Technische Universiteit
**Eindhoven**
University of Technology

Embedded Visual Control
5HC99

# Design and Implementation of a Quadcopter with Visual Control

By
Stef Louwers (0590864)
Sunil Chokkanathapuram Ramanarayanan (0826874)
Qian Qian (0827493)
GROUP 3

**Date:** October 15, 2013
**Professor:** Prof. dr. Henk Corporaal, Lr. Roel S. Pieters, MSc, Lr. Mark Wijtvliet, MSc

## Contents

# 1 Introduction

In this project, a quadcopter is designed and implemented to autonomously fly and demonstrate on-board image processing capabilities. The project is aimed to control the quadcopter by virtue of two control systems; the local control, and the global control. The local control is intended for the stabilisation of the quadcopter, so that it is able to fly manually through the RC receiver and the global control is intended to make the quadcopter to fly autonomously with the aid of visual control through the camera. These two control loops are implemented on dedicated processors with appropriate capabilities.

# 2 System Architecture

In this section, we will discuss the used hardware and system architecture. First we will take a look at the local control, global control and used sensors, and after that, we will address communication used between these components.

## 2.1 Local Control

The local control is powered by an STM32F3Discovery development board, powered by the STM32F303VCT6 processor (a 72 MHz Cortex-M4 MCU), 48 KB RAM and 256 KB flash memory. This board also contains a gyroscope, accelerometer and a magnetic sensor, all measuring three axis. This makes it an ideal board for a quadcopter's local control, because it contains enough processing power, and has all the basic sensors available.

## 2.2 Global Control

For the global control, we have chosen for the Raspberry PI. We chose for this platform because it was cheap, we already had it available, and the camera module was expected soon after the start of the project.

Because we wanted two cameras on the quadcopter (one facing to the front, and one facing down), we decided to add two Raspberry PIs to the quadcopter, as each Raspberry PI only has the possibility to connect one camera. This also gave us more processing power available, as each Raspberry PI only has to process one camera.

## 2.3 Sensors

The local control board already contained the basic sensors required to fly a quadcopter: a gyroscope, an accelerometer and a magnetic sensor. To this list, we added a battery monitor, a barometer and two cameras.

### 2.3.1 Battery monitor

The battery monitor was a simple voltage divider that connected the battery voltage to one of the ADC pins of the STM32F3Discovery board. The voltage divider was necessary to convert the voltage to the 0 – 3V domain.

### 2.3.2 Barometer

In order to facilitate altitude hold, we needed some means of measuring the current altitude of the quadcopter. Because other teams have reported some problems with the ultrasound sensor, we decided to do something different and try a barometer.

Compared to an ultrasound sensor, a barometer has less accuracy. Also, a barometer provides only a absolute height (to sea level). Thus a command like "stay at 1 meter from the ground" is not possible. Instead, we should say "stay at the current height". The advantage of the barometer over the ultrasound sensor is that it has no height limits, and that there are no false measurements as reported with the other groups.

### 2.3.3 Cameras

We have chosen for the Raspberry PI cameras, because they are affordable, of relative good quality, small and light, and they uses few system resources on the Raspberry PI. Because each Raspberry PI can only interface with one camera, we have placed two Raspberry PIs and cameras on the quadcopter.

These cameras are able to film in 1080p resolution with 30 fps, and take still images with 2592 x 1944 resolution.

## 2.4 Communication

There are many communication-paths, as can be seen in figure 1. In this section, these will all be discussed.

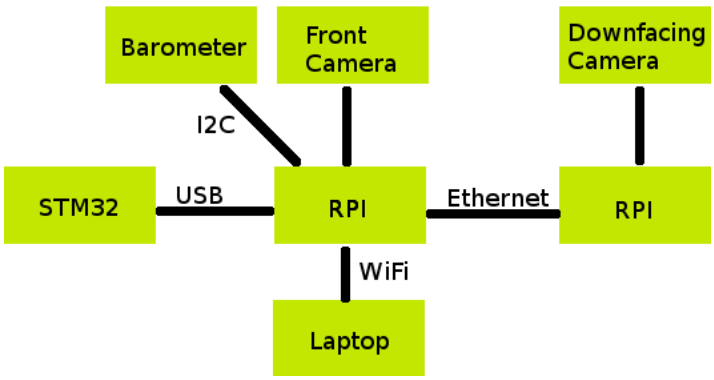


Figure 1: Overview of different communication protocols.

### 2.4.1 Local control and global control

The local control on the STM32F3Discovery communicates with the first Raspberry PI using an USB-cable. They use the UAVTalk[1] protocol, and thus communicate high level objects. Because the UAVTalk protocol is already seamless integrated in the local control (it is also used for communication with the Ground Control Station), all useful pieces of data are available as a UAVObject.

This makes it an extremely powerful interface. Global control algorithms have access to all the information that it available to the local control, and it can influence and override all aspects of the local control by writing new values to these objects.

The interface is also flexible. Currently it uses USB, but UART or other protocols are also possible. The UAVObjects on the local control are also accessible on a laptop and on the other Raspberry PI, using a WiFi or an ethernet connection, thus eliminating the need for a separated (bluetooth) telemetry module.

A presentation on this topic is also available online on `http://youtu.be/uEVFPBQtt0U`.

### 2.4.2 Raspberry PIs

The two Raspberry PIs on the quadcopter need to be able to communicate, and they do this by a standard ethernet (network) cable. We chose for this option because it is supported out of the box and widely supported, and it is fast. Secondly, it allows the first Raspberry PI to share its WiFi connection easily with the second Raspberry PI, allowing easier remote management.

### 2.4.3 Laptop

On the Raspberry PI runs a normal Linux installation, and that gives the option to add a WiFi adaptor so that it can communicate with the outside world using WiFi. We did this, and even shared the internet connection to the second Raspberry PI, so both could easily download and install new software.

### 2.4.4 Barometer

The barometer is connected to the first Raspberry PI using $I^2C$. It is not connected to the STM32F3Discovery, because there where some problems with the $I^2C$ driver in the TauLabs system, that we where unable to properly debug. The Raspberry PI sends the altitude information from the barometer to the STM32F3Discovery using the UAVObjects and the UAVTalk protocol, as discussed in section 2.4.1.

[1] `http://wiki.openpilot.org/display/Doc/UAVTalk`

### 2.4.5 Camera

The cameras are attached to the Raspberry PIs, and use a flatcable for the connection to a CSI[2]-interface.

## 3 Software

To get all this hardware flying, we need some software to control everything. We have software for local and global control, and both will be discussed in this section.

### 3.1 Local control

For the local control, we chose for the open source project TauLabs[3]. This project is a fork from the OpenPilot project. We choose for this project, because is was already ported to the hardware we where using (the STM32F3Discovery).

This project provided not only software for the local control, there was also a "Ground Control Station" (GCS). This program runs on your computer and communicated with the local control to provide configuration options and means of debugging and monitoring.

### 3.2 Global control

We have written our own global control program. This program runs on a Raspberry PI, and is a stripped down version of the GCS provided by TauLabs. This allows it to communicate with the local control using UAVObjects, as is discussed in section 2.4.1. Our global control code is also available on GitHub[4].

The global control has a fixed core, and allows functionality to be added by modules. We did not have enough time to write really interesting modules, but we do have some "proof-of-concept" modules, which will be discussed next. These plugins show that it is possible to read and write data from/to the local control.

#### 3.2.1 Recorder plugin

This plugin monitors the "Armed" flag in the "FlightStatus" UAVObject, and as soon as the quadcopter gets armed, it starts recording video on both Raspberry PIs. On disarming, the recording stops. This allows us to easily create in-flight footage with the cameras.

[2] Camera Serial Interface
[3] `https://github.com/TauLabs/TauLabs`
[4] `https://github.com/fhp/TauLabs`

### 3.2.2 Barometer plugin

The barometer plugin reads the barometer that is connected to the Raspberry PI using $I^2C$. It parses the results and writes the resulting altitude information to the "BaroAltitude" UAVObject that will then be send to the local control.

## 4 Objectives

### 4.1 Building Quadcopter and flying

### 4.2 Tuning Local control

### 4.3 Autonomous flight

### 4.4 360 degrees rotation

### 4.5 Altitude hold

### 4.6 Drift correction in horizontal axis

### 4.7 Camera capture while rotation

### 4.8 Stiching for Panorama image

### 4.9 Face detection

During our development period of the quadcopter, the Raspberry PI Foundation has released a camera module (figure 2) which fits specifically into Raspberry PI's CSI interface. The camera has a 5 megapixel sensor and supports 1080p, 720p, and 640x480p video. The footprint dimensions are 25 x 20 x 9 mm. Compared to a usb webcam which is widely used for personal computers, the Raspberry PI camera has a relatively smaller size and Raspberry PI has opensource driver for the camera which gives larger throughput in terms of image processing. So we decided to use the Raspberry PI camera as the front facing camera.

The goal of this face detection task is to make the quadcopter follow and track the movement of a human face while flying. We intended to use OpenCV[5] library as a tool to implement the face detection program on Raspberry PI, and after processing the image, we will find the position of the human face relative to the boundary of the image. Based on that position we can decide whether the quadcopter should fly higher or lower, go to the left or to the right. Moreover, and estimated distance of the face from the camera can be calculated by measuring the area of the face within the image taken by the camera. So the quadcopter can track the face when the face is moving away or towards the quadcopter.

However, when we implement the face detection program, we found out that the API from OpenCV library which will invoke the camera to take video is working. Because the Raspberry

[5] `http://opencv.org/`

Figure 2: Raspberry PI with camera module.

PI camera uses a different driver model as the conventional usb webcams, the API which is responsible for invoking the camera was not able to locate the camera.

In the Raspbian system, we where provided with two commands to operate the camera; raspivid and raspistill. raspivid is a command line application that allows you to capture video with the camera module, while the application raspistill allows you to capture images.

In order to extract the driver, we looked at the source code of the two applications in the Raspberry PI github repository[6]. We figured out that the Raspberry PI system uses MMAL functions to communicate with the camera. And we managed to trim down the the camera driver and integrated it into the face detection code. By feeding the haar cascade filter with the image get from the camera, the program is able to detect faces in the image. The output of the program is the coordinate of the face within the image. Figure 3 shows that the Raspberry PI can successfully detect faces.

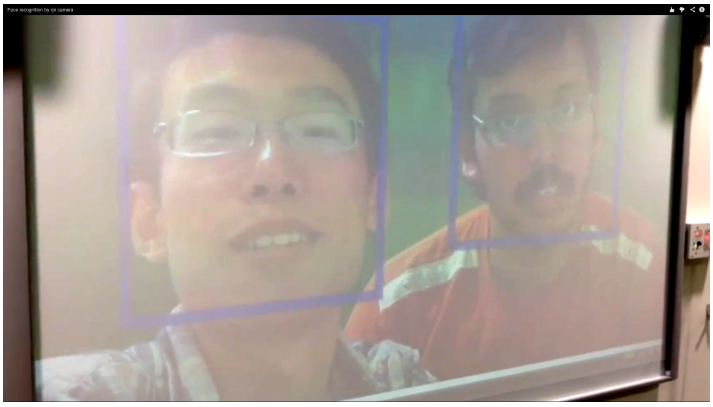

Figure 3: The Raspberry PI has detected faces.

[6] `https://github.com/raspberrypi/userland/tree/master/host_applications/linux/apps/raspicam`

Since the face detection program is based on the haar cascade filter, it will have some drawbacks. First, it will be able to find multiple faces in the camera scene. This may seem to be an advantage, but if two faces appear on the scene, then the quadcopter may not know which one to track, so it may lead to unexpected results. Furthermore, according to the way that haar filter works, it will only detect a face that is placed vertically straight in the image, other orientations will not be detected.

### 4.10 Marker identification

The expected output of our marker detection is that the quadcopter can use the front facing camera to detect and recognise some specific markers on the wall, so that it can do some pre-programmed action, that depends on the detected marker.

As an aerial vehicle, a quadcopter is very susceptible to disturbances from the surrounding. So a quadcopter has to be constantly tuning its position according to the data from the sensors in order to get its stable position. This has disadvantages for the cameras, because for the cameras, a constantly moving quadcopter make the image blurred and therefore make the detection harder. In order to confront this problem, we need to make the markers robust, so that even for a moving quadcopter, the camera can still recognise the markers with high detection rate.

What we use is called ArUco[7]: a minimal library for Augmented Reality applications based on OpenCV. It is a open-source marker detection library developed by University of Cordoba. The library uses markers as shown in figure 4.

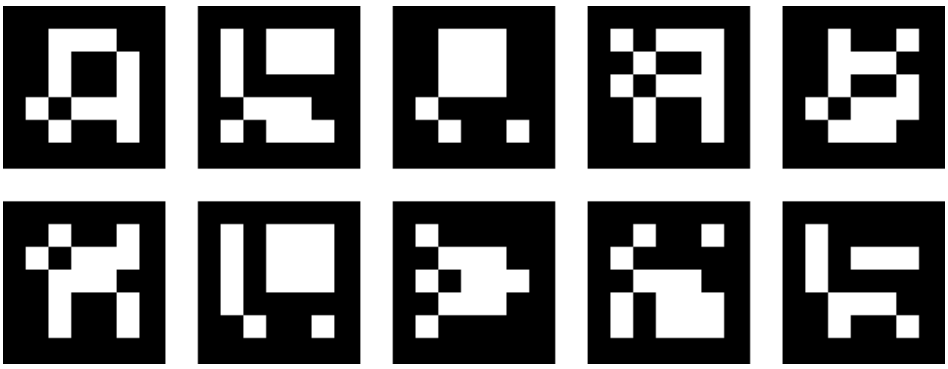

Figure 4: ArUco markers

Each marker has an internal code given by 5 words of 5 bits each (as shown in figure 5). The codification employed is a slight modification of the Hamming Code. In total, each word has only 2 bits of information out of the 5 bits employed. The other 3 are employed for error detection. As a consequence, we can have up to 1024 different ids. With the help of Hamming coding, the marker is able to tolerate more errors, and the robustness of the detection is increased.

The main difference with the Hamming Code is that the first bit (parity of bits 3 and 5) is inverted. So, the id 0 (which in hamming code is 00000) becomes 10000 in our codification.

[7] http://www.uco.es/investiga/grupos/ava/node/26

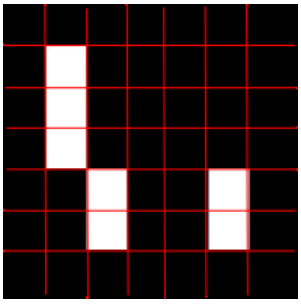

Figure 5: Marker structure

The idea is to prevent a completely black rectangle from being a valid marker id with the goal of reducing the likelihood of false positives with objects of the environment.

Since we already have the experience of face detection program, we successfully integrated the marker detection code with raspberry pi camera driver as well. As a result of the program it can detect the marker printed on a paper as well as knowing the corresponding id of this marker (figure 6). Based on the id, the quadcopter can be assigned to do some pre-defined actions like turn around or go forward, etc.

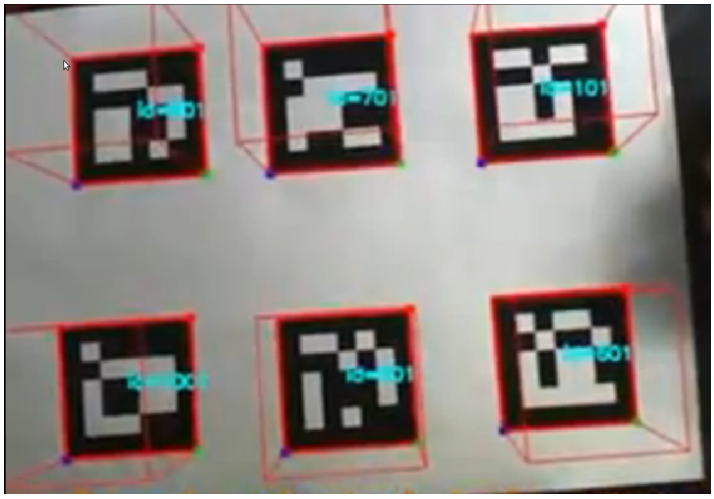

Figure 6: Detected markers

# A Who did what?

## A.1 Hardware

| | |
|---|---|
| Building the basic quadcopter | Stef, Sunil, Qian |
| Basic PID tuning | Stef, Sunil, Qian |
| More PID tuning | Stef |
| Attaching the cameras | Stef, Sunil |
| Battery monitor | Stef |
| Barometer | Stef |

## A.2 Software

| | |
|---|---|
| UAVTalk on Raspberry PI | Stef |
| Global control | Stef |
| Face recognition | Qian |
| Marker recognition | Qian |

## A.3 Report

| | |
|---|---|
| System Architecture (section 2) | Stef |
| Software (section 3) | Stef |
| Face and marker detection (section 4.9, 4.10) | Qian |

# B Partlist