Custom Autopilot Capabilities for a Quadcopter

Independent study EML6907 (May 2020)
under Dr.Tansel Yucelen and Stefan Ristevski
by Mrudit Trivedi
Laboratory for Autonomy, Control, Information and Systems at University of South Florida

Project synopsis:

The project is based on building a custom quadcopter, in which my study is focused on designing a part of the autopilot that will control the quadcopter and execute the commands sent by a user. This will allow for the implementation of a custom controller. The capability to implement custom controller will allow later, different theoretically proven control strategies to be tested and supported with experimental results.

Hardware implementation:

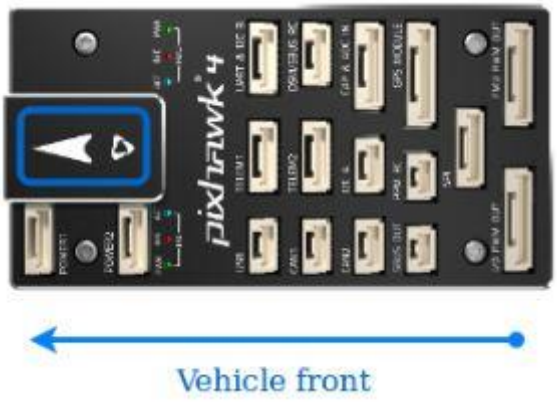


Fig: Holybro Pixhawk 4 flight controller with orientation direction (PX4 Autopilot Users Guide [2])

The custom quadcopter body frame has an F450 build with landing gears. On it, is mounted a Pixhawk 4 flight controller (manufactured by Holybro), a power management (PWM) board, powered by a set (3S) of Li-Po batteries and an ESP8266 wi-fi module. All four devices are stacked on top of each other and centrally aligned to keep the center of gravity symmetric (see image below). The power management board connects to four position sensors and actuator motors at each arm of the frame with their propellers. Input to the PWM board is 11.1V from the batteries and it converts to an output of 5V to all its connections.

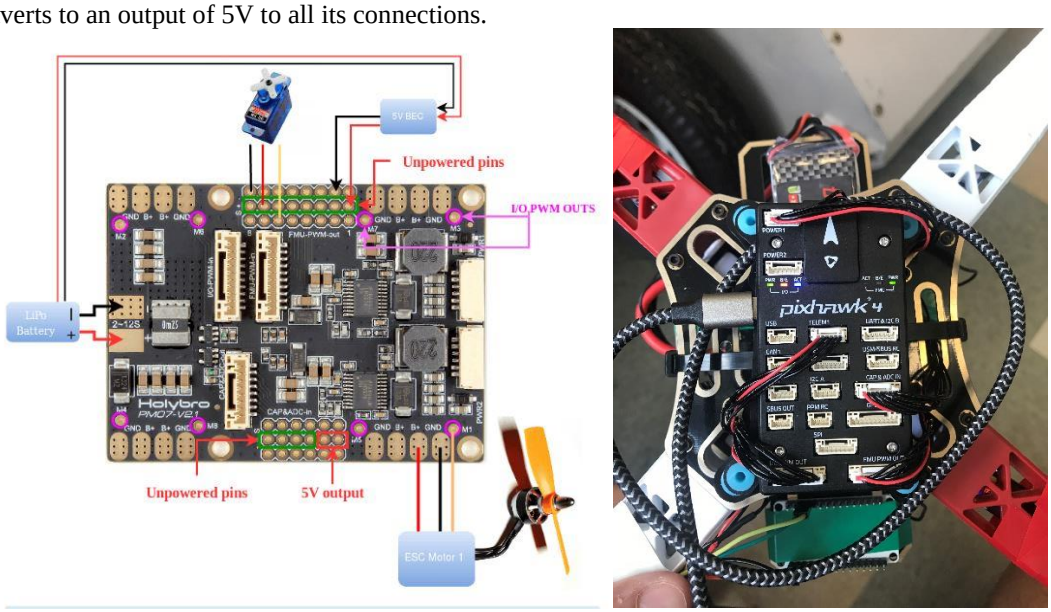


Fig: (Left) Power management board with its peripheral connections. (Right) Pixhawk, Batteries, Wi-fi module and PWM (not seen, at bottom) stacked on the quadrocopter frame. (PX4 Autopilot Users Guide [2]; LACIS at USF)

Software configuration and communication:

The following softwares and environments have been created and tested on both a Windows and Linux pc, which is still an on-going process as there have been certain configuration issues. The reason for testing on both operating systems is because the Pixhawk documentation and resources available online at the time was not comprehensive and certain port drivers are not correctly detected in Linux.

SOFTWARE / ENVIRONMENT	*WINDOWS 10 INSTALL*	*UBUNTU 18.1 INSTALL*	*REMARKS*
Cygwin toolchain	Yes	No	Environment for PX4
QGroundControl	Yes	Yes	Ground control system
Build for NuttX	Yes	Yes	`make px4_fmu-v5_default`
Px4 firmware 10.1	Yes	Yes	Clone from github
MAVLink Bridge	Yes	Yes	Build from shell using binaries file. Firmware v 1.2.2
Putty/Arduino IDE	Yes	Yes	For communication testing over wi-fi.
MATLAB 2019a	Yes	Yes	Designing custom control codes
Embedded coder support package for PX4 autopilot	Yes	Yes	Long install procedure. Requires PX4 firmware version downgraded to 1.8.0. Cannot install otherwise.
Ubuntu 16.04 distribution on WSL	Yes	No	Required for MATLAB PX4 package on windows.
Global protect vpn client	Yes	Yes	Required for MATLAB professional license.
Visual studio code	Yes	Yes	Developing and modifying source code

Table: Software and environment required to run PX4 source code effectively

Windows config:
Steps used in configuring on Windows 10

- Setting up a developer environment PX4 target – Installing the Cygwin toolchain that supports NuttX shell (https://dev.px4.io/master/en/setup/dev_env_windows_cygwin.html)

- Install QGroundControl (GUI application) (https://docs.qgroundcontrol.com/en/releases/daily_builds.html)

- Installing the source code (PX4 Firmware) – this is done with either connecting the Pixhawk via USB to QGroundControl (QGC - a ground communication software) or cloning the specific firmware version directly from PX4 GitHub. (https://dev.px4.io/master/en/setup/building_px4.html)

- Build the firmware for NuttX
```
cd Firmware
make px4_fmu-v5_default
```

- Uploading the Mavlink Bridge firmware on the ESP8266 wifi module and configuring the settings on the local ip (192.168.2.211). See complete configuration in the below image. (https://docs.px4.io/v1.9.0/en/telemetry/esp8266_wifi_module.html)

- PuTTY for testing message transfer over wi-fi, commonly called ‘heartbeat’. (https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html)

- Install MATLAB 2019a (https://www.mathworks.com/products/new_products/release2019a.html)

- The installation of PX4 toolchain requires that the Ubuntu 16.04 distribution on WSL is already completed on the host computer. (https://docs.microsoft.com/en-us/windows/wsl/install-manual)

- Install Embedded coder support package for PX4 Autopilots in MATLAB. Within the process it will ask for the PX4 firmware to be 1.8.0 version. Hence it is ideal to downgrade that using bash or QGC before

starting the installation. (https://www.mathworks.com/matlabcentral/fileexchange/70016-embedded-coder-support-package-for-px4-autopilots)

- Install PX4 toolchains – since above package requires installation of a development environment within Bash on Windows. This development environment is used to build firmware for all the Pixhawk Series flight controller boards. It will be asked when the package installation is initiated.

- Global Protect VPN client (used mainly to run encoder on MATLAB) (https://vpn.usf.edu/)

- Download and install Visual Studio Code for developing the source code (https://dev.px4.io/master/en/setup/vscode.html)

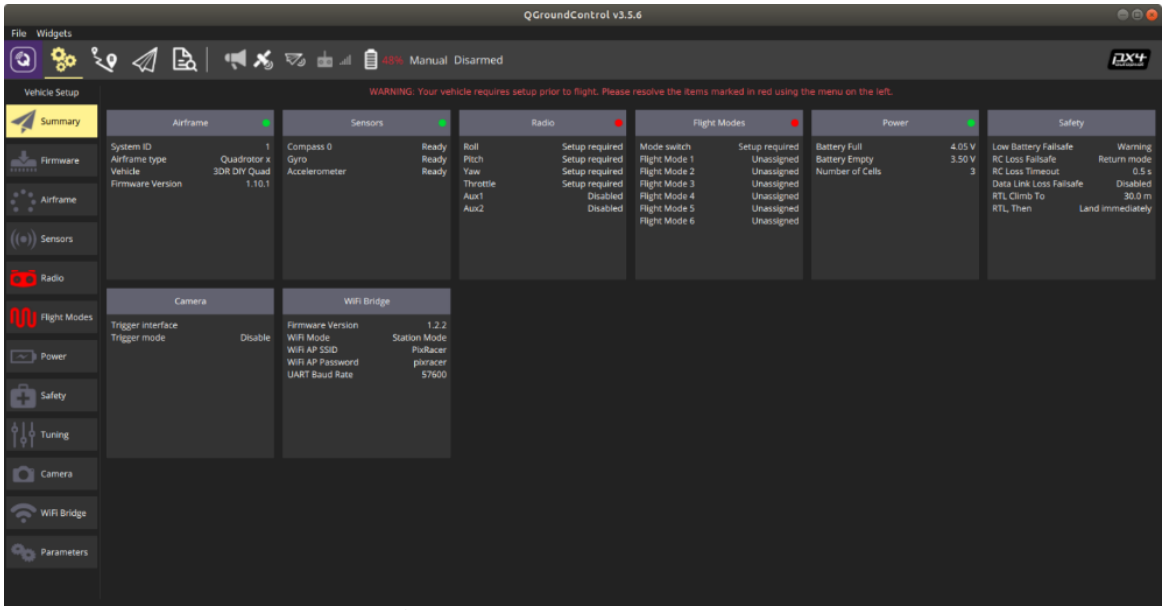


Fig: QGroundControl settings interface connected to Pixhawk 4 over wireless router and module. (LACIS at USF)

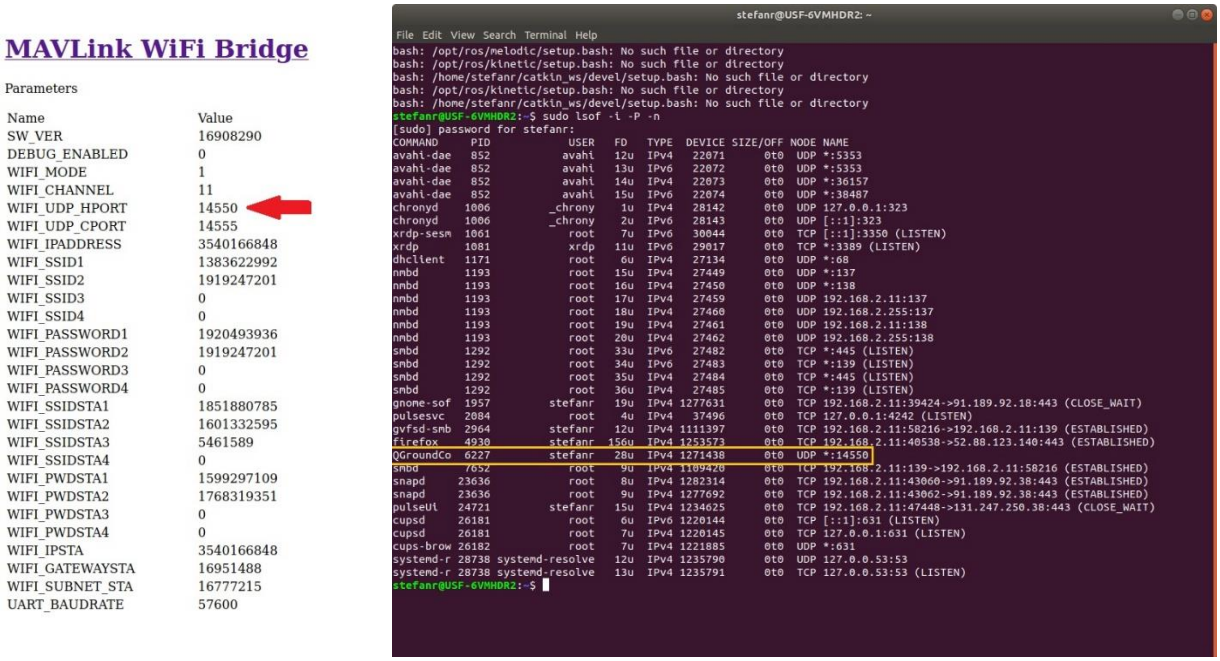


Fig: (Left) Parameters settings windows for MAVLink Wifi Bridge on local ip address 192.168.2.211. (Right) Terminal window on Ubuntu showing active connection of Pixhawk to the ground communication system. Notice that the UDP PORT which was set by the user in the MAVLink setting can be seen in the terminal. (LACIS at USF)

Linux config:
Steps used in configuring on Ubuntu 18.1

- Install QGroundControl (ground communication software) (https://docs.qgroundcontrol.com/en/releases/daily_builds.html)

- Install the source code (PX4 Firmware) via QGC or clone from GitHub (https://dev.px4.io/master/en/setup/building_px4.html)
```
git clone https://github.com/PX4/Firmware.git
```

- Run NuttX installation (https://dev.px4.io/master/en/setup/dev_env_linux_ubuntu.html)

- Build firmware for Nuttx
```
make px4_fmu-v5_default
```

- Uploading the Mavlink Bridge firmware on the ESP8266 wifi module and configuring the settings on the local ip (192.168.2.211). See complete configuration in the above image. (https://docs.px4.io/v1.9.0/en/telemetry/esp8266_wifi_module.html)

- PuTTY or Arduino IDE – for detecting 'heartbeat' (https://www.arduino.cc/en/main/software)

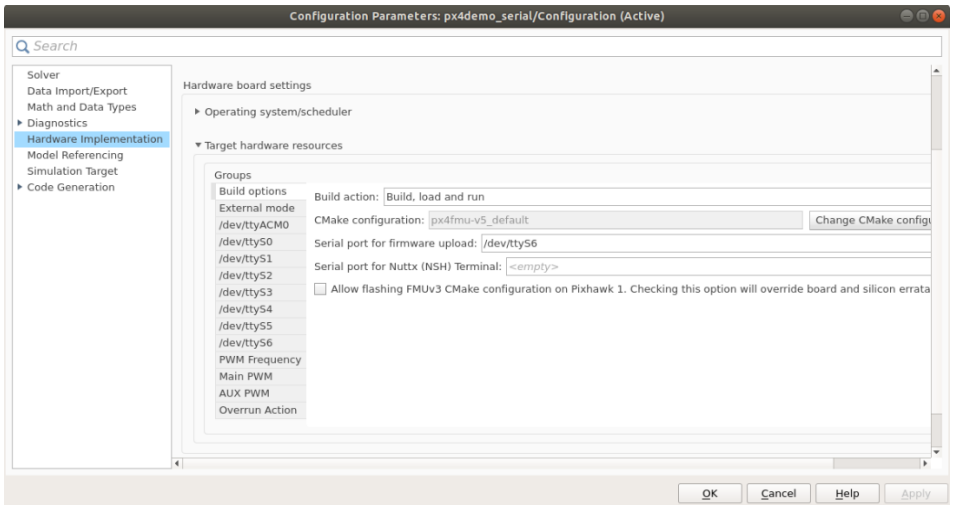
- MATLAB/Embedded coder for PX4 - Utilizing custom codes in MATLAB on Linux was tried but getting an active connecting wirelessly (i.e. configuring the correct UDP port) has not been successful. (https://www.mathworks.com/products/new_products/release2019a.html)

- Global Protect VPN client (used mainly to run encoder on MATLAB) (https://vpn.usf.edu/)

- Download and install Visual Studio Code for developing the source code (https://dev.px4.io/master/en/setup/vscode.html)

MATLAB/Simulink implementation:

Codes can also be generated in the embedded code format from a custom MATLAB/Simulink control file. Once a control algorithm is created, the Embedded Coder for PX4 Autopilot package can be used to generate a .cpp and .hpp file. This can then be either added to the 'Flighttask library' in the firmware files (which shall be explained below) and run the quadrocopter on autopilot or we can create an active communication (wired or wireless) directly to the Pixhawk 4 and upload the algorithm just by clicking a designated encoder button on the Simulink application. In order to do the latter, we need to first set board and port parameters in the hardware implementation tab in Simulink (Toolbar > Configuration Parameters > Hardware Implementation) as shown below:



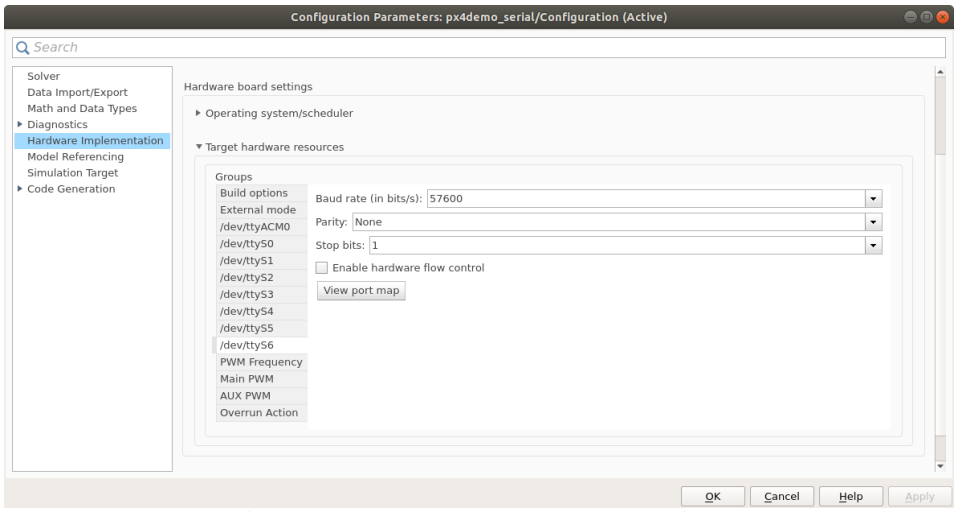


Fig: (Above) Setting the serial port number for wireless communication to upload the updated firmware. (Below) Setting the baud rate same as mentioned in the MAVLink setup (see last line in that image) (LACIS at USF).

Example to run a custom code on Simulink:

Below link is an example of a program that logs signals to SD card on the PX4:
https://www.mathworks.com/help/releases/R2018b/supportpkg/px4/ref/log-signals-sdcard-px4-example.html

When running a firmware code developed in MATLAB, it is often required to customize the default boot. If the complete boot should be replaced, create a file /fs/microsd/etc/rc.txt, which is located in the etc folder on the microSD card [8]. If this file is present nothing in the system will be auto started, and only the commands set in the txt file will follow. (more detail given in https://dev.px4.io/v1.9.0/en/concept/system_startup.html under Customizing the System Startup)

This is the custom rc.txt file which loads px4_simulink_app on start-up (note that the communication setting is same as mentioned in the command lines at the bottom – in bold):

```
usleep 1000
uorb start
usleep 1000
tone_alarm start
usleep 1000
px4io start
#gps start -f #Starts GPS driver and Fake a GPS signal (useful for testing)
usleep 1000
sh /etc/init.d/rc.sensors
usleep 1000
#Uncomment the below 2 lines to use LPE estimator
attitude_estimator_q start
local_position_estimator start
#Using EKF2 estimator by default as PX4 does build LPE on px4fmu-v2 due to a limited
flash.
ekf2 start
usleep 1000
mtd start
usleep 1000
param load /fs/mtd_params
usleep 1000
rgbled start
usleep 1000
fmu mode_pwm # This is required for AUX PWM channels
usleep 1000
px4_simulink_app start
mavlink start -u 14550 -b 57600 -m onboard -r 80000
mavlink boot_complete
#exit #4-Jan-2019
```

Unfortunately, the upload process stopped towards the end as the encoder was unable to find the serial port on the computer. This process has been replicated number of times on Ubuntu and Windows system but has always failed over wi-fi. However, it did run when connected via USB (setting the port to /dev/ACM01). But this setting is not ideal for real-time testing purposes of custom control codes. Hence it is still an on-going find.

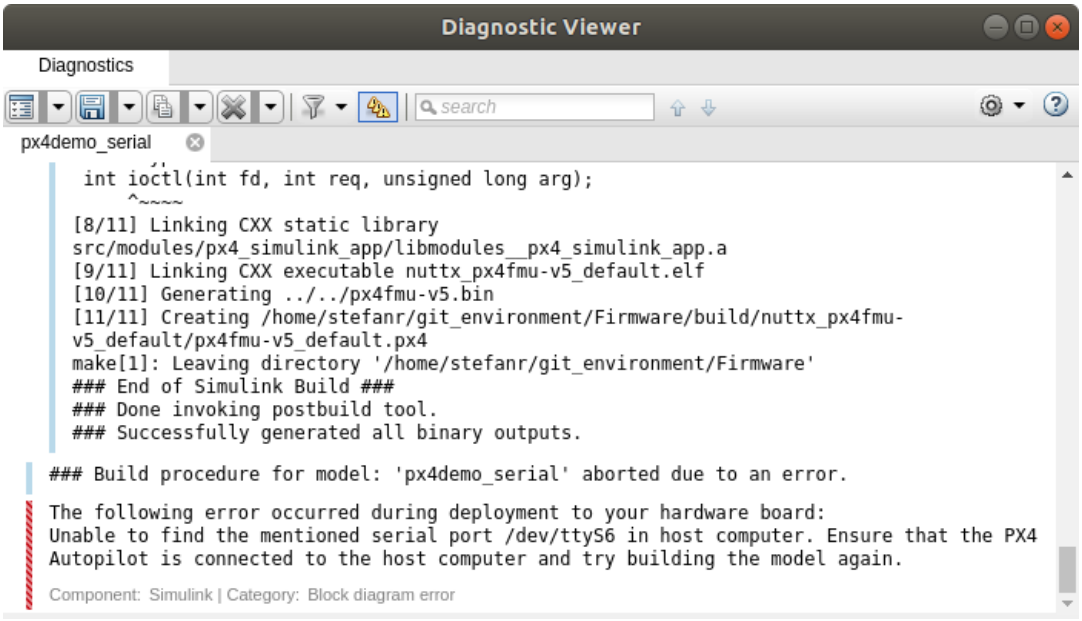


Fig: Diagnostic viewer on Simulink displaying the build error (LACIS at USF)

Firmware architecture:
Code structure, focus more on control code

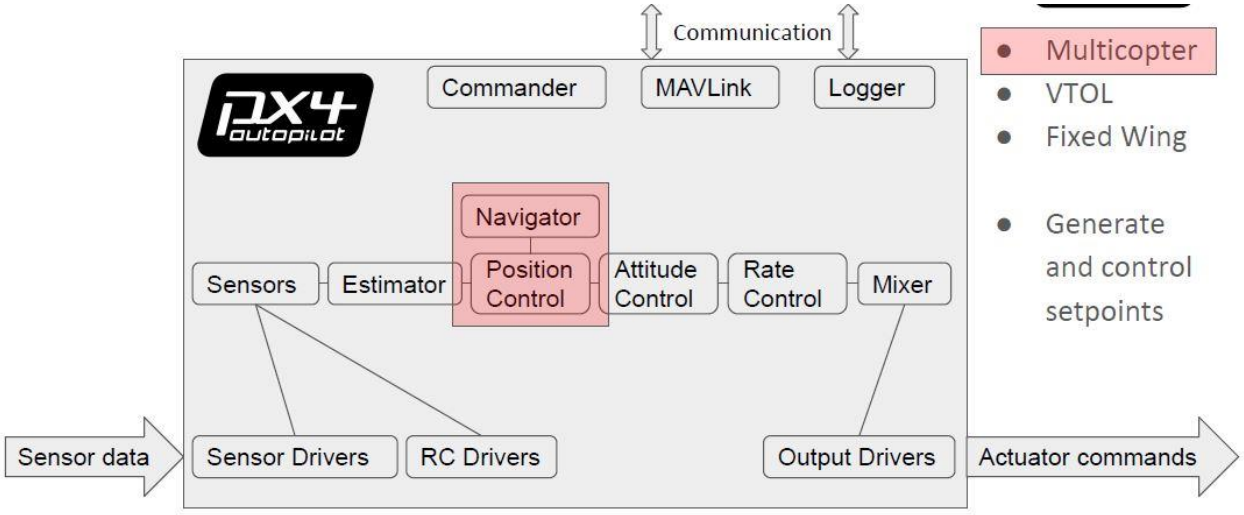


Fig: Firmware architecture overview (PX4 Developers Summit 2019 [5])

The firmware is divided into 3 sections:

1. Communication
2. Control code
3. Input data and output commands

1. Communication:

These code files launch the autopilot system on the target launcher (NuttX, SITL, etc). At the same time, it also sends messages to ground communication system(GCS) via MAVLink bridge and within the flight task by uORB topics (uORB is an asynchronous publish() / subcribe() messaging API used for inter-thread/inter-process communication).

Files contained in the following folders of the firmware is part of the communication code:

- `boards` – bootloader support for different brands of boards (Holybro, etc)
- `cmake` – .cmake files for initiating on the target
- `launch` – .launch files for launching SITL (Software in the Loop) simulator communication
- `mavlink` – message definition files for GCS
- `msg` – contains all the .msg files used by uORB topics and messages from various sensors
- `platforms` – simulator support (NuttX, Posix, Qurt)
- `posix-configs` - .config files for driver startup
- `ROMFS`

Follow this link for an example to understand how these codes work:
https://dev.px4.io/master/en/apps/hello_sky.html

2. Control code:

This is the brain of the autopilot firmware. It consists of all the algorithms used in filtering sensor data, making sensor value estimations (Estimation and Control Library – Estimated Kalman Filters (ECL-EKF)), determining position and attitude control, rate control and providing accurate mixers for MultiCopter, VTOL and Fixed Wing applications.

Files contained in the following folders is part of the control code:

- `Tools` – Math and control tools used for calculation in the control codes.
- `src`
  - `include` – standard header to include in other codes
  - `lib` – contains function libraries for calculations (ekf, math, matrix, pid, mathlib, etc)
  - `modules` – contains all the estimator and control codes

Control theory:
Since the model being worked on is a Quadrocopter, only MC position and attitude control is explained below:

Multicopter Position Controller

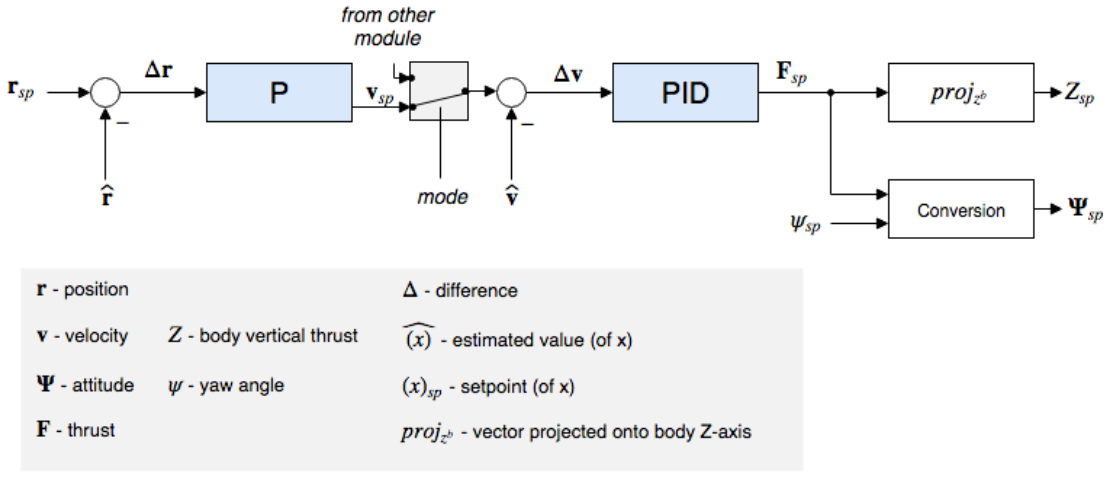


Fig: Multicopter (MC) Position control system (PX4 Developers Guide [1]).

- Estimates of all the states come from Extended Kalman Filter (EKF2).
- This is a standard cascaded position-velocity loop.
- Depending on the mode, the outer (position) loop is bypassed (shown as a multiplexer after the outer loop). The position loop is only used when holding position or when the requested velocity in an axis is null.
- The integrator in the inner loop (velocity) controller includes an anti-reset windup (ARW) using a clamping method.

Core Position-Control for MC.

This class contains P-controller for position and PID-controller for velocity.

Inputs are:

- vehicle position/velocity/yaw
- desired set-point position/velocity/thrust/yaw/yaw-speed
- constraints that are stricter than global limits

Outputs are:

- thrust vector and a yaw-setpoint

If there is a position and a velocity set-point present, then the velocity set-point is used as feed-forward. If feed-forward is active, then the velocity component of the P-controller output has priority over the feed-forward component. If there is a position/velocity- and thrust-setpoint present, then the thrust-setpoint is omitted and recomputed from position-velocity-PID-loop.

Core Attitude-Control for MC.

The nonlinear multicopter attitude control is based on unit quaternions to avoid the gimbal lock, wherein it controls only the yaw angle and angular body rate. The two states in turn compute the values of roll and pitch angles to determine a precise attitude control with a time constant between 0.2-0.4s. This implements the multicopter attitude controller. It takes attitude setpoints as inputs and outputs a rate setpoint. The controller has a P loop for angular error.

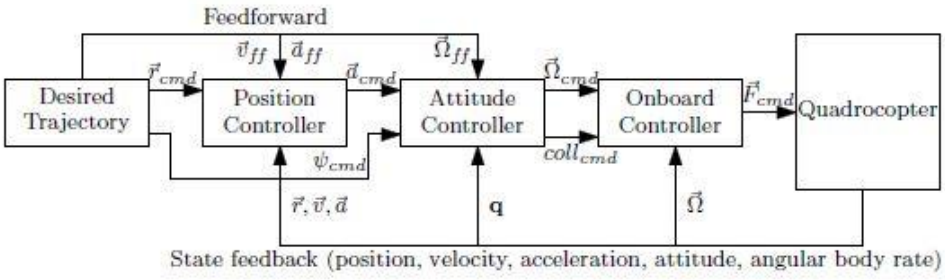


Fig: Attitude control for Multicopter (Nonlinear Quadrocopter Attitude Control [6])

Control Design:

As a first step, the commanded acceleration from the position loop and the commanded yaw angle must be converted into a desired attitude q_{cmd}. The objective is then to design a feedback law which stabilizes the quadrocopter at any desired physical attitude. Since any physical attitude in SO(3) corresponds to two antipodal quaternions in S3, this can be achieved by stabilizing the attitude q at $+-q_{cmd}$. When neglecting this fact, quaternion-based controllers can cause undesirable phenomena such as unwinding, where the rigid body rotates unnecessarily through a full rotation. To solve the problem of unwinding, the controller must satisfy

$$\Omega_{cmd}(q) = \Omega_{cmd}(-q)$$

Control Law:

$$\vec{\Omega}_{cmd}(\mathbf{q}) = \frac{2}{\tau}\mathrm{sgn}(q_{e,0})\mathbf{q}_{e,1:3}, \qquad \mathrm{sgn}(q_{e,0}) = \begin{cases} 1, & q_{e,0} \geq 0 \\ -1, & q_{e,0} < 0 \end{cases}$$

τ = first-order system time constant [s],

$q_e = q^{-1} \cdot q_{cmd}$ error measure, representing the rotation from q to q_{cmd}. Which is G.A.S. equilibrium point.

The control task is divided into two parts: Reduced and Full attitude control.
In Reduced AC, only the crucial pointing direction of the thrust is controlled. The yaw angle is not controlled directly, but target orientation is always chosen such that no rotation about the yaw axis is induced.
Whereas in Full AC, the pointing direction of the thrust vector as well as the yaw angle is controlled. The target orientation is chosen such that the corresponding z-axis is aligned with commanded acceleration direction and yaw angle is equal to the commanded yaw angle.

The Attitude control algorithm for MC can be found in `Firmware/src/modules/mc_att_control/AttitudeControl/`
It contains the `AttitudeControl.cpp` & `AttitudeControl.hpp` files and a test file `AttitudeControlTest.cpp` which tests the code using constant values for every input parameter. The file `mc_att_control_main.cpp` in its parent folder calls different flight tasks like update setpoints, publish rates, performs certain action during throttle, etc.

Extended Kalman Filter Algorithm:

The Estimation and Control Library (ECL) uses an Extended Kalman Filter (EKF) algorithm to process sensor measurements and provide an estimate of the following states:

- Quaternion defining the rotation from North, East, Down local earth frame to X, Y, Z body frame
- Velocity at the IMU - North, East, Down (m/s)
- Position at the IMU - North, East, Down (m)
- IMU delta angle bias estimates - X, Y, Z (rad)
- IMU delta velocity bias estimates - X, Y, Z(m/s)
- Earth Magnetic field components - North, East, Down (gauss)
- Vehicle body frame magnetic field bias - X, Y, Z (gauss)
- Wind velocity - North, East (m/s)

The EKF runs on a delayed 'fusion time horizon' to allow for different time delays on each measurement relative to the IMU. Data for each sensor is FIFO buffered and retrieved from the buffer by the EKF to be used at the correct time.

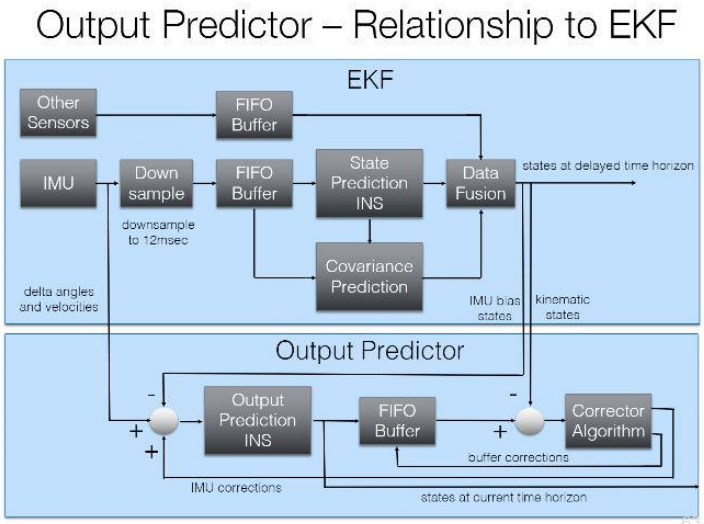


Fig: Extended Kalman Filter – Output Predictor relationship chart (Paul Riseborough – GitHub [7])

Achieves 2 objectives:
1) It takes data from different points in time, and able to fuse them in delayed time horizon
2) It also outputs attitudes, velocities, position states and produce a very up to date estimate from the IMU data that does not suffer from minimum latency and lag.

Most important piece of data is from the IMU. The buffer takes all the data from different sensors (barometer, temp, GPS, magnetometer, etc) uses them to correct the motion that is calculated from the IMU at the correct time for that piece of data. So, buffers fix the motion at the correct point in time. IMU data is used in initial navigation

calculation and it is corrected it using data from other sensors, and then calculate the uncertainty in the estimates. From EKF the state estimates is thus generated.

3. Input data and output commands:

These codes consist of sensor drivers and actuator commands which are sent directly or through uORB messages. They convert and send readable input values to controller codes and take output values from it which are then sent to the actuators after precise mixing.

Everything in the following folders:

- `src`
  - `drivers` – drivers for all the sensors and actuators
  - `systemcmds` – mixer codes and command generators

Designing a custom Flight task:

A new flight task can be implemented in `Firmware/src/lib/flight_tasks/tasks` pointing towards any control module where the messages are sent via uORB.
It has easy implementation in the code where we can choose any:

- Factory class: Flighttask(s) like getPositionSetpoint(), update() to define the algorithm class.
- Base class: Flighttask #_setpoints, #_state to call the states.

The added flight tasks can be called at convenience via MAVLink commands (MAV_CMD) as well.

Below is the complete Flight task architecture, with its parameters that can be chosen according to the task needs:

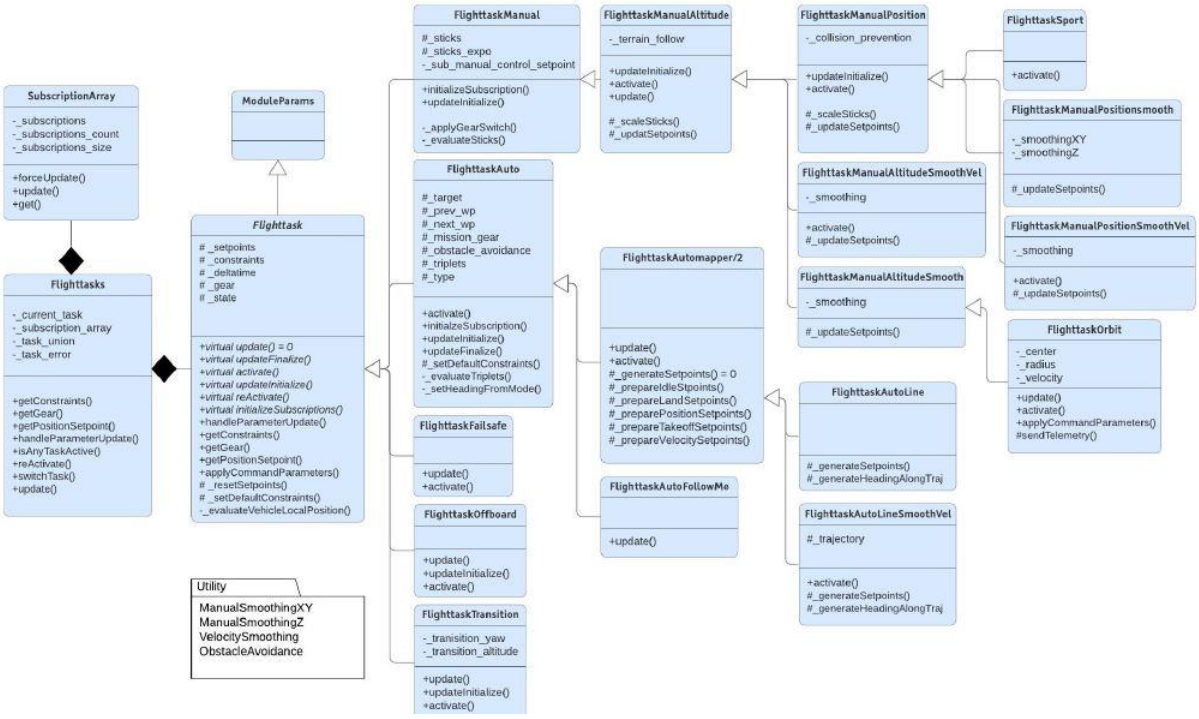


Fig: Flight task architecture (PX4 Developers Summit 2019 [5])

There is one class for each behavior. Initiated in the controller but only when necessary. It can be seen, the four columns on the right has all the flight task modules found in the folder mentioned above, with their classes.
For example, if we call getPositionSetpoints() from 'Flighttasks' (first column) it will call getPositionSetpoint() from 'Flighttask' (second column) and take values from _setpoints. We then call update() to update the values every time, else the output will just stay constant. Every task must have an update method. Another function Activate() is used at the start of the task to run it, and so on.
A good example can be seen in the video in the link below:
https://dev.px4.io/master/en/concept/flight_tasks.html

Next step:

The Pixhawk flight controller algorithm is definitely a complicated structure, but its capabilities are very powerful and to fully develop a custom controller that can work and be tested efficiently,

- It is best to start by mimicking simple examples found on the PX4 developer documentations.
- Conduct tests of these codes on SITL simulators (FlightGear, Gazebo, JMavSim).
- This will give an understanding of what steps must be followed during the initial process when launching the firmware and later how messages are transferred on-board and off-board between softwares and devices.
- Collecting log files from these simulations will better our comprehension of the parameters and functions affecting the controller codes which can later be tweaked and implemented to our needs. Many parameter values can be seen on QGroundControl as well, for example certain uORB topics, position and attitude parameters, etc.
- Simulating similar custom codes on MATLAB as its embedded coder is very useful.
- Perform variations of custom system startups with files in the microSD card folder.

References:

[1] *PX4 Developer Guide* - https://dev.px4.io/master/en/index.html

[2] *PX4 Autopilot User Guide* - https://docs.px4.io/master/en/

[3] *PX4 GitHub* - https://github.com/PX4

[4] *PX4 AutoPilot Discussion forum* - https://discuss.px4.io/

[5] *PX4 Developers Summit 2019* - https://px4.io/px4-developer-summit-zurich-2019/

[6] *Nonlinear Quadrocopter Attitude Control – Technical Report. Dario Brescianini, Markus Hehn and Raffaello D'Andrea. Institute for Dynamic Systems and Control (IDSC) ETH Z¨urich October 10, 2013.*

[7] *Estimation & Control Library for PX4 (Paul Riseborough)* - https://github.com/priseborough/ecl

[8] *Embedded Coder for PX4 Autopilot* - https://www.mathworks.com/help/supportpkg/px4/index.html