系统启动App过程

1. 系统编译规则文件的结构

Px4程序是以自己定义的Makefile组织编译程序以获取能够供Cpu工作的二进制文件(\*.bin)。具体的Makefile组织结构如下（我们采用从上层到底层的方式加载）：

1 总体Makefile: 其出现在firmware/; 它包括了程序编译的整体过程，包括make命令的伪命令(all, upload, clean,distclean等)所需执行的命令。

2 功能性makefile: 其出现在firmware/makefile文件夹下，其中包括相应的功能性makefile(如模块加载的module.mk,系统编译的nuttx.mk,固件编译规则firmware.mk,以及相关的配规则congfig\_\*.mk,子系统编译规则board\_\*.mk)。在这些文件中实现目标文件的编译。

3 模块级Makefile: 这些文件出现在firmware/src/driver 和firmware/src/module之中，主要包含模块的命令名称（MODULE\_COMMAND）,源文件的相对地址（SRCS），这两个是必须要的。以及需要分配的堆栈大小以及优先级等。注意这并不是所有的模块都需要的，在firmaware.Mk 中已经定义了一个默认的值。只有在需要修改的时候才需要进行定义。文件结构如下所示：

Firmware----

|

|--makefile.mk

|——makefiles

|-----board\_\*mk

|-----config\_\*.mk

|-----firmware.mk

|-----module.mk

|-----nuttx.mk

|-----upload.mk

|---- -等\* .Mk

|--- src

|----divers

|----相关的驱动文件夹中的module.mk

|----modules

|----相关的功能模块的文件夹中的module.mk

1. 编译过程及生成的文件

通过上述的文件结构可以看出整个文件编译的过程也是这样层层深入的，在外层的Makefile中调用内层的makefile,同时，通过全局变量传递相关的地址及命令。 其所生成的文件在firmware/build文件夹下。该文件夹的内容是在编译后生成的。其包含每个子系统生成.bin文件所需要的所有源文件及中间生成的文件。该文件的结构如下：

---Build

|--子系统.build

|---子系统.biuld

|---builtin\_commands: 生成NSH命令的临时文件

|---C : 包含所有的源程序，是从Firmware中的相关文件复制而来。需要复制那些文件由config\_.\*mk中所包含的modul决定。

|---nuttx-export: 该文件应该是系统编译的相关文件及中间的代码，包括NSH等系统的操作。

|--romfs\_scratch:包括子系统的.bin文件和相关的初始化系统的nsh文件以及MIXERS文件。

|--biutin\_commands.c : 其定义了相关模块的\*\_main函数的外部定义以及命令结构体，从Bultin\_command文件夹中的相关文件中获取。

|--Firmware.Bin：固件的二进制文件。

|- 其他

编译过程：1 在eclipse软件的make target 窗口下创建相关的make命令。All --- build all , archieves---biuld nuttx, clean ----清除编译文件(删除Build文件夹)。Upload ----下载相关的文件。

2 将相关源文件复制到C文件夹下

3 生成相关的命令biutin\_commands文件。

4 生成buitin\_commands.c文件。

1. 生成相关相关的.bin 文件。

3.加载自己的APP

1 加载自己的App只需要在相应的/driver 和/ module文件夹中添加需要的模块文件夹。

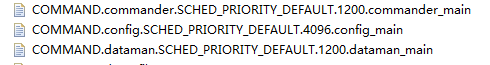
2 在上述文件夹中添加相关的源程序文件，和该模块的module.mk。在module.mk中添加相关的 module\_command 和 srcs 及相关配置命令。

3 在源文件中添加 module\_command\_main() 函数。

4 在对应的 /makefiles/config\_\*.mk文件中添加 MODULES += \*/module.mk

5 编译相关模块即可。

6 检查builtin\_commands文件加下是否有类似如下文件名的文件。



7 检查buitin\_commands.c文件中是否有类似如下的外部文件定义



8 调用nsh 输入模块名称按回车启动相关命令进行相关调试

系统串口调试

本部分旨在实现在PX4基础上调试其串口通信方实现收发功能。再在此基础上实现MTI驱动程序的书写。

1 串口的实现

定义UART设备口，定义文件地址“dev/test\_uart”,定义波特率。

2 驱动的实现

消息的发布

1. 简介

PX4程序流程中关于消息的发布并非完全采用nuttx操作系统的信息机制，而是在nuttx信息机制的基础上，创建了一种新的信息收发机制，作者命名为UORB，采用类似于报文收发的方式，有UORB向系统全局广播相应的数据，需要使用相关数据的子系统，可直接获取。其原程序在SRC/module/UORB下。

1. 消息的发布流程

1 创建一个subscribe: 调用orb\_subscribe(ORB\_ID( topic name))

2 设置信息发送间隔 orb\_set\_interval(创建的subscribe,time(ms));

3 创建初始topic 用于数据的发布： orb\_advertise( ORB\_ID() , 数据指针（void \*data）)

4 创建poll的文件指针，用于等待消息。

5 调用poll()函数将当前定义的文件指针挂载到等待队列中。函数原型：

int poll(FAR struct pollfd \*fds, nfds\_t nfds, int timeout);

6 在时间到达后没有数据更新，返回值为：0如果出现错误返回值小于零，否则大于零。同时对fds中的相关状态值进行修改。

7 根据POLL的状态，采用orb\_copy（）函数将 subscribe topic中的内容复制相应的变量中。函数原型: int

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

buffer:指向需要存储数据的变量。

Handle: 创建的subscrible topic

Meta : handle 对应的ORB\_ID,采用ORB\_ID(命名)宏定义创建。

8 数据的发布采用orb\_publish(ORB\_ID(vehicle\_attitude), att\_pub, &att);函数实现。