Cal Poly UAV Pixhawk User Guide

INTRODUCTION

This guide is intended to help users quickly familiarize themselves with the Pixhawk. Most of the information found in this guide and more can be found documented at the Pixhawk website, pixhawk.org, but to speed up the learning process the most relevant information is compiled here.

1 INITIAL CONFIGURATION

1.1 BUILD ENVIRONMENT AND TOOLCHAIN INSTALLATION

1.1.1 Linux

The full installation guide can be found at: http://dev.px4.io/starting-installing-linux.html

The online guide is thorough and correct for the most part. The only issue is that you will likely end up installing the wrong version of GCC. If you reach the point where you're trying to compile and you run into the error "No rule to make target '<something-related-to-gcc>'" then it means it downloaded the wrong version. To fix this:

1. Delete the currently installed version. If you used "`sudo apt-get install gcc-arm-none-eabi`", then it's just "`sudo apt-get remove gcc-arm-none-eabi`".
2. Go to https://launchpad.net/gcc-arm-embedded/+download
3. Search for the 4.8-2014-q3-update release and download the linux tarball.
4. Open a terminal and navigate to where you'd like to install the toolchain (/usr/local/ is recommended).
5. Unpack the tarball using "`sudo tar xjf ~/Downloads/gcc-arm-none-eabi-4.8-2014q3-20140805-linux.tar.bz2`". Modify the path as necessary if using a different folder for downloads.
6. Add the toolchain to your PATH variable. Open /etc/environment and add to the front of the PATH variable: "`/usr/local/gcc-arm-none-eabi-4_8-2014q3/bin:`"
7. To verify functionality type "`arm-none-eabi-gcc --version`". It should output the version.
8. You're done.

1.1.2 Mac

The full installation guide can be found at: https://pixhawk.org/dev/toolchain_installation_mac

There haven't been any issues following the instructions posted.

1.1.3 Windows

The full installation guide can be found at: https://pixhawk.org/dev/toolchain_installation_win

There's been limited success with configuring Eclipse to work with the px4 environment. If setting up Eclipse is proving too troublesome, it's possible to forgo setting up Eclipse entirely. Alternative steps are

as follows (these aren't actual instructions, just what I did to get the toolchain installed on my computer):

1. Install MSysGit (Only download if you need a GUI, toolchain comes with bash Git)
2. Download and run the PX4 Toolchain Installer v14 for Windows (http://firmware.diydrones.com/Tools/PX4-tools/px4_toolchain_installer_v14_win.exe) There will be warnings about unverified publishers. It's safe to install the drivers. (Probably)
3. Run the PX4 Toolchan -> PX4 Software Download program that was installed. This will download all the necessary tools and source code to get started.
4. Open the PX4 Console and navigate the px4/ directory generated in the last step.
5. Delete the Firmware folder ("`rm -rf Firmware`")
6. Navigate into the Firmware folder
7. Clone our Firmware ("`git clone https://github.com/CalPolyUAV/Firmware.git`")
8. Initialize ("`git submodule init`") & update ("`git submodule update`") the submodules.
9. Build the Archives ("`make archives`"). This is a long process.

1.2 Building the Pixhawk software

More information can be found at: http://dev.px4.io/starting-building.html

The current build of the PX4 Firmware utilizes makefiles, so in order to compile the Firmware simply type "`make px4fmu-v2_default`" and to flash it to the Pixhawk type "`make upload p4xfmu-v2_default`".

1.2.1 Customizing the Makefile

More information can be found at: https://pixhawk.org/dev/px4_simple_app

In order to add custom modules to the Firmware, these files must be added to the makefile as well. Source code is added to the Firmware/src/modules/ folder and each process has its own folder. To demonstrate the template for adding files to the makefile we'll look at the bottle_drop process. Navigate into the folder and open the module.mk file. The MODULE_COMMAND line indicates the argument that will be used to run the process when inside the Nuttshell terminal. The SRCS line indicates all files within the folder that need to be compiled. Navigate to the Firmare/makefiles/ folder and open the config_px4fmu-v2_default.mk file. Any folders added to the Firmware/src/ directory need to be added in here following the standard syntax. Adding the necessary line in the config file and setting up the module.mk file is all that's necessary to add custom modules. Understanding the Firmware

1.2.2 Opening the Nuttshell Terminal

More information can be found at: https://pixhawk.org/users/serial_connection

The Nuttshell terminal (NSH) is used to interact with processes while the Pixhawk is running. Currently accessing NSH requires a usb connection, however it's documented as possible to access NSH via telemetry. Whenever accessing NSH it may be necessary to press enter a few times to get the prompt to show up.

Linux & Mac

Download minicom "`sudo apt-get install minicom`". Type "`minicom -D /dev/ttyACM0`" to access NSH.

Windows

Run TeraTerm, which was installed with the PX4 Toolchain, to access NSH

1.3 PIXHAWK STARTUP PROCEDURES

More information can be found at: https://pixhawk.org/dev/system_startup.

Upon startup the Pixhawk follows a series of scripts in order to initialize. The scripts can be located within the *Firmware/ROMFS/px4fmu_common/init.d/* folder. Within the folder are a number of specific, preconfigured startup scripts for a variety of vehicle types and geometries (files prefaced with a number) and the general startup scripts that will run on most systems (files prefaced with rc). The systems begins by running the rcS script which does basic checks on hardware availability and configuration variables to adjust the rest of the startup procedure accordingly and then runs the other startup scripts as necessary. Following the execution of the startup scripts it executes any additional scripts placed on the SDcard within the etc/ folder. Configuration Variables

While most of the variables are described here https://pixhawk.org/dev/system_startup#configuration_variables some of the descriptions could be described as lackluster.

The current Pixhawk configuration is a combination of the previous px4fmu and px4io modules. As such USE_IO should be set to yes but the OUTPUT_MODE should be set to fmu.

The PWM_RATE is typically set to 50Hz, however modern ESCs should be able to support, and it's recommended to use, 400MHz. If there are undiagnosed issues with the motors, try setting this value to 50.

The PWM_DISARMED rate in order to prevent motor beeping on startup. Without a pwm signal the motors will emit beeps indicating no power. Keep this value below the minimum threshold to prevent the motors from spinning.

The PWM_MIN and PWM_MAX values are meant to stay within the default pwm values that the motor can accept. The standard range is between 1000 and 2000.

1.3.1 Extras.txt

There is an additional startup file on the SDcard at etc/extras.txt. The file can be found in the git under CalPolyUAV/misc/extras.txt. This file manually sets the sensor calibrations, PID settings, and the required mixer. The 'Commander Stop' and 'Commander Start' lines are necessary because the Pixhawk won't recognize the calibrations otherwise. Normally the Commander waits for the user to run calibrations, however to speed up the development process previous calibration data was saved and is written to the Pixhawk after normal startup. The Commander has to be reset to recognize the calibration data.

1.3.2 Preflight Checks

The Commander app runs a number of preflight checks before allowing the system to arm itself. If the system consistently refuses to arm it may be necessary to debug the Preflight Checks. The preflight checks can be found in Firmware/src/modules/commander/PreflightCheck.cpp.

1.4 Process Communication

More information can be found at: https://pixhawk.org/dev/shared_object_communication

Processes on the Pixhawk communicate via the uORB (micro Object Request Broker) system. uORB utilizes a shared data system and a publisher-subscriber pattern. In order to send data to or receive data from another process, uORB uses topics to identify specific data busses that processes can subscribe and publish to. This is necessary because the Pixhawk runs the NuttX RTOS (Real-time operating system). Instead of being able to run processes simultaneously and using a system of interrupts and context switching, processes are run one without being able to interrupt one another. Information is passed by a process placing data in a shared space and raising a flag indicating new data is available for any process watching that particular topic. (This also means it's very easy to lock up the OS with a loop or by running heavy calculations.)

1.4.1 Subscribing

Subscribing to a topic allows the process to read information from the other processes either at will or following an information update. This is necessary in order to read vehicle state information, sensor information, or messages received via telemetry/RF communications.

```
// Create the struct for passing information along the vehicle status topic
struct vehicle_status_s vstatus;
memset(&vstatus, 0, sizeof(vstatus));

// Generate the file descriptor for identifying the desired topic
int vstatus_sub_fd = orb_subscribe(ORB_ID(vehicle_status));

// Pull the information from the topic into our struct
orb_copy(ORB_ID(vehicle_status), vstatus_sub_fd, &vstatus);

printf("Vehicle main state has changed: %d\n", vstatus.main_state);
```

The code above allows the process to check information as needed. However in the case that a process needs to react to new information a poll system can be setup.

```
// Create the struct for passing information along the vehicle status topic
struct vehicle_status_s vstatus;
memset(&vstatus, 0, sizeof(struct vehicle_status_s));

// Generate the file descriptor for identifying the desired topic
int vstatus_sub_fd = orb_subscribe(ORB_ID(vehicle_status));

// Setting up the struct for polling
// Waiting on multiple topics is a matter of adding more lines
struct pollfd fds[1] = {
        { .fd = vstatus_sub_fd,             .events = POLLIN },
/*      { .fd = other_fd,                   .events = POLLIN }, */
};

while(true) {
        // Wait for updates on 1 file descriptor every 1000ms
        int poll_ret = poll(fds, 1, 1000);

        if(poll_ret == 0) {
                // Timeout - No data updates
        }
        else if(poll_ret < 0)
                printf("Error! Check the error value: %d\n", poll_ret);
        }
        else {
                if(fds[0].revents & POLLIN) {
                        //Have new data

                        orb_copy(ORB_ID(vehicle_status), vstatus_sub_fd, &vstatus);
                        printf("Vehicle main state has changed: %d\n", vstatus.main_state);
                }
        }
}
```

1.4.2 Publishing

Publishing to a topic allows a process to send information to other processes as needed. This is necessary to send requests to change the flight mode or control the motors.

```
// Create the struct for passing information along the vehicle arming topic
struct actuator_armed_s arm;
memset(&arm, 0, sizeof(arm));

// Indicate to uORB that we'll be writing information to the bus
orb_advert_t actuator_pub_fd = orb_advertise(ORB_ID(actuator_armed), &arm);

// Fill the struct with information
arm.timestamp = hrt_absolute_time();
arm.ready_to_arm = true;
arm.armed = true;

// Write the information to the bus
orb_publish(ORB_ID(actuator_armed), arm_pub_fd, &arm);
```