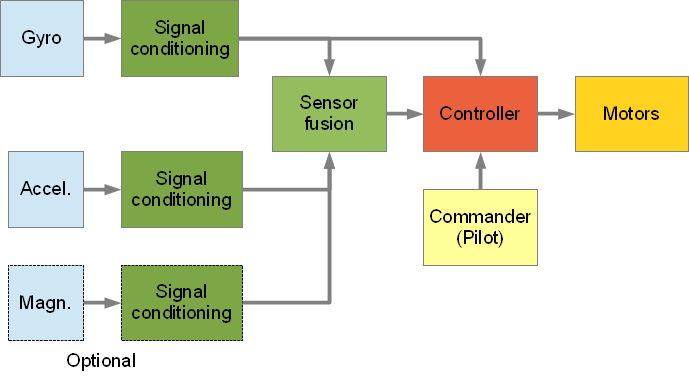
# Navigating the Original Crazyflie 2.0 firmware by Bitcraze

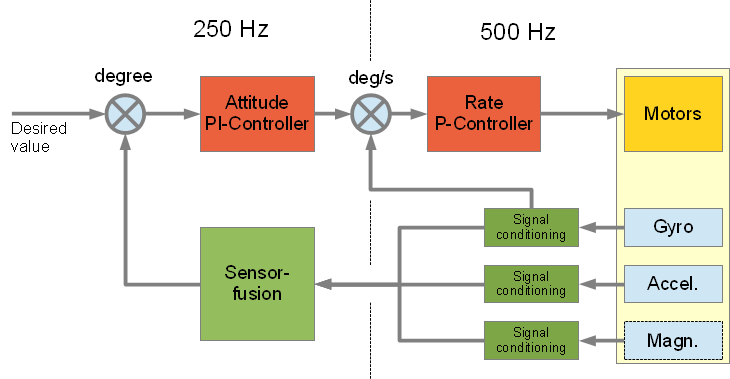
By Shamoon Irshad, August 2017.

To succeed in any DEVS based implementation, it is necessary to have a good sense of how the original firmware from the manufacturer was written. This gives an idea for the overall objectives of the software design. Also, many of the driver subroutines can be re-used.

Any real documentation for the firmware is unavailable as of today on the Crazyflie wiki website. Two general system flow diagrams are provided as a map for where to look for in the code.

Source : https://wiki.bitcraze.io/doc:crazyflie:dev:fimware:sensor\_to\_control





In broad terms, the controller is being fed with input data from the motion sensors: the Accelerometer, Gyroscope and Magnetometer (optional), as well as the from the user input (Commander) through a radio link. The sensors provide a constant feedback to the controller so that it can calculate the offset between the desired value (from commander) and the actual value (from the sensors). This offset is used by PID controllers for estimating the output for the motors. For an unchanged commander input, the PID controllers ensure a smooth minimization of this offset over time and thus ultimately the quadcopter is made to fly in the required orientation.

The original Crazyflie 2.0 firmware is developed to run on top of FreeRTOS, an open-source Real-Time Operating System. After system initialization, various real-time tasks, i.e . stabilizer, controller, and memory control, etc. are executed by the RTOS.

The commander task polls the user input using the CRTP radio protocol. The input device could be a Bluetooth connected smartphone, a wireless gaming joystick etc, for controlling the speed/direction of the quadcopter flight. The orientation is measured in terms of the 3D euler angles called the roll, pitch and yaw (R,P, and Y). RPY together is called the attitude of the aircraft.

The ‘Stabilizer’ task is of fundamental importance: it encompasses the entire work loop for the flow of data between the sensor input, commander input, controller and the motor drivers. Various subroutines in the stabilizer task working loop are:

1. **imu9read**: reads the motion sensor values, in each orientation: x,y,z from each of the accelerometer, gyroscope and the magnetometer. Thus there are a total of 9 parameters. Theses are declared as global variables in ‘stablilzer.c’ file. The magnetometer readings are not used in flight control.
2. **commanderGetRPY**: Reads the desired roll, pitch & yaw values. These are user input, polled by the commander task.
3. **sensfusion6UpdateQ**: The raw accelerometer x,y,z and gyroscope x,y,z are used in determining the attitude of aircraft. The first step is to determine the “Quaternion”, Q values. These are concerned with aerodynamic mechanics. The equations in this subroutine calculates the Q values.
4. **sensfusionGetRPY**: Here, the Q values calculated above are used to find the Roll, Pitch and Yaw values in degrees. These RPY values is the actual, i.e. current attitude of the quadcopter.
5. **controllerCorrectAttitudePID**: Here, PID objects for each of axis, i.e. Roll, Pitch and Yaw are created. The input parameters are the actual RPY (from step 4) and desired RPY (from step 2). Since PID controllers give a time-based output, the required RPY rates are found. These are degree changes required per second.
6. **stabilizerAltitudeHoldUpdate**: Checks if the “Altitude Hold” , i.e. hovering mode is enabled by the commander input.  
   A base thrust value is declared in stabilizer.c which is later used to set the motor powers and ultimately the vertical speed of the quadcopter. if the Altitude Hold mode is enabled, the current altitude is set as the target altitude so that the vertical speed outcome would be zero. The current altitude is estimated roughly using the pressure sensor (barometer). If the Alt Hold is recently enabled, PID object for thrust is created so that the vertical speed is gradually driven down to zero.
7. **controllerCorrectRatePID** and **controllerGetActuatorOutput**: Creates a copy of RPY rate values from PID outputs in step 5. These are later used calculating required motor powers.
8. **ControllerGetActuatorThrust:** In case when the Alt hold mode is not enabled, gets sets the base thrust variable after reading from the commander input.
9. **DistributePower:** Determines output drive rates for each of the four motors. These rates are found by appropriate additions/subtractions of the RPY rates with the base thrust. After this, the MotorsSetRatio function is called for each motor, to implement the required PWM motor signals.
10. Repeat from step 1, for the continuous loop operation.