于产生马达转速控制输出的。

多旋翼混控的参数有：

* 飞行器的几何外观（X型、+型、六旋翼、Y型）
* 单独的滚转roll、俯仰pitch、航向yaw、油门thrust的缩放参数
* 马达输出死区（限制范围）

滚转roll、俯仰pitch、航向yaw的输入范围为-1.0 ~ 1.0，而油门thrust的输入范围为0 ~ 1.0 。

**比例钳位（Ratio Clamping）：**多旋翼混控的输出是被钳位的，使得做多只有一个电机的输出达到饱和，来避免出现翻滚的情况，同时输出值也不允许进入死区限制范围，来防止出现在飞行中电机重启的情况。此外，任何钳位都是根据输入的比例进行的，来防止多旋翼翻滚。如果一个电机出现正饱和或者负饱和，那么总的油门将会减小，从而单独各个电机间的比例将会得到满足而不会饱和。

钳位例子：

输入（四旋翼，四个马达），限制为100：

150 75 75 75

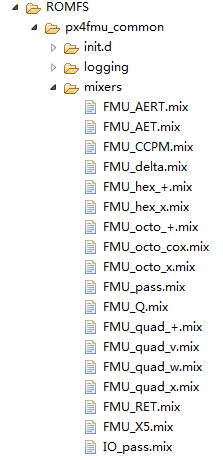
钳位输出结果：

100 50 50 50

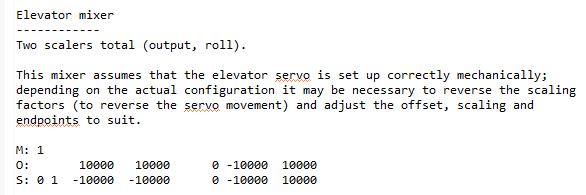
注意，此例子中，钳位后，马达1的转速依然是另外三个的两倍，所以结果姿态会是正确的。相比与没有进行钳位的情况，飞行器将不会增加高度。如果只是对马达1进行了限制（100 75 75 75），飞行器将出现翻滚或者不稳定的情况，这是因为电机间的比例发生了变化。

**下面是关于程序中，Mix配置的讲解。**

PX4工程中对Mix的配置是使用“\*.mix”文件来进行的，这些文件存放在ROMFS/px4fmu\_common/mixers 文件夹下，如下图所示（每个文件对应一种配置）：



这些文件都是文本文件，有意义的行的格式都是以一个大写字母开头，之后跟一个冒号；而不是以大写字母开头并跟一个冒号的行则作为注释行。如下面的例子所示：



其中只有后三行是有效的，其余的都是注释。

每个配置文件可以定义多个混控；混控的分配都是根据特定的飞行器进行定义的，混控产生的执行器输出的个数也是与混控特性相关的。

混控定义的格式为：

<tag>: <mixer arguments>

tag选取混控的格式；“M”表示一个简单混控，“R”表示一个多旋翼混控，等等。

**空混控（Null Mixer）:**

一个空混控是指一个没有输入，输出一直为0的混控。一般情况下，空混控用来在一组混控中为达到一种特别的格式的输出而作为一个占位符。

它的格式如下：

Z:

**简单混控（Simple Mixer）：**

一个简单混控是将0个或者多个控制输入整合到一个执行器输出中。输入经过了缩放，混控功能将结果在输出前进行了相加。

它的格式如下：

M: <control count>

O: <-ve scale> <+ve scale> <offset> <lower limit> <upper limit>

S: <group> <index> <-ve scale> <+ve scale> <offset> <lower limit> <upper limit>

在第一行中，如果control count是0，那么总和实际上为0，混控会输出一个由偏移（offset）和上下限（lower limit , upper limit）决定的固定值。

第二行定义了输出缩放规则。这里的数值都进行了缩放，放大了10000倍，即：用-5000表示-0.5。

第三行定义了控制输入和它们的缩放规则。group定义了控制组号，index指的是在控制组中的偏移值。当用来混控飞行器控制，混控控制组0是飞行器姿态控制组，它的index从0到3依次为roll、pitch、yaw和thrust。剩下的部分是定义输入的缩放规则的，也是放大了10000倍。

**多旋翼混控（Multirotor Mixer）:**

多旋翼混控将四个控制输入（roll、pitch、yaw和thrust）整合成一系列执行器输出用来控制马达速率。

它的格式如下，只有一行：

R: <geometry> <roll scale> <pitch scale> <yaw scale> <deadband>

geometry（飞行器几何外形）包含如下几种：

4x – X型四旋翼

4+ - +型四旋翼

6x – X型六旋翼

6+ -+型六旋翼

8x – X型八旋翼

8+ -+型八旋翼

每个roll、pitch和yaw缩放参数决定了roll、pitch和yaw相对于总油门（thrust）的缩放比率。这些参数也都放大的10000倍。

Roll、pitch和yaw的如入在-1.0到1.0之间，而thrust的范围在0到1.0之间（因为油门没有负的）。各个执行器的输出范围则为-1.0到1.0。，

在一个执行器饱和的情况下，所有执行器的值将被重新缩放来使得饱和的那个执行器的值限制在1.0以内（由于是同时缩放所有的执行器的值，因此各个执行器见大小的比例不变，从而防止出现侧翻的情况，当然，这样会导致掉高度等情况出现）。

注意：\*.mix文件只是定义了飞行器的几何外形（R:开头的行），以及剩下的通道的混控（M：开头的行）。比如X型四旋翼前四个通道对应四个马达通道，后四个通道为辅助通道，可以自定义为别的用途，在FMU\_quad\_x.mix文件中通过“R： ”开头的行定义飞行器为X型四旋翼，前四个通道为马达输出，然后通过四个“M：”开头的行定义后四个控制通道为简单的辅助控制输出；而对于X型六旋翼，则通过“R：”开头的行定义前6个通道为马达输出，然后通过两个“M:”开头的行定义后面两个通道为简单的辅助控制输出。

**具体对“R： ”开头行的使用**，则是在src / modules / systemlib / mixer / mixer\_multirotor.cpp文件中，该文件中的函数可以识别”R：” 开头的行，并提取其中的参数，然后根据提取的参数，将roll、pitch、yaw等转化为各个马达的控制输出。具体在某种几何形状的飞行器中，某个电机的混控参数，则是直接在这个“.cpp”文件中，“.mix”文件只是选择机型。

**同样，具体对“M：”开头行的使用**，则是在/ src / modules / systemlib / mixer / mixer\_simple.cpp文件中。

## 自定义应用程序

参考链接：<http://pixhawk.org/dev/px4_simple_app>

**步骤一：文件设置**

1、从Github上下载源文件，链接：<https://github.com/px4/Firmware/>。

2、将工程导入PX4 Eclippse中，导入方法参考链接：

<http://pixhawk.org/dev/toolchain_installation_win>

3、make archives ， 当子模块更改或者Nuttx配置更改过后，需要进行这一步。一般情况下有版本迁移或者更新后都要进行这一步。

4、在本地硬盘的“Firmware/src/modules”目录下新建一个自己命名的文件夹，如“px4\_simple\_app”，在这个文件夹中创建一个新的名字为“module.mk”的文件（注意，此文件名必须为module.mk），在该文件中添加如下代码：

MODULE\_COMMAND = px4\_simple\_app

SRCS = px4\_simple\_app.c

**步骤二：编写应用程序**

在第一步创建的文件夹中创建一个名为“px4\_simple\_app.c”的文件，向其中添加如下代码（注意，Nuttx系统下的模块的主函数名字都是以“\_main”开始的，而调用的时候，不加\_main，而是直接用“px4\_simple\_app”）：

*/\*\**

*\* @file px4\_simple\_app.c*

*\* Minimal application example for PX4 autopilot.*

*\*/*

*#include <nuttx/config.h>*

*#include <stdio.h>*

*#include <errno.h>*

\_\_EXPORT int px4\_simple\_app\_main(int argc, char \*argv[]);

int px4\_simple\_app\_main(int argc, char \*argv[])

{

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Hello Sky!\n");

return OK;

}

**步骤三：在NuttShell中注册该程序并编译**

在Firmware/makefiles/config\_px4fmu-v2\_default.mk文件中添加如下行：

MODULES += modules/px4\_simple\_app

执行如下操作：

make clean

make px4fmu-v2\_default

如果没有新的程序被注册，只执行make px4fmu-v2\_default即可，这样更快一点。

**步骤四：上传并测试程序**

执行上传命令：

make upload px4fmu-v2\_default

在你重启板子之前，界面会返回如下信息：

Generating /Users/user/src/Firmware/Images/px4fmu.px4

Loaded firmware for 5,0, waiting for the bootloader...

重启板子，开始上传，会返回如下信息：

Found board 5,0 on /dev/tty.usbmodem1

erase...

program...

verify...

done, rebooting.

现在切换到NSH连接，在shell中单击回车键，会返回如下界面：

nsh>

输入help并按回车键，弹出如下信息：

nsh> help

help usage: help [-v] [<cmd>]

[ df kill mkfifo ps sleep

? echo losetup mkrd pwd test

cat exec ls mh rm umount

cd exit mb mount rmdir unset

cp free mkdir mv set usleep

dd help mkfatfs mw sh xd

Builtin Apps:

reboot

perf

top

..

px4\_simple\_app

..

sercon

serdis

可以发现px4\_simple\_app已经出现在可用内建程序列表中，输入px4\_simple\_app并按回车，出现如下信息：

nsh> px4\_simple\_app

Hello Sky!

这个程序运行成功，下一步可以将其进一步扩充。

**步骤五：订阅传感器信息**

为了做些有用的事，我们需要将程序修改一下，使它可以订阅一些输入数据，并能发布一些输出数据。注意，PX4平台的真实硬件抽象概念的优势也在此处体现：我们不需要与底层硬件驱动打交道，当电路板或者传感器改变时，我们的应用程序不需要做任何改动。

在PX4中，应用程序间通信通道成为topic（本文“进程间通信”章节有详细讲述），此处例程，我们使用名为“sensor\_combined”的topic，它包含了整个系统的传感器信息，并且是同步的。

订阅一个topic是非常迅速并且简洁的：

*#include <uORB/topics/sensor\_combined.h>*

..

int sensor\_sub\_fd = orb\_subscribe(ORB\_ID(sensor\_combined));

其中，“sensor\_sub\_fd”是一个文件描述符，可以用来非常高效地实现对新数据的阻塞式等待。当前线程进入休眠状态，当新数据可用时，它会自动地被调度程序唤醒，因此在数据等待时，不会占用任何CPU资源。为实现这个功能，我们用到了poll()系统函数（这个函数在Linux操作系统中很常用）。

添加完poll()函数后整个程序如下：



编译并进行测试，首先打开uorb程序，这个程序可能已经启动了，但是再次启动它不会产生问题。

uorb start

启动传感器：

sh /etc/init.d/rc.sensors

启动自定义程序（注意此处与上面的不同，结尾添加了一个”&”符合）：

px4\_simple\_app &

二者的区别在于uorb和传感器程序都似乎Daemon（守护程序，或称后天程序）。这使得即使我们忘了加“&”，我们依然可以启动、停止它们，而不会失去对NuttShell的控制。

自定义的程序现在就好快速输出一堆传感器信息：

[px4\_simple\_app] Accelerometer: 0.0483 0.0821 0.0332

[px4\_simple\_app] Accelerometer: 0.0486 0.0820 0.0336

[px4\_simple\_app] Accelerometer: 0.0487 0.0819 0.0327

[px4\_simple\_app] Accelerometer: 0.0482 0.0818 0.0323

[px4\_simple\_app] Accelerometer: 0.0482 0.0827 0.0331

[px4\_simple\_app] Accelerometer: 0.0489 0.0804 0.0328

因为它不是一个daemon，我们没有办法去停止它。要么让它继续运行，要么输入以下命令重启飞控：

reboot

或者按飞控板上的复位按钮。本文后面的章节会讲如何将一个程序转换为一个daemon。

**步骤六：限定订阅频率**

测试程序会将传感器信息以数据洪流般淹没窗口，事实上我们往往不需要全速率的数据，要跟上全速率的数据往往会将整个系统变慢。

将数据速率改为1Hz只需添加如下行：

orb\_set\_interval(sensor\_sub\_fd, 1000);

对于我们的程序，在下面的这一行（大约第55行）下面添加该行即可：

*/\* subscribe to sensor\_combined topic \*/*

int sensor\_sub\_fd = orb\_subscribe(ORB\_ID(sensor\_combined));

现在控制台就会以1Hz的速率输出传感器信息。

**步骤七：发布信息**

要使用计算后的输出，下一步就是发布结果。如果我们使用一个已知mavlink程序会将其传递到地面站软件的的topic，我们甚至可以在地面站中查看结果。为达到此目的，让我们拦截attitude（姿态）topic。

接口很简单：初始化将要发布的topic结构体，并通告它。

*#include <uORB/topics/vehicle\_attitude.h>*

*/\* advertise attitude topic \*/*

struct vehicle\_attitude\_s att;

memset(&att, 0, sizeof(att));

int att\_pub\_fd = orb\_advertise(ORB\_ID(vehicle\_attitude), &att);

在主循环中，当信息准备好后发布它：

orb\_publish(ORB\_ID(vehicle\_attitude), att\_pub\_fd, &att);

修改后的完整程序如下：



运行最终的程序，这次需要先启动一些别的程序：

uorb start

sh /etc/init.d/rc.sensors

mavlink start -d /dev/ttyS1 -b 115200

px4\_simple\_app &

如果你启动地面站软件QGroundControl或者Mission Planner，你可以实时观测传感器信息的曲线图（Main Menu：Main Widget -> Realtime Plot），这些信息正是你的程序发布的。

本章节包含了用户自定义PX4飞控程序的所有要点。注意飞控中所有topic的列表链接如下：

<https://github.com/PX4/Firmware/tree/master/src/modules/uORB/topics>

你也可以在工程中“/src/modules/uORB/topics”目录下找到一系列头文件，这些头文件就包含了所有的topic。

## 创建Daemon程序（后台程序）

参考链接：<http://pixhawk.org/dev/px4_daemon_app>

在Unix和其他多任务操作系统中，daemon(/ˈdeɪmən/ or /ˈdiːmən/)程序是指作为一个后台进程运行的计算机程序，而不是由用户直接控制的程序。daemon概念的好处是它不需被用户或者shell发送到后台就能被启动，并且当它在运行时可以通过shell查询它的状态，它也可以被终止。

**步骤一：创建一个小的普通程序**

\_\_EXPORT int px4\_daemon\_app\_main(int argc, char \*argv[]);

int px4\_daemon\_app\_main(int argc, char \*argv[])

{

while (**true**) {

warnx("Hello Daemon!\n");

sleep(1);

}

return 0;

}

这个程序的问题很明显：如果启动它时没有加“&”，它会阻塞shell（Nuttx不会阻塞，支持Ctrl + Z /fg /bg）。为了规避这个问题，下面的部分会将这个程序转换为一个daemon。

**步骤二：创建一个Daemon控制函数**

主函数现在由一个daemon控制函数代替，主函数中的原来的部分现在有一个后台任务（task）/线程（thread）来代替。

*#include <systemlib/systemlib.h>*

..

\_\_EXPORT int px4\_daemon\_app\_main(int argc, char \*argv[]);

..

int mavlink\_thread\_main(int argc, char \*argv[]);

..

int mavlink\_thread\_main(int argc, char \*argv[])

{

while (**true**) {

warnx("Hello Daemon!\n");

sleep(1);

if (thread\_should\_exit) **break**;

}

return 0;

}

..

int px4\_daemon\_app\_main(int argc, char \*argv[])

{

if (argc < 1)

usage("missing command");

if (![strcmp](http://www.opengroup.org/onlinepubs/009695399/functions/strcmp.html)(argv[1], "start")) {

if (thread\_running) {

warnx("daemon already running\n");

*/\* this is not an error \*/*

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(0);

}

thread\_should\_exit = **false**;

daemon\_task = task\_spawn\_cmd("daemon",

SCHED\_RR,

SCHED\_PRIORITY\_DEFAULT,

4096,

px4\_daemon\_thread\_main,

(argv) ? (const char \*\*)&argv[2] : (const char \*\*)NULL);

thread\_running = **true**;

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(0);

}

usage("unrecognized command");

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(1);

}

这个会启动一个拥有4096个字节的栈，并将非daemon特殊指令行选项传递给后台主函数。一个典型的调用如下：

px4\_daemon\_app start

上述代码并不报告一个状态，并且无法防止程序被多次启动。

**步骤三：添加终止/状态命令和安全措施**

如下程序添加了终止和状态命令以及安全措施：

*/\*\**

*\* @file px4\_daemon\_app.c*

*\* daemon application example for PX4 autopilot*

*\**

*\* @author Example User <mail@example.com>*

*\*/*

*#include <nuttx/config.h>*

*#include <nuttx/sched.h>*

*#include <unistd.h>*

*#include <stdio.h>*

*#include <systemlib/systemlib.h>*

*#include <systemlib/err.h>*

static bool thread\_should\_exit = **false**; */\*\*< daemon exit flag \*/*

static bool thread\_running = **false**; */\*\*< daemon status flag \*/*

static int daemon\_task; */\*\*< Handle of daemon task / thread \*/*

*/\*\**

*\* daemon management function.*

*\*/*

\_\_EXPORT int px4\_daemon\_app\_main(int argc, char \*argv[]);

*/\*\**

*\* Mainloop of daemon.*

*\*/*

int px4\_daemon\_thread\_main(int argc, char \*argv[]);

*/\*\**

*\* Print the correct usage.*

*\*/*

static void usage(const char \*reason);

static void

usage(const char \*reason)

{

if (reason)

warnx("%s\n", reason);

errx(1, "usage: daemon {start|stop|status} [-p <additional params>]\n\n");

}

*/\*\**

*\* The daemon app only briefly exists to start*

*\* the background job. The stack size assigned in the*

*\* Makefile does only apply to this management task.*

*\**

*\* The actual stack size should be set in the call*

*\* to task\_create().*

*\*/*

int px4\_daemon\_app\_main(int argc, char \*argv[])

{

if (argc < 1)

usage("missing command");

if (![strcmp](http://www.opengroup.org/onlinepubs/009695399/functions/strcmp.html)(argv[1], "start")) {

if (thread\_running) {

warnx("daemon already running\n");

*/\* this is not an error \*/*

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(0);

}

thread\_should\_exit = **false**;

daemon\_task = task\_spawn\_cmd("daemon",

SCHED\_DEFAULT,

SCHED\_PRIORITY\_DEFAULT,

2000,

px4\_daemon\_thread\_main,

(argv) ? (const char \*\*)&argv[2] : (const char \*\*)NULL);

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(0);

}

if (![strcmp](http://www.opengroup.org/onlinepubs/009695399/functions/strcmp.html)(argv[1], "stop")) {

thread\_should\_exit = **true**;

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(0);

}

if (![strcmp](http://www.opengroup.org/onlinepubs/009695399/functions/strcmp.html)(argv[1], "status")) {

if (thread\_running) {

warnx("\trunning\n");

} else {

warnx("\tnot started\n");

}

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(0);

}

usage("unrecognized command");

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(1);

}

int px4\_daemon\_thread\_main(int argc, char \*argv[]) {

warnx("[daemon] starting\n");

thread\_running = **true**;

while (!thread\_should\_exit) {

warnx("Hello daemon!\n");

sleep(10);

}

warnx("[daemon] exiting.\n");

thread\_running = **false**;

return 0;

}

测试改程序，将会产生如下输出：

nsh> px4\_daemon\_app start

[daemon] starting

Hello Daemon!

完整程序已经包含在PX4工程中了，路径为：

src/examples/px4\_daemon\_app/px4\_daemon\_app.c

将该程序添加到Firmware/makefiles/config\_px4fmu\_default.mk中，该文件中已经包含了这个程序，不过被注释掉了，取消注释即可。

## 创建一个固定翼控制主程序

参考链接：<http://pixhawk.org/dev/fixedwing_control>

本例程非常有利于全面了解PX4飞行控制流程！

程序比较大，不在本文列出，源程序在src/examples/fixedwing\_control下，编译之前，在Firmware/makefiles/config\_px4fmu-v2\_default.mk文件中取消MODULES += examples/fixedwing\_control行的注释即可。

运行方法：

ex\_fixedwing\_control start

## NuttX操作系统

参考链接：<http://nuttx.org/doku.php?id=documentation:userguide>

NuttX是一个flat addresss的操作系统，也就是说它不提供想Linux那样的进程（processes）。NuttX只支持简单的运行在同一地址空间的线程。但是，它的程序模型使得task（任务）与pthread（线程）间有一定区别。

* tasks：有一定独立性的线程；
* pthreads：共享某些资源的线程。

## 文件系统（File System）

参考链接：<http://pixhawk.org/dev/file_system>

PX4有一个虚拟的文件系统，被保存带2到3个不同的器件中：

1、只读文件系统（read-only file system , ROMFS），作用为保存启动脚本，这个文件系统保存在MCU（STM32）的内部Flash中（编译时直接编译到程序中了）。这个文件被挂载在/etc下。

2、可写microSD卡文件系统（通常为FAT32格式）。作用是用来存储log文件。它被挂载在/fs/microsd下。

3、可写FRAM文件系统（被映射到FRAM中的单个文件），挂载在/fs/mtd\_params和/fs/mtd\_waypoints下。用来存储参数（parameters）和航点（waypoints）。对于FMUV1.x板子（PX4电路板，而非Pixhawk），上面没有FRAM，用的是EEPROM，因此映射到了EEPROM中。

**microSD注意事项：**

因为microSD卡需要时间去写文件，所以在板子正在log（写日志）的时候断电有可能会损坏系统。Logging在disarm的时候会自动被停止，因此系统断电前需要disarm。

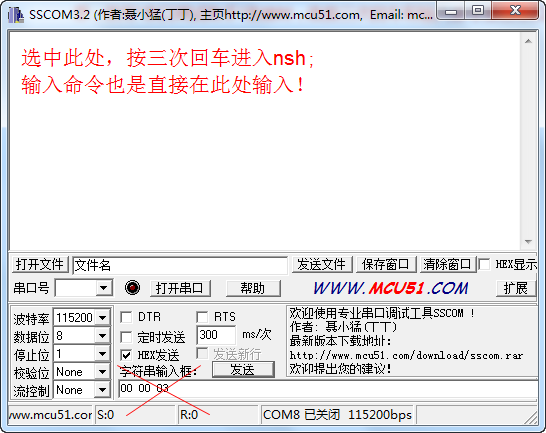
## nsh（NuttShell）

参考链接：<http://pixhawk.org/dev/nuttx/nsh>

1、nuttx/configs/px4fmu-v2/nsh/defconfig中有关于nsh的配置，其中第547行开始是各个串口的配置。

2、对于PX4板子，nsh默认对应USB和串口1；对于Pixhawk板子，nsh默认对应的是USB和串口8。如果需要修改为其他串口，可以根据本章的参考链接进行修改。

3、在USB连接情况下，打开USB对应的那个串口，在终端界面按三次回车键，将进入nsh（nsh启动脚本在系统启动时会立即启动，其输出会默认输出到/dev/null，并不会输出到nsh端口）。如果连不上，可以修改波特率试一下。注意，使用“串口调试助手”连接时，连按三次回车和输入命令是在串口上方的大的显示窗口输入，而不是在下面的字符串输入栏输入，如下图：



## 底层驱动

**1、类的概念。**

PX4中不少程序是用C++写的，大量使用了类的概念。如MPU6000这个芯片，与STM32的SPI接口相连（几个传感器都是通过同一个SPI与STM32相连，只是用CS引脚来加一区分）。在src/drivers/device/spi.h中定义了SPI类，而在MPU6000的驱动源文件（src/drivers/mpu6000/mpu6000.cpp）中，使用了继承的方法创建了mpu6000类：

class MPU6000 : public device::SPI

其他几个与SPI相连的传感器也是如此进行初始化。

进一步研究发现，最底层的初始化还是用的C语言编写，并且调用的是STM32官方固件库， 如src/drivers/boards/px4fmu-v2/px4fmu\_spi.c定义了SPI引脚的初始化以及几个作为片选（CS）的GPIO的初始化。

2、**FMU与IO连接**

FMU的CPU（STM32F427）通过其USART6与IO的CPU（STM32F103）的相连。

## 线程优先级（Thread Priorities）

参考链接：<http://pixhawk.org/dev/thread_priorities>

尽管PX4中，各个线程是运行在操作系统NuttX的主循环之上，其核心部件的运行依然是时间确定的（实时的），并且永远都是按照固定顺序依次执行的。这个是靠带优先级的循环调度来实现的。

一个标准的运行在250Hz的子程序示例如下：

1、读出传感器数据；

2、依靠新读出的传感器数据进行姿态估计；

3、姿态控制器更新（使用新的姿态和当前设定位置点）。

**优先级集合（Priority Bands）**

PX4固件中的程序被被分为了不同优先级的集合：

1、（中断级）快速传感器驱动；

2、看门狗/系统状态监控；

3、执行器输出（PWM输出驱动线程，IO通信发送线程）；

4、姿态控制器；

5、慢速/阻塞式驱动（必须不能阻塞姿态控制器）；

6、目标/位置控制器；

7、默认优先级——普通用户程序，shell命令，随机废弃物（？）……

8、logger，参数同步；

9、空闲。

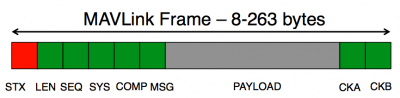
## Mavlink

参考链接：<http://qgroundcontrol.org/mavlink/start>

MAVLink是一个轻量级的、只由头文件组成的信息集编库，常被用在微型飞行器上。它能将C语言结构的串行数据高效低打包并发送给地面控制软件。它已经被大量使用在如下系统中：PX4、Pixhawk、APM和Parrot AR.Drone等几种飞控中，同时，也用在微控制器（MCU）和惯性测量装置（IMU）间的通信，此外，也用在Linux进程与地面链路间的通信。

**帧结构：**

Mavlink的帧结构参考了CAN和SAE AS-4标准，其帧结构如下：



|  |  |  |  |
| --- | --- | --- | --- |
| **字节序号** | **内容** | **值** | **备注** |
| 0 | 帧头 | V1.0 : 0xFE  V0.9：0x55 | 指示帧的起始（一位足够，官网有解释） |
| 1 | 帧长 | 0 - 255 | 表示帧数据长度 |
| 2 | 帧序号 | 0 - 255 | 每发一帧，自动增一，溢出归零。用来判断是否丢帧。 |
| 3 | 系统ID (System ID) | 1 - 255 | 发送方系统ID。允许同一网络中有不同的系统。 |
| 4 | 组件ID (Component ID) | 0 - 255 | 发送组件ID。允许同一系统中有多个组件，如IMU和飞控。 |
| 5 | 信息ID (Message ID) | 0 -255 | 信息ID：指示帧的定义 |
| 6 to (n+6) | 数据（Data） | (0 - 255)bytes | 数据，根据Message ID进行编码 |
| (n+7) to (n+8) | Checksum(低字节、高字节) | ITU X.25/SAE AS-4 hash，不包含帧头，即1~（n+6） | |

* 最短的帧长度为8个字节（命令状态返回帧）；
* 最长的帧长度为263个字节。

**支持的数据类型**

Mavlink支持固定大小整形数据，IEEE 754单精度浮点数据，这些数据组成的数组（如：char[10]），已经特殊的mavlink\_version域，被协议自动添加。如下类型是被支持的：

* **char** - Characters / strings
* **uint8\_t** - Unsigned 8 bit
* **int8\_t** - Signed 8 bit
* **uint16\_t** - Unsigned 16 bit
* **int16\_t** - Signed 16 bit
* **uint32\_t** - Unsigned 32 bit
* **int32\_t** - Signed 32 bit
* **uint64\_t** - Unsigned 64 bit
* **int64\_t** - Signed 64 bit
* **float** - IEEE 754 single precision floating point number
* **double** - IEEE 754 double precision floating point number
* **uint8\_t\_mavlink\_version** – 8位无符号数，被自动添加，用于指示Mavlink版本；无法被修改，只能被作为一个普通的uint8\_t数被读出。

**性能**

这个协议注重两个特性：传输速度和安全性。它允许检测信息内容，也允许检测丢失的信息，但依然只需要在每帧数据前加上6个字节的头部（即帧头、各种ID等等）。

发送示例：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **数据链速率** | **硬件** | **更新速率** | **有效帧长** | **等效float数个数** |
| 115200 | XBee Pro 2.4 GHz | 50 Hz | 224 bytes | 56 |
| 115200 | XBee Pro 2.4 GHz | 100 Hz | 109 bytes | 27 |
| 57600 | XBee Pro 2.4 GHz | 100 Hz | 51 bytes | 12 |
| 9600 | XBee Pro XSC 900 | 50 Hz | 13 bytes | 3 |
| 9600 | XBee Pro XSC 900 | 20 Hz | 42 bytes | 10 |

**Mavlink板级集成教程（MAVLink Onboard Integration Tutorial）**

参考链接：

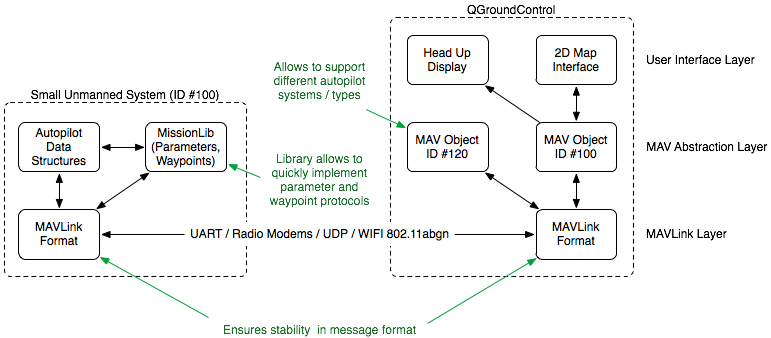
<http://qgroundcontrol.org/dev/mavlink_onboard_integration_tutorial>

Mavlink是一个只由头文件组成的库，这意味着你在MCU编程时不需要编译它。你只需要将Mavlink/include添加到程序列表中即可。（注意：你可以创建自定义的信息帧，然后使用Mavlink官方提供的工具转换得到C语言程序）

Mavlink是stateless的，但是地面站软件QGroundControl会通过心跳信息（heartbeat message）来确定系统是否处于活跃状态。因此需要确保每个60、30、10或者1秒（1Hz是推荐值，但是非必须）发送一次心跳信息，只有这样这个系统才会被认为是活跃的。

**集成结构**

如下图所示，集成Mavlink是非入侵式的。它不需要成为板子架构的核心部分。Mavlink提供的missionlib掌管参数和任务以及路径点的传输，飞控只需要从特性的结构体中读出其值即可。



快速集成：发送数据

下面的示例方法会需要很多行（与实际应用相比），但是能够帮助你快速入门。

*/\* The default UART header for your MCU \*/*

*#include "uart.h"*

*#include <mavlink/v1.0/common/mavlink.h>*

mavlink\_system\_t mavlink\_system;

mavlink\_system.sysid = 20; *///< ID 20 for this airplane*

mavlink\_system.compid = MAV\_COMP\_ID\_IMU; *///< The component sending the message is the IMU, it could be also a Linux process*

mavlink\_system.type = MAV\_TYPE\_FIXED\_WING; *///< This system is an airplane / fixed wing*

*// Define the system type, in this case an airplane*

uint8\_t system\_type = MAV\_TYPE\_FIXED\_WING;

uint8\_t autopilot\_type = MAV\_AUTOPILOT\_GENERIC;

uint8\_t system\_mode = MAV\_MODE\_PREFLIGHT; *///< Booting up*

uint32\_t custom\_mode = 0; *///< Custom mode, can be defined by user/adopter*

uint8\_t system\_state = MAV\_STATE\_STANDBY; *///< System ready for flight*

*// Initialize the required buffers*

mavlink\_message\_t msg;

uint8\_t buf[MAVLINK\_MAX\_PACKET\_LEN];

*// Pack the message*

mavlink\_msg\_heartbeat\_pack(mavlink\_system.sysid, mavlink\_system.compid, &msg, system\_type, autopilot\_type, system\_mode, custom\_mode, system\_state);

*// Copy the message to the send buffer*

uint16\_t len = mavlink\_msg\_to\_send\_buffer(buf, &msg);

*// Send the message with the standard UART send function*

*// uart0\_send might be named differently depending on*

*// the individual microcontroller / library in use.*

uart0\_send(buf, len);

**接收数据**

上述例子讲述了如何使用不太方便的函数发送数据（这种函数需要更多的行）。下面的程序讲述如何接收数据。这里为了示例，直接把接收放在主循环中了，这样会尽可能快地清空串口接收缓存。

*#include <mavlink/v1.0/common/mavlink.h>*

*// Example variable, by declaring them static they're persistent*

*// and will thus track the system state*

static int packet\_drops = 0;

static int mode = MAV\_MODE\_UNINIT; */\* Defined in mavlink\_types.h, which is included by mavlink.h \*/*

*/\*\**

*\* @brief Receive communication packets and handle them*

*\**

*\* This function decodes packets on the protocol level and also handles*

*\* their value by calling the appropriate functions.*

*\*/*

static void communication\_receive(void)

{

mavlink\_message\_t msg;

mavlink\_status\_t status;

*// COMMUNICATION THROUGH EXTERNAL UART PORT (XBee serial)*

while(uart0\_char\_available())

{

uint8\_t c = uart0\_get\_char();

*// Try to get a new message*

if(mavlink\_parse\_char(MAVLINK\_COMM\_0, c, &msg, &status)) {

*// Handle message*

switch(msg.msgid)

{

case MAVLINK\_MSG\_ID\_HEARTBEAT:

{

*// E.g. read GCS heartbeat and go into*

*// comm lost mode if timer times out*

}

**break**;

case MAVLINK\_MSG\_ID\_COMMAND\_LONG:

*// EXECUTE ACTION*

**break**;

default:

*//Do nothing*

**break**;

}

}

*// And get the next one*

}

*// Update global packet drops counter*

packet\_drops += status.packet\_rx\_drop\_count;

*// COMMUNICATION THROUGH SECOND UART*

while(uart1\_char\_available())

{

uint8\_t c = uart1\_get\_char();

*// Try to get a new message*

if(mavlink\_parse\_char(MAVLINK\_COMM\_1, c, &msg, &status))

{

*// Handle message the same way like in for UART0*

*// you can also consider to write a handle function like*

*// handle\_mavlink(mavlink\_channel\_t chan, mavlink\_message\_t\* msg)*

*// Which handles the messages for both or more UARTS*

}

*// And get the next one*

}

*// Update global packet drops counter*

packet\_drops += status.packet\_rx\_drop\_count;

}

**使用便捷的函数**

Mavlink提供了方便发送帧的便捷函数。如果你想使用这些函数，你需要在你的代码中添加一些特殊的函数、这种方法在发送数据时需要更少的代码，但是需要你自己去写适配头文件。下面的例子中将会讲到如何使用这些函数（注意：我们建议初学者从非便捷函数用起，能成功发送消息到地面站软件后，如果需要，再去添加适配头文件）。

*#include "your\_mavlink\_bridge\_header.h"*

*/\* You have to #define MAVLINK\_USE\_CONVENIENCE\_FUNCTIONS in your\_mavlink\_bridge\_header,*

*and you have to declare: mavlink\_system\_t mavlink\_system;*

*these two variables will be used internally by the mavlink\_msg\_xx\_send() functions.*

*Please see the section below for an example of such a bridge header. \*/*

*#include <mavlink.h>*

*// Define the system type, in this case an airplane*

int system\_type = MAV\_FIXED\_WING;

*// Send a heartbeat over UART0 including the system type*

mavlink\_msg\_heartbeat\_send(MAVLINK\_COMM\_0, system\_type, MAV\_AUTOPILOT\_GENERIC, MAV\_MODE\_MANUAL\_DISARMED, MAV\_STATE\_STANDBY);

**Mavlink适配头文件**

下面的例程是为一个单片机写的，大部分情况下大家都是这样用的。

**发送函数：**

根据你的MCU定义这个函数。之后，你可以使用'mavlink\_msg\_xx\_send(MAVLINK\_COMM\_x, data1, data2, ..)' 函数来便捷地发送数据。your\_mavlink\_bridge\_header.h如下：

*/\* MAVLink adapter header \*/*

*#ifndef YOUR\_MAVLINK\_BRIDGE\_HEADER\_H*

*#define YOUR\_MAVLINK\_BRIDGE\_HEADER\_H*

*#define MAVLINK\_USE\_CONVENIENCE\_FUNCTIONS*

*#include <mavlink/v1.0/mavlink\_types.h>*

*/\* Struct that stores the communication settings of this system.*

*you can also define / alter these settings elsewhere, as long*

*as they're included BEFORE mavlink.h.*

*So you can set the*

*mavlink\_system.sysid = 100; // System ID, 1-255*

*mavlink\_system.compid = 50; // Component/Subsystem ID, 1-255*

*Lines also in your main.c, e.g. by reading these parameter from EEPROM.*

*\*/*

mavlink\_system\_t mavlink\_system;

mavlink\_system.sysid = 100; *// System ID, 1-255*

mavlink\_system.compid = 50; *// Component/Subsystem ID, 1-255*

*/\*\**

*\* @brief Send one char (uint8\_t) over a comm channel*

*\**

*\* @param chan MAVLink channel to use, usually MAVLINK\_COMM\_0 = UART0*

*\* @param ch Character to send*

*\*/*

static **inline** void comm\_send\_ch(mavlink\_channel\_t chan, uint8\_t ch)

{

if (chan == MAVLINK\_COMM\_0)

{

uart0\_transmit(ch);

}

if (chan == MAVLINK\_COMM\_1)

{

uart1\_transmit(ch);

}

}

*#endif /\* YOUR\_MAVLINK\_BRIDGE\_HEADER\_H \*/*

## 板间通信（Offboard Communication）

参考链接：<http://pixhawk.org/dev/offboard_communication>

本章讲述如何通过MAVLink将传感器信息从飞控板发送到地面计算机。在参考程序中，缩放后的整型的传感器原始读数通过MAVlink以500Hz的频率发送到计算机。

硬件条件：

* 飞控板
* 两根连接飞控板的UART1和UART2的USB线
* 一台运行Linux的电脑（Debian，Ubuntu等）
* 一根micro

**Requirements**

* PX4FMU
* Two serial adapters from the computer to UART1 and UART2 of PX4FMU
* A computer running \*nix (e.g. Debian, Ubuntu, Mac OS, BSD, Archlinux, etc.) with C compiler installed
* A micro USB cable to power PX4FMU

## 设置飞行模式（Set Flight Mode）

参考链接：<http://pixhawk.org/dev/set_flight_mode>

PX4的飞行模式（flight mode）包含了主飞行模式（main flight mode）（如手动MANUAL、自动AUTO等等）和一个子飞行模式（sub-mode），子飞行模式在很多情况下代表了当前的导航状态（如定高、定点或者自动巡航等等）。这两种模式是通过名为vehicle\_status的topic传输，并通过Mavlink中的custom\_mode域传输。

主和子模式的定义在src/modules/commander/ px4\_custom\_mode.h中：

**enum** PX4\_CUSTOM\_MAIN\_MODE {

*PX4\_CUSTOM\_MAIN\_MODE\_MANUAL* = 1, // 手动

*PX4\_CUSTOM\_MAIN\_MODE\_ALTCTL*, // 姿态控制

*PX4\_CUSTOM\_MAIN\_MODE\_POSCTL*, // 位置控制

*PX4\_CUSTOM\_MAIN\_MODE\_AUTO*, // 自动控制

*PX4\_CUSTOM\_MAIN\_MODE\_ACRO*,

*PX4\_CUSTOM\_MAIN\_MODE\_OFFBOARD*,

};

**enum** PX4\_CUSTOM\_SUB\_MODE\_AUTO {

*PX4\_CUSTOM\_SUB\_MODE\_AUTO\_READY* = 1,

*PX4\_CUSTOM\_SUB\_MODE\_AUTO\_TAKEOFF*, // 起飞

*PX4\_CUSTOM\_SUB\_MODE\_AUTO\_LOITER*, // 悬停

*PX4\_CUSTOM\_SUB\_MODE\_AUTO\_MISSION*, // 自动航线

*PX4\_CUSTOM\_SUB\_MODE\_AUTO\_RTL*, // 返回起飞点

*PX4\_CUSTOM\_SUB\_MODE\_AUTO\_LAND*, // 自动降落

*PX4\_CUSTOM\_SUB\_MODE\_AUTO\_RTGS*

};

**union** px4\_custom\_mode {

**struct** {

uint16\_t reserved;

uint8\_t main\_mode; // 当前主模式

uint8\_t sub\_mode; // 当前子模式

};

uint32\_t data;

**float** data\_float;

};

注意：上面的定义与《PX4调机笔记》中“飞行模式设置”章节对应，与之一起研究更易懂一些。

**通过uORB设置飞行模式**

struct vehicle\_command\_s vcmd;

[memset](http://www.opengroup.org/onlinepubs/009695399/functions/memset.html)(&vcmd, 0, sizeof(vcmd));

*/\* copy the content of mavlink\_command\_long\_t cmd\_mavlink into command\_t cmd \*/*

vcmd.param1 = 0; */\* This field can be used to send MAVLink-defined mode commands,*

*\* but will be ignored if param2 / the custom main mode is set \*/*

vcmd.param2 = main\_mode; *// value from enum PX4\_CUSTOM\_MAIN\_MODE*

vcmd.param3 = 0;

vcmd.param4 = 0;

vcmd.param5 = 0;

vcmd.param6 = 0;

vcmd.param7 = 0;

vcmd.command = VEHICLE\_CMD\_DO\_SET\_MODE;

vcmd.target\_system = <id of this system>;

vcmd.target\_component = <component id of this [system](http://www.opengroup.org/onlinepubs/009695399/functions/system.html), 0 for all components>;

vcmd.source\_system = <id of this system>;

vcmd.source\_component = <component id of this system>;

vcmd.confirmation = 0; *// Set to 1 to send a confirmation of the command to the GCS*

*// Publish the command via uORB here 下面添加uORB发送代码，此处省略*

**通过Mavlink设置飞行模型**

如果用户模式域（custom mode field）为0，则Mavlink就会将其映射为与之最近的PX4特定飞行模式。用户模式域只有在非零的情况下才会被解析。发送飞行模式需要将主模式和子模式放入Mavlink的SET\_MODE消息的用户模式域中（或者MAV\_CMD\_DO\_SET\_MODE命令的param2）。模式应该作为32位无符号整形发送，小端模式。

union px4\_custom\_mode {

struct {

uint16\_t reserved;

uint8\_t main\_mode;

uint8\_t sub\_mode;

};

uint32\_t data;

};

也可以不用union，直接如下用：

uint8\_t main\_mode = PX4\_CUSTOM\_MAIN\_MODE\_AUTO;

uint8\_t sub\_mode = PX4\_CUSTOM\_SUB\_MODE\_AUTO\_LAND;

uint32\_t custom\_mode = (main\_mode << 8) | (sub\_mode);

## 参数（Parameters）

PX4的参数可以通过地面站软件进行设定，它是一个很简单的调试固件性能的方法。典型的参数（parameters）包含控制器的PID参数；其他参数包含飞行环境、校准信息（传感器偏移）等等。

你可以在飞行中调整参数（比如PID参数），但是只有在你知道你在干什么的情况下才建议你这么做（尽量不要在飞行时调参）。控制程序（controller apps）会周期地复制并更新它们内部的相应参数。

**所有板载参数（on-board Parameters）在如下链接中列出：**

<http://pixhawk.org/firmware/parameters>

**通过地面站软件Qgroundcontrol设置参数的方法**

1、确保你打开了“Calibration and On-board Parameters”小工具的显示（菜单栏/Tool Widgets/Calibration and Parameters）。

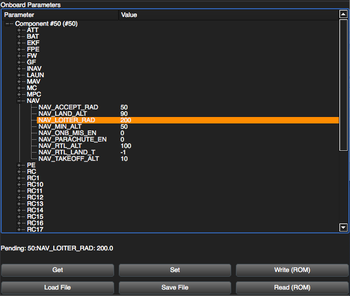
2、确保已经链接飞控。

3、连接后，你将能看到当前参数值的列表。如果看不到，按“Refresh”。

4、改变参数值。

5、按“set”。

6、如果你感觉新的参数值很好，并且想永久地保存它，按“Write（ROM）”。



**自定义参数**

参考链接：<http://pixhawk.org/dev/parameters>

在你的文件的全局范围内，使用如下三行的中一行来创建一个新的参数（这三行对应不同的参数类型）：

PARAM\_DEFINE\_INT32(GROUP\_NAME,default\_value);

PARAM\_DEFINE\_FLOAT(GROUP\_NAME,default\_value);

PARAM\_DEFINE\_STRUCT(GROUP\_NAME,default\_value);

这些行创建了一个在名字为“GROUP”的组（group）中，名字（name）为”NAME”、值（value）为“default\_value”的变量。

获取指向该变量的指针方法如下：

#include <systemlib/param/param.h>

param\_t param\_pointer = param\_find("GROUP\_NAME");

获取变量值的方法如下：

int32\_t param;

param\_get(param\_pointer,&param);

地面站软件QGroundcontrol（或者其他的基于Mavlink的地面站）就可以查看并修改该参数。

**参数文档**

所有参数都应该按照如下格式书写：

*/\*\**

*\* Gyro X offset*

*\**

*\* This is an X-axis offset for the gyro.*

*\* Adjust it according to the calibration data.*

*\**

*\* @min -10.0*

*\* @max 10.0*

*\* @group Gyro Config*

*\*/*

PARAM\_DEFINE\_FLOAT(SENS\_GYRO\_XOFF, 0.0f);

这样就会自动产生如下说明文档：

**Gyro X offset FIXME (SENS\_GYRO\_XOFF)**

This is an X-axis offset for the gyro. Adjust it according to the calibration data.

\* Minimal value: -10.0 \* Maximal value: 10.0 \* Default value: 0.0

在代码中，参数将会在四个地方被调用：

1、uORB app：承担参数的定义和存储；

2、commander app：从eeprom中加载（load）参数，将参数保存（save）到eeprom中。

3、Mavlink app：通常参数是由地面站软件设置的。地面站软件与Mavlink app通信，查询并修改参数。

4、all apps：访问参数。

通常，当你要添加一个自定义参数时，你需要修改第1条对应的代码；如果你需要访问该参数，你当然需要修改第4条对应的代码；然而，修改第2条和第3条对应的代码是非必须的。

## 蜂鸣器驱动(Tone Alarm Driver)

参考链接：<http://pixhawk.org/dev/tone_alarm>

蜂鸣器驱动可以使用蜂鸣器来播放各种频率的声音，这些声音以QBasic PLAY/ANSI格式存储。

默认的一些音调被存储在蜂鸣器驱动文件中（src/drivers/stm32/tone\_alarm），可以通过如下命令进行播放任何声音：

tone\_alarm MFT225O3L8GL8GL8GL2E-P8L8FL8FL8FMLL2DL2DMNP8

注意：为了使得tone\_alarm程序识别，传递给该app的声音序列需要以“M”开头。有些ANSI格式的声音文件在其头部和尾部包含跳出序列，这个需要移除，否则会导致出错。

默认的声音的含义如下：

|  |  |  |
| --- | --- | --- |
| **Index** | **Short name** | **Description** |
| 0 | STOP | Stops the current tune |
| 1 | BOOT | 'PX4 startup sound'. Is auto-played by the tone alarm driver on system startup |
| 2 | BATT\_LOW | The battery is low, but not fully empty. Running on reserve. |
| 3 | BATT\_EMPTY | The battery is completely flat, the system is not able to maintain safe flight. |
| 4 | CMD\_ACK | The last command has been accepted |
| 5 | CMD\_REJ | The last command has been rejected |
| 6 | POS\_LOCK | The position estimator obtained a valid position (emitted once on first lock) |
| 7 | ARMING | Emitted on the transition to armed |
| 8 | DISARMING | Emitted on the transition to disarmed |
| 9 | EMERGENCY | General, potentially fatal error. NOT due to battery. Land immediately. |

注：常用的生成ANSI格式声音文件的软件为Melody Master。