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("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：