阶段汇报

2016.4

系统结构

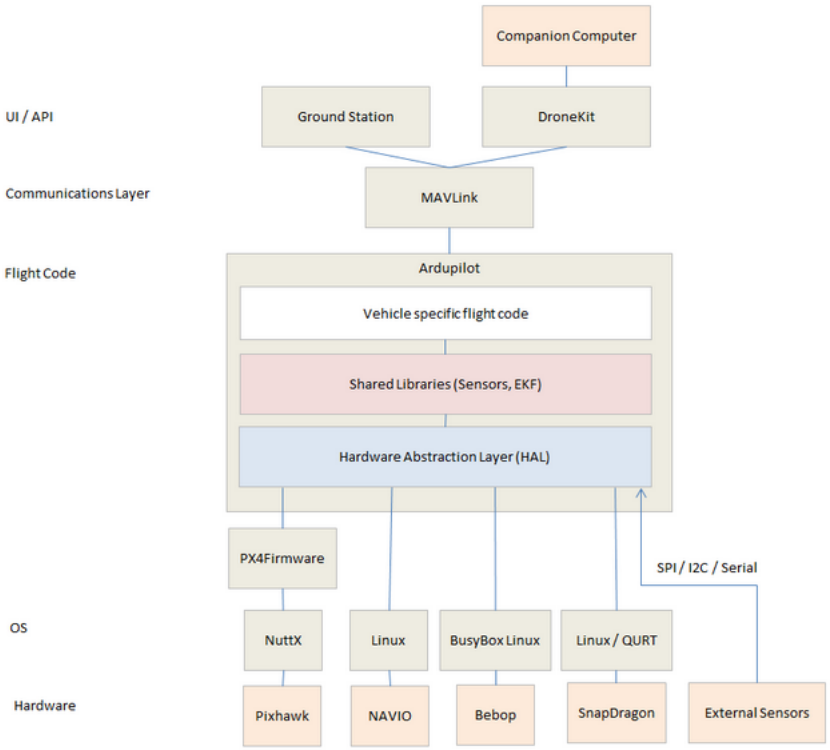

图 1: 系统结构图

主要组成:

- ArduCopter
- libraries
- HAL(AP_HAL & AP_HAL_PX4)
- 外部模块（PX4Firmware，NuttX，uavcan，mavlink）

串口定义

包含 USB 接口在内，pixhawk 共有 6 个串口。具体定义如下表:

表 1: 串口定义

序号	UART 定义	外部定义	默认 tty 设备	波特率
0	UARTA	USB	/dev/ttyACM0	115200
1	UARTC	telem1	/dev/ttyS1	38400
2	UARTD	telem2	/dev/ttyS2	57600
3	UARTB	1st GPS	/dev/ttyS3	57600
4	UARTE	serial4(2nd GPS)	/dev/ttyS6	57600
5	UARTF	serial5	/dev/null	57600

其中，serial5 是底层操作系统 nuttx 默认的控制台 (Console)，打印系统启动信息，并可作为 NSH 终端供手动控制操作系统中程序的运行。serial5 通过 USB-TTL 连接 linux 终端时对应的设备为**/dev/ttyUSB0**。

USB 口是 ArduPilot 程序的默认控制台，在 ArduPilot 程序运行最初用于打印信息，之后以 Mavlink 协议进行数据传输，通过 USB 线或数传与地面站通信。某些情况下（如不插入 *SD* 卡启动），*USB* 口也会被作为 *NSH* 终端使用。

注意：**ArduPilot 只是 nuttx 操作系统上运行的程序之一**，因此两个控制台的工作层面是不同的。对两个串口的查看和操作，在 windows 系统下，可使用任意串口程序或 PX4 Toolchain 自带程序 TeraTerm；在 Linux 系统下，可用的串口工具有 screen、minicom 等，如可分别使用命令：**screen /dev/ttyACM0 115200 8N1** 和 **screen /dev/ttyUSB0 57600 8N1** 来打开 USB 口和 serial5。

telem1&2 支持硬件流控制 (RTS/CTS 管脚)(主程序启动成功后被禁用)。

主程序编译方式

主要命令：

make px4-v2 为 pixhawk 硬件编译固件；

make px4-v2-upload 编译固件并刷写；

make px4-clean 清理编译产生的部分文件；

make px4-cleandef 清理编译中使用的依赖文件 (*.d)；

make sitl 编译 SITL 软件。

固件启动流程

系统的引导由 nsh 脚本文件控制。脚本文件最初位于 ardupilot 源程序的*/mk/PX4/ROMFS/init.d* 文件夹中，编译时被存入固件，最后刷写固件时存

入 pixhawk 的 Flash 中 (*/etc/init.d*)。此文件夹中包含 3 个文件: **rcS**,**rc.APM** 和 **rc.error**。

1.pixhawk 启动时，自动运行 rcS，LED 灯启动并为白色:

```
    # show startup white
    rgbled rgb 16 16 16
```

之后寻找 microSD 卡，找到则蜂鸣器发声:

```
    tone_alarm /etc/tones/startup
```

否则，LED 灯变为红色:

```
    rgbled rgb 16 0 0
```

如果找到了 MicroSD 卡且对应位置有 rc 脚本文件，则运行脚本。

2. 检查是否连接了 USB。如果没有连接，则调用同目录下的 rc.APM 文件:

```
    # if APM startup is successful then nsh will exit
    sh /etc/init.d/rc.APM
```

否则，将 USB 口设置为 nsh 终端:

```
    nshterm /dev/ttyACM0 &
```

3.rc.APM 的执行: 首先进行硬件模块和各传感器的启动。完成后，开始运行 ArduPilot 主程序:

```
echo Starting ArduPilot $deviceA $deviceC $deviceD
if ArduPilot -d $deviceA -d2 $deviceC -d3 $deviceD start
then
    echo ArduPilot started OK
else
    sh /etc/init.d/rc.error
fi
```

4. 如果脚本运行过程中出错，则调用 rc.error，LED 灯变为红色，USB 口被设置为 NSH 终端。

```
    nshterm /dev/ttyACM0 &
```

ArduPilot 启动流程

1. 程序入口在主程序文件最后的宏定义。

对于 ArduCopter:

```
    AP_HAL_MAIN_CALLBACKS(&copter);
```

对于其他测试例程:

```
    AP_HAL_MAIN();
```

宏定义位于文件*/libraries/AP_HAL/AP_HAL_Main.h* 中：

```
#define AP_HAL_MAIN() extern "C" { \
    int AP_MAIN(void); \
    int AP_MAIN(void) { \
        AP_HAL::HAL::FunCallbacks callbacks(setup, loop);\
        hal.run(0, NULL, &callbacks); \
        return 0; \
    } \
}

#define AP_HAL_MAIN_CALLBACKS(CALLBACKS) extern "C" { \
    int AP_MAIN(void); \
    int AP_MAIN(void) { \
        hal.run(0, NULL, CALLBACKS); \
        return 0; \
    } \
}
```

即程序的真实起点为

```
hal.run(0, NULL, &callbacks);
```

2. 要运行 run() 函数，首先程序需要获得 hal：

```
const AP_HAL::HAL& hal = AP_HAL::get_HAL();
```

(例子程序的此语句位于主程序文件开头，ArduCopter 的此语句位于 Copter.cpp 而非主程序文件)。对于 pixhawk，hal 对应的类型为 AP_HAL_PX4。函数 get_HAL() 在 AP_HAL 的每个子类中都有定义，在编译时，系统使用了条件编译：

```
#if CONFIG_HAL_BOARD == HAL_BOARD_PX4
```

从而只编译了 PX4 的一个子类 AP_HAL_PX4，AP_HAL::get_HAL() 的实现为：

```
const AP_HAL::HAL& AP_HAL::get_HAL() {
    static const HAL_Empty hal;
    return hal;
}
```

而全局搜索宏 CONFIG_HAL_BOARD，并没有找到其定义。实际上，这个宏定义是通过 makefile 中的-D 选项实现的：

```
    SKETCHFLAGS=$(SKETCHLIBINCLUDES) ……
        -DCONFIG_HAL_BOARD=HAL_BOARD_PX4 ……
```

此语句存在于 mk/PX4_target.mk，其调用逻辑为：
(1) 编译时所在目录的 makefile 调用/mk/apm.mk；
(2)/mk/apm.mk 调用/mk/environ.mk；
(3)/mk/environ.mk 利用变量 MAKECMDGOALS(即输入 make 命令时的参数，如 px4-v2)，将变量 HAL_BOARD 定义为 HAL_BOARD_PX4;
(4)/mk/apm.mk 根据变量 HAL_BOARD 调用/mk/board_px4.mk；
(5)/mk/board_px4.mk 调用/mk/px4_target.mk。

AP_HAL_PX4 类的 run() 函数位于文件 HAL_PX4_Class.cpp：

```
void HAL_PX4::run(int argc, char * const argv[], Callbacks*
    callbacks) const
```

函数的输入参数 (argv[ ]) 为脚本文件 rc.APM 调用 ArduPilot 时的参数：

```
ArduPilot -d $deviceA -d2 $deviceC -d3 $deviceD start
```

3.run() 函数通过创建 nuttx 操作系统任务的方式 (px4_task_spawn_cmd())，建立一个守护进程 (daemon_task)，用于运行 main_loop() 函数：

```
daemon_task = px4_task_spawn_cmd(SKETCHNAME,
                                 SCHED_FIFO,
                                 APM_MAIN_PRIORITY,
                                 APM_MAIN_THREAD_STACK_SIZE,
                                 main_loop,
                                 NULL);
```

在 nsh 中可通过 ps 命令显示操作系统上运行的进程，此守护进程的名字为所编译程序的名字 (SKETCHNAME，如 ArduCopter(),UART_test())，优先级 (PRI) 为 180。

4.main_loop() 函数首先为 ArduPilot 进行一系列初始化设置：

```
    hal.uartA->begin(115200);
    hal.uartB->begin(38400);
    hal.uartC->begin(57600);
    hal.uartD->begin(57600);
    hal.uartE->begin(57600);
    hal.scheduler->init();
    hal.rcin->init();
    hal.rcout->init();
    hal.analogin->init();
    hal.gpio->init();
```

其中需特别注意 scheduler 的初始化。在 HAL_PX4_Class.cpp 中发现，init() 函数创建了 4 个新的线程：

```
// setup the timer thread - this will call tasks at 1kHz
```

```
2  pthread_create(&_timer_thread_ctx, &thread_attr, &PX4Scheduler
       ::_timer_thread, this);
3  // the UART thread runs at a medium priority
4  pthread_create(&_uart_thread_ctx, &thread_attr, &PX4Scheduler::
       _uart_thread, this);
5  // the IO thread runs at lower priority
6  pthread_create(&_io_thread_ctx, &thread_attr, &PX4Scheduler::
       _io_thread, this);
7  // the storage thread runs at just above IO priority
8  pthread_create(&_storage_thread_ctx, &thread_attr, &
       PX4Scheduler::_storage_thread, this);
```

这些线程分别进行计时器、UART、IO 和存储的操作。在 NSH 终端利用 ps 命令，打印出各线程的具体信息，如下表：

表 2: 操作系统进程列表

PID	PRI	SCHD	TYPE	NP	STATE	NAME
0	0	FIFO	TASK		READY	Idle Task()
1	192	FIFO	KTHREAD		WAITSEM	hpwork()
2	50	FIFO	KTHREAD		WAITSIG	lpwork()
3	100	FIFO	TASK		RUNNING	init()
10	240	FIFO	TASK		WAITSEM	px4io()
43	180	FIFO	TASK		WAITSEM	ArduCopter()
44	181	FIFO	PTHREAD		WAITSEM	<pthread>(20003b70)
45	60	FIFO	PTHREAD		WAITSEM	<pthread>(20003b70)
46	58	FIFO	PTHREAD		WAITSEM	<pthread>(20003b70)
47	59	FIFO	PTHREAD		WAITSEM	<pthread>(20003b70)

最后 4 个线程即 hal.scheduler->int() 创建的线程，其 PID 是连续的，因为它们由守护进程 (PID=43) 依次创建。其名字为 <pthread>(XXX)，XXX 是 hal.scheduler 的首地址。多线程的运用，可以更好地调度执行速度慢的任务 (如读写 MicroSD 卡) 而不影响核心飞控程序的运行。

之后，降低守护进程的优先级并调用主程序的 setup() 函数：

```
1  /*
2    run setup() at low priority to ensure CLI doesn't hang the
3    system, and to allow initial sensor read loops to run
4   */
5  hal_px4_set_priority(APM_STARTUP_PRIORITY);
6
7  schedulerInstance.hal_initialized();
```

```
8
9  g_callbacks->setup();
10 hal.scheduler->system_initialized();
11
12 perf_counter_t perf_loop = perf_alloc(PC_ELAPSED, "APM_loop");
13 perf_counter_t perf_overrun = perf_alloc(PC_COUNT, "APM_overrun
       ");
14 struct hrt_call loop_overtime_call;
15
16 thread_running = true;
```

最后，恢复守护进程的优先级，循环调用 loop() 函数：

```
1  /*
2    switch to high priority for main loop
3   */
4  hal_px4_set_priority(APM_MAIN_PRIORITY);
5
6  while (!_px4_thread_should_exit) {
7      ... ...
8      g_callbacks->loop();
9      ... ...
10
11     /*
12       give up 250 microseconds of time, to ensure drivers get a
13       chance to run. This relies on the accurate semaphore wait
14       using hrt in semaphore.cpp
15      */
16     hal.scheduler->delay_microseconds(250);
17 }
```

常用的 hal 函数

- hal.console(或 uartA...E)->printf(),print(),println()
- hal.scheduler->millis(),micros(): 系统启动以来的时间
- hal.scheduler->delay(),delay_microseconds(): 休眠
- hal.gpio->pinMode(),read(),write()
- hal.i2c

- hal.spi

详细的接口函数参考硬件抽象层函数库 AP_HAL_PX4。

PX4 系统控制台

前面已经提到，可通过两种方式操作系统控制台：直接使用串口 5；或拔出 SD 卡，连接 USB。进入控制台，界面显示 nsh>，此时可进行命令行操作。输入 help 可显示所有命令和刷入的程序 (Builtin Apps)。常用命令包括：ls 列出文件结构（不支持输入参数）；ps 列出正在运行的进程。内嵌程序包括各硬件模块的驱动程序和主程序 ArduPilot，可对各传感器进行启动 (start)、停止 (stop) 和测试 (test) 等操作。

2016.4.25
方酉