Software Engineering for Quadcopter Control

CSE303 - Computer Science Project

Author: Vrushank Agrawal

Supervisor: Dr. Timothy Bourke

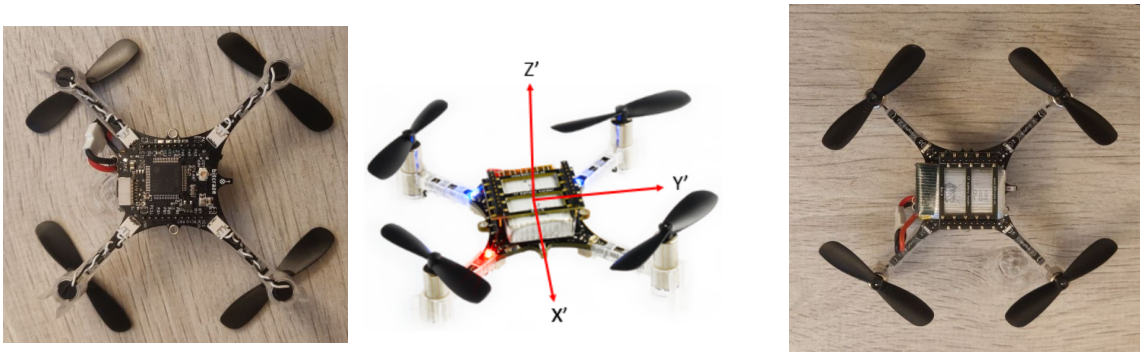
Abstract

A quadcopter (Unmanned Aerial Vehicle) is a naturally unstable embedded system that uses different control algorithms to guide its movement. Usually, quadcopters are programmed in languages like `C` or `C++` which are complex to directly use for understanding and composing reactive behaviours of embedded systems. In this report, we will study the Bitcraze Crazyflie 2.1 quadcopter and describe its static structure including main tasks and components of its feedback control loop. We will create a block diagram to study the dataflow between these different tasks to concretely understand its control algorithm and eventually analyze the possibility to reprogram it in a dataflow synchronous language like Lustre that offers an alternative and simpler approach to study control algorithms through high-level block diagrams.

Contents

1 Introduction

A quadcopter (a type of drone) is a four-propeller helicopter, usually arranged in a square, with four independent rotors. The need for aircraft with greater maneuverability and hovering ability has led to a rise in quadcopter research. The four-rotor design allows quadcopters to be relatively simple in design yet highly reliable and maneuverable, allowing them to be programmed for remote piloting by a ground control operator. Active research in the area is continuing to increase the abilities of quadcopters which are now used for various civil and military applications such as environment exploration, land surveys, photography, remote strike execution, etc. With further research, quadcopters would be capable of advanced autonomous missions that are currently not possible with other vehicles.



(a) Bottom View. (b) Axes (c) Top View

Figure 2: Crazyflie 2.1 quadcopter

Quadcopters are usually written in languages like C/C++ that are complex and hard to use for replicating reactive behaviors of embedded systems. In this report we will study a quadcopter and analyze the possibility to replicate parts of its code in a dataflow synchronous language like Lustre where it is simpler to model and study reactive behaviours of embedded systems. We will use the Bitcraze Crazyflie 2.1 quadcopter which has been specifically developed for testing and research purposes shown in Figure 2. For the hardware, the crazyflie uses an ARM Cortex-M4, a 1Mb flash, and the BMI088 3 axis accelerometer/gyroscope and BMP388 high precision pressure sensor both manufactured by Bosch electronics [1].

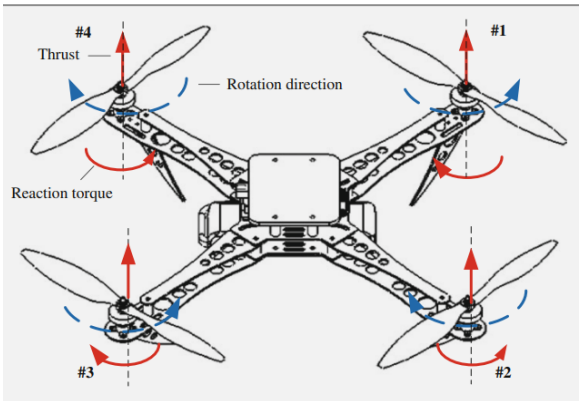
2 Design of Quadcopter

A quadcopter uses 4 propellers divided into 2 sets of diagonal pairs which rotate clockwise and anti-clockwise respectively. A pair of propellers on the same side spin in the opposite direction , i.e. diagonal propellers spin in the same direction. This is done to nullify the torque generated as a reaction of the upward force generated when the propellers rotate with some angular speed as shown in Figure 3a. A quadcopter has 6 degrees of freedom where 3 degrees are for transitional movement along the x, y, and z axes while 3 degrees are for rotational movement along x, y, and z axes termed roll, pitch, and yaw respectively as shown in Figure 3b [3].

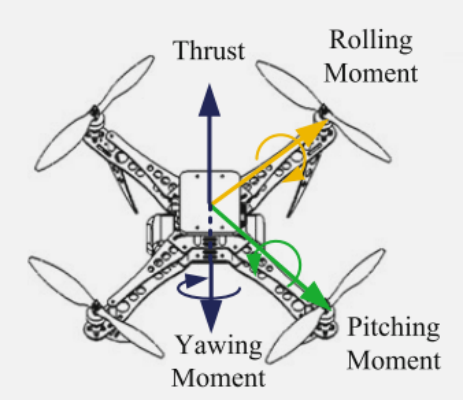
2.1 Movement

Even though a quadcopter has six degrees of freedom, the following four variables can be used to control its movements [2] which have been illustrated in Figure 4.

- `Thrust`: This is characterized as the transitional movement of the quadcopter along the z-axis or *upward-and-downward movement* [2]. This movement is controlled by changing the angular speed of the propellers by an equal amount as shown in Figure 4a. Here, all motors change angular speed with the same amount and as a result, there is a net upward force on the quadcopter. So,



(a) Propeller rotation



(b) Rotational Degrees of Freedom

Figure 3: Quadcopter Rotation and Degrees of Freedom [2]

for example, if the quadcopter needs to rise, the angular speed of the propellers will be increased proportionally to increase the upward force acting on it against the gravitational force pulling it downwards and vice-versa if it needs to fall.

- `Roll`: This is characterized as the rotational movement of the quadcopter along the x-axis or *leftward-and-rightward movement* [2]. So, for example, if the quadcopter wants to roll right, it would speed up motors on the left side $(\#3, \#4)$ by some amount and slow down the two on the right $(\#1, \#2)$ by the same amount as shown in Figure 4b. Doing this creates an increased net force on the left side which lifts it on that side to produce a right roll.

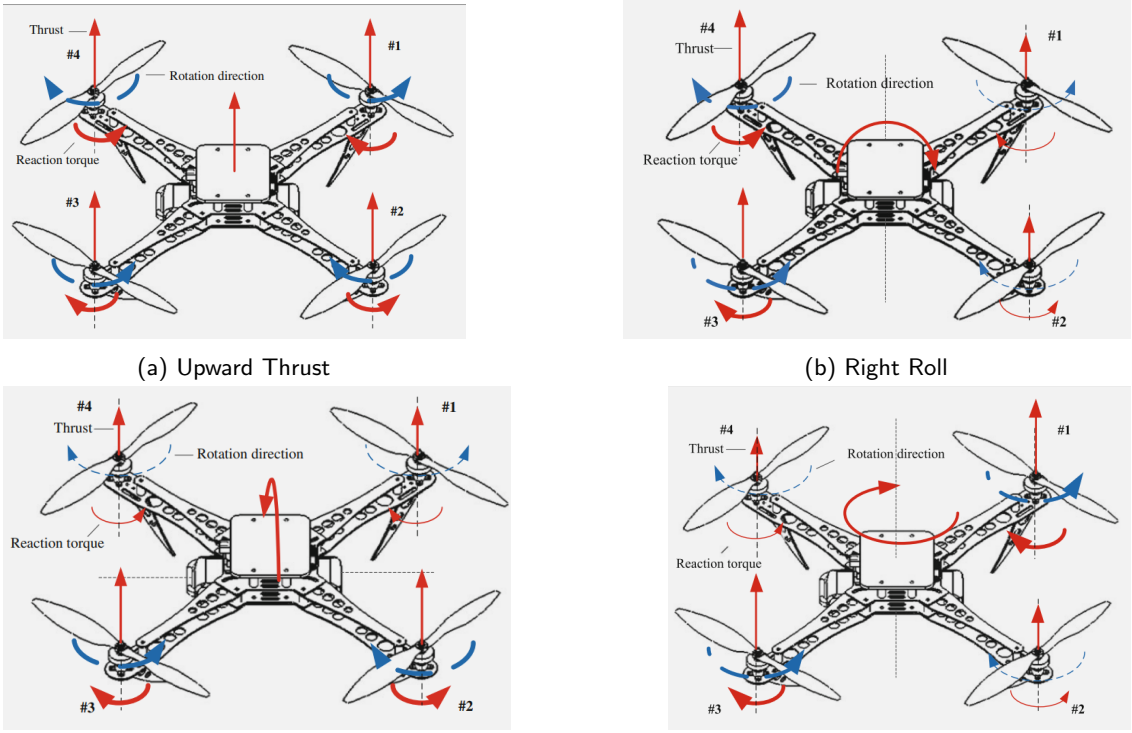
- `Pitch`: This is characterized as the rotational movement of the quadcopter along the y-axis or *forward-and-backward movement* [2]. If the quadcopter wants to pitch forward, it speeds up the two motors at the back $(\#3, \#2)$ and slows down the two at the front $(\#1, \#4)$ as shown in Figure 4c. Doing this increases the torque at the back-end of the quadcopter and it rises up relative the inertial frame while generating a forward pitching movement

- `Yaw`: This is characterized as the rotational movement of the quadcopter along the z-axis or *clockwise-and-anticlockwise movement*. This movement is realized by increasing and decreasing the angular speed of the diagonal propellers by the same amount. So, if the quadcopter wants to yaw clockwise, the angular speed of the propellers rotating clockwise $(\#4, \#2)$ will be decreased while the angular speed of the propellers rotating anti-clockwise $(\#1, \#3)$ will be increased by the same amount as shown in Figure 4d. A net unequal reactionary torque would be generated in the clockwise direction relative to the copter's z-axis hence turning it clockwise.

2.2 Control Algorithms

Control algorithms are methods that let a quadcopter decide in real time which movements (described above) to use to reach its final destination (desired setpoint). Control algorithms are employed in a feedback loop mechanism where sensors provide data to the control algorithm that calculates the error between the current position, determined using the measured data, and the desired setpoint and then chooses the optimal movement to minimize the error while repeating the process until the desired setpoint is achieved.

The Crazyflie 2.1 quadcopter software provides three different control algorithms— PID (Proportional-Integral-Derivative), INDI (Incremental Nonlinear Dynamic Inversion), and Mellinger. The INDI and Mellinger controllers are direct applications of [5] and [6] which have only recently been integrated into the Crazyflie firmware without rigorous testing or documentation. The PID control algorithm, on the other hand, has been rigorously tested and widely documented in the firmware and is also a simple and standard algorithm hence, we will study it further.



(a) Upward Thrust

(b) Right Roll

(c) Forward Pitch

(d) Clockwise Yaw

Figure 4: Quadcopter movements as a result of change in motor angular speeds [2].

2.2.1 PID Controller

The PID control algorithm is a linear algorithm that is widely used in robotics. It functions by calculating the error or difference between a measured output (sensors data) and a desired setpoint and adjusts the system control inputs such that the calculated error is corrected in a controlled manner in several loops to reach the desired set point to prevent sudden hiccups in the system. The PID algorithm consists of three control parameters, P–Proportional, I—Integral and D—Derivative. The mathematical expression of the discrete-time PID algorithm is given as the following control function:

$$u_p = K_p e(t) + K_d \frac{de(t)}{dt} + K_i \int_0^t e(\tau) d\tau$$

Here, the term with K_p determines the reaction to the current error (Proportional), the term with K_i determines the reaction based on a sum of recent errors (Integral) while the term with K_d responds to the rate at which the error has been changing (Derivative) [3]. A block representation of a PID controller is show in Figure 5.

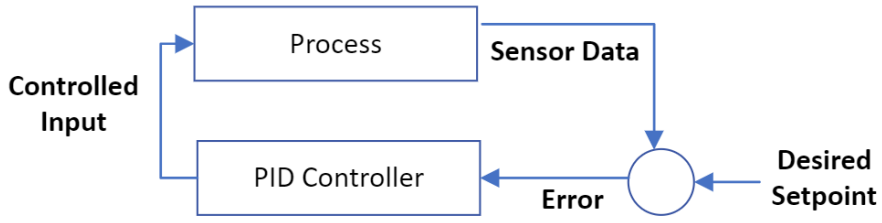


Figure 5: Block diagram for a PID controller in a quadcopter's control loop.

3 Control in Crazyflie

The crazyflie 2.1 quadcopter uses the open-source FreeRTOS real-time operating system for its microcontroller [7] which is responsible for the real-time operations of the quadcopter.

3.1 Source Code

To study the crazyflie, we studied its code repository hosted on github [9] that contains over 1000 files and more than $375,000$ lines of code using tools like `CScope` and `CTags` that are specficially desgined to navigate large `C` code repositories. The root directory contained several subdirectories for `build, tests, tools, documentation, and vendor` (which contained code for third-party software integrations like FreeRTOS), but the main directory useful for our analysis was *src* that contained the crazyflie source code. This *src* directory contained subdirectories shown in Appendix C for the expandable decks [11] (*decks*), sensors (*drivers*), hardware abstraction layer (*hal*), third-party libraries (*lib*), stm32 integrated circuit for the microcontroller (*platform*) that are all used to support the main crazyflie code in the *modules* subdirectory.

The */src/modules* directory is divided into two subdirectories—*interface* that contains all header files and *src* that contains all `C` files. For our analysis, we stayed in the */src/modules/src* directory that contained all `C` source files for the feedback control loop (*stabilizer.c*) and its five phases described below. The *estimator.c* contained calls to estimator algorithms (3.3.2), *crtp_commander.c* contained the `Commander` functions (3.3.3) and calls to other `C` files supporting the `Commander`'s functionalities, *controller_pid.c* contained functions for the PID controller (3.3.4), and *power_distribution_quadrotor.c* contained functions for power distribution in the quadcopter's motors (3.3.5).

3.2 Tasks

A real time application can be structured as a set of independent tasks run by a scheduler where each task executes within its own context with no dependency on other tasks within the system or the scheduler itself. At any instant, the scheduler decides which task executes on the microcontroller. A task itself has no knowledge of the scheduler activity and it is the responsibility of the scheduler to ensure that the processor context (register values, stack contents, etc) when a task is swapped in is the same as when the same task was swapped out. This is achieved by providing individual stacks for each task [8].

Initially, we assumed that the control algorithm was divided into several individual tasks handled by the FreeRTOS scheduler. We thus searched for functions like *xTaskCreate* and *xTaskCreateStatic* in the crazyflie-firmware repository [9] that are required to create tasks for handling by the scheduler [10]. We retrieved the complete list of tasks created using *xTaskCreate* and listed them in Appendix A.1. Out of these 21 tasks, 20 are for optional decks (that can be added to improve quadcopter's movement [11]) while one is for led lighting sequence. None are directly related to the feedback control loop. For *xTaskCreateStatic*, we used the classical `DEBUG PRINTING` method to check which functions are being called as static tasks when the crazyflie 2.1 is connected and flown and these have been listed in Appendix A.2. Out of the listed functions crteaed as static tasks, only *stabilizerInit* and *sensorsTaskInit* functions are directly related to the control loop, but both are implemented one after the other and do not switch amongst each other through semaphores during program execution. The *estimatorKalmanTaskInit*, also related to the feedback control loop, is initialized but not ultimately used as the estimator defaults to the less computationally expensive *estimatorComplementary* function instead. The *sensorsTaskInit*, as the name suggests, is to retrieve information from the sensors and is not particularly useful for our analysis, and thus, we will only study the *stabilizerInit* static task below.

3.3 Feedback Control Loop

The feedback control loop is the main loop of a quadcopter that constantly evaluates the quadcopter's position and plans its future trajectory using the control algorithms. The feedback control loop for the crazyflie 2.1 is initialized as the *stabilizerInit* task that calls the *stabilizerTask* function present in the *stabilizer.c* file of its firmware repository. This control loop can be divided into five main phases namely: `Sensors`, `Estimator`, `Commander`, `Controller`, `Motors` that are explained below and have been drawn as a block diagram in Figure 6 representing the main data flow in the feedback control loop. In the diagram, the main `Drone Process` initiates the feedback control loop by calling the `Sensors` block which is followed until the `Motors` block that gives back control to the drone process to complete the loop.

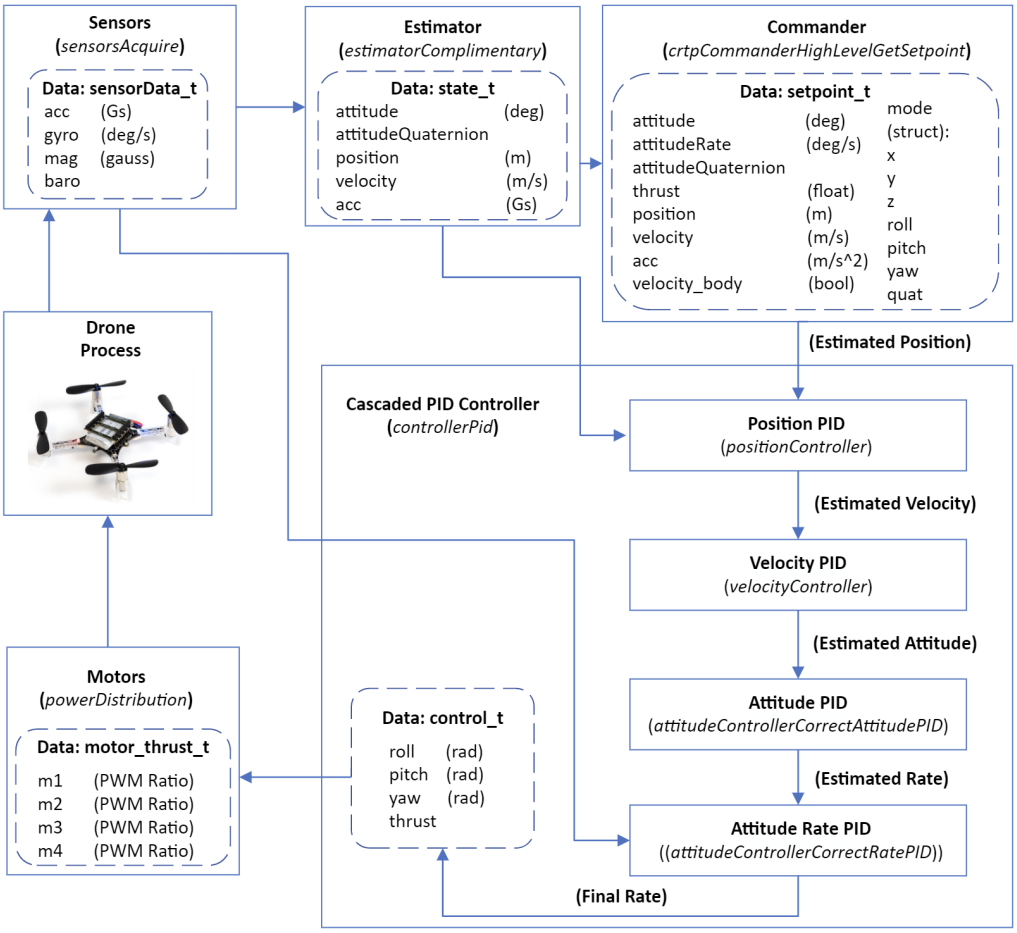


Figure 6: Feedback control loop dataflow block diagram

3.3.1 Sensors

The crazyflie 2.1 uses two IMU (Inertial Measurement Unit) sensors BMI088 (3 axis accelerometer) and BMP388 (high precision pressure sensor). The control loop starts by first fetching data from the sensors through the *sensorsAcquire* function in the following struct:

```
typdef struct sensorData_s {
    Axis3f acc;       // Gs
    Axis3f gyro;      // deg/s
    Axis3f mag;       // gauss
    Axis3f baro;
    uint64_t interruptTimestamp;
} sensorData_t;
```

This data is sent to the `Estimator` and the `Controller` which use the readings as explained below.

3.3.2 Estimator

The estimator is where the crazyflie uses one of three estimator algorithms (*estimatorComplementary, estimatorKalman, or estimatorOutOfTree*) to decrease the noise in the sensors data and provide a better estimate of the state variables required by the control algorithm. The defualt estimator is *estimatorComplementary* which uses the *gyro* and *acc* readings to calculate the attitude of the quadcopter at a rate of 250Hz. Attitude is a combination of `roll`, `pitch`, and `yaw`. The velocity component is updated using the *acc* readings at the same frequency as attitude and the position readings are updated using the velocity readings at a frequency of 100Hz. The estimator algorithm also calculates *attitudeQuaternion* but these are not yet used and are there for future integration in an improved state estimate. The crazyflie firmware uses the *stateEstimator* function to access these algorithms and the final output is stored in the following struct:

```
typdef struct state_s {
    attitude_t attitude;     // deg
    quaternion_t attitudeQuaternion;
    point_t position;        // m
    velocity_t velocity;     // m/s
    acc_t acc;               // Gs
} state_t;
```

This struct is passed to the `Commander` and the `Controller`.

3.3.3 Commander

The commander is where the quadcopter sets the desired destination setpoint. The crazyflie 2.1 uses the *crtpCommanderHighLevelGetSetpoint* function to evaluate the next setpoint. This function uses the *plan_current_goal* function which has a *planner* variable that keeps track of the setpoints of the trajectory in a *traj_eval* struct. The *planner* uses the previous setpoint data to evalute the new setpoint and updates this new setpoint as the latest setpoint to be used for the next loop calculation. If the quadcopter is in an active trajectory, the new setpoint is updated using the *commanderSetSetpoint* function otherwise it is not changed and current setpoint is nullified. Finally, if the setpoint is updated, then the *commanderGetSetpoint* function fetches this latest setpoint to fly to in the following struct:

```
typdef struct setpoint_s {
    attitude_t attitude;      // deg
    attitude_t attitudeRate; // deg/s
    quaternion_t attitudeQuaternion;
    float thrust;
    point_t position;        // m
    velocity_t velocity;     // m/s
    acc_t acceleration;      // Gs
    bool velocity_body;      // true if velocity in body frame
    struct {
        stab_mode_t x;
        stab_mode_t y;
        stab_mode_t z;
        stab_mode_t roll;
        stab_mode_t pitch;
        stab_mode_t yaw;
        stab_mode_t quat;
    } mode;
} setpoint_t;
```

This data structure is passed to the `Controller`.

3.3.4 Controller

The controller is the phase where the *setpoint*, *state*, and *sensors* data generated in the previous three steps is used to calculate the appropriate thrust required by each motor to reach the desired setpoint. The controller can use any of the three controller algorithms (`PID`, `INDI`, and `Mellinger`) but as mentioned earlier, we will study `PID` and look at the *controllerPid* function. This function uses the *mode* variable from the *setpoint* data struct that specifies if the quadcopter is static (`modeAbs`) or in flight (`modeVelocity`) to cap the desired *yaw* between -180° and 180° which might overflow this interval if the drone is in velocity mode when the *attitudeRate* is regularly appended.

The PID controller implemented in the crazyflie is actually a Cascaded PID controller which linearly evaluates `position`, `velocity`, `attitude`, and `attitudeRate` to eventually determine the thrust passed into motor power distribution. The PID controller constants (K_p, K_i, K_d) used in the crazyflie are given in Appendix B. The `position` is evaluated by the *positionController* function at a frequency of 100Hz when it runs the *runPid* function to calculate the desired `velocity` which is updated in the *setpoint*. The *positionController* directly calls the *velocityController* function (100Hz) which uses these recently updated `velocity` readings to evaluate the desired `attitude` by again using the *runPid* function. Afterward, the *attitudeControllerCorrectAttitudePID* function uses these recently calculated desired `attiude` readings with the previous readings from the *state* struct to evaluate the desired `attitudeRate` readings. Finally, the *attitudeControllerCorrectRatePID* function sets the desired `roll`, `pitch`, and `yaw` rate using the *pidSetDesired* function and also uses *gyro* readings from the *sensors* struct to further precise the *attitudeRate* readings even though they are not taken into account in the current implementation. The controller then updates the desired `rate` outputs in the following struct:

```
typdef struct control_s {
    int16_t roll;
    int16_t pitch;
    int16_t yaw;
    float thrust;
} control_t;
```

which is passed to the `Motors`.

3.3.5 Motors

In this final phase of the loop, the *powerDistribution* function uses the data from *control* struct to calculate the appropriate thrust for each of the four motors and set their values in the following struct:

```
typdef struct motors_thrust_s {
    int16_t m1;
    int16_t m2;
    int16_t m3;
    int16_t m4;
} motors_thrust_t;
```

This struct is used by the *motorsSetRatio* function to set the desired `thrust` for each motor. This step completes the loop which passes the control back to the `Drone Process` that runs the next iteration of the feedback control loop with the `Sensors` phase described in 3.2.1.

4 Conclusion

In this report, we introduced quadcopters and studied in detail the Bitcraze crazyflie 2.1 to understand its control structure and the general dataflow. From the analysis, we located the feedback control loop in the `stabilizer.c` file implemented in the *stabilizerTask* function. On further study, we discovered that this control loop runs in a simple linear fashion in a cyclic algorithm that executes each task one by one contrary to our initial assumption where the loop would have been divided into

several FreeRTOS tasks running in parallel and communicating via semaphores. It would have been useful to have this information before starting the project as that would have saved us time studying the FreeRTOS functions. We would have used that time to perform experiments with the drone to further study its feedback control loop.

5 Further Steps

The analysis provided in Section 3 and the block diagram in Figure 6 provide a promising first step for the gradual implementation of crazyflie's `C` code in a dataflow synchronous language like `Lustre` to study how much of a typical UAV controller can be rewritten in a block-diagram language. This is particularly useful for engineers who use dataflow synchronous languages (block diagram languages) to understand, describe, and compose reactive behaviours of embedded systems.

References

[1] Bitcraze, "Crazyflie 2.1", *bitcraze.io:*, https://www.bitcraze.io/products/crazyflie-2-1/. Accessed 28 November 2022.

[2] Quan, Quan. Introduction to Multicopter Design and Control. Springer Singapore, 2017. 1. pp.7-10.

[3] Ostojic, Gordana & Stankovski, Stevan & Tejic, Branislav & Dukic, Nikola & Tegeltija, Srdjan. (2015). Design, control and application of quadcopter. International Journal of Industrial Engineering and Management. 6. 43-48.

[4] Manimaraboopathy, M. & Christopher, H. & Vignesh, S. & selvan, P.. (2017). Unmanned Fire Extinguisher Using Quadcopter. International Journal on Smart Sensing and Intelligent Systems. 2017. 471-481. 10.21307/ijssis-2017-264.

[5] Smeur, Ewoud & Chu, Q.P. & Croon, Guido. (2015). Adaptive Incremental Nonlinear Dynamic Inversion for Attitude Control of Micro Air Vehicles. Journal of Guidance, Control, and Dynamics. 39. 1-12. 10.2514/1.G001490.

[6] Mellinger, Daniel et al. "Trajectory generation and control for precise aggressive maneuvers with quadrotors." The International Journal of Robotics Research 31 (2010): 664 - 674.

[7] Bitcraze, "Adding a new system task", *bitcraze.io*, https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/development/systemtask/. Accessed 06 December 2022

[8] FreeRTOS, "Tasks and Co-routines", *freertos.org:*, https://www.freertos.org/taskandcr.html. Accessed 29 November 2022.

[9] Github, "bitcraze/crazyflie-firmware", *github.com*, https://github.com/bitcraze/crazyflie-firmware/. Accessed 29 November 2022.

[10] FreeRTOS, "RTOS xTaskCreate", *freertos.org*, https://www.freertos.org/a00125.html. Accessed 29 November 2022.

[11] Bitcraze, "Expansion decks of the Crazyflie 2.X", *bitcraze.io*, https://www.bitcraze.io/documentation/system/platform/cf2-expansiondecks/. Accessed 03 December 2022

A xTask Callers

A.1 Functions calling xTaskCreate

- *activeMarkerDeckInit*
- *bigQuadInit*
- *flowdeckInit*
- *flowdeck2Init*
- *gtgpsInit*
- *lighthouseInit*
- *drm1000Init*
- *mrInit*
- *oaInit*
- *Init*
- *uart1testInit*
- *uart2testInit*
- *usdInit*
- *usdLogTask*
- *zRangerInit*
- *zRanger2Init*
- *cpxInit*
- *cpxExternalRouterInit*
- *cpxUARTTransportInit*
- *dtrStartProtocolTask*
- *ledSeqInit*

A.2 Functions calling xTaskCreateStatic

- *pmInit*
- *sysLinkInit*
- *crtpserviceInit*
- *logInit*
- *memInit*
- *paramInit*
- *platformserviceInit*
- *estimatorKalmanTaskInit*
- *sensorsTaskInit*
- *stabilizerInit*

B PID `Controller` constants

B.1 Position Controller

- X
  - $K_p = 2.0$
  - $K_i = 0.0$
  - $K_d = 0.0$
- Y
  - $K_p = 2.0$
  - $K_i = 0.0$
  - $K_d = 0.0$
- Z
  - $K_p = 2.0$
  - $K_i = 0.5$
  - $K_d = 0.0$

B.2 Velocity Controller

- V_x
  - $K_p = 25.0$
  - $K_i = 1.0$
  - $K_d = 0.0$
- V_y
  - $K_p = 25.0$
  - $K_i = 1.0$
  - $K_d = 0.0$
- V_z
  - $K_p = 25.0$
  - $K_i = 15.0$

- $K_d = 0.0$

B.3 Attitude Controller

- Roll
  - $K_p = 6.0$
  - $K_i = 3.0$
  - $K_d = 0.0$
- Pitch
  - $K_p = 6.0$
  - $K_i = 3.0$
  - $K_d = 0.0$
- Yaw
  - $K_p = 6.0$
  - $K_i = 1.0$
  - $K_d = 0.35$

B.4 Rate Controller

- Roll Rate
  - $K_p = 250.0$
  - $K_i = 500.0$
  - $K_d = 2.5$
- Pitch Rate
  - $K_p = 250.0$
  - $K_i = 500.0$
  - $K_d = 2.5$
- Yaw Rate
  - $K_p = 120.0$
  - $K_i = 16.7$
  - $K_d = 0.0$

C *src/* sub-directory tree in the root folder

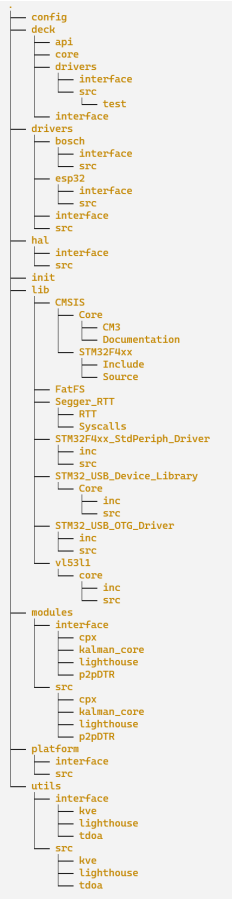


Figure 7 */src/* tree