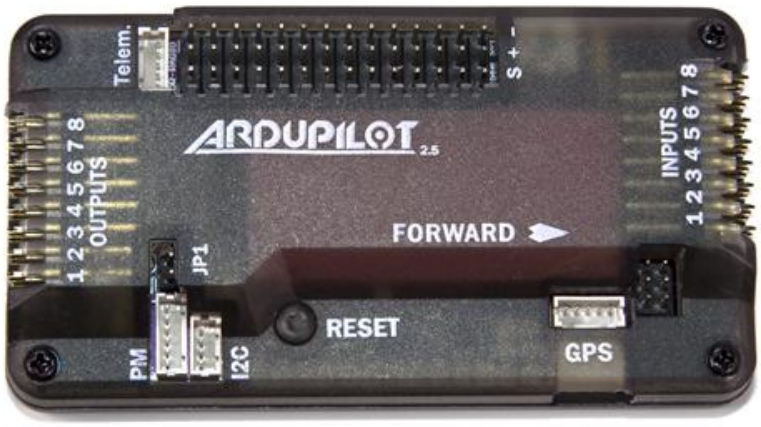
Pixhawk 原始固件 PX4 之常用函数解读

PX4Firmware

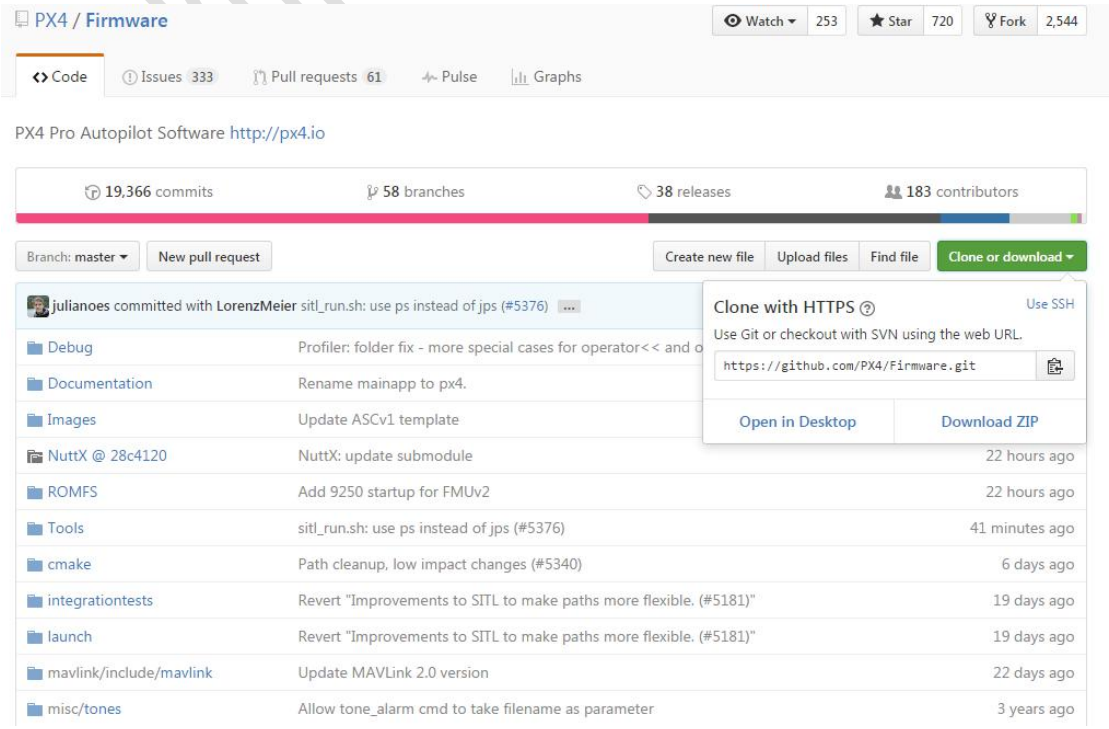
经常有人将 Pixhawk、PX4、APM 还有 ArduPilot 弄混。这里首先还是简要说明一下：

Pixhawk 是飞控**硬件**平台，PX4 和 ArduPilot 都是开源的可以烧写到 Pixhawk 飞控中的自驾仪**软件**，PX4 称为**原生固件**，专为 Pixhawk 打造。APM（Ardupilot Mega）早期也是一款自驾仪硬件，到 APM3.0 版本，这款基于 Arduino Mega 的自驾仪已经走到了它的终点。ArduPilot 早期是 APM 自驾仪的固件，Pixhawk 作为 APM 的升级版，也兼容 ArduPilot 固件，APM 自驾仪卒了之后，ArduPilot 现在全面支持 Pixhawk，现在大家亲切的**称 ArduPilot 固件为 APM**。

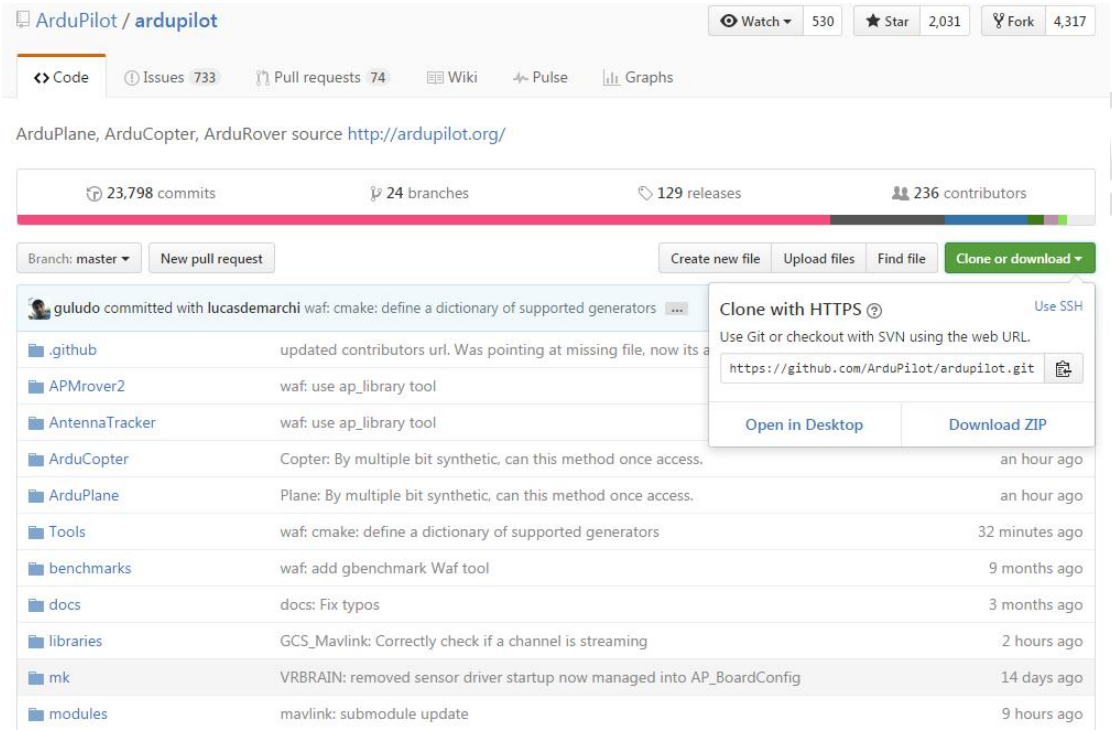
Pixhawk


Telem.
S + -
ARDUPILOT 2.5
INPUTS 1 2 3 4 5 6 7 8
OUTPUTS 1 2 3 4 5 6 7 8
FORWARD
JP1
RESET
PM
I2C
GPS

APM 2.5


PX4 / Firmware
Watch 253
Star 720
Fork 2,544
Code
Issues 333
Pull requests 61
Pulse
Graphs
PX4 Pro Autopilot Software http://px4.io
19,366 commits
58 branches
38 releases
183 contributors
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
julianoes committed with LorenzMeier sitl_run.sh: use ps instead of jps (#5376)
Clone with HTTPS
Use SSH
Use Git or checkout with SVN using the web URL.
https://github.com/PX4/Firmware.git
Open in Desktop
Download ZIP
Debug
Profiler: folder fix - more special cases for operator<< and o
Documentation
Rename mainapp to px4.
Images
Update ASCv1 template
NuttX @ 28c4120
NuttX: update submodule
22 hours ago
ROMFS
Add 9250 startup for FMUv2
22 hours ago
Tools
sitl_run.sh: use ps instead of jps (#5376)
41 minutes ago
cmake
Path cleanup, low impact changes (#5340)
6 days ago
integrationtests
Revert "Improvements to SITL to make paths more flexible. (#5181)"
19 days ago
launch
Revert "Improvements to SITL to make paths more flexible. (#5181)"
19 days ago
mavlink/include/mavlink
Update MAVLink 2.0 version
22 days ago
misc/tones
Allow tone_alarm cmd to take filename as parameter
3 years ago

PX4

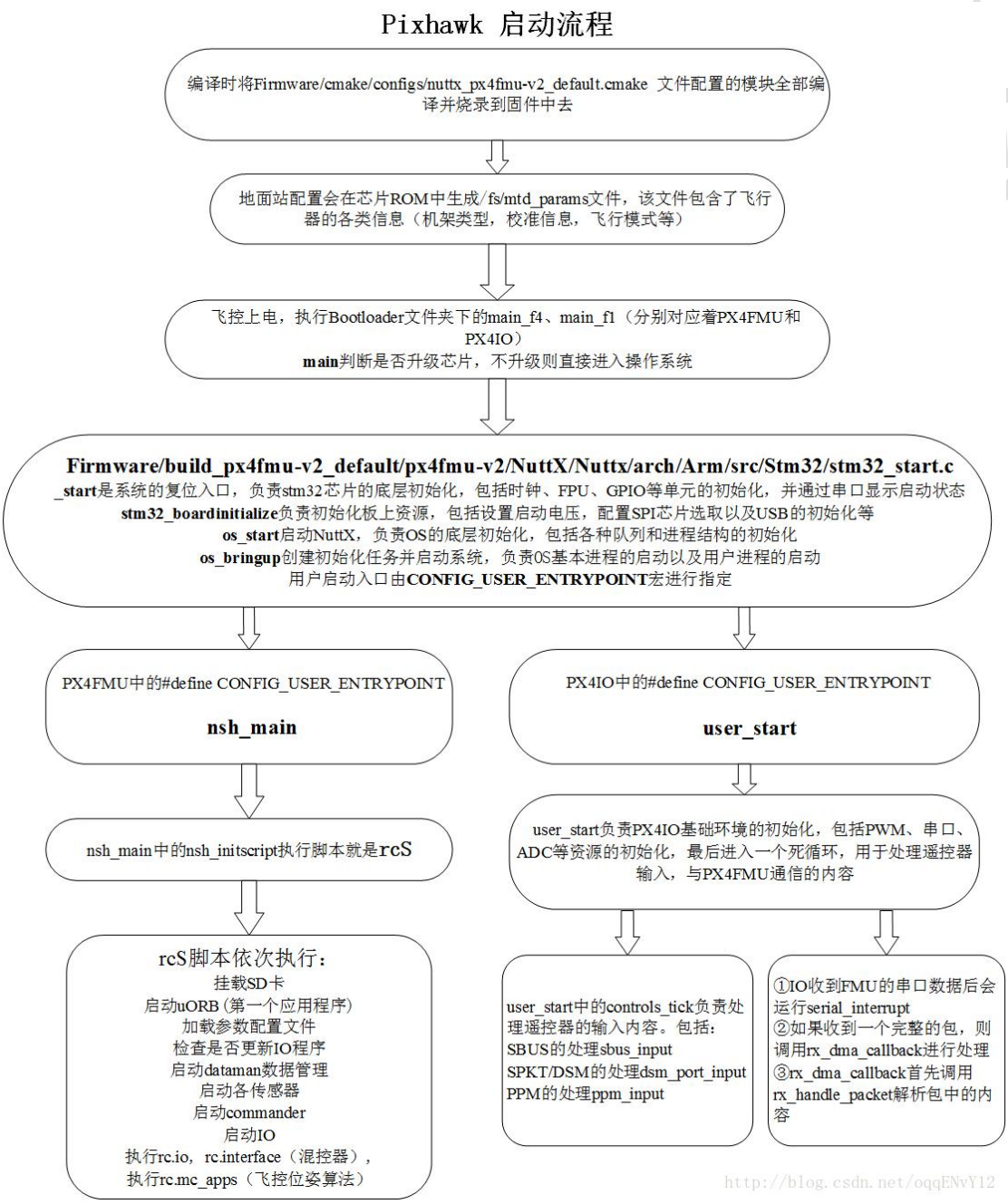


ArduPilot

笔者一直使用的是 Pixhawk 飞控，研究 PX4Firmware。用 Source Insight 看代码是极好的。

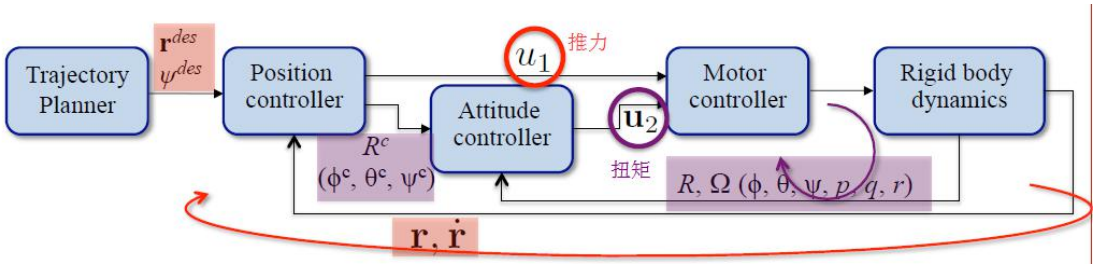
PX4 固件主要是用 C++语言编写，真是学好 C++，走遍天下都不怕。使用了 NuttX 实时操作系统，整体软件架构不可谓不庞大。

不知道如何开头，复述一个人工智能和机器人领域著名的莫拉维克悖论：和传统假设不同，对计算机而言，实现逻辑推理等人类高级智慧只需要相对很少的计算能力，而实现感知、运动等低等级智慧却需要巨大的计算资源。且从系统说起吧。



地面站配置的文件应该在芯片 flash 中，格式化 SD 卡同时擦除芯片后配置信息依然存在。

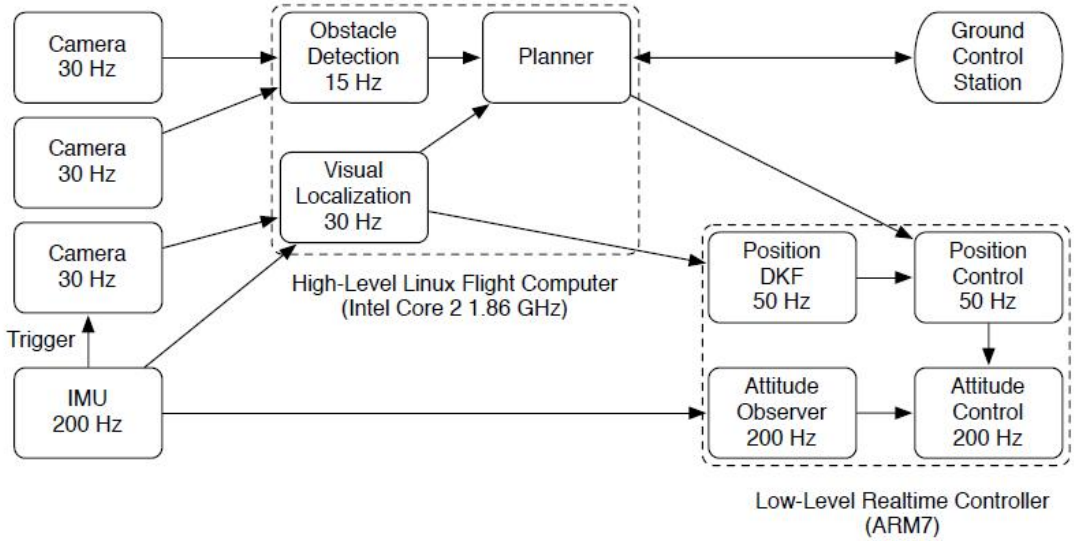
RTFSC



Pixhawk 整体逻辑大致为：

- commander 和 navigator 产生期望位置
- position_estimator 估计当前位置
- 通过 pos_ctrl 产生期望姿态
- attitude_estimator 估计当前姿态
- 通过 att_estimator 产生 PWM 数值
- 最后通过 mixer 和 motor_driver 控制电机

一直都还只是停留在底层，什么时候能感受一下 ETHz 这帮人的成果呢，这才是 pixhawk 啊。



启动函数

Pixhawk 是没有 main 函数的，飞控上电后，会自动执行 Firmware/ROMFS/px4fmu_common/init.d 文件夹下的 **rcS** 启动脚本（startup script）。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有 app(任务软件 ，包括位置和姿态估计，控制遥测等)。所有属于自启动程序的脚本文件可以在 init.d 文件夹中找到。

uORB 是 Pixhawk 系统中非常重要且关键的一个模块，它肩负了整数据传输任务，所有的感器、 数据传输任务、 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 PX4 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。

rcS 启动顺序

extern “C” __EXPORT int main(int agrc, char *agrv[ ]);

argc 和 argv 是 main 函数的形参，它们是程序的“命令行参数”。agrc(argument count 的缩写，意思是参数个数)，argv(argument vector 的缩写，意思是参数向量)，它是一个*char 指针数组，数组中每一个元素指向命令行中的一个字符串。
main 函数是操作系统调用的，实参只能由操作系统给出。在操作命令状态下，实参是和执行文件的命令一起给出的。例如在 DOS、UNIX 或 **Linux** 等系统的操作命令状态下，在命令行中包括了命令名和需要传给 main 函数的参数。

命令行的一般形式为：

1 命令名 参数 1 参数 2 …… 参数 n

命令名和各参数之间用空格分隔。命令名是可执行文件名(此文件包含 main 函数)。

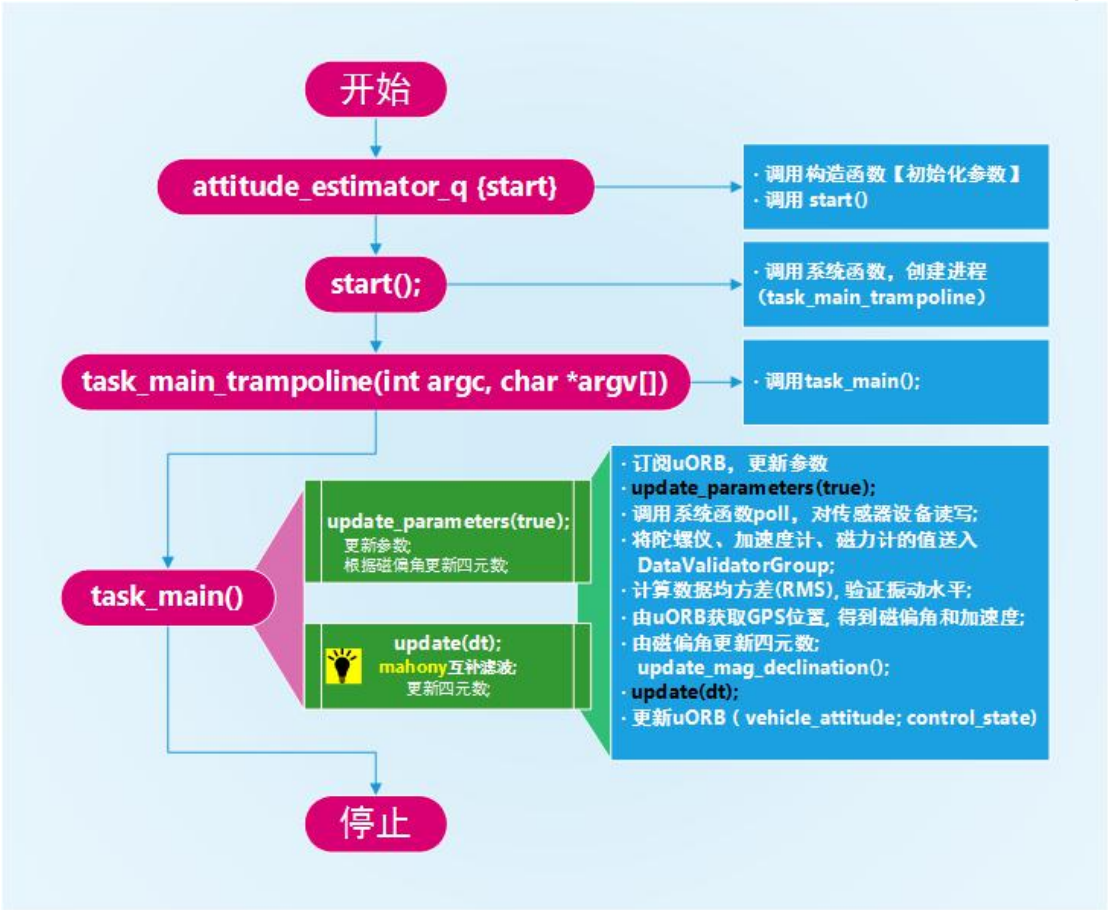
在 rcS 执行的时候，比如 attitude_estimator_q_main start 那么 agrc 就等于 2，agrv[0]就是 attitude_estimator_q_main 这个字符串，argv[1]就是 start。 所以要判断 agrv[1]是 start 还是 stop。就像你在 dos 命令行里输入 attitude_estimator_q start 自然就给 agrc 和 agrv[]赋值。NuttX 系统下的模块的主函数名字都是以”_main”开始的，但是调用的时候不加“_main”。

不管了，就地举个栗子，还是 attitude_estimator_q_main.cpp 这个文件。

```
1  extern "C" _EXPORT int attitude_estimator_q_main(int argc, char *argv[]);
2
3  ……
4
5  int attitude_estimator_q_main(int argc, char *argv[])
6
7  {
8
9  if (argc < 1) {
10    warnx("usage: attitude_estimator_q {start|stop|status}");
11    return 1;
12 }
13
14 if (!strcmp(argv[1], "start")) {
15    if (attitude_estimator_q::instance != nullptr) {
16        warnx("already running");
17        return 1;
18     }
19
20    attitude_estimator_q::instance = new AttitudeEstimatorQ;
21
22    if (attitude_estimator_q::instance == nullptr) {
23       warnx("alloc failed");
24       return 1;
25     }
26
27    if (OK != attitude_estimator_q::instance->start()) {
28        delete attitude_estimator_q::instance;
29        attitude_estimator_q::instance = nullptr;
30        warnx("start failed");
31        return 1;
32    }
33
```

```
34          return 0;
35  }
36
37  if (!strcmp(argv[1], "stop")) {
38      if (attitude_estimator_q::instance == nullptr) {
39          warnx("not running");
40          return 1;
41      }
42
43      delete attitude_estimator_q::instance;
44      attitude_estimator_q::instance = nullptr;
45      return 0;
46  }
47
48  if (!strcmp(argv[1], "status")) {
49      if (attitude_estimator_q::instance) {
50          attitude_estimator_q::instance->print();
51          warnx("running");
52          return 0;
53
54      } else {
55          warnx("not running");
56          return 1;
57      }
58  }
59
60  warnx("unrecognized command");
61  return 1;
62  }
```

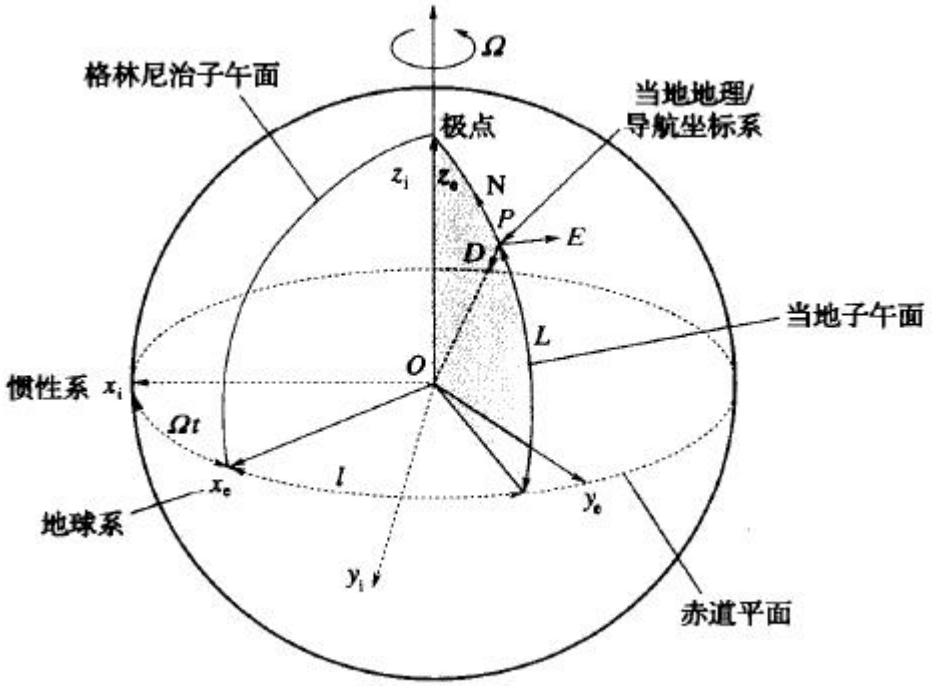
在一系列头文件之后，这里 **extern “C”**告诉编译器在编译 attitude_estimator_q_main 这个函数时按照 C 的规则去翻译相关的函数名而不是 C++的； **__EXPORT** 表示将函数名输出到链接器(Linker)。然后跳转到函数的定义部分 int attitude_estimator_q_main(int argc, char *argv[])，判断系统给出的命令行的参数，一系列的判断，C++在大型项目上的优势这里有没有发挥出来！总之你要的是 start 就对了。和 attitude_estimator_q 相似，一个正常的应用程序启动如下图所示，直接 task_main( ) 吧：



坐标系

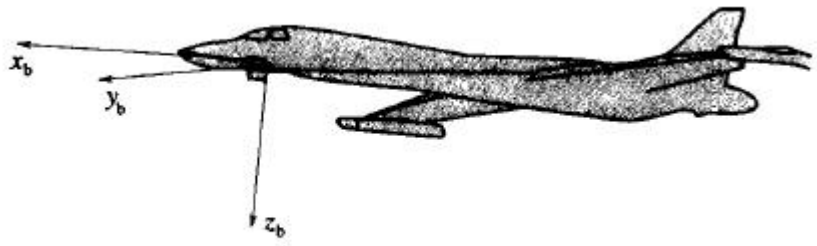
惯性导航的基础是精确定义一系列的笛卡儿参考坐标系，每一个坐标系都是**正交的右手坐标系**或轴系。

对地球上进行的导航，所定义的坐标系要将惯导系统的测量值与地球的主要方向联系起来。也就是说，当在近地面导航时，该坐标系具有实际意义。因此，习惯上将原点位于地球中心、相对于恒星固定的坐标系定义为惯性参考坐标系，下图给出了用于陆地导航的固连于地球的参考坐标系和当地地理导航坐标系以及惯性参考坐标系。



参考坐标系

- 地球坐标系（e 系）。原点位于地球中心，坐标轴与地球固连，轴向定义为 0xe, 0ye, 0ze 。其中，0ze 沿地球极轴方向，0xe 轴沿格林尼泊子午面和地球赤道平面的交线。地球坐标系相对于惯性坐标系绕 0zi 轴以角速度 Ω 转动。
- 导航坐标系(n 系)。是一种当地地理坐标系，原点位于导航系统所处的位置 P 点，坐标轴指向北、东和当地垂线方向(向下)。导航坐标系相对于地球固连坐标系的旋转角速率 wen 取决于 P 点相对于地球的运动，通常称为转移速率。
- 载体坐标系(b 系)。一个正交坐标系，轴向分别沿安装有导航系统的运载体的横滚轴、俯仰轴和偏航轴。



载体坐标系

在 PX4 中，

Local position setpoint in NED frame → 导航坐标系(以起飞点 home 为原点)北东地 xyz

Global position setpoint in NED frame → 由位置估计器发布的位置信息，融合了包括 GPS 原始信息的多个传感器数据

GPS position in WGS84 coordinates → 为 GPS 全球定位系统使用而建立的坐标系统（原始 GPS 信息）

NED earth-fixed frame →~~个人觉得是 GPS 投影到地面的坐标系,原点？~~

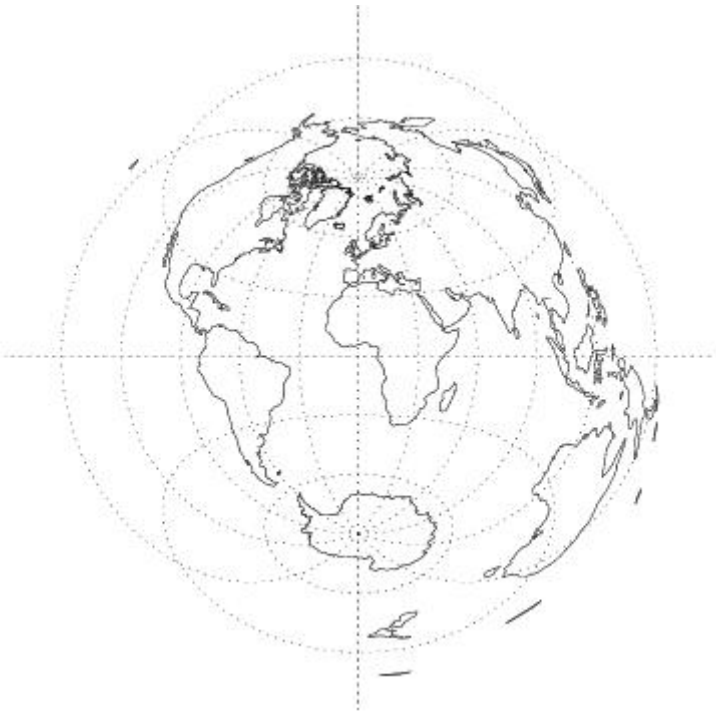
NED body-fixed frame →机体坐标系，x 轴正方向为机头，z 轴正方向下

这里有两个函数不得不提：

```
//将地理学坐标系(geographic coordinate system)中的点(球)投影到本地方位等距平面(XOY)中
int map_projection_project(const struct map_projection_reference_s *ref, double lat, double lon, float *x,float *y)
  {
    if (!map_projection_initialized(ref)) {
        return -1;
    }

    double lat_rad = lat * M_DEG_TO_RAD; // 度 -> 弧度    A/57.295
    double lon_rad = lon * M_DEG_TO_RAD; // GPS数据角度单位为弧度

    double sin_lat = sin(lat_rad); //程序中三角运算使用的是弧度
    double cos_lat = cos(lat_rad);
    double cos_d_lon = cos(lon_rad - ref->lon_rad);

    double arg = ref->sin_lat * sin_lat + ref->cos_lat * cos_lat * cos_d_lon;

    if (arg > 1.0) {
        arg = 1.0;

    } else if (arg < -1.0) {
        arg = -1.0;  //限幅
    }

    double c = acos(arg);
    double k = (fabs(c) < DBL_EPSILON) ? 1.0 : (c / sin(c));// c为正数

    *x = k * (ref->cos_lat * sin_lat - ref->sin_lat * cos_lat * cos_d_lon) * CONSTANTS_RADIUS_OF_EARTH;
    *y = k * cos_lat * sin(lon_rad - ref->lon_rad) * CONSTANTS_RADIUS_OF_EARTH;

    return 0;
}
```

将球面坐标转化为平面坐标的过程便称为投影。这里将经纬度转换成地坐标系 xy 值,也就是说是基于 GPS 的位置自动控制 。

采用的是等距方位投影的方法(Azimuthal Equidistant Projection)。



方位投影既不是等面积也不是保形的。让φ1 和λ0 作为投影中心的纬度和经度，则变换方程由下式给出

$$x=k'\cos\phi\sin(\lambda-\lambda_0)$$

$$y=k'[\cos\phi_1\sin\phi-\sin\phi_1\cos\phi\sin(\lambda-\lambda_0)]$$

这里

$$k'=c/\sin c$$

并且

$$\cos c=\sin\phi_1\sin\phi+\cos\phi_1\cos\phi\cos(\lambda-\lambda_0)]c$$

在这里代表距中心的角距离(一定点到两物体之间所量度的夹角)。公式的逆表达如下：

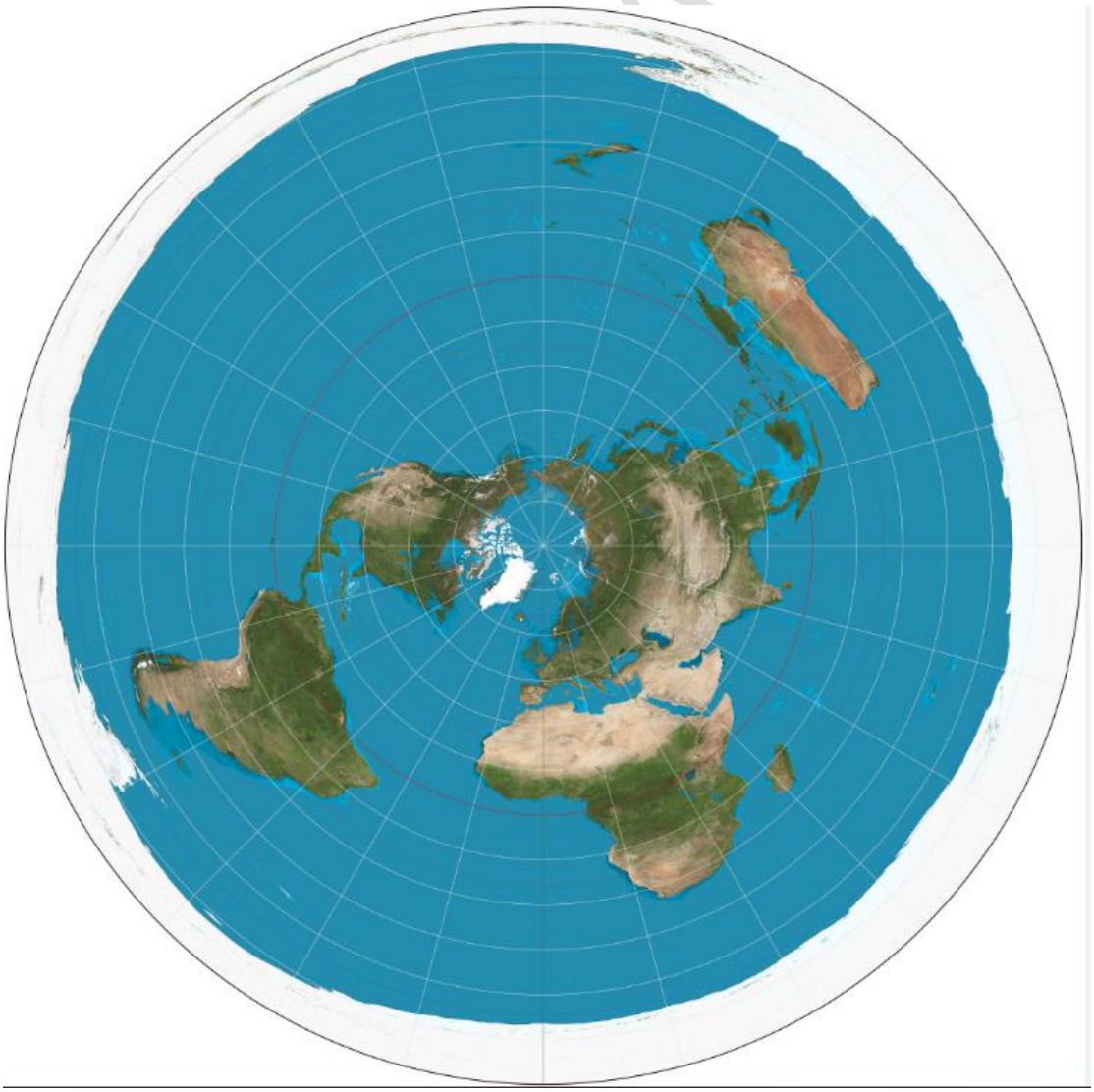
$$\phi = \sin^{-1}(\cos c \sin\phi_1 + \frac{y \sin c \cos\phi_1}{c})$$

以及

$$\lambda = \begin{cases} \lambda_0 + \tan^{-1}\left(\dfrac{x\sin c}{c\cos\phi_1\cos c - y\sin\phi_1\sin c}\right), & \text{for}\ \phi_1 \neq \pm 90^\circ \\ \lambda_0 + \tan^{-1}\left(-\dfrac{x}{y}\right), & \text{for}\ \phi_1 = 90^\circ \\ \lambda_0 + \tan^{-1}\left(\dfrac{x}{y}\right), & \text{for}\ \phi_1 = -90^\circ \end{cases}$$

到中心的角距离由下式给出：

$$c = \sqrt{x^2 + y^2}$$



```
//将本地方位等距平面中的点投影到地理学坐标系
int map_projection_global_reproject(float x, float y, double *lat, double *lon)
{
    return map_projection_reproject(&mp_ref, x, y, lat, lon);
}

__EXPORT int map_projection_reproject(const struct map_projection_reference_s *ref, float x, float y, double *lat,
                      double *lon)
{
    if (!map_projection_initialized(ref)) {
        return -1;
    }

    double x_rad = x / CONSTANTS_RADIUS_OF_EARTH; // 地球半径
    double y_rad = y / CONSTANTS_RADIUS_OF_EARTH;
    double c = sqrtf(x_rad * x_rad + y_rad * y_rad);
    double sin_c = sin(c);
    double cos_c = cos(c);

    double lat_rad;
    double lon_rad;

    if (fabs(c) > DBL_EPSILON) {
        lat_rad = asin(cos_c * ref->sin_lat + (x_rad * sin_c * ref->cos_lat) / c);
        lon_rad = (ref->lon_rad + atan2(y_rad * sin_c, c * ref->cos_lat * cos_c - x_rad * ref->sin_lat * sin_c));

    } else {
        lat_rad = ref->lat_rad;
        lon_rad = ref->lon_rad;
    }

    *lat = lat_rad * 180.0 / M_PI; // 弧度 -> 度
    *lon = lon_rad * 180.0 / M_PI;

    return 0;
}
```

先 map_projection_reproject( )再 map_projection_project( )。这种方式将位置转换为经纬度和高度， 然后用位置估计参数来更新经纬度和高度，接着转换回位置参考点，属于 GPS 数据转换的方式。

poll

```
1    int poll(struct pollfd fds[], nfds_t nfds, int timeout)
```

功能：监控文件描述符（多个）；

说明：timemout=0,poll()函数立即返回而不阻塞；timeout=INFTIM(-1),poll()会一直阻塞下去，直到检测到 return > 0；

参数：

fds:struct pollfd 结构类型的数组；

nfds:用于标记数组 fds 中的结构体元素的总数量；

timeout:是 poll 函数调用阻塞的时间，单位：毫秒；

返回值：

>0：数组 fds 中准备好读、写或出错状态的那些 socket 描述符的总数量；

==0:poll()函数会阻塞 timeout 所指定的毫秒时间长度之后返回;

-1:poll 函数调用失败；同时会自动设置全局变量 errno；

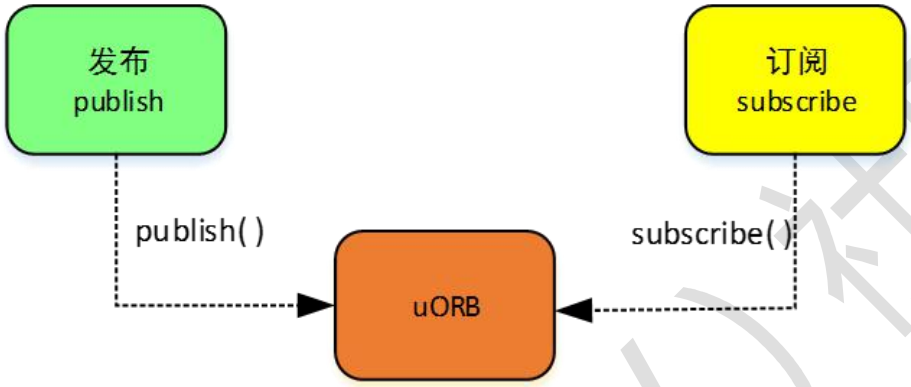
poll( )函数用于监测多个等待事件，若事件未发生，进程睡眠，放弃 CPU 控制权。若监测的任何一个事件发生，poll 函数将唤醒睡眠的进程，并判断是什么等待事件发生，并执行相应的操作。poll( )函数退出后，struct polldf 变量的所有值被清零，需要重新设置。

uORB

uORB(Micro Object Request Broker,微对象请求代理器)是 PX4/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。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 使用的是 NuttX 实时 ARM 系统，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。进程通过命名的「总线」交换的消息称之为「主题」(topic)，在 Pixhawk 中，一个主题仅包含一种消息类型，通俗点就是数据类型。每个进程可以「订阅」或者「发布」主题，可以存在多个发布者，或者一个进程可以订阅多个主题，但是一条总线上始终只有一条消息。

应用层中操作基础飞行的应用之间都是隔离的，这样提供了一种安保模式，以确保基础操作独立的高级别系统状态的稳定性。而沟通它们的就是 uORB。



源码中关于 uORB 有几个常见的函数：

- 公告主题

在数据被发布到一个主题前，它必须被公告，发布者可以使用下面的 API 来公告一个新的主题

```
1    extern int orb_advertise(const struct orb_metadata *meta, const void *d
     ata);
```

参数：

meta:uORB 元对象，可以认为是主题 id，一般是通过 ORB_ID(主题名)来赋值；

data:指向一个已被初始化，发布者要发布的数据存储变量的指针；

返回值：

错误则返回 ERROR;成功则返回一个可以发布主题的句柄；如果待发布的主题没有定义或声明则会返回-1，然后会将 errno 赋值为 ENOENT；

公告也可以发布初始化数据到主题，meta 参数是传递给 API 的一个指针，指向由 ORB_DEFINE()宏定义好的数据，通常使用 ORB_ID()宏来根据主题名称获取该指针。请注意，虽然主题更新可以从中断处理函数发布，公告主题必须在常规的线程上下文中执行。

- 发布更新

一旦公告了一个主题，公告主题后返回的句柄可使用下面的 API 来发布主题更新。

```
1  extern int orb_publish(const struct orb_metadata *meta, int handle, cons
   t void *data);
```

参数：

meta:uORB 元对象，可以认为是主题 id，一般是通过 ORB_ID(主题名)来赋值;

handle:orb_advertise 函数返回的句柄；

data:指向待发布数据的指针；

返回值：

OK 表示成功；错误返回 ERROR；否则则有根据的去设置 errno;

uORB 不会缓存多个更新，当用户检查一个主题，他们将只能看到最新的更新。

- 订阅主题

订阅主题的要求如下：

- 调用 ORB_DEFINE()或 ORB_DEFINE_OPTIONAL()宏（在订阅者的头文件中包含他们）
- 发布到主题的数据结构定义（通常与发布者使用同一头文件）

如果满足上面的条件后，订阅者可以使用下面的 api 来订阅一个主题：

```
1  extern int orb_subscribe(const struct orb_metadata *meta);
```

参数：

meta:uORB 元对象，可以认为是主题 id，一般是通过 ORB_ID(主题名)来赋值；

返回值：

错误则返回 ERROR;成功则返回一个可以读取数据、更新话题的句柄；如果待订阅的主题没有定义或声明则会返回-1，然后会将 errno 赋值为 ENOENT;

即使订阅的主题没有被公告，但是也能订阅成功；但是在这种情况下，却得不到数据，直到主题被公告；如果可选主题不存在于固件之中，订阅到可选的主题将会失败，但其他主题即便发布者没有进行公告也会订阅成功，这样可大大降低系统对启动顺序的安排。

- 取消订阅

要取消订阅一个主题，可以用下面的 API

```
1  extern int orb_unsubscribe(int handle);
```

- 拷贝数据

订阅者不能引用 ORB 中存储的数据或其他订阅共享的数据，而是在订阅者请求时从 ORB 拷贝数据到订阅者的临时缓冲区。副本拷贝的方式可以避免锁定 ORB 的问题，并保持两者之间（发布者，订阅者）的 API 接口简单。它也允许订阅者在必要的时候直接修改拷贝副本的数据供自己使用。从订阅的主题中获取数据并将数据保存到 buffer 中。

当订阅者想要把主题中的最新数据拷贝一份全新的副本，可以使用：

```
1  extern int orb_copy(const struct orb_metadata *meta, int handle, void *
   buffer);
```

参数：

meta:uORB 元对象，可以认为是主题 id，一般是通过 ORB_ID(主题名)来赋值;

handle:订阅主题返回的句柄；

buffer:从主题中获取的数据；

返回值：

返回 OK 表示获取数据成功，错误返回 ERROR;否则则有根据的去设置 errno;

拷贝是以原子操作进行的，所以可以保证获取到发布者最新的数据。

- 检查更新

订阅者可以使用下面的 API 来检查一个主题在发布者最后更新后，有没有人调用过 orb_copy 来接收，处理：

```
1    extern int orb_check(int handle, bool *updated);
```

参数：

handle:主题句柄；

updated:如果当最后一次更新的数据被获取了，检测到并设置 updated 为 ture;

返回值：

OK 表示检测成功；错误返回 ERROR;否则则有根据的去设置 errno;

如果主题在被公告前就有人订阅，那么这个 API 将返回“not-updated”直到主题被公告。

- 发布时间戳

订阅者可以使用下面的 API 来检查一个主题最后发布的时间：

```
1    extern int orb_stat(int handle, uint64_t *time);
```

参数：

handle:主题句柄；

*time:时间指针；

返回值：

主题最后的发布时间。

需要注意的是，要小心的使用这个调用，因为不能保证再调用返回后不久主题就不会被发布（调用返回后不久，主题可能马上又被发布，导致最后更新时间错误）

注意：

1. 必须有了 orb_advertise( )函数和 orb_subscribe( )函数(通过它们可以获取 orb_copy( )函数中的参数 handle 句柄)之后才可以使用 orb_copy( )函数来获取订阅的主题数据。
2. 源码中常用的调用顺序为：orb_subscribe( ) - > orb_check( ) -> orb_copy( )

Topics

所有的主题都可以在 Firmware/Build_px4fmu-v2_default/Src/Modules/uORB/Topics 文件夹下找到。对于 xxx_sub = orb_subscribe(ORB_ID(xxx))这类语句，很多时候 Source Insight 无法跳转到这些主题（**ORB_ID**）定义的地方，可以去 Topics 中查看。

或者也可以从 Msg 中查看这些结构体成员的具体使用情况，消息在 Firmware/Msg 下都可以找到。

Daemon

守护进程 daemon 是运行在后台的进程。

在 NuttX 中守护进程是一个任务，在 POSIX(Linux/Mac OS)中是一个线程

```
daemon_task = px4_task_spawn_cmd("commander",
                  SCHED_DEFAULT,
                  SCHED_PRIORITY_DEFAULT + 40,
                  3600,
                  commander_thread_main,
                  (char * const *)&argv[0]);
```

以下是参数：

- arg0: 进程名 commander
- arg1: 调度类型（RR or FIFO）the scheduling type (RR or FIFO)
- arg2: 调度优先级
- arg3: 新进程或线程堆栈大小
- arg4: 任务/线程主函数
- arg5: 一个 void 指针传递给新任务,在这种情况下是命令行参数

在 Unix 和其他多任务操作系统中 daemon 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要被用户或者 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。

后台应用程序只是暂时存在用与开始后台作业，在 MakeFile 中指定的堆栈大小仅适用于这个管理任务。实际的堆栈大小应在 task_create( )调用中设置。

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

数据操作

- **memset**

```
1   void *memset(void *s, int c, size_t n)
```

参数：

dest 目标指针；

src 数据源；

n 数据长短

功能：

作用是在一段内存块中填充某个给定的值，将 s 中当前位置后面的 n 个字节 （typedef unsigned int size_t ）用 c 替换并返回 s

例子：

```
1  memset(&actuator_controls, 0, sizeof(actuator_controls));
```

表示将 actuator_controls 的各位置 0

- **memcpy**

```
1  FAR void *memcpy(FAR void *dest, FAR const void *src, size_t n)
2  {
3  FAR unsigned char pout = (FAR unsigned char)dest;
4  FAR unsigned char pin  = (FAR unsigned char)src;
5  while (n-- > 0) *pout++ = *pin++;
6  return dest;
7  }
```

参数：

dest 目标指针；

src 数据源；

n 数据长短

功能：

从源 src 所指的内存地址的起始位置开始拷贝 n 个字节到目标 dest 所指的内存地址的起始位置中。

例子：

```
1  memcpy(val, v, param_size(param));
```

表示将 v 中的前 n 个值复制到 val 中

- **param_find**

```
uintptr_t param_find(const char *name)   //unsigned int pointer
{
return param_find_internal(name, false);
}

uintptr_t param_find_internal(const char *name, bool notification)
{
uintptr_t param;
/* perform a linear search of the known parameters */
  // 对已知参数执行线性搜索
for (param = 0; handle_in_range(param); param++) {
    if (!strcmp(param_info_base[param].name, name)) {
        if (notification) {
            param_set_used_internal(param);
        }
        return param;
    }
}
/* not found 未找到对应的参数*/
return PARAM_INVALID;  // 参数无效
}
```

参数：

*name 指针名

功能：

返回指针指向的值

例子：

```
param_t _param_system_id = param_find("MAV_SYS_ID");
// 其中
PARAM_DEFINE_INT32(MAV_SYS_ID, 1);
```

表示将 MAV_SYS_ID 所代表的值 1 赋给_param_system_id

- **param_get**

```
int param_get(param_t param, void *val)
{
int result = -1;
param_lock(); // lock和unlock主要就是通过sem信号量控制对某一数据的互斥访问
const void *v = param_get_value_ptr(param);
if (val != NULL) {
    memcpy(val, v, param_size(param));
    result = 0;
}
param_unlock();
return result;
}
```

- **参数：**

  param 源参数；

  val 目标参数指针

  功能：获取 param 的地址并赋给 val

  例子：

  1 param_get(_param_system_id, &(status.system_id));

  表示将_param_system_id 的地址写入 status.system_id 中 param_find 与 param_get 是一起使用的，先 find 定义的值再 get 到目标指针变量中

nsh

关于 terateam 中的 nsh 乱码问题 先拔内存卡

依然乱码

修改掉 rcS

具体做法改为

- 新固件 (line766)

```
1    # Start MAVLink
2    mavlink start -r 800000 -d /dev/ttyACM0 -m config -x
```

按老固件方法加一个判断

```
1    # Start USB shell if no microSD present, MAVLink else
2    if [ $LOG_FILE == /dev/null ]
3    then
4        # Try to get an USB console
5        nshterm /dev/ttyACM0 &
6    else
7        mavlink start -r 800000 -d /dev/ttyACM0 -m config -x
8    fi
```

进而 make 并 upload

掉高

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执行器的值，根据比例钳位（Ratio Clamping）的原理，各个执行器间大小的比例不变，从而防止出现侧翻的情况。当然，这样会导致掉高情况的出现。 以上是对 pixhawk 掉高的解释，在 mixer 中进行了说明，位置是 Firmware/Romfs/Px4fmu_common/Mixers

任何钳位都是根据输入的比例进行的，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，从而淡出各个电机点的比例将会得到满足而不会饱和，保证了飞行器不会出现侧翻的情况。

现举例说明四旋翼的钳位：

输入（四个电机），限制为 100：

150 75 75 75

钳位结果：

100 50 50 50

本例中，进行钳位后，电机 1 的转速依然是另外三个的两倍，所以目标姿态会是正确的。相比于没有进行钳位的情况，飞行器将不会增加高度。如果值是对电机 1 进行了限制（输出 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。

查看 Firmware 版本

- Console / shell

```
#../px4/Firmware

git describe --always --tags
```

- 将输出 Git clone 下来的固件版本

```
v1.5.2-661-gc2db188
```

- NSH

```
nsh>ver all
```

将输出

```
HW arch: PX4FMU_V2
FW git-hash: c2db1886ac12475677ef691a84a62975c162e8ae
FW version: 1.5.2 0 (17105408)
OS: NuttX
OS version: Release 1.8.0 (17301759)
Build datetime: Jan 16 2017 19:30:23
Build uri: localhost
Toolchain: GNU GCC, 5.4.1 20160609 (release) [ARM/embedded-5-branch revision 237715]
MCU: STM32F42x, rev. 1
UID: 40002D:31345116:36383835
```

可知当前固件版本为 **v1.5.2**

文章可能有诸多不知出处的引用，侵删。

有错误的地方，提就改。