```
#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 */
staticint 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.
 */
staticvoidusage(constchar*reason);

staticvoid
usage(constchar*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(argv[1],"start")){

                if(thread_running){
                        warnx("daemon already running\n");
                        /* this is not an error */
                        exit(0);
                }

                thread_should_exit =false;
                daemon_task = task_spawn_cmd("daemon",
                                             SCHED_DEFAULT,
                                             SCHED_PRIORITY_DEFAULT,
                                            2000,
                                             px4_daemon_thread_main,

        (argv)?(constchar**)&argv[2]:(constchar**)NULL);
                exit(0);
        }

        if(!strcmp(argv[1],"stop")){
                thread_should_exit =true;
                exit(0);
        }

        if(!strcmp(argv[1],"status")){
                if(thread_running){
                        warnx("\trunning\n");
                }else{
                        warnx("\tnot started\n");
                }
                exit(0);
        }
```

```
        usage("unrecognized command");
        exit(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;

        return0;
    }
```

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。

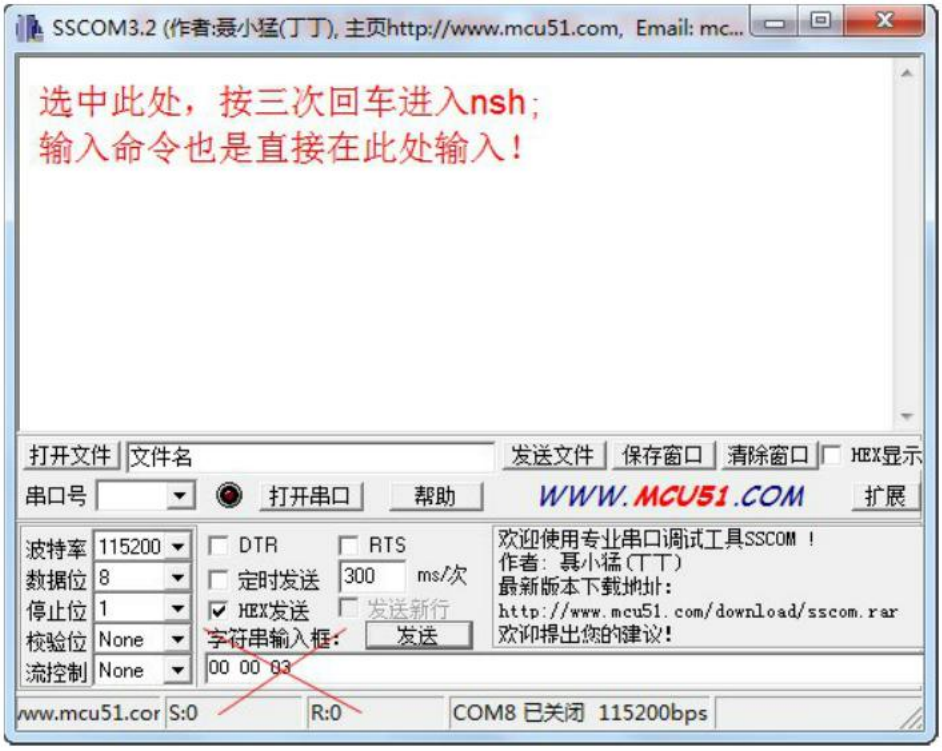
nsh（NuttShell）

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（？）.....