Design of a vision-enabled quadrocopter

Sander van Gameren (0607274)
Patrick Wijnings (0660382)

September 21, 2012

Contents

I. Introduction 6

1. Introduction 7

II. Hardware platform 10

2. Hardware platform overview 11
   2.1. Hardware components of the vision 12

3. The low-level embedded system: Arduino and the Aeroquad code 13

4. ESC control using the Arduino 15
   4.1. Arduino ESC signaling 16

5. Hardware selection: power and weight balancing 18
   5.1. Propeller selection 20
   5.2. Carbon fibre frame design and construction 21

III. Local control 23

6. The Aeroquad code: overview and operation 24
   6.1. The Aeroquad control loop 26
   6.2. Interfacing with the Aeroquad code 27

IV. Vision system 29

7. BeagleBoard explored: software-hardware interface 30
   7.1. DM3730 architecture 30
   7.2. Memory management and caching 33
   7.3. Start-up sequence 33

8. Toolchain set-up: abstraction of the software-hardware interface 35
   8.1. Choice of operating system 35
   8.2. Installing Angstrom Linux on target 36

8.3. Cross-compiling for Angstrom on Linux host 38
8.4. Accessing the DSP 40
8.5. Texas Instruments DSP libraries 43
8.6. Automating the compilation process: Makefile design 43
8.7. Limitations and caveats of the toolchain 48

9. Visual odometry: camera-based position tracking **50**
9.1. Algorithm selection 50
9.2. Step 1: Image Preprocessing 53
9.3. Step 2: Feature Extraction 54
9.4. Step 3: Initial Rotation Estimation 55
9.5. Step 4: Feature Matching 56
9.6. Step 5: Inlier Detection 56
9.7. Step 6: Motion Estimation 57

10. Algorithm implementation: from mathematics to real-time code **59**
10.1. Balancing between ARM and DSP 59
10.2. Step 1: Image Preprocessing 60
10.3. Step 2: Feature Extraction 61
10.4. Step 3: Initial Rotation Estimation 62
10.5. Step 4: Feature Matching 63
10.6. Remaining steps 63

V. Conclusion **64**

11. Conclusion **65**

12. Recommendations **66**

List of abbreviations

DSP Digital signal processor
MMU Memory management unit
MPU Microprocessor unit
IVA Image, video and audio accelerator
GPIO General-purpose input/output
TLB Translation look-aside buffer
IPC Interprocessor communication
VFP Vector floating point
SIMD Single instruction multiple data
OS Operating system
UAV Unmanned aerial vehicle
DOG Degrees of freedom
ESC Electronic speed controller
BLDC Brushless DC motor
PPM Pulse phase modulation
PWM Pulse width modulation
DCM Directed cosine matrix

Part I.

Introduction

1. Introduction

Recently, interest in unmanned aerial vehicles (UAVs) has increased a lot: due to advancements in technology, UAVs have become much more versatile as well as affordable. Besides military applications, UAVs are also suitable for numerous civil applications. Examples include: surveillance, search and rescue, wild fire suppression and formation of ad-hoc communication networks.[23] Additionally, UAVs are well suited as challenging platform for researchers, because many disciplines are united in them, such as:

- mechanical engineering (e.g. hardware design);
- control engineering (e.g. stabilization of the vehicle);
- power electronics (e.g. battery design and powertrain);
- algorithm design (e.g. object recognition or swarm intelligence);
- and embedded engineering (e.g. providing a platform for above algorithms).

At this point, a brief intermezzo about manned aerial vehicle designs will be made to illustrate the incredible amount of design parameters of aerial vehicles. Besides the well-known designs from e.g. the brothers Wright or Anthony Fokker, many more less-known designs can be found throughout the chronicles of history, often with a quite unconventional form-factor. The first of these designs range from the time of Leonardo DaVinci and new vehicles are still being designed today. Unfortunately, many have never managed to lift off: sometimes because of technical impossibilities, sometimes because of a lack of commercial adoption, but most often because of a combination of these two. Consider for example the Avro Canada VZ-9 Avrocar from the 1950s (figure 1.1). This 'flying saucer' started as an idea by aircraft designer John Frost, and managed to attract interest from the U.S. Air Force. However, many years and numerous stability and performance problems later, funding from the U.S. Air Force ran out and the design was finally abandoned.[24]

A similar fate almost happened to the manned quadrocopter designs from the 1920s and 1930s (figure 1.2). As described in [25], "early prototypes suffered from poor performance, and latter prototypes required too much pilot work load, due to poor stability augmentation and limited control authority." The quadrocopter would never have gotten popular, were it not for recent technological advancements in inertial measurement sensors, electrical motors and low-cost computing. Today, the quadrocopter has become one of the most popular UAVs. Its main advantages are its great maneuverability, allowing for both indoor and outdoor deployment; and its ease of construction when compared to e.g. helicopters with mechanically much more involved rotor design. Because of this, it has been embraced by both the hobbyist and academic communities.

Figure 1.1.: Avro Canada VZ-9 Avrocar. Picture taken from [24].

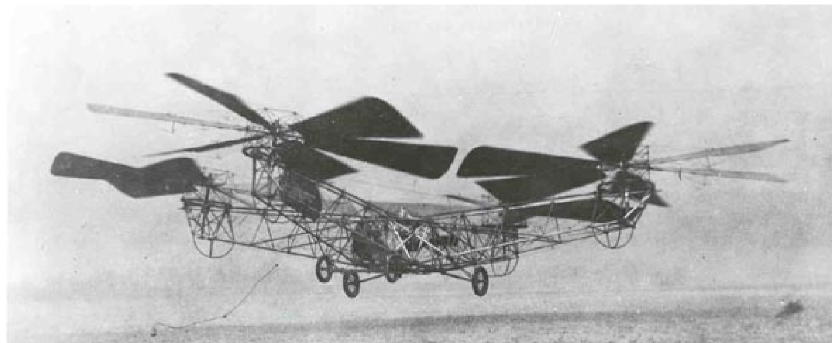


Figure 1.2.: De Bothezat Quadrotor, 1923. Picture taken from [25].

The authors of this report decided to design and build a vision-enabled quadrocopter UAV in the context of the *Embedded Visual Control* course [26] as taught at Eindhoven University of Technology. However, in spite of the large amount of hobbyist and academic literature, this turned out to be much harder than expected. Even when just considering the quadrocopter UAV design, the number of possible design choices is still incredibly high. Also, as mentioned above, these design choices often span multiple disciplines. We were not very familiar with some of these disciplines, such as mechanical engineering and power electronics. Furthermore, as always, Hofstadter's law[1] holds for this project as well.

In the end, like the many forgotten vehicles from history, our quadrocopter never managed to make a controlled flight. However, with this report we aim to prevent our gained knowledge from being forgotten and to document our design choices. This report aims to help the reader understand which trade-offs need to be made, so that he can hopefully get his own quadrocopter up in the air. Do note, however, that no significant academic improvements are described in this report: most algorithms and techniques that will be used have already been described in other papers. It is by explaining and combining these ingredients that this report aims to gain its value.

This report has been divided in several parts. In part II, the hardware design will be elaborated. Topics include frame design, choice of motors and propellers, and avionics selection. Then the low-level control will be explained in part III. Further on, in part IV, focus will be on the software layer of the quadrocopter. Among other things, it contains a description of the embedded platform with its toolchain, the vision algorithm, and its implementation. Finally, part V will contain notes about performance and our recommendations as based on our experiences.

[1] Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

Part II.

Hardware platform

2. Hardware platform overview

The main emphasis of our project lies on the embedded visual control of quadrotors. However, before we can focus on this, a reliable hardware platform must be assembled. This platform should be as 'ready-to-fly' as possible, so that we can spend more development time on the main emphasis of the project. On the other hand, the platform should still be customizable enough so that it can handle our quite heavy payload (see section 2.1 and chapter 5 for details). Furthermore, we also want to learn something about the hardware of a quadrocopter. This means that a complete off-the-shelf solution is not the best option. Instead, it was decided to use a modular quadrotor system as base. In this way, all the modules can still be selected and customized, but it is not needed to completely reinvent the wheel.

There are several companies that offer such modular quadrotor systems. The best system is Aeroquad. The main advantages of this system are its hardware modularity and the use of an Arduino as embedded system. Arduino code is easy to learn and extended documentation is available for it. The Aeroquad system can fly any quadrotor weighting from 200 grams till 2+ kilograms. The Aeroquad system ranks hardware in three categories:

1. Priority one: the embedded system, Arduino, and sensorshield.
2. Priority two: the motors, ESC's, batteries, frame, etc.
3. Priority three: battery chargers, power distribution, etc.

There are several choices in each category. For the priority one components, Aeroquad v1.9 is the best choice. Other versions with more advanced sensors (e.g. compass, height or even GPS) are also available, but these are significantly more expensive. This system contains a sensor shield and an Arduino. The sensor shield has to be assembled on you own. The code operating on the Arduino is available on the Aeroquad website for free. Next to that, the forums offer 24H support for the system. Also, a tutorial is available for how to assemble to whole system on the website. The operation and the use of the code will be discussed globally in chapter 3 and more extensively in part III. In section 4 it will be discussed how to control ESCs using an Arduino.

After selecting the priority one core components, a selection for the priority two modules has to be made. In order to do so, a weight estimation is required. From that estimation the system can be iteratively designed. We start the iteration by weighting the most important components. After the first iteration more components will be added and after a few iterations the design will be complete. This iteration will be further discussed in section 5. The remainder of this chapter will focus on the components required for the vision system.

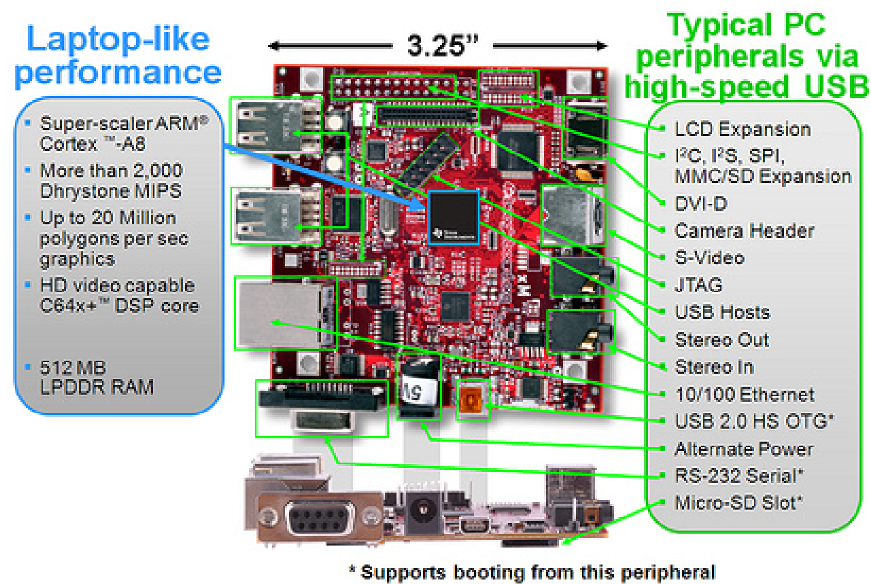


Figure 2.1.: BeagleBoard-Xm specifications. Taken from [43].

2.1. Hardware components of the vision

To make the quadrocopter vision-enabled, a suitable camera and processing platform must be selected.

For the camera, we opted for the Microsoft Kinect. This camera can capture 640x480 RGB images as well as a 640x480 depth map. This depth map is created using stereo triangulation technology from PrimeSense. See [42] for a more detailed explanation. Using the Kinect has two advantages. Firstly, the depth map can be used cleverly by the vision algorithm (see part IV for details). Secondly, because the Kinect is intended for consumer usage, it is quite affordable and there are a lot of software libraries available for it. Its most important disadvantage is its weight. Luckily, it is possible to strip down the Kinect and remove unnecessary components, such as the 4-microphone array and motorized stand. (However, it should be noted that this process of stripping down is quite difficult, because Microsoft put a lot of effort in making the device tamper-proof. For example, Security Torx screws are used all over the place and the housing is glued together so that it is difficult to take apart. Also, the USB connector is non-standard and uses an additional 12V line.)

As processing platform the BeagleBoard-Xm was selected. Consult figure 2.1 for its specifications. This board has quite a lot of processing capacity and connectivity, but still a small form factor and low power requirements. Other candidates included the PandaBoard (faster, more expensive), Raspberry Pi (very good price, but out of stock at the time we needed it) and the Gumstix (much smaller, but more expensive and with less documentation).

3. The low-level embedded system: Arduino and the Aeroquad code

To properly use the Aeroquad code, the Arduino IDE must be installed first. The Arduino IDE can be obtained for free from [36]. After installing the IDE make sure to properly configure the drivers so that Windows is able to connect to the Arduino. Then download the Aeroquad configurator and flight software. These are available from [37]. The configurator is a program used to configure and calibrate, for example: the ESCs, sensors, controller parameters and much more. The flight software is the software that operates on the Arduino. Make sure to select the *AeroQuadConfiguratorFull_ v3.0.2Win.zip* configurator and *AeroQuad v3.0.1 with bug fixes* for the flight software. These two are the most stable versions.

Be sure to install the configurator first. After that, upload the flight software using the configurator. The interface of the configurator will look as depicted in figure 3.1. After that, the settings need to be configured. First go to the *Upload flight software* button in the upper left corner. The screen from figure 3.2 will appear. In this screen, the Aeroquad uploader, options and configurations can be selected. The attitude calculation algorithms are the first three select boxes: *MARG*, *ARG* and *Rate mode*. The attitude can be calculated using several sensors and mainly two algorithms. The used algorithms are either: quaternion model derivation or directed cosine matrix (DCM).

Each of the algorithms (*MARG*, *ARG* or *Rate mode*) calculates the attitude in a different way. MARG and ARG use quaternion model derivation for the attitude calculation. *Rate mode* uses DCM to calculate the attitude. Furthermore, each algorithm uses different sensors. MARG uses magnetometers, accelerometers and gyroscopes, while *Rate mode* uses only a gyroscope. In our case, for the Aeroquad v1.9 shield, select the ARG mode.

Select for Flight Control Board: Aeroquad v1.8 or greater. Then, select the appropriate com port. It is then possible to change the propeller configuration by clicking on the image. In our case select the Quad X configuration. After that select upload. The code will then be compiled and uploaded to the Arduino.

Now all the sensors can be calibrated using the appropriate buttons in the left hand of the screen. In the upper part of the screen, different setup menus can be selected. At the beginning the initial setup is selected. This can be changed to for example motor tests, serial monitor or vehicle attitude.

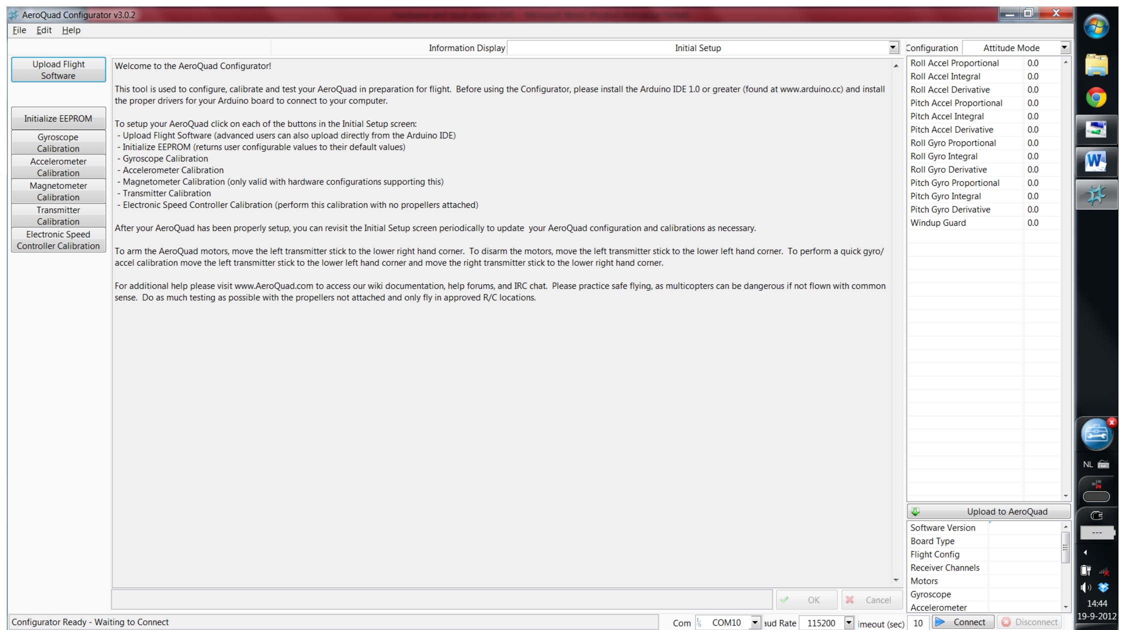


Figure 3.1.: Interface of Aeroquad configurator.

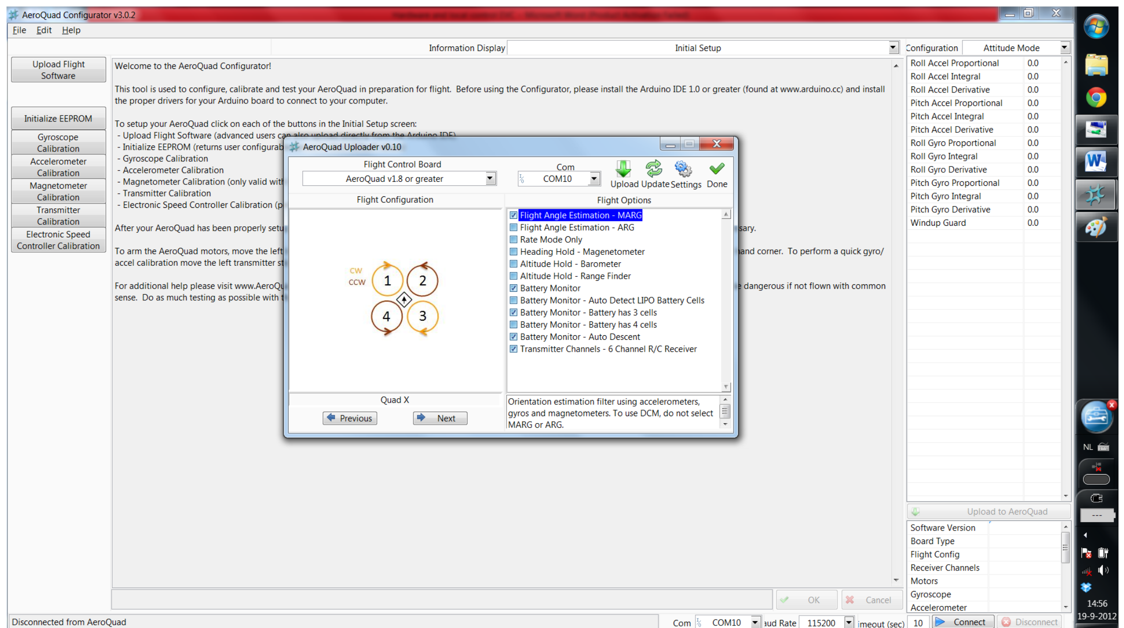


Figure 3.2.: Interface of Aeroquad configurator after *Upload flight software* has been chosen.

4. ESC control using the Arduino

This chapter contains a brief intermezzo on how to interface with the ESC (electronic speed controller) modules using the Arduino. Besides providing some insight in how the Aeroquad code works, this intermezzo can be especially useful when one wants to test the motors without using the complete Aeroquad code.

BLDC motors are controlled using ESCs. However, these ESCs need a reference input in order to control. This reference input can be generated by an Arduino. The signal pin of the signal connector of the ESC will be connected to a PWM pin of the Arduino. The two other pins of the signal connector are a +5 voltage supply pin and a ground pin. The 5 Volt pin can be used to supply the Arduino. (Do note that this is not advised, due to some instability in the voltage at high rotational speeds of the motor.) The ground pin will be connected to the Arduino ground.

A BLDC motor is a type of synchronous motor. All synchronous motors are controlled using ESCs (electronic speeds controllers) or EPCs (electronic position controllers). The feedback signal from the motor to the controller is a back EMF signal.

There is an Arduino library for servo control, servo.h. The header allows to define servo objects with several functions related to it. An example of such code is shown below:

```
1  #include <Servo.h>
2
3  Servo myservo1;
4  int val;
5
6  void setup() {
7      myservo1.attach(3);
8      for(val = 7; val <= 8; val++) {
9          myservo1.write(val);
10         delay(1000);
11     }
12 }
13
14 void loop() {
15     for(val = 63; val <= 78; val++) {
16         myservo1.write(val);
17         delay(5000);
18     }
19 }
```

Line number three defines the object *myservo1*. The Arduino is capable of controlling four servos in total. After this the setup function is initiated. In the Arduino code the setup usually arms or initiates the peripheral hardware. In this case it arms the

ESCs: most ESCs must be armed before the motors can actually be started. The arming functions as synchronization between the Arduino and the ESC but serves also as a sign to the user that the motor is armed and dangerous.

It does so by first attaching the servo to the pin it is connected to on the Arduino board. After that itsends two specific signals. The send function is *myservo1.write*. These signals sent to arm, have values 7 and 8.

The arming values, seven and eight in this case, are found by looping through all the possible values. (The user manual of the ESCs do not contain these values, so they must be found by trial and error.) The range of all possible values is from 0 to 180 degrees. The range is 0 to 180 degrees because the BLDC-motor is treated as a servo motor. The relation between degrees and motor speed is derived in the following manner:

- The maximum speed of the motor is 6000 rpm.
- The minimum speed is 100 rpm.
- The motor starts to rotate from 20 degrees till 180 degrees.
- The required speed is 3000 rpm.
- The required value in degrees will then be: $(160/5900) \times 3000 = 81.3559 = 82$degrees.

The *delay(1000)* is a delay of one second. This is long enough for the motor to effectively arm. While the delay is active the Arduino will keep writing the signal on pin three to the motor.

After the for loop of the setup function is finished the code will continue to the loop function. The Arduino will operate this part of the code indefinitely. In this case the Arduino will send out values between 63 and 78 degrees. That is approximately 100 rpm to 2000 rpm with the motor/ESC combination used.

4.1. Arduino ESC signaling

To clarify, there is a lot of debate online about what kind of signal is sent over the signal line from the Arduino to the ESC (e.g. [39] and [40]). Some debate that it is PWM (pulse width modulation) and some say it is PPM (pulse phase modulation). Both sides are right. Figure 4.1 explains the situation.

PPM is a method to multiplex several data signals over one link. This is shown in the first line of the picture. After the PPM signal is received, it is decoded into several PWM signals, represented by the PPM frame decoder output. The length of each PWM signal is determined by the next rising edge. The last bar shows a synchronization signal. This signal is used to let the decoder know when a new sequence of signals is arriving. In our case, the PWM signals are used to directly control the motor with the ESC.

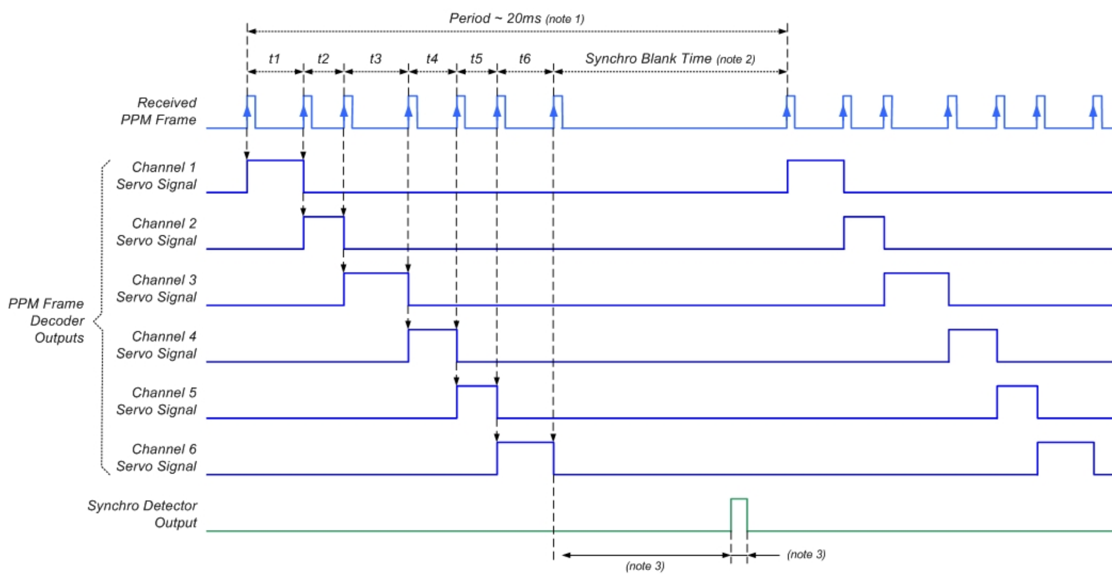


Figure 4.1.: Signals between Arduino and ESC. Picture taken from [41].

5. Hardware selection: power and weight balancing

Now that the priority one core has been chosen in chapter 2 and ESC operation has been explained in chapter 4, it is time to select the remaining hardware components. These selected components have to satisfy one main goal: sustain flight for at least 10 minutes. To achieve this, an initial estimation will be made of the total weight. The initial estimation is based on the components that are already in the design and estimated additional amount of weight for a realistic model. From that point on, multiple iterations are made to achieve the optimal power to weight ratio.

As written in the introduction of this part, the Aeroquad categorizes the required hardware in the following way:

1. Priority one: the embedded system, Arduino en sensor shield.
2. Priority two: the motors, ESC's, batteries, frame, etc.
3. Priority three: battery chargers, power distribution, etc.

To make a proper weight estimation, the essential components that have already been chosen are measured:

- BeagleBoard-Xm and Kinect: 550 grams total weight. This weight includes the cables.
- Expected frame weight: 200 grams. Constructed of carbon fibre material.
- Heavy duty BLDC motors, as advised on the Aeroquad website: 80 grams a motor.
- ESC's for that motor: 45 grams for each ESC.
- Total weight of: 1302 grams.

This selection excludes battery weight. In order to have at least ten minutes of sustained flight a large battery capacity is needed. Estimating that the total weight of quadrotor is about 2000 grams, including batteries, each motor should at least generate 1000 grams of thrust to effectively *lift* the quadrotor into the air. This allows to suitably select the propellers. The choice for the propellers is discussed thoroughly in section 5.1. For now it is enough to state that the selected propeller weights 58 gram a piece. This changes the weight overview to:

- BeagleBoard-Xm and Kinect: 550 grams total weight. This weight includes the cables.
- Expected frame weight: 200 grams. Constructed of carbon fibre material.
- Heavy duty BLDC motors, as advised on the Aeroquad website: 80 grams a motor.
- ESCs for these motors: 45 grams for each ESC.

- Propellers for delivering sufficient thrust, 14x3.7 inch: 58 grams for each propeller.
- Total weight of: 1534 grams.

For *sustained* flight, for a total quadrotor weight of about 2000 grams, 500 grams of lift per motor should be enough. At 500 grams of thrust a motor will draw about twelve amperes from the batteries and each motor has an operating voltage of 11.1 volts. The motors consume a total amount of: $11.1 \times 12 \times 10 \times 60 \times 4 = 319680$J.

The BeagleBoard, the Arduino and the Kinect draw together three amperes at 5 volts. These three consume a total amount of: $5 \times 3 \times 10 \times 60 = 9000$J.

The total power consumed in ten minutes is then: $9000\text{J} + 319680\text{J} = 328680\text{J}$.

The battery that suits this requirement the best is the Zippy Flightmax 4000 mAh and 7.4 V. Four of these should balance out the quadrotor and deliver the required power. The total battery power is: $7.4 \times 4000 \times 3.6 \times 4 = 426240$J.

This calculation did not take into account that lifting off requires much more power than stable hovering. However an additional amount of at least 97 KJ is present in the batteries if the quadrotor only hovers. This 97 KJ should be more than enough to lift the quadrotor in the air.

The four Zippy Flightmax batteries weigh 765 grams. The total weight overview now becomes:

- BeagleBoard-Xm and Kinect: 550 grams total weight. This weight includes the cables.
- Aeroquad shield and Arduino: 52 grams.
- Expected frame weight: 200 grams. Constructed of carbon fibre material.
- Heavy duty BLDC motors, as advised on the Aeroquad website: 80 grams a motor.
- ESC's for that motor: 45 grams for each ESC.
- Propellers for delivering sufficient thrust, 14x3.7 inch: 58 grams for each propeller.
- Four Zippy flightmax batteries 4000 mAh 7.4 V: 768 grams.
- Estimated weight of additional cabling and voltage regulator: 200 grams.
- Total weight of: 2502 grams.

This is much more than the estimated 2000 grams. In order to reduce the weight the following things were considered:

- Reducing the weight of the frame even more by constructing it ourselves from carbon fibre. We successfully constructed our own frame with a total weight of 140 grams. The frame construction will be discussed in paragraph 1.4.2.
- Stripping down the Kinect and shortening and modifying its cable: Stripping down the Kinect and modifying the connection cable reduced the total weight of the Kinect by an additional 250 grams. The total weight of the Kinect and BeagleBoard is now 300 grams.
- Shortening the USB cable between the BeagleBoard and the Arduino: Creating a very small connector cable between the two reduced the weight by 50 grams.

These improvements reduce the total weight of the system to: 2142 grams. The price tag of all the components is approximately $1440:

- 4 Motors and ESCs: $120
- 4 Batteries: $150
- Frame: $200
- BeagleBoard-Xm: $200
- Arduino and Aeroquad shield: $120
- Four propellers: $40
- Kinect: $160
- Connectors, charger and other accessories: $250
- Import taxes and shipping (from America, China, ...): $200

5.1. Propeller selection

To select the right kind of propeller it is important to keep track of two parameters. These two parameters are a shared property of all propellers:

- *The propeller diameter*: The distance from one blade tip to the other blade tip. Propeller with a larger diameter generates more thrust.
- *The propeller pitch*: The twisting of the blade from the center to the end. The propeller pitch determines how far a propeller would travel if the propeller would turn one time. The pitch also strongly affects the stalling power of the propeller. The stalling power is the amount of power needed to stall the propeller. When a propeller is stalled it cannot propel air effectively anymore. Propellers with a high pitch have a low stalling power.

To generate a large amount of thrust the propeller pitch must remain as low as possible. A low pitch means that most of the work is transferred to thrust (or force) instead of distance. Also a low pitch means a high stalling power. This high stalling power is a great advantage since a relatively high amount of power will be transferred to the propeller. Furthermore, a low pitch means that, from a control point of view, the total gain in the system is lower. This allows for a more stable flight of the quadrocopter.

Internet provides great support of selecting the right kind of propeller. Static thrust calculators (e.g. [38]) provide a great opportunity to get a good feeling of the importance and the impact of the different propeller properties on the static thrust.

A good propeller for the system would be a propeller around 15x5 inches. High diameter, low pitch. The market does not offer a propeller with these dimensions. The propeller that has very similar properties is the 14x3.7 from APC. This propeller should deliver about 1.2 kg of thrust at 5000 RPM. More than enough to get the system in the air.

Next to the two standard propeller parameters, propellers are made of different plastic composites. Mainly there are three types:

- Standard: cheapest composite, maximum RPM lies usually about 6200 RPM. Do not come in very large sizes.
- SF, also Slow Fly: meant for large quadrotors to attain more stability in flight. SF-propellers are lighter than standard propellers. Because of their lighter weight their maximum RPM is also lower. This maximum RPM about 4000 RPM. Come in large sizes.
- Electric: mean for large and heavy quadrotors. The main advantage of electric propellers over SF-propellers is their higher maximum RPM. The maximum RPM of electrical propellers typically lies around the 7000 RPM. However the increase of their maximum RPM goes at the expense of the weight of the propeller. Electric propellers usually weigh twice as much as SF-propellers. Come in large sizes.

Since the limitations in size on the standard propellers and the too low RPM of SF-propellers, only one choice remains, the electric propellers. APC offers an electric propeller with the same dimensions noted above (14x3.7).

5.2. Carbon fibre frame design and construction

The frame design and construction (i.e. how to assemble all components) is a delicate part of quadrotor development. A good frame results in:

- Stability of the quadrotor.
- Longer flight time due to light weight.
- Durability of the design.

In our quest to achieve this, a lot of practical problems were encountered. For example, the majority of the electronic parts require connectors to be soldered on. The connectors we bought were not fully compatible with the thickness (gauge) of our wires. It proved to be a difficult challenge to solder the wires properly without any loose contacts.

Another practical problem was the choice of frame material. To reduce weight, we decided to use carbon fibre: the target frame weight as used in the weight estimation above is only 200 grams. This is not easy to achieve when one uses only metal to construct the frame.

Inspired by [45], we first tried to cast our own frame by using several layers of carbon fibre sheet combined with epoxy to harden it. This construction procedure requires the production of negative molds as a reference for the casting process. The mold making proved to be very difficult and therefore this procedure was discarded. In addition, it turned out to be very hard to actually achieve a similar stiffness as the pre-produced carbon fibre sticks one can also buy. The reason for this is that the pre-produced sticks are produced in an industrial process (using e.g. vacuum molding). It is impossible to copy this process at home.

Thus, in the end it was decided to buy a pre-produced two meter long square tube from [46] instead. We selected a diameter of $5 \times 5\text{mm}^2$, but using slightly thicker tubes or circular hollow tubes might provide even better results.

The tube was cut into four equally long pieces and notches were created by filing so that all the tubes could be connected. To make the connection between the tubes solid, a combination of (e.g. epoxy) glue and tie wraps can be used. The tie wraps hold the tubes in place while the glue is drying and also provides additonal strength afterwards. Then cable lugs (Dutch: *kabelschoenen*) were attached to the end of the tubes: each arm of the quadrocopter consists of two parallel sticks, so that a motor can be attached between those two sticks using the lugs. In the end, the total frame weighted 140 grams.

The last practical problem is mounting the propellers properly. If your propellers are not mounted fully horizontal, the quadrotor will be destabilized severely. We asked for professional help from Sjoerd van Driel, the mechanic who works in the workshop of our department.

Part III.

Local control

6. The Aeroquad code: overview and operation

Now that all the hardware components have been chosen, our focus can shift towards the stabilization of the quadrocopter. This section examines the internals of the Aeroquad code in more details, so that a better understanding of it can be gained.

The Aeroquad system is a system designed to be very flexible. The software has to be able to operate on all the different versions that were produced by Aeroquad. One of the main advantages that come from this flexibility is hardware generalization. It allows the system to be tested using only an Arduino without any supplied hardware. With hardware generalization it is also possible to add your own hardware in the configuration. This allows the system to be modified rapidly for current needs.

At the most basic level of the Aeroquad code lies a simple Arduino operation. Arduino code is very similar[1] to traditional C++ code: each Arduino code setup generally looks like this:

```
1  #Arduino.h
2  #Some_other_headers.h
3
4  // Declaration of some global variables
5  void setup() {
6      /* Initialization of all the attached peripherals, typically some low-level
           ↪ hardware.*/
7  }
8
9  void loop() {
10     /* Execute all the tasks, and loop indefinitely. */
11 }
```

A difference with traditional C++ applications is that the traditional *main* function is split in two parts: *setup* and *loop*. This difference originates from the intended use of the Arduino: it is much more oriented at prototyping and low-level hardware such as servo motor, sensor systems, etc. These peripheral systems all need to be initialized, which is why a separete *setup* function is very convenient. Moreover, because the Arduino code runs on an Atmel microprocessor without any operating system, it is desired that the application never exits. Because of this, *loop* is continuously called after setup has returned. The intended use as prototyping is also visible in the many available libraries for the platform. Consider for example the servo library as already described in chapter

[1] In fact, Arduino code is compiled using the Atmel AVR C++ compiler behind the covers.

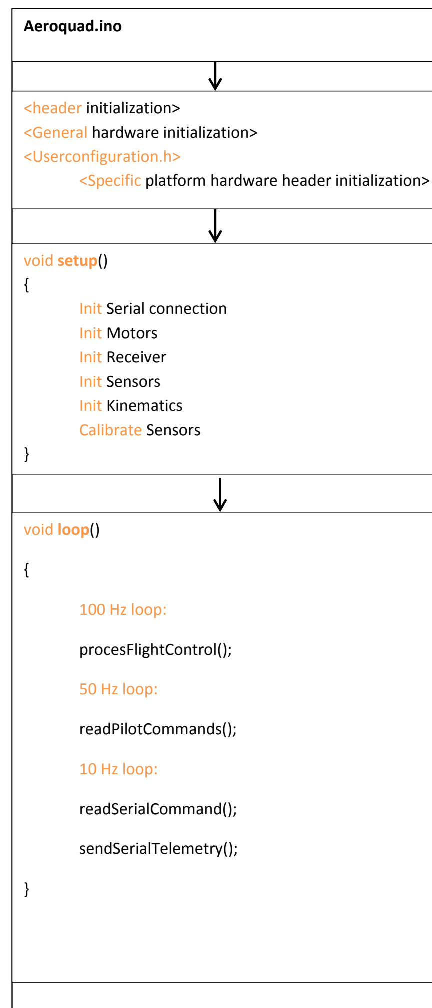


Figure 6.1.: Overview of Aeroquad program flow.

4. Below the covers, this library automatically sets all the relevant Atmel timer and PWM output registers, so that the user does not need to worry about these details. To summarize, because of the orientation towards prototyping, the Arduino an ideal platform for developing a quadrotor.

The Aeroquad shares the same code setup but is much more complicated. For example, there are 20 libraries involved and 10 other function *cpp* files in which the library functions are executed. To get a better grasp of the code, consider the overview diagram in figure 6.1.

The *Aeroquad.ino* file contains the *setup* and *loop* functions. First, the most general headers are initialized, those are summed under *<header initialization>* in the overview diagram. Then the general hardware is initialized. The general hardware are for example *motors.h*, *sensors.h*, *kinematics.h*, etc. General hardware contains usually only some basic

functions and commonly used variables. When the code hits <*Userconfiguration.h*> it initializes the specific hardware. Specific hardware is a header specialized in a single hardware piece. For example: <*Accelerometer_ADXL345_9DOF.h*>. This header is only used for controlling a specific nine degrees of freedom accelerometer from Analog Devices.

After all the platform specific hardware is initialized, all the platform specific variables will be initialized and startup sequences will be send to the peripheral hardware in the setup function. This is represented by the init for each system.

Next follows the *loop* function. This function actually implements three loops. Each loop is executed at a specific interval using timers. By using timers in each loop, the loop time is added to some total time. At the start of a total loop an *if* statement checks for each inner loop whether the total time matches a certain time division. For example, assume that the total time spent is 0.04 seconds. This number is divisible by 0.01 (100 Hz) and 0.02 (50 Hz) without remainder, but not by 0.1 (10 Hz). The 10 Hz loop leaves a remainder after the division and is therefore skipped.

The control loop, represented by the function *procesFlightControl()*, will be discussed in the next section. The 50 Hz loop and the 10 Hz loop will be discussed below.

The 50 Hz loop only reads the pilot commands. The commands are received and processed in *receiver.h*, the specific receiver hardware header and *flighCommandProcessor.h*. In the next loop iteration, the commands received will be directly processed by the *flightcontrolprocessor.h*. Due to the 100 Hz speed of the control loop, every command read in, in the 50 Hz loop, will be processed. This ensures a stable command flow and good flight stability.

The 10 Hz loop, ensures a low priority serial connection with a laptop with a serial interface program. Putty for example. The serial connection in the code is used only for testing, reading and writing PID values. The commands are all defined in SerialCom.h of the Arduino code.

6.1. The Aeroquad control loop

As noted in the previous section, the Aeroquad code iterates in three loops at the same time. The most important loop is the 100 Hz loop. This loop controls the quadrotor with the function *procesFlightControl()*. The commands, or reference input for the control loop, comes from the *readPilotCommands()* in the 50 Hz loop.

To get a better understanding of the control loop, take a look at figure 6.2. The blue parts represent the code being executed in *receiver.h*. In these files the reference signal is derived from movement of the controller stick. *Receiver.h* smooths the signals before deriving the control signal from it. After *receiver.h* the signal consists out of two parts:

- A reference angle in the pitch angle, *Opr*.
- A reference angle in the roll angle, *Orr*.

From that signal the current pitch and roll angle are subtracted, to generate an error signal. This represented in the green sum subtractor. After the sum subtractor the

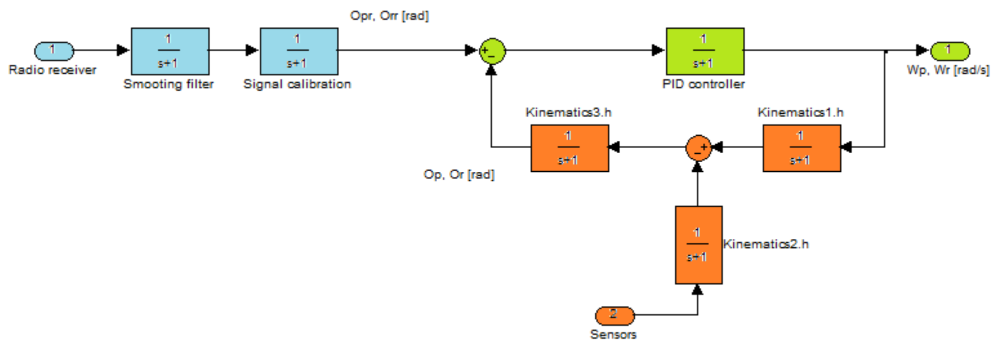


Figure 6.2.: Aeroquad control loop.

signal is send through a PID controller. Each error signal goes through its own PID controller. After the PID controller the two signals have become rotary speeds for all the engine axes. Note that these PID controllers need to be adjusted for the specific dynamics of the quadrocopter hardware in use. Finding suitable PID values can be a very tedious and time-consuming task. Luckily, the Aeroquad configurator can assist in this procedure. Still, because of limited time available for us, we did not attempt to complete this procedure.

The orange parts are the kinematic functions. The kinematic functions are responsible for deriving the current pitch and roll angles from the sensors, with the help of a dynamic model of the quadrocopter. As mentioned before, this dynamic model of the quadrotor can either be:

- Directed cosine matrix
- Model derivation based on quaternions

The kinematic functions will deliver the current pitching and roll angles, represented by *Op* and *Or*.

6.2. Interfacing with the Aeroquad code

An important problem that still remains is how to interface the BeagleBoard with the Aeroquad code. The latter has two supported interfaces:

1. Using a serial USB connection
2. Using a radio receiver

Since it is necessary to connect the BeagleBoard with the Arduino using only a serial USB connection and some digital pins, the first option is preferred. In order to use the *serialCom.h* header to read in the serial commands, it needs to be heavily modified. The *serialCom.h* is only used for reading and writing configuration data such as PID values and transmitter smoothing values, but not for a reference input for the control loop. One problem with modifying the *serialCom.h* is that the Aeroquad configurator

program (described in section 1.2) relies on *serialCom.h* to properly upload the files to the Arduino. For example these functions are also used in the Aeroquad configurator and are defined in *serialCom.h*: reportVehicleState(); fastTelemetry(); readFloatSerial(); printPID(); and others. In summary, before the serial USB connection can be used, the serialCom.h code needs to be fully sorted out (i.e. thoroughly understood) and rewritten.

The other issue is with the *FlightCommandProcessor.h* file. In this function, the pilot commands are read in with the function *readPilotCommands* and processed so that the *FlightControlProcessor.h* can take it over to control the quadrotor. This *readPilotCommands* function is a very complex function and uses an additional six general and five specific hardware headers to perform the necessary actions. These headers are: Gyroscope.h; Accelerometer.h; Kinematics.h; Receiver.h; Motors.h; and Aeroquad.h.

Modifying one part of the *FlightCommandProcessor.h* influences the how control loop operates severely. In order to fully grasp the effects of modifying this file, testing needs to be done to verify whether the control algorithm still operates properly. Concluding, there was simply not enough time to fully understand this code, next to modifying it.

However, one option still remains: use the radio interface of the Aeroquad code. This interface supports several radio transceivers, such as the XBee or a Spektrum digital transceiver.

Of course, using such a transceiver is not very practical, because we actually want a *wired* connection to the BeagleBoard instead of a *wireless* connection to the user. Furthermore, e.g. the Spektrum transceiver would cost an additional $400. It seems the way to go here is to make a hardware module that emulates such a transceiver, i.e. that translates serial port commands from the BeagleBoard to the required signals for the receiver input of the Arduino. Alternatively, one could try to make such a module in software, either in the Arduino or in the BeagleBoard. Again, because of time limitations, we have not investigated this possibility in more detail. Instead, we decided to focus on testing the low-level hardware platform and high-level vision system separetely.

Part IV.

Vision system

7. BeagleBoard explored: software-hardware interface

As noted in the previous part, the BeagleBoard will be used as the central processing board. In this chapter, the hardware capabilities and architecture of the BeagleBoard will be explored. This is important, because good understanding of the hardware is necessary to be able to achieve optimal software performance.

First, the architecture of the BeagleBoard processor will be discussed. Then, some remarks about memory management and caching will be made. Finally, the start-up sequence of the board will be described.

7.1. DM3730 architecture

Figure 7.1 on the facing page contains a block diagram of the DM3730. As can be seen, the ARM (denoted as MPU in the figure), DSP (denoted as IVA in the figure) and PowerVR GPU are all connected to a central interconnect bus. The SDRAM memory controller also connects to this bus.

Inter-processor communication between the ARM and DSP is achieved using a mailbox-interrupt mechanism. This allows the software to communicate between the two processors by transceiving messages from/to the mailboxes. Also, power management can automatically put the mailboxes in idle mode to save power. The mailboxes can also be used to communicate with the integrated GPU.

The ARM processor has a 32-bit wide superscalar architecture. This means that it is mainly suitable for general-purpose tasks. It also includes a NEON coprocessor with support for floating point and SIMD instructions. Because of this, the ARM can also be used for light to modest signal processing tasks such as MP3 decoding. Figure 7.2 on the next page gives an impression of the ARM architecture. More details can be found in [2].

The DSP processor has a 16-bit wide VLIW architecture and is mainly suitable for heavy signal processing tasks (with sufficient parallelism and regular memory access patterns). Also, its instruction set is targeted towards signal processing. Figure 7.3 on page 32 contains an overview of the datapath of the DSP. Further details can be found in [3].

Because our vision algorithm will be quite computationally intensitive, we expect that just using the ARM processor will not provide sufficient performance. In addition to the fact that the DSP is more suitable for signal processing, using it simply provides more processing power.

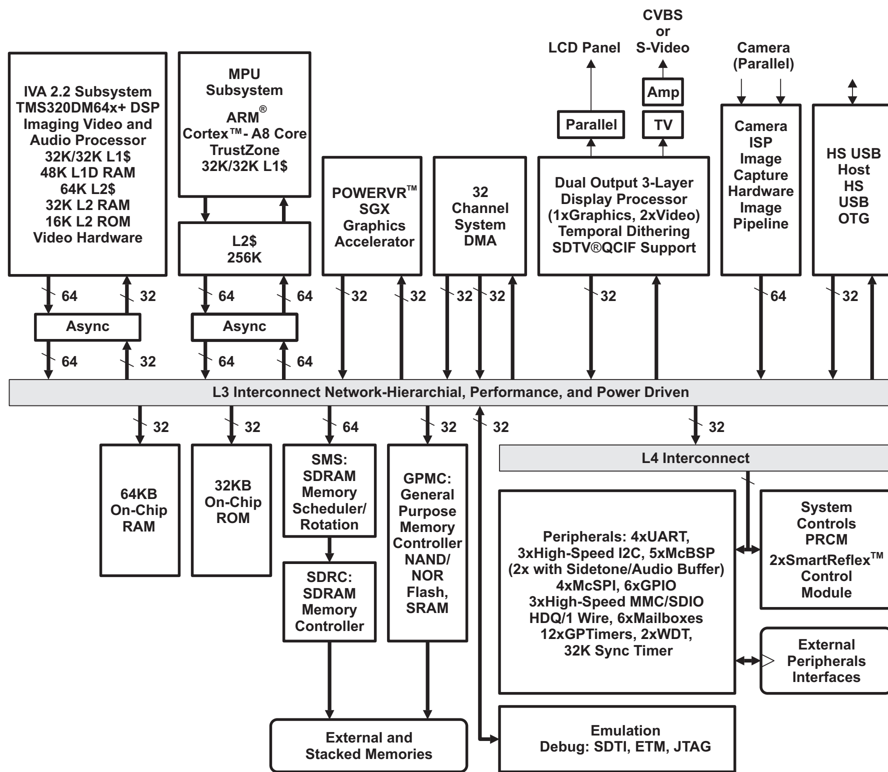


Figure 7.1.: DM3730 functional block diagram. Reproduced from [1].

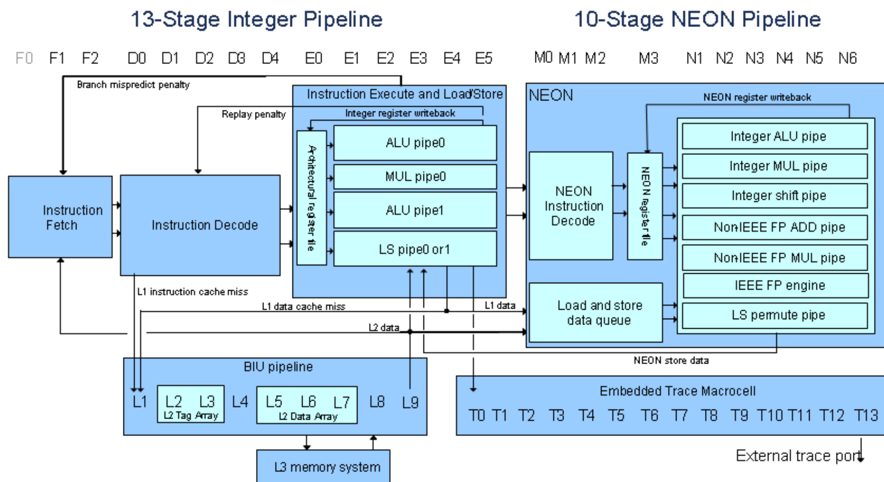
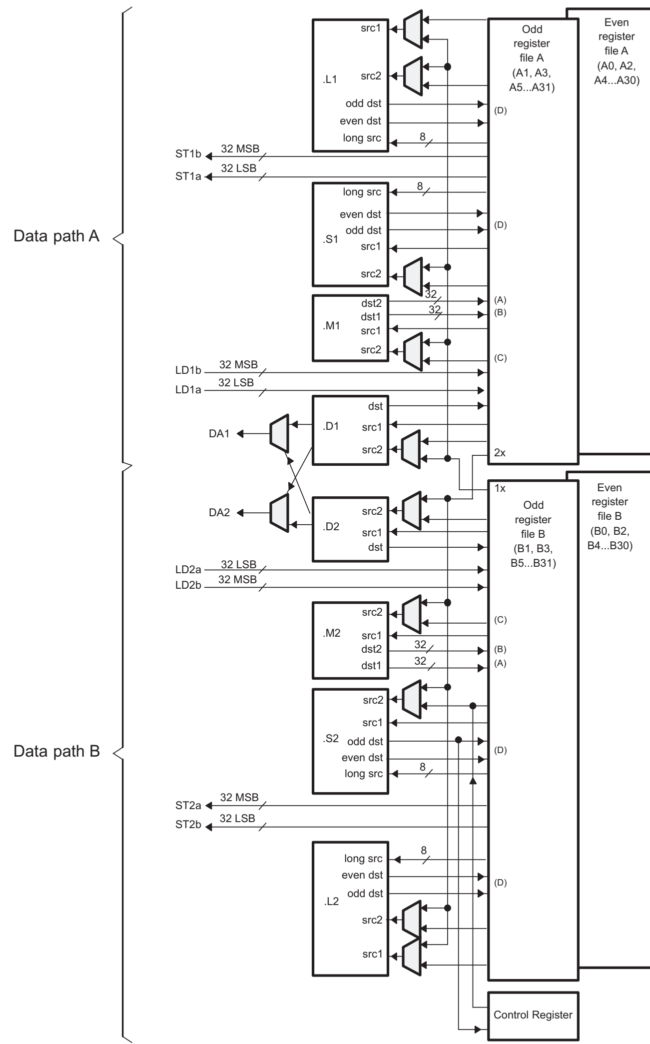


Figure 7.2.: ARM processor pipeline diagram. Reproduced from [2].



A. On .M unit, dst2 is 32 MSB.
B. On .M unit, dst1 is 32 LSB.
C. On C64x CPU .M unit, src2 is 32 bits; on C64x+ CPU .M unit, src2 is 64 bits.
D. On .L and .S units, odd dst connects to odd register files and even dst connects to even register files.

Figure 7.3.: DSP processor datapath diagram. Reproduced from [3].

The same could be said about the PowerVR GPU. However, Texas Instruments does not support general-purpose programming on the GPU (i.e. OpenCL [4]). Because of this, the value of the GPU is limited for our purposes. Therefore, we decided not to invest any time in utilizing the GPU.

7.2. Memory management and caching

It should be noted that the system memory is shared between ARM and DSP. A major advantage of this is that data does not need to be copied between ARM and DSP: both processors can use the same pointers to physical memory. Of course, the price for this is that the data bus must be shared. When both processors are accessing memory at the same time, a performance penalty arises. To alleviate this, both processors have their own L1 and L2 caches. (For more information on ARM cache architecture, see [2]. For DSP cache operation, consult [6].)

Two complications arise with above configuration:

1. Because the ARM typically runs a high-level operating system such as Linux, its memory management interface (MMU) supports virtual memory addressing (using a translation look-aside buffer (TLB)). This means that pointers in ARM userspace point to *virtual* instead of physical memory. The DSP does not have a TLB, and hence does not support virtual memory addressing in hardware. Thus, it it not possible to pass ARM virtual memory pointers directly to the DSP.
2. Care must be taken that memory operations of one processor do not invalidate the cache of the other processor. Hence, cache flushes must be performed when data is shared between the ARM and DSP. This implies a potentially large performance penalty.

Both complications can be taken care of by careful software programing. This will be elaborated upon in the next chapters.

In addition to the external system memory, the DM3730 also supports a small amount of on-chip memory (see figure 7.1). The on-chip ROM is used by the Texas Instruments bootloader. Hence, is not user-accessible [5]. Presumably, the bootloader also uses the on-chip RAM. Although it might be possible to utilize the on-chip RAM for other purposes (e.g. as scratchpad memory), the authors could not find any documentation about how to do so in Linux. Therefore, the on-chip memory will not be considered in this paper.

7.3. Start-up sequence

The BeagleBoard does not have any on-board ROM, and hence always boots from MicroSD card. When power is applied, the ARM processor executes the code from its on-chip ROM. This code scans the SD card for an active primary partition, FAT formatted, with the boot flag set. On this partition, it executes the file named *MLO*. This file can load a fully featured bootloader such as *U-Boot*. This bootloader can then load the operating system (e.g. the *Linux kernel*) [7].

The DSP processor is not enabled during the start-up sequence. After the operating system has loaded, the ARM processor must manually enable the DSP and instruct it to start executing code.

8. Toolchain set-up: abstraction of the software-hardware interface

In the previous chapter, an overview of the BeagleBoard hardware has been given. This chapter will explain what software is required for us to be able to access this hardware in an easy way, i.e. being able to compile our own C/C++ applications. By the end of this chapter, a complete toolchain will have been set up, supporting both the ARM and the DSP. In addition, information will be given about installing drivers for the WiFi module and Kinect camera.

The authors were quite surprised by the incomplete documentation available on the internet. Especially on interfacing the DSP, information was often missing, outdated, or even incorrect. Because of this, a considerable amount of time was spent on trial-and-error. To exempt future BeagleBoard users from this, we decided to describe the installation and configuration process of the software quite extensively. We hope this chapter will be a valuable addition to the documentation available on the internet.

First, the choice and installation of the operating system of the BeagleBoard will be described. Next, the installation of the Angstrom toolchain will be explained. Then, the DSP cross-compiler will be added to this toolchain. The Texas Instruments DSP libraries will be discussed after this. Finally, some remarks on automation of the toolchain and its limitations will be made.

8.1. Choice of operating system

The first choice to be made is about the operating system. There are many OSs available for the ARM architecture, all with their own strengths and weaknesses. They can roughly be categorized in three groups:

- High-level operating systems. These OSs offer a lot of features and services, at the price of a relatively high system load. Also, because of the many background processes in such an operating system, real-time execution of user code can become problematic. Notable examples are Linux or Windows CE.
- Real-time operating systems. These OSs offer less features and are specifically designed to allow real-time execution of user code. An example with BeagleBoard support is QNX Neutrino.
- Bare-metal approach. Instead of installing an OS, user code can also directly run on the ARM processor. On the one hand, all system resources are available for user code, thereby allowing for maximal performance. On the other hand,

however, all features must be manually written or included from libraries. This means development can become considerably more complicated, especially when support for e.g. USB devices or Ethernet is required. This approach is commonly used when developing for the Arduino board.

Because our development time is limited, we want to *outsource* as many services as possible. For this reason, we chose to use a high-level OS. As the Arduino board will be responsible for 'keeping the Quadrocopter in the air', we require just a soft real-time guarantee. This means that the user code should generally run in real-time, but is allowed to lag behind occasionally. Thus, the possible impact of background processes is limited.

Now the only task remaining is the choice of a specific (high-level) OS. The BeagleBoard has been demonstrated using general Linux distributions such as Ubuntu, Angstrom, Fedora, Gentoo, Arch Linux; mobile phone operating systems such as Android or Symbian; and even Windows CE. Our main criterium for selecting the operating system was the amount of documentation and libraries available. Ubuntu and Angstrom seem to be the most commonly used OSs for the BeagleBoard.

Ubuntu was tried first, because the authors had previous experience with that OS. In addition, Ubuntu seems to have a larger software repository. However, it turns out that Ubuntu does not fully support the DM3730 processor: it can only run at 800 MHz instead of the full 1 GHz. By default, it even runs at 600 MHz [8]. In addition, we ran into trouble with our 16 GB SD card: it caused I/O errors in combination with Ubuntu.

Thus, we selected Angstrom, which does support the DM3730 at full speed.

8.2. Installing Angstrom Linux on target

First, we created a 2011.03 image using the Narcissus online image builder [9]. Then, we created a SD card by following[1] the instructions from [10]. Note that it is very difficult to access a partitioned SD card in Windows, because the default SD card driver does not provide low-level block access. Hence, these instructions should be performed on a Linux host. We used an Ubuntu 12.04 LTS Live CD.

The next step is setting up the Angstrom Linux installation. We performed the following tasks:

Set-up user account

For the sake of simplicity, we used the default *root* user account. However, because the system will be accessable over Wifi, we did set up a password as basic security measure using the *passwd* command.

[1] As noted in [8], the *j* flag in the *tar* commands should be replaced by an *x* flag, because the compression format of Angstrom has changed.

Install native toolchain

A native (i.e. local) C/C++ toolchain was also installed. This was easily done with the command *opkg install task-native-sdk*.

Set-up Wifi dongle

A Zydas ZD1211 based Wifi dongle will be used for wireless communication. The driver for this dongle is already included in Angstrom and can be installed using *opkg install kernel-module-zd1211rw zd1211-firmware*.

Initially, we tried to set up the Wifi dongle as access point. However, it turns out the version of the zd1211 driver included in Angstrom does not support this. Thus, the Wifi dongle had to be configured in ad hoc mode instead, by adding the following lines to the */etc/network/interfaces* file:

```
1  auto wlan0
2  iface wlan0
3      inet static
4      wireless-mode ad-hoc
5      wireless-essid Beagleboard
6      wireless-channel 4
7      address 10.0.4.0
8      netmask 255.255.255.0
```

A DHCP server for dynamic IP address assignment was also installed using *opkg install dhcp-server*. The following lines were added to the configuration file */etc/dhcp/dhcpd.conf*:

```
1  INTERFACES="wlan0";
2  subnet 10.0.4.0 netmask 255.255.255.0 {
3      range 10.0.4.1 10.0.4.20;
4      option routers 10.0.4.0;
5  }
```

After a restart, a host system can connect to the wireless network *Beagleboard* and automatically get a dynamic IP address in the range 10.0.4.1-20. The BeagleBoard itself has fixed IP address 10.0.4.0.

Install and set-up serial-over-USB drivers

The Arduino needs a serial-over-USB driver. Recent Arduinos such as the Uno need the Atmel *cdc-acm* driver; older ones need the FTDI *ftdi-sio* driver. [11] We installed both drivers on the Beagleboard using the following commands:

```
1  opkg install kernel-module-cdc-acm kernel-module-ftdi-sio
2  echo usbserial > /etc/modutils/usbserial
3  echo cdc_acm > /etc/modutils/cdc_acm
4  echo ftdi_sio > /etc/modutils/ftdi_sio
5  update-modules
```

Install Kinect driver

Before images can be aquired from the Kinect, a driver library must be installed. We used the OpenKinect *libfreenect* library [13], because it is very lightweight and easy to use. We manually compiled the latest version of *libfreenect*, because this version offers significantly better performance than the one included with Angstrom. The (cross-)compilation was performed on a Linux host. See section 8.3 for details.

Do note that *libfreenect* also has some disadvantages: it offers no built-in support for camera calibration and image and depth frames are not necessarily time-synchronized with each other. These disadvantages might decrease quality of the visual odometry. However, we expect that it is possible to work around these disadvantages in user code, should quality turn out to be insufficient.

Install SSH server

Finally, an SSH server was installed on the BeagleBoard using *opkg*. This allows for safe data transfer and remote login on the BeagleBoard. To prevent the user from needing to enter a password every time, public key authentication is used. See [14] for details on how to generate and install these keys.

8.3. Cross-compiling for Angstrom on Linux host

Because compiling is a computationally intensive task, doing this on the BeagleBoard may take a long time. This means that debug cycles also take a lot of time. To alleviate this, a more powerful host system is used. In our case, the host system is running Ubuntu 12.04 LTS. (We installed it in a VirtualBox virtual machine, so that the Ubuntu installation can be easily transferred to a different host system.)

Installing a cross-compiler

Because the architecture of the host (x86) is not the same as that of the target (ARM), a regular ARM compiler cannot be used. Instead, a *cross-compiler* must be installed. This cross-compiler can generate the ARM code, but is itself written for x86.

However, just a cross-compiler is not sufficient: the code it is compiling may refer to files (libraries, headers, ...) that are only available on the BeagleBoard. Therefore, a complete copy of the Angstrom distribution should be installed on the host as well. The combination of this distribution, cross-compiler and all other utilities is called a *toolchain*.

Luckily, Angstrom provides a pre-built toolchain [15]. It can be installed by unpackaging it using `tar -xf angstrom-2011.03-i686-linux-armv7a-linux-gnueabi-toolchain.tar.bz2 -C` and then setting up the environment by adding `source /usr/local/angstrom/arm/environment-setup` to `~/.bashrc`. (Do not forget to restart your shell after editing `~/.bashrc`.)

The copy of the Angstrom distribution resides in `/usr/local/angstrom/arm/arm-angstrom-linux-gnueabi`. Packages can be installed on the host system using

the *opkg-target* command. In addition, the GNU compiler tools can be invoked using the prefix *arm-angstrom-linux-gnueabi-*. For example, to run a cross-compiling *g++*, run *arm-angstrom-linux-gnueabi-g++*.

Note that by default the Angstrom toolchain is only accessible for the *root* user. To make it available for your own user account, change ownership by executing the following commands from your own account:

```
sudo chown `whoami`:`whoami` -R /usr/local/angstrom
sudo chown `whoami`:`whoami` -R /var/lib/opkg
```

(Of course, there are more elegant ways of changing these permissions, e.g. using user groups. However, above approach seems to be the most straight-forward one.)

Makefile cross-compiling

Most larger applications include a Makefile that automates the compilation process. Compilation of these applications is usually done using three commands:

1. *./configure*. This command adapts the Makefile so that it will work on this host system. Do not forget to use the `--host=arm-angstrom-linux-gnueabi` flag, so that it will use the correct cross-compiler. In addition, the `--prefix=<folder>` option should be used to specify the location of the output files.
2. *make*. This command performs the actual compilation.
3. *make install*. This command copies the compiled files to the location specified in the first step.

CMake cross-compiling

For some programs, such as *libfreenect*, *cmake* is used instead of *./configure* to generate the relevant *Makefiles*. It can be installed on the host by using the *sudo apt-get install cmake* command. Then, to run it in cross-compile mode, a *Toolchain.cmake* file is required:

```
# this one is important
SET(CMAKE_SYSTEM_NAME Linux)
# this one not so much
SET(CMAKE_SYSTEM_VERSION 1)

# specify the cross compiler
SET(CMAKE_C_COMPILER /usr/local/angstrom/arm/bin/arm-angstrom-linux-gnueabi-gcc)
SET(CMAKE_CXX_COMPILER /usr/local/angstrom/arm/bin/arm-angstrom-linux-gnueabi-g++)

# where is the target environment
SET(CMAKE_FIND_ROOT_PATH $ENV{PKG_CONFIG_SYSROOT_DIR})

# search for programs in the build host directories
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
```

```
15
16 # for libraries and headers in the target directories
17 SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
18 SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

Finally, *cmake* can be invoked using the *-DCMAKE_TOOLCHAIN_FILE=Toolchain.cmake* parameter. For more information, consult the *cmake* documentation [12].

8.4. Accessing the DSP

As mentioned earlier, the DM3730 processor of the BeagleBoard also includes a DSP core. Because this DSP core has a VLIW architecture, instruction scheduling must be done in software. Therefore, a specialized compiler is needed, i.e. *gcc* cannot be used. Fortunately, TI provides such a compiler: the *C6000 Code Generation Tools* [16].

For communication between the ARM and DSP, three Linux kernel modules are required:

- *lpm* - Local Power Manager. This module handles the power management of the DSP core.
- *dsplinkk* - DSP/BIOS Link. This module provides a framework for communication between the ARM and DSP cores.
- *cmemk* - CMEM. This module allows the user to allocate physical memory in Linux, thereby solving the first complication of section 7.2.

Additionally, Texas Instruments provides a tool called *C6EZRun* [17] that abstracts away the communication between ARM and DSP.

Texas Instruments describes C6EZRun as follows: [17]

> The C6EZRun project allows you to seamlessly use the DSP from the ARM core on TI's ARM+DSP devices, without having to deal with any advanced, and potentially complicated, frameworks and software stacks. C6EZRun is a set of tools which will take in C files and generate either an ARM executable, or an ARM library which will leverage the DSP to execute the C code.
> *[...]*
> The project consists of two main components:
>
> 1. A build system to create back-end libraries composed of various TI software technologies and the code of the C6EZRun framework itself.
> 2. Front-end scripts that wrap the TI C6000 code generation tools in a GCC-like interface. These scripts make use of the back-end libraries and build system to create ARM-side components that transparently make use of the DSP.

C6EZRun also supports several advanced functions such as cache control (see section 7.2) and multi-threaded support. See [21] for details.

Instructions by Texas Instruments on how to use C6EZRun on the BeagleBoard can be found in [18]. However, we found that several modifications to these instructions are necessary. In the remainder of this section, the corrected instructions will be presented.

Step 1: installing the kernel modules on the BeagleBoard

Although Texas Instruments suggests building the kernel modules from scratch using the OpenEmbedded *bitbake* tool, it is much easier to use the pre-built modules from the Angstrom package repository. More importantly, due to changes in OpenEmbedded, the instructions from TI are outdated.

To install the kernel modules, run the command `opkg install ti-cmem-module ti-dsplink-module ti-lpm-module`. Next, they can be enabled by issuing the following commands:

```
echo "cmemk phys_start=0x86300000 phys_end=0x88000000 allowOverlap=1" > /etc/
    ↪ modutils/cmemk
echo "dsplinkk" > /etc/modutils/dsplinkk
echo "lpm_omap3530" > /etc/modutils/lpm_omap3530
update-modules
```

This also configures *cmemk* to use the physical memory between address 0x86300000 and 0x88000000. Note that Linux should also be made aware of this. Failure to do so may lead to system crashes because the same memory can then be used both by the Linux memory manager and *cmemk*.

To prevent the Linux memory manager from using this memory, a bootloader option must be modified. First, mount the bootloader partition with:

```
mkdir ~/temp
mount /dev/mmcblk0p1 ~/temp
```

Next, create a file *uEnv.txt* in the *~/temp* directory and add the following line: *optargs="mem=99M@0x80000000 mem=384M@0x88000000"*.

Finally, unmount the bootloader partition and then reboot the system:

```
umount ~/temp
rmdir ~/temp
reboot
```

If everything went correctly, the kernel modules are now loaded. This can be verified with the *lsmod* (list modules) and *dmesg* (driver message) commands.

No other steps are required on the BeagleBoard: the C6EZRun tool runs completely on the Ubuntu host system. All required DSP support libraries are linked and included into the output executable.

Step 2: installing the DSP compiler on Ubuntu host

Now, the DSP compiler can be installed on the Ubuntu host. First, download the Code Generation Tools from [16]. We used version 7.3.5, but other versions may also be supported. Then, install the tools by issuing the following commands:

```
cd ~/Downloads
chmod +x ti_cgt_c6000_7.3.5_setup_linux_x86.bin
```

```
./ti_cgt_c6000_7.3.5_setup_linux_x86.bin --mode silent --prefix /usr/local/
    ↪ angstrom/arm/ti_cgt_c6000_7.3.5
```

Step 3: installing and compiling C6EZRun

In this final step, the C6EZRun tool is installed. First, download C6Run 0.98.03.03 from [19] and unpack it using:

```
cd /usr/local/angstrom/arm
tar zxvf ~/Downloads/C6Run_0_98_03_03.tar.gz
cd C6Run_0_98_03_03
```

Next, edit the *Rules.mak* file and change the following variables:

```
DSPLINK_VERSION=1_65_00_03
LPM_VERSION=1_24_02_09
BIOS_VERSION=5_41_07_24

XDCTOOLS_VERSION=3_10_05_61
LINUXUTILS_VERSION=2_25_05_11

SDK_PATH ?= /usr/local/angstrom/arm

CODEGEN_INSTALL_DIR ?= $(SDK_PATH)/ti_cgt_c6000_7.3.5
ARM_TOOLCHAIN_PATH ?= $(SDK_PATH)
ARM_TOOLCHAIN_PREFIX ?= $(TARGET_SYS)-
```

It is very important that the version numbers of the tools used by C6EZRun match with the kernel modules on the BeagleBoard. The other changes in the *Rules.mak* file configure C6EZRun to use the ARM compiler from the Angstrom toolchain and the DSP compiler from the TI Code Generation Tools.

The required dependencies for C6EZRun can be downloaded and installed automatically by issuing the command *make get_ components*. (We noticed that occasionally C6EZRun failed to download some packages. To work around this, manually download the relevant package from the TI site, copy it to */usr/local/angstrom/arm/C6Run_ 0_ 98_ 03_ 03/downloads* and make it executable using *chmod +x <file>*. Then re-run *make get_ components*.)

Next, compile C6EZRun by executing:

```
make beagleboard_config
make all
source ./c6run-environment.sh
make examples tests
```

Finally, add *source ./c6run-environment.sh* to the end of `~/.bashrc` to setup the C6EZRun environment.

Some C6EZRun demos can be found in the *examples* and *test* directories. You can test whether everything is working correctly by copying them to the BeagleBoard and then executing them.

8.5. Texas Instruments DSP libraries

To speed up development, Texas Instruments provides several optimized DSP libraries containing often used functions, such as Fourier transformations and matrix multiplications. An overview of all available libraries can be found on [20]. (For libraries with multiple versions, use the *C64XPLUS* version.)

Note that TI also offers the *Video Analytics & Vision Library (VLIB)* which is specifically meant for video processing. Unfortunately, access must be requested before this library can be downloaded. We never got any response from Texas Instruments...

Installing the libraries is quite easy: just run the downloaded file and install to the */usr/local/angstrom/arm/<library_name>* directory. Nothing needs to be compiled, except for the *IMGLIB* library. To achieve this, simply edit its *Rules.mak* file so that it points to the TI Code Generation Tools and then run *make all*.

To use the libraries, simply include the relevant header file in your code. Then, when compiling with C6EZRun, specify the directory with the library header files using the *-I* flag. Finally, add the actual library (extension *.a64p*, *.l64p* or *.lib*) as an extra input file so that the linker can include it in the output executable.

8.6. Automating the compilation process: Makefile design

A block diagram of the complete toolchain can be found in figure 8.1. To automate the compilation process, a Makefile has been designed, based on the Makefiles from the C6EZRun examples:

```
1  # -----------------------------------------------------------------------------
2  # Configuration variables
3  # -----------------------------------------------------------------------------
4  
5  # NOTE: make sure that c6run_environment.sh is sourced before running this
6  # Makefile.
7  
8  # Hostname, username and path to install application to
9  INSTALL_HOST ?= beagleboard
10 INSTALL_USER ?= root
11 INSTALL_PATH ?= code
12 
13 # Name of application
14 PROG_NAME = hello
15 
16 # Debugging mode (comment to disable)
17 #DEBUG = 1
18 
19 # Include files
20 CINCLUDES += -I$(C6RUN_TOOLCHAIN_PATH)/include
21 CINCLUDES += -I$(SDK_PATH)/c64plus-imglib_2_02_00_00/include
22 CINCLUDES += -I$(SDK_PATH)/c64xplus-iqmath_2_01_04_00/include
23 CINCLUDES += -I$(SDK_PATH)/dsplib_c64Px_3_1_0_0/inc
```

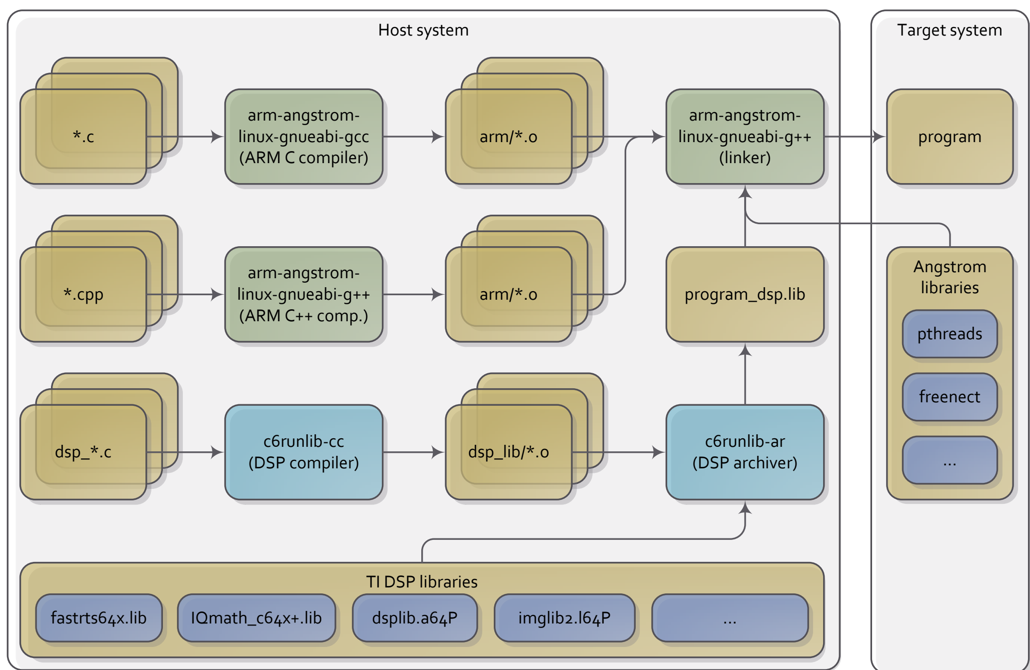


Figure 8.1.: Block diagram of complete toolchain

```
24 CINCLUDES += -I$(SDK_PATH)/dsplib_c64Px_3_1_0_0/packages
25 CINCLUDES += -I$(SDK_PATH)/fastRTS_c62xc64x_1_42/c6400/mthlib/include
26 CINCLUDES += -I$(SDK_PATH)/fastRTS_c62xc64x_1_42/c6400/C_fastRTS/include
27 CINCLUDES += -I$(SDK_PATH)/arm-angstrom-linux-gnueabi/usr/local/include/
       ↪ libfreenect
28
29 # ------------------------------------------------------------------------------
30 # ARM compiler setup
31 # ------------------------------------------------------------------------------
32
33 ARM_TOOLCHAIN_PREFIX ?= arm-none-linux-gnueabi-
34 ifdef ARM_TOOLCHAIN_PATH
35     ARM_CC := $(ARM_TOOLCHAIN_PATH)/bin/$(ARM_TOOLCHAIN_PREFIX)gcc
36     ARM_CPP := $(ARM_TOOLCHAIN_PATH)/bin/$(ARM_TOOLCHAIN_PREFIX)g++
37     ARM_AR := $(ARM_TOOLCHAIN_PATH)/bin/$(ARM_TOOLCHAIN_PREFIX)ar
38 else
39     ARM_CC := $(ARM_TOOLCHAIN_PREFIX)gcc
40     ARM_CPP := $(ARM_TOOLCHAIN_PREFIX)g++
41     ARM_AR := $(ARM_CROSS_COMPILE)ar
42 endif
43
44 # Set compiler flags
45 ARM_CFLAGS = $(CFLAGS)
46 ARM_CFLAGS += -std=gnu99 -Wdeclaration-after-statement -Wall -Wextra \
47 -fno-strict-aliasing -fno-common -c -O3
48
49 ARM_CPPFLAGS = $(CFLAGS)
50 ARM_CPPFLAGS += -Wall -Wextra -fno-strict-aliasing -fno-common -c -O3 -fno-
       ↪ exceptions
51
52 ifdef DEBUG
53     ARM_CFLAGS += -D_DEBUG_ -fno-omit-frame-pointer
54     ARM_CPPFLAGS += -D_DEBUG_ -fno-omit-frame-pointer
55 endif
56
57 # Set linker flags
58 ARM_LDFLAGS = $(LDFLAGS)
59 ARM_LDFLAGS += -lm -lpthread -lfreenect_sync
60 ARM_ARFLAGS = rcs
61
62 # Set library search paths
63 ARM_LDFLAGS += -L$(SDK_PATH)/arm-angstrom-linux-gnueabi/usr/local/lib
64
65 # ------------------------------------------------------------------------------
66 # DSP compiler setup
67 # ------------------------------------------------------------------------------
68
69 C6RUN_TOOLCHAIN_PREFIX ?= c6runlib-
70 ifdef C6RUN_TOOLCHAIN_PATH
71     C6RUN_CC := $(C6RUN_TOOLCHAIN_PATH)/bin/$(C6RUN_TOOLCHAIN_PREFIX)cc
72     C6RUN_AR := $(C6RUN_TOOLCHAIN_PATH)/bin/$(C6RUN_TOOLCHAIN_PREFIX)ar
73 else
```

```
    C6RUN_CC := $(C6RUN_TOOLCHAIN_PREFIX)cc
    C6RUN_AR := $(C6RUN_TOOLCHAIN_PREFIX)ar
endif

# Set compiler flags
C6RUN_CFLAGS = -Wall -c -O3

ifdef DEBUG
    C6RUN_CFLAGS += -D_DEBUG_
endif

# DSP libraries
C6RUN_LIBS += $(SDK_PATH)/c64plus-imglib_2_02_00_00/lib/target/imglib2.l64P
C6RUN_LIBS += $(SDK_PATH)/c64xplus-iqmath_2_01_04_00/lib/IQmath_c64x+.lib
C6RUN_LIBS += $(SDK_PATH)/dsplib_c64Px_3_1_0_0/lib/dsplib.a64P
C6RUN_LIBS += $(SDK_PATH)/fastRTS_c62xc64x_1_42/c6400/mthlib/lib/fastrts64x.lib

# Set linker flags
# NOTE: using --C6Run:replace_malloc here causes SegFault on exit
C6RUN_ARFLAGS = rcs

ifdef DEBUG
    C6RUN_ARFLAGS += --C6Run:debug
endif

# ---------------------------------------------------------------------------
# List of source files
# ---------------------------------------------------------------------------

# List the files to run on the ARM here (all files not starting with dsp_)
EXEC_SRCS := $(shell ls *.c|grep -v ^dsp_|grep -v ^_)
EXEC_SRCS_CPP := $(shell ls *.cpp)
EXEC_DSP_OBJS := $(EXEC_SRCS:%.c=arm/%.o)
EXEC_DSP_OBJS_CPP := $(EXEC_SRCS_CPP:%.cpp=arm/%.o)

# List the files to run on the DSP here (all files starting with dsp_)
LIB_SRCS := $(shell ls dsp_*.c|grep -v ^_)
LIB_DSP_OBJS := $(LIB_SRCS:%.c=dsp_lib/%.o)

# ---------------------------------------------------------------------------
# Makefile targets
# ---------------------------------------------------------------------------

.PHONY : dsp_exec dsp_lib all clean install

# Build everything
all: dsp_exec

# Clean everything
clean:
    @rm -Rf $(PROG_NAME)_dsp $(PROG_NAME)_dsp.lib
    @rm -Rf arm dsp_lib
```

```
126
127 # Link ARM objects and DSP library --> Binary
128 dsp_exec: arm/.created dsp_lib $(EXEC_DSP_OBJS) $(EXEC_DSP_OBJS_CPP)
129     $(ARM_CPP) $(ARM_LDFLAGS) $(CINCLUDES) -o $(PROG_NAME)_dsp $(EXEC_DSP_OBJS) $(
        ↪ EXEC_DSP_OBJS_CPP) $(PROG_NAME)_dsp.lib
130     @echo "=================="
131     @echo "Built $(PROG_NAME) on:"
132     @echo "  ARM: $(EXEC_SRCS) $(EXEC_SRCS_CPP)"
133     @echo "  DSP: $(LIB_SRCS)"
134     @echo "=================="
135
136 # Link DSP objects --> DSP library
137 dsp_lib: dsp_lib/.created $(LIB_DSP_OBJS)
138     $(C6RUN_AR) $(C6RUN_ARFLAGS) $(PROG_NAME)_dsp.lib $(C6RUN_LIBS) $(LIB_DSP_OBJS
        ↪ )
139
140 # Compile ARM C sources --> ARM objects
141 arm/%.o : %.c
142     $(ARM_CC) $(ARM_CFLAGS) $(CINCLUDES) -o $@ $<
143
144 # Compile ARM C++ sources --> ARM objects
145 arm/%.o : %.cpp
146     $(ARM_CPP) $(ARM_CPPFLAGS) $(CINCLUDES) -o $@ $<
147
148 # Compile DSP sources --> DSP objects
149 dsp_lib/%.o : %.c
150     $(C6RUN_CC) $(C6RUN_CFLAGS) $(CINCLUDES) -o $@ $<
151
152 # Create arm directory
153 arm/.created:
154     @mkdir -p arm
155     @touch arm/.created
156
157 # Create dsp_lib directory
158 dsp_lib/.created:
159     @mkdir -p dsp_lib
160     @touch dsp_lib/.created
161
162 # Install binary --> Host
163 install: all
164     scp $(PROG_NAME)_dsp $(INSTALL_USER)@$(INSTALL_HOST):$(INSTALL_PATH)/$(
        ↪ PROG_NAME)
165
166 # ---------------------------------------------------------------------------
167 # Debug information
168 # ---------------------------------------------------------------------------
169
170 # Set DUMP=1 to print below variables for debugging purposes
171 ifdef DUMP
172     $(warning ARM_CC:        $(ARM_CC))
173     $(warning ARM_AR:        $(ARM_AR))
174     $(warning ARM_CFLAGS:    $(ARM_CFLAGS))
```

```
    $(warning ARM_LDFLAGS:   $(ARM_LDFLAGS))
    $(warning ARM_ARFLAGS:   $(ARM_ARFLAGS))

    $(warning C6RUN_CC:      $(C6RUN_CC))
    $(warning C6RUN_AR:      $(C6RUN_AR))
    $(warning C6RUN_CFLAGS:  $(C6RUN_CFLAGS))
    $(warning C6RUN_ARFLAGS: $(C6RUN_ARFLAGS))

    $(warning EXEC_DSP_OBJS: $(EXEC_DSP_OBJS))
endif
```

This Makefile compiles all *dsp_ *.c* files to DSP C code, all other **.c* files to ARM C code, and all **.cpp* files to ARM C++ code. It can also copy the output executable to the BeagleBoard using SSH (*make install)*.

8.7. Limitations and caveats of the toolchain

In this final section of this chapter, several limitations and caveats of the toolchain will be mentioned. Keep these in mind when trying to use the toolchain.

C6EZRun compiler bugs

It has been found that C6EZRun contains some curious compiler bugs. Most notably, the following code causes the compiler to hang indefinitely:

```
int test() {
    return 0;
}
```

Curiously, the following code is compiled correctly:

```
int test()
{
    return 0;
}
```

The only difference is the location of the open bracket...

Usage of the *restrict* keyword

Some of the TI DSP libraries use the *restrict* keyword in their header files. However, this keyword only exists in C. Hence, this will result in compile errors when the header files are included in C++ code.

To work around this, use the following code in all header files which include DSP library headers.

```
#ifdef __cplusplus
#define restrict __restrict__
#endif
```

This defines the *restrict* keyword in C++ code using the GNU-specific *__restrict__* keyword.

C and C++ naming convention

Another problem when mixing C and C++ code is that their symbol names follow a different naming convention. This may result in the linker not being able to resolve certain symbols when trying to call DSP code.

To solve this, define all functions that run on the DSP as follows:

```
#ifdef __cplusplus
extern "C" {
#endif

extern int my_dsp_function();

#ifdef __cplusplus
}
#endif
```

"Terminate called without an active exception"

At some point, we encountered "Terminate called without an active exception" error messages during execution of our code on the BeagleBoard, which resulted in execution being aborted.

According to a conversation on the TI forums [22], it has been found that C6EZRun uses the *pthread* library internally. This library is designed for C only, and is hence not aware of C++ stack unwinding.

To work around this, include the *-fno-exceptions* compiler flag. This completely disables exceptions, thereby suppressing the error condition. Unfortunately, this also means that use of C++ exceptions in user code is not possible.

Access violations

When access violations occur in code being executed on the BeagleBoard, the *dsplinkk* kernel module may become unusable. Hence, C6EZRun executables will not run anymore after a faulty application has caused an access violation, even if the *dsplinkk* module is reloaded. The only solution seems to be a complete reboot of the system.

9. Visual odometry: camera-based position tracking

In the previous chapters, we described both the hardware and software side of our quadrocopter platform quite extensively. However, one element is still missing from the equation: the actual algorithm that is going to control the platform. It turns out that one can find quite a lot of different algorithms in the literature, each with their own advantages and disadvantages. Therefore, we chose to devote this entire chapter to the selection and description of the algorithm. Then, in chapter 10, the actual implementation of this algorithm on our quadrocopter platform will be described.

9.1. Algorithm selection

Let us start with a quick overview of several possible tasks the vision system of a quadrocopter could perform:

- *Stabilization (i.e. control).* The most 'low-level' purpose of the vision system might be stabilization of the quadrocopter. For example, a down-facing camera might be used to reduce drift in the horizontal plane or to estimate the altitude. The commercially available Parrot AR Drone quadrocopter includes such a vision system. [27]
- *Object tracking and/or recognition.* On a higher level, the vision system might perform object tracking or recognition. Applications can range from following a ball to face recognition in a surveillance setting.
- *Pathfinding and/or obstacle avoidance.* Vision-based object recognition can also help with pathfinding. Pathfinding means moving the quadrocopter from position A to position B without hitting any obstackes. This task becomes especially challenging when the environment is dynamic, i.e. when the obstacles move or change in shape.
- *Mapping the environment.* When one takes pathfinding to the next level, the vision system can help with mapping the environment. A typical output might be a 3D model of the environment. See [28]for an interesting video which illustrates the creation of such a 3D model.
- *Autonomously exploring the environment.* When the vision system is able to map an environment, one can combine it with an intelligent exploration strategy. At this stage, the quadrocopter becomes especially interesting for (autonomous) surveillance and reconnaissance missions (e.g. "find the bomb" missions without any

human agents actually entering the building).

Of course, design and/or implementation of such an algorithm quickly becomes very complicated: one could easily fill a 4-year PhD position with such a task. For this reason, we have been searching for an algorithm that fits within the scope of this project, but is still challenging to implement. Key point is the balance between required effort and learning opportunity. To achieve this, the task of the vision system was narrowed down to *visual odometry*.

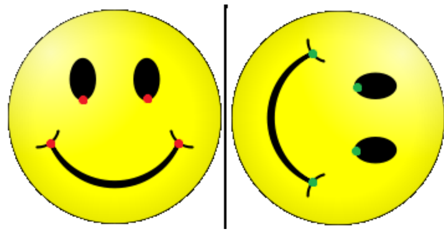
Visual odometry means reconstructing the movement of a camera through time, based on its image stream. Or, worded in a different way: given two similar images, find how they can be transformed into each other. This is a very relevant problem, because a lot of the tasks mentioned above actually require visual odometry as part of their solution. For example, when mapping an environment, separate camera images must be stitched together to form one big map. One of the key ingredients in this stitching procedure is calculating what the relative movement between the images is, i.e. visual odometry. On the other hand, by just focusing on visual odometry, a lot of complications can be ignored. Examples of such complications include preventing error accumulation, converting images to 3D models, and pathfinding. This greatly reduces our required effort.

Note that there are several approaches to solve the visual odometry problem. The most straight-forward one is by direct optimization: find the camera transformation that minimizes a cost function based on the difference between two image frames. More commonly used, however, is an approach based on *keypoints* (also: feature points). This method is outlined in figure 9.1. We will use this latter approach for our project, because it seems to be more commonly used. Also, keypoint detection is part of several object recognition algorithms, which makes it more interesting to study than the direct optimization approach.

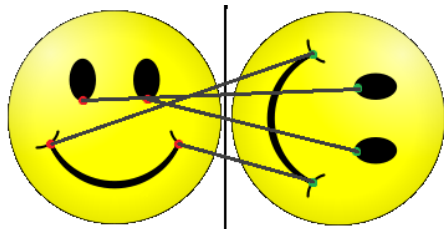
In the remainder of this chapter, the visual odometry will be described in more detail. This description is heavily based on [29]. We chose this paper because of the following reasons:

- The intented application as described in the paper is autonomous flight. This matches with our application.
- The algorithm is designed for RGB-D camera images, i.e. images including depth information. This matches with our hardware platform.
- The paper is quite recent (2011).
- The paper describes how the algorithm can be extended with SLAM (simultaneous localization and mapping) to produce a 3D map of the environment while improving[1] visual odometry accuracy at the same time.
- The visual odometry steps are described quite extensively and the paper provides a lot of references to background information.

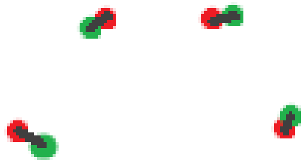
[1] Visual odometry provides just a relative displacement. Although one can easily integrate this to get an absolute displacement, this also accumulates the errors, thus causing significant drift in the long run. SLAM can be used to periodically 'reset' this drift to zero. Consult the paper [29] for details.



(a) Extract interesting points (*keypoints*) from both images.



(b) Match the extracted keypoints between the two images.



(c) Estimate the camera motion which minimizes the error between the matched keypoints.

Figure 9.1.: Outline of keypoint-based visual odometry.

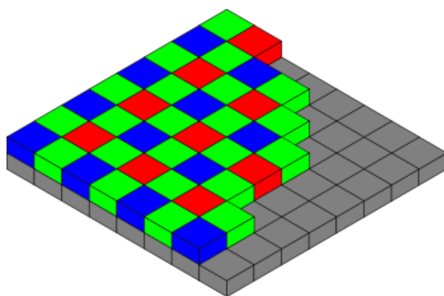


Figure 9.2.: Bayer CCD sensor.

- The hardware platform used is based on a 1.86 GHz Core2Duo processor. This makes for an interesting challenge to implement or modify the algorithm so that it runs in real-time on our hardware platform, which certainly has less processing power.

The extraction, matching and motion estimation step of the visual odometry each consist of multiple substeps. Each substep will be described in its own section below.

9.2. Step 1: Image Preprocessing

The first step is the preprocessing of the images from the Kinect. The purpose of this step is to make the images more suitable for feature detection:

- Because the feature detection does not support color images, the RGB component of the Kinect images must first be converted to grayscale. The paper does not provide too much detail on this step. However, it should be noted that the Kinect contains a Bayer-pattern CCD sensor (figure 9.2). This means that the 640x320 RGB output of the camera actually contains demosaiced (i.e. interpolated) color data. Thus, first interpolating the colors and then converting to grayscale seems very redundant. Therefore, we request the raw Bayer data from the Kinect and downsample each 2x2 Bayer block to produce one grayscale pixel. In addition of simplifying the grayscale conversion, this also reduces image resolution to 320x240, thereby significantly speeding up the rest of the visual odometry algorithm.

  The downsampling is performed by simply calculating the average of the 2x2 Bayer block. Note that this means that the green component has more weight than the red and blue components: our grayscale intensity is given by $I = 0.25R + 0.5G + 0.25B$. This turns out to be similar to luminance as traditionally defined in the YUV color space: $Y = 0.30R + 0.59G + 0.11B$. Moreover, this roughly corresponds with the physiology of the human eye: luminance perception is most sensitive to green light [31]. Ergo, the feature detection and matching is actually performed on the luminance component of the RGB input images.
- To provide scale invariance for the feature detection, a Gaussian pyramid of 3 levels

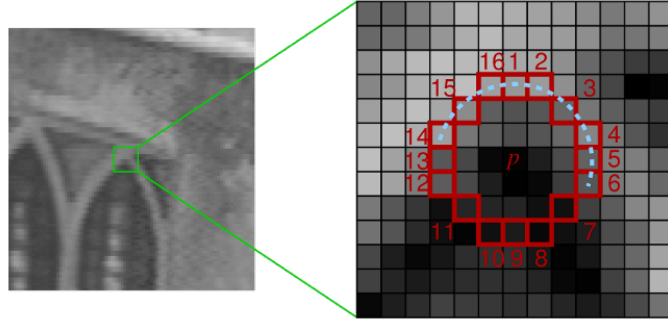


Figure 9.3.: FAST feature extraction. Image taken from [30].

is computed. As described in the paper, "features at the higher scales generally correspond to larger image structures in the scene, which generally makes them more repeatable and robust to motion blur." features at the higher scales generally correspond to larger image structures in the scene, which generally makes them more repeatable and robust to motion blur. The first level is produced by filtering above grayscale image with a 5×5 $\sigma = 0.85$ Gaussian kernel. The other levels are produces by downscaling with a factor of 2 and then filtering again with the same kernel.

- The 640x320 depth channel from the Kinect is not downsampled. The reason for this is that the depth channel is only used after keypoint detection. Because the number of keypoints is significantly less than the number of pixels in the image, it makes sense to perform this downsampling on areas around the keypoints only at a later stage.

9.3. Step 2: Feature Extraction

Now that the camera images have been preprocessed, the actual feature extraction can be performed. For this, the FAST feature extraction algorithm [30] has been used. Sometimes, FAST is also called a *corner point* detector; this hints towards the kind of feature points it extracts. As depicted in figure 9.3, FAST returns whether a given pixel (p in the figure) is a feature point by considering a circle of pixels around it (numbered 1..16 in the figure). When at least 8 consecutive pixels (dotted blue in the figure) in the circle are either much brighter or much darker than p, it is considered a feature point. This "much brighter or much darker" is quantified with the inequality $|\mathrm{I}(p) - \mathrm{I}(k)| \geq T$ where k is one of the pixels in the circle and T is the detection threshold. The function $\mathrm{I}(\cdot)$ returns the intensity (i.e. luminance in our case) value of a pixel.

Each feature point can also be assigned a score. We used the score as defined in formula (8) from [30]: $V = \max\left(\sum_{k\in\{\text{circle}|\text{much darker}\}} |\mathrm{I}(p) - \mathrm{I}(k)| - T, \sum_{k\in\{\text{circle}|\text{much brighter}\}} |\mathrm{I}(k) - \mathrm{I}(p)| - T\right)$. Note that, for implementation purposes, the absolute value signs and "$-T$" terms can be omitted. The higher the score, the 'sharper' the corner and hence the 'stronger' the

feature point.

To find all the feature points in a image, one can simply scan all pixels with the FAST algorithm. (This scan is performed independently on every level of the Gaussian pyramid.) However, sometimes neighbouring feature points are detected. This may decrease the performance of the visual odometry, either by confusing the feature matching process or by decreasing the framerate due to the higher number of keypoints. To alleviate this complication, a *non-maximum suppression* step is recommended. In this step, all feature points with a lower score than their neighbouring feature points are discarded.

Note that three more complications arise, as explained in [29]:

- The optimal threshold T depends on the lighting conditions of the camera image. To solve this, after each frame the threshold is proportionally updated: $T := \text{clip}\{T + \alpha(N_{\text{desired}} - N_{\text{actual}})\}$. Here α is the update rate and $N_{\dots}$ is the number of keypoints. The $\text{clip}(\cdot)$ function limits the threshold to reasonable values to prevent runaway when e.g. the light is turned off for a few seconds.
- The keypoints may not be distributed uniformly throughout the image. From the paper: "to maintain a more uniform distribution of features, each pyramid level is discretized into 80×80 pixel buckets, and the 25 features in each bucket with the strongest FAST corner score are retained."
- Not every pixel may have a valid associated depth value. For example, on surfaces such as a mirror, the Kinect laser-based depth detection may fail. To prevent problems in the feature matching step, all keypoints without valid depth must be discarded at this point.

9.4. Step 3: Initial Rotation Estimation

Next, as a first step in the feature matching process, an initial rotation estimate between two captured images should be calculated. We denote these images with A and B. From the paper: "for small motions such as those encountered in successive image frames, the majority of a feature's apparent motion in the image plane is caused by 3D rotation."

Although the paper bases this initial rotation estimation on the images itself, we opted to use the gyroscope data from the Arduino. It is expected that this decision reduces computational load on the BeagleBoard.

Two complications arise due to the fact that the gyroscope data is not synchronized with the image frames from the Kinect:

- The serial link between Arduino and BeagleBoard (or its Linux driver stack) can introduce a non-deterministic latency. This may greatly reduce accuracy of the gyroscope data. To solve this, we let the Arduino add a timestamp to the gyroscope data. For details on how this timestamp is synchronized with the BeagleBoard, consult chapter 10.
- The gyroscope data may need to be interpolated. Because both the image frames and the gyroscope data are timestamped, this complication can easily be solved by

using one of the many smooth motion interpolation algorithms from the literature (e.g. Catmull-Rom spline interpolation [44]).

9.5. Step 4: Feature Matching

Now that an initial rotation estimate is available, the actual feature matching can be performed. This consists of multiple steps and, like the feature extraction, is performed independently on every level of the Gaussian pyramid:

1. A feature descriptor vector $\mathbf{d}$ is assigned to each keypoint by using the intensity values from a 9×9 pixel square around the keypoint.
2. A score is calculated for every pair of feature points (a_A, a_B) (one point being in image A and the other in B). The score is given by the sum-of-absolute-differences (SAD): $S_{(a_A,a_B)} = \|\mathbf{d}_1 - \mathbf{d}_2\|_1$, where $\|\cdot\|_1$ denotes the L_1 or Manhattan norm.
3. When, for a given pair (a_A, a_B), there exists no $b_A \in A$ for which $S_{(b_A,a_B)} < S_{(a_A,a_B)}$, and no $b_B \in B$ for which $S_{(a_A,b_B)} < S_{(a_A,a_B)}$, a match is declared. This is called a mutual consistency check.
4. The match is discarded if the location of the keypoints is not consistent with the initial rotation estimate. The paper does not go into detail about how to check this consistency. We suggest re-using the buckets from the feature extraction step, because it is both easy to implement and computationally efficient.

   For a given bucket, the initial rotation estimate could be used to determine which buckets might contain matched keypoints. In this way, a score need only be calculated for keypoints from certain combinations of buckets. In this way, the consistency check is then implicitly included in the score calculation. A disadvantage of this method may be that the buckets could cause a 'staircase effect' in performance due to severe discretization of the rotation estimate. However, this can easily be alleviated by reducing the bucket size as used in this step.
5. The paper recommends an additional sub-pixel refinement step by minimizing the sum-of-squared error $SSE = \|\mathbf{d}_1 - \mathbf{d}_2\|_2$ using a nonlinear optimization algorithm. However, for the sake of simplicity, we decided not to implement this step. As described on page 10 of [29], this does have some consequences for the quality of the visual odometry.
6. What the paper does not mention, is how to handle the three levels in the Gaussian pyramid. It turns out that this is a good point to 'merge' the three levels into one big bin of feature points: the actual image around the extracted feature points is not used anymore in the next steps.

9.6. Step 5: Inlier Detection

The next step is *inlier detection.* In this step, all matched keypoint pairs are checked for mutual consistency. Key for understanding this step is the observation that the

distance between two keypoints does not change when the camera moves. Thus, when two keypoints in image A are correctly matched to the same keypoints in image B, the distance between a_A and b_A should be (approximately) equal to the distance between a_B and b_B. Of course, distance here means the actual 3D distance between the keypoints: this is where the depth channel from the Kinect becomes important.

The inlier detection consists of two steps, again based on the paper:

1. A graph is created with each node corresponding to a matched keypoint pair (a_A, a_B). An edge between the *nodes* (a_A, a_B) and (b_A, b_B) is created iff $|\mathrm{P}_A(a_A) - \mathrm{P}_A(b_A)| - |\mathrm{P}_B(a_B) - \mathrm{P}_B(b_B)| < \epsilon$. Here $\mathrm{P}_X(k)$ returns the space coordinates of k, based on its location in image X and its associated depth value. Of course, coordinates depend on the image X, hence the *absolute* location in space is not known. This is not a problem, because only *relative* location matters when calculating the distance. ϵ is the tolerance of the distance matching.
2. The *maximum clique* is selected from beforementioned graph. The *maximum clique* is defined as the largest set of nodes for which one can travel from any node to any other node by only traversing *one* edge, i.e. for which all nodes are *directly* connected to each other. Actually, finding the maximum clique is an NP-complete problem, which means that there is (presumably) no efficient algorithm that finds the exact solution. Therefore, this problem is approximated by a simple greedy algorithm.

   This algorithm works as follows:

   a) Select the node with maximum *degree* and add it to the set of output nodes. (The *degree* of a node is the number of edges that are connected to it.)
   b) Add the node with maximum degree that is connected to all nodes already in the set of output nodes.
   c) Repeat step (b) until no more such nodes can be found.

Alternatively, a method based on RANSAC (**Ran**dom **Sa**mple **C**onsensus) could be used. RANSAC can also use the idea of invariant distance between keypoint pairs to form *constellations* of points. These constellations are then iteratively improved to find a good set of inliers. See [32] for more information about RANSAC. The paper [29] actually compares RANSAC with the approach described above, and it turns out that the latter actually performs better. Moreover, RANSAC seems very complicated to understand. For these reasons, we did not use RANSAC for our visual odometry.

9.7. Step 6: Motion Estimation

If everything went correctly, a consistent set of matched feature points has now been found. Thus, the final step of the visual odometry can be started: the motion estimation. In this step, the actual camera motion between two images is calculated. Again, we will walk through the motion estimation step by step, as suggested by the paper:

1. The Euclidean distances between the matched feature points is minimized. This is achieved by using Horn's method [33]. Horn derived a closed-form solution to find the scale, rotation matrix, and translation matrix that minimize the sum of Euclidean distances between the feature points.
2. The result from above step is improved by minimizing *reprojection error*. Given two points a and b, and a camera projection operation $\text{proj}_X(\cdot)$, the reprojection error is defined as $E = \|\text{proj}_X(a), \text{proj}_X(b)\|$. Here, $\|\cdot,\cdot\|$ is used to denote Euclidean distance. This reprojection error is used to create a cost function that can be minimized using a nonlinear least-squares solver. This procedure is sometimes also called *bundle adjustment*. Bundle adjustment simultaneously finds the most likely 3D location of the keypoints *and* projection matrices for image A and B. These projection matrices provide information about the transformation of the camera between these two images. More details can be found in [34]. Of course, before this step can be implemented, the procedure must be worked out in more detail. Unfortunately, the paper does not provide this detail. Due to time limitations, we decided not to further examine the bundle adjustment procedure.

   As noted in the paper, one advantage of this procedure is that it implicitly accounts for the fact that the depth data of the Kinect becomes less reliable when distance increases.
3. Matched feature points with too large a reprojection error are discarded. Then, step (2) is repeated once more.

This concludes the description of the visual odometry algorithm. In the next chapter, the actual implementation will be discussed in more detail.

10. Algorithm implementation: from mathematics to real-time code

Now that the visual odometry algorithm has been defined in the previous chapter, our focus can shift to its implementation. This chapter contains important implementation notes that help with achieving good performance of the algorithm. Do not that a lot of detail has been left out: the main purpose of this chapter is to give the reader some more implementation-oriented background information that complements the previous chapter. It is not intended to fully documentate our own implementation. Reason for this is that we did not manage to finish the complete implementation on time: it turned out that even implementing 'just' the visual odometry is still too labour-intensive for the scope of this project. A copy of the current (incomplete) implementation code is available from the authors on request.

10.1. Balancing between ARM and DSP

One important point to consider is the balancing between ARM and DSP. For some steps, the ARM may perform better while for others the DSP is more suitable. For example, image acquisition and motion estimation have to be performed on the ARM because of the required communication with the Arduino and Kinect. However, the construction of the Gaussian pyramid, feature extraction and Horn's method in the motion estimation are tasks that are more suitable for the DSP. The reason for this is that they process a lot of data in a predictable way (i.e. unrollable loops, few branches, . . .). On the other hand, we expect the inlier detection to perform better on the ARM, because the program flow of the maximum clique detection depends more heavily on the values of the data.

Some tasks, such as feature matching, could perform well on both ARM (using the NEON SIMD instructions) and DSP. These tasks can then be used to balance the workload between ARM and DSP: because they can be used in parallel, it is desirable that they both spend the same amount of time per frame.

In summary, we suggest the following division of the workload:

Timeslot	ARM	DSP
A	Step 1: image acquisition	
	Step 3: motion estimation	
B		Step 1: construction of Gaussian pyramid
		Step 2: feature extraction (FAST)
C	Step 4: feature matching	
	Step 5: inlier detection	
D		Step 6: motion estimation

In this way, the processing of the different frames can be pipelined in the following fashion:

ARM	DSP
A3	D1
C2	B3
A4	D2
C3	B4
A5	D3
C4	B5
...	...

In above table, the letters correspond to the timeslots as defined above, and the numbers to the frame that is being processed.

Unfortunately, again because of time limitations, we did not manage to completely implement and benchmark this division of workload. The division is roughly based on the benchmark from [29], but it is to be expected that actual timing figures will be different because the architecture is not the same. Therefore, finetuning of this workload division is most probably required to achieve optimum performance.

The remainder of of this chapter contains implementation notes, presented in a bullet-point style because the notes can most often be considered separately from each other.

10.2. Step 1: Image Preprocessing

- The Bayer to grayscale conversion can be implemented efficiently using one load, three load-accumulate, one right shift and one store operations on the ARM processor. At the same time, the image frames are copied from a Linux to a CMem memory buffer, so that the image data can also be accessed by the DSP.
- The downsampling in the Gaussian pyramid construction is performed in a similar fashion, but then by the DSP.
- The Gaussian kernel is implemented using the Texas Instruments IMGLIB. The coefficients have been generated with Matlab: `int16(round(2^17 * fspecial('gaussian', [5 5], 0.85)))`. The 2^{17} term scales the coefficients for optimal dynamic range.

- All the memory buffers are pre-allocated at the start of the application and then reused for all the frames. This prevents costly and unnecessary memory allocation and freeing operations.
- The depth (i.e. number of bits for each pixel) of the images is always 8 bit at this stage. This allows for fast filtering on the DSP. The intermediate accumulator as used in the the Gaussian filter operation has more bits, and is only shifted back to 8 bit afterwards. This step also compensates for the 2^{17} scale factor in the coefficients.

10.3. Step 2: Feature Extraction

- For the feature extraction, we first tried to use the FAST detector as included in the OpenCV library [35] (i.e. on the ARM core). Unfortunately, this code turned out to perform much too slow: just the feature extraction would require almost all the time available for 30 frames per second. Additionally, the code needed to be rewritten from C++ to C, so that it can be compiled for the DSP.
- The most-called part of the FAST detector is the check whether 8 concecutive 'circle pixels' are either much brighter or much darker. This check can be sped up significantly by using the following observation. If the current pixel is a keypoint, either circle pixel 1 or 9 should be a threshold away from p (the terminology refers to figure 9.3). This means that for non-keypoint pixels, the check is often already conclusive after just examing two of the circle pixels. Because most pixels are no keypoint, the check can be sped up by 'short-circuiting ' after examining just some of the circle pixels. In code, this looks like this:

```
1  /* First bit of d is set (using the threshold table tab) if current pixel is
    ↪ threshold lighter than circle pixel under consideration.  Second bit is
    ↪ set if current pixel is threshold darker. ptr is a pointer to the current
    ↪ pixel. pixel[] contains offsets to go to the pixels in the circle. */
2  unsigned char d = tab[ptr[pixel[0]]] | tab[ptr[pixel[8]]];
3
4  if (d == 0) { // Intermediate check to short-circuit if this direction already
    ↪ fails
5      continue;
6  }
7
8  // .. more gradient directions ...
9  d &= tab[ptr[pixel[2]]] | tab[ptr[pixel[10]]];
10 d &= tab[ptr[pixel[4]]] | tab[ptr[pixel[12]]];
11 d &= tab[ptr[pixel[6]]] | tab[ptr[pixel[14]]];
12
13 if(d == 0) { // Another intermediate check
14     continue;
15 }
16
17 // ... and even more gradient directions
```

```
d &= tab[ptr[pixel[1]]] | tab[ptr[pixel[9]]];
d &= tab[ptr[pixel[3]]] | tab[ptr[pixel[11]]];
d &= tab[ptr[pixel[5]]] | tab[ptr[pixel[13]]];
d &= tab[ptr[pixel[7]]] | tab[ptr[pixel[15]]];

if(d == 0) { // Last intermediate check
    continue;
}

// Now check if the pixels are actually consecutive
```

- The feature point score calculation routine from OpenCV has been discarded and completely rewritten. The main reason for this is that this OpenCV routine was very difficult to comprehend. It did not become clear to us what actual formula the implementation is based on. Our replacement implementation uses the formula as reproduced in section 9.3 and is much shorter than the OpenCV code.
- The functioning of the FAST detector has been verified by comparing out output to the reference Matlab implementation by Edward Rosten, the author of the FAST algorithm [30].
- Selecting the strongest 25 keypoints from each bucket can be done efficiently by using a priority queue (e.g. based on a heap data structure). Insertion of a keypoint into the queue requires only $O(\log n)$ time, while selecting the 25 strongest keypoints can be done in constant time. A good reference implementation is available in the standard C++ <*queue*> library. The downside of using this implementation is that C++ cannot be compiled for the DSP. Luckily, there are also numerous C priority queue libraries available which can be used instead. We made a C++ reference implementation of the bucketing step, but to save time, we did not port it (yet) to the DSP.

10.4. Step 3: Initial Rotation Estimation

- Due to time limitations we only implemented the Arduino side of the rotation estimation step. The BeagleBoard implementation is left as future work. Below are some of the ideas that have been used.
- Although the Arduino can easily calculate time deltas using its internal timers, they are useless when the BeagleBoard does not have at least one reference timestamp. To solve this, we propose connecting a GPIO on the BeagleBoard to an interrupt pin on the Arduino. The BeagleBoard code can access its GPIO pins with a jitter of around 1ms, and the Arduino can immediately reset a timer when an interrupt occurs.
- Reading out the serial port can best be done in a separate thread so that it interferes as little as possible with the rest of the vision algorithm. When no new data is available, the Linux scheduler can simply block the serial port processing thread.

10.5. Step 4: Feature Matching

- As also mentioned in [29], the bottom-right pixel of the 9×9 patch around the feature points can be omitted so that the descriptor length becomes 80, which is nicely divisible by 8.
- The sum-of-absolute-difference calcualtions in this step can be significantly sped up by using the NEON instructions. The code without NEON is printed below. As can be seen, every element of the feature descriptor requires four loads, one difference, one comparison and one accumulate operation. (Overhead due the loop is not counted, because it can easily be unrolled.)

```
for (int k = 0; k < 80; k += 1) {
    // Difference ...
    short diff = pix1[offset[k]] - pix2[offset[k]];

    // .. and absolute sum
    if (diff < 0) {
        score -= diff;
    } else {
        score += diff;
    }
}
```

- With NEON, every element requires just four loads and 1/8th *Vector Absolute Difference and Accumulate* (VABA) operation (eight elements are processed at the same time due to the SIMD nature of the instruction). Then, in the end, three more *Vector Pairwise Add* (VPADD) instructions are required to add the 8 partial results together.

10.6. Remaining steps

The inlier detection and motion estimation have not been implemented: we simply ran out of time. For this reason, no implementation notes on these steps have been included in this report.

Part V.

Conclusion

11. Conclusion

In this report, we described the process of designing a vision-enabled quadrocopter. First, the hardware design was elaborated. While we have managed to select a well-balanced set of hardware components, we had insufficient time to actually test and tune the quadrocopter hardware, because of complications with the frame and interface to the Aeroquad code. Next, the inner workings of the Aeroquad code have been discussed. In the process of examining this Aeroquad code, we learned to appreciate the amount of work that is required to just stabilize a quadrocopter. Further on, the embedded hardware platform was examined. Most important lesson from this part is that the availability of an easy-to-use toolchain is as important as the hardware platform itself. We also discussed some of the differences between the ARM and DSP cores on the BeagleBoard. Finally, a vision algorithm was described. As noted before, implementing a complete 3D mapping algorithm would be outside the scope of a project like this. Instead, we thoroughly investigated a visual odometry algorithm. Although the implementation of this algorithm was not finished due to time limitations, we certainly got a feeling for several commonly-used steps and algorithms in computer vision. Also, based on the parts of the algorithm that were completely implemented, we are optimistic about the possibility of achieving 30 FPS real-time performance: the feature extraction (FAST) step alone performs at over 70 FPS.

To conclude this project, it was very hard to complete a fully functional quadrotor with the available amount of time and materials. This project became very challenging because the problems we encountered were strongly multidisciplinary. On the other hand, this made the project highly interesting with a lot of potential for learning. Because a quadrotor is not as "conventional" as a traditional plane there is a lot to discover on what the optimal system is. There is a big spectrum of design choices, in which each choice can hold a significant impact on performance of the final system. For example, we are electrical engineers by education, with virtually no knowledge of material properties and proper construction techniques. When we were busy developing this quadrotor we expected that PID controllers and Kalman filters can stabilize nearly any system. Now we know that the mass distribution and rigidness of the quadrocopter frame have significant influence on controller performance. Thus, failure to design a good frame may cause the quadrotor to oscillate and crash. After this project we are much more knowledgeable about these types of multidisciplinary systems.

12. Recommendations

This last chapter contains several recommendations for the brave readers who want to build their own quadrocopter.

- Be sure to check if the ready to fly platform you buy is easy to interface with. If not, be sure to check very thoroughly if it is easy to modify it, so it becomes easy to interface with.
- Make sure not to order essential parts such as motors, ESCs and embedded hardware from far abroad. These parts can take very long to receive and you pay a lot of import tax. In our case, our BLDC motors came from HobbyKing (China). Because of a mistake by the post office, the package was returned to China and had to be resent. It took an additional four weeks to receive the package. We waited about three months just to receive the motors.
- For heavy quadrocopters, there are a few issues with propellers:
  - It is very hard to mount the propellors perfectly on the motor axis: any error in mounting can cause severe frame oscillations which can in turn confuse the sensors. There is even the risk of propeller shattering. We asked help from a professional mechanic who mounted them properly on the motor shaft using a milling machine.
  - Propellers, especially for the heavy quadrotors, are like blades. They are razor sharp and can cut off hands and fingers. Make sure to develop a secure test environment, which allows you to test your whole system without a health hazard.
- A much lighter frame can be developed using carbon fibre. However, make sure you have the following items in stock for your security:
  - A gas mask: for keeping the small cancerous dust particles out of your lungs.
  - Good gloves: for handling epoxy, which is highly toxic.
- Make sure there is up-to-date documentation available for your chosen hardware platform. Getting the BeagleBoard DSP toolchain to work proved to be a more difficult task than expected due to lack of decent documentation: although there is a lot of general documentation for the BeagleBoard, specific documentation for using the DSP was missing.
- It is not always possible to use off-the-shelf processing libraries for easy implementation of a(n) (vision) algorithm: their performance may not be good enough. For example, this was the case with the OpenCV FAST implementation.

- Reserve more time for testing and verification of the design. Because we spent all our time on implementation, we did not get to the point where we could start with finetuning of the design. Examples include tuning the low-level PID controller values and the visual algorithm workload division between ARM and DSP core.

Bibliography

[1] Texas Instruments (2011). *DM3730, DM3725 Digital Media Processors (Rev. D)*. http://www.ti.com/lit/gpn/dm3730.

[2] Texas Instruments (2010). *Cortex-A8 Architecture*. http://processors.wiki.ti.com/index.php/Cortex-A8_Architecture.

[3] Texas Instruments (2012). *TMS320DM647/TMS320DM648 Digital Media Processors (Rev. H)*. http://www.ti.com/litv/pdf/sprs372h.

[4] Imagination Technologies Forum (2009). *OpenCl support - PowerVR Insider Forums*. http://www.imgtec.com/forum/forum_posts.asp?TID=194.

[5] TI E2E Community (2011). *AM1808 On-Chip ROM and RAM*. http://e2e.ti.com/support/dsp/omap_applications_processors/f/42/t/134879.aspx. *Note: this reference only applies to the AM1808. It has been assumed that this information also holds for the DM3730. No specific information for the DM3730 could be found.*

[6] Texas Instruments (2003). *TMS320C6000 DSP Cache User's Guide*. http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=spru656&track=no.

[7] "nguillaumin" (2011). *Understanding the BeagleBoard-xM boot process*. https://github.com/nguillaumin/beagle-carputer/wiki/system-BeagleBoot.

[8] "dwatts" (2011). *BeagleBoard XM and Ubuntu 11.04 - The Big Dog Still Has a Burr in its Paw*. http://www.gigamegablog.com/2011/08/20/beagleboard-xm-and-ubuntu-11-04-%E2%80%93-the-big-dog-still-has-a-burr-in-its-paw/.

[9] *Narcissus - Online image builder for the Angstrom distribution*. http://narcissus.angstrom-distribution.org/.

[10] Trey Weaver (2010). *Installing Angstrom on the BeagleBoard-xM*. http://treyweaver.blogspot.nl/2010/10/installing-angstrom-on-beagleboard-xm.html.

[11] eLinux.org (2012). *BeagleBoard with Arduino*. http://elinux.org/index.php?title=BeagleBoard_with_Arduino.

[12] CMake (2011). *CMake Cross Compiling*. http://www.cmake.org/Wiki/CMake_Cross_Compiling.

[13] OpenKinect (2012). *Main Page - OpenKinect*. http://openkinect.org/wiki/Main_Page.

[14] Ubuntu (2011). *SSH/OpenSSH/Keys - Community Ubuntu Documentation.* https://help.ubuntu.com/community/SSH/OpenSSH/Keys.

[15] Angstrom Linux (2011). *Standalone toolchain for Angstrom 2011.03.* http://www.angstrom-distribution.org/toolchains/angstrom-2011.03-i686-linux-armv7a-linux-gnueabi-toolchain.tar.bz2.

[16] Texas Instruments (2012). *Code Generation Tools for Texas Instruments Processors : Downloads.* https://www-a.ti.com/downloads/sds_support/TICodegenerationTools/download.htm.

[17] Texas Instruments (2012). *C6EZRun.* http://processors.wiki.ti.com/index.php/C6EZRun.

[18] Texas Instruments (2012). *Getting Started With C6EZRun On Beagleboard.* http://processors.wiki.ti.com/index.php/Getting_Started_With_C6Run_On_Beagleboard.

[19] Texas Instruments (2012). *C6EZRun Software Development Tool for TI DSP+ARM Devices.* http://www.ti.com/tool/c6run-dsparmtool.

[20] Texas Instruments (2012). *C6x Software Libraries.* http://processors.wiki.ti.com/index.php?title=Software_libraries.

[21] Texas Instruments (2012). *C6RunLib Documentation.* http://processors.wiki.ti.com/index.php/C6RunLib_Documentation.

[22] TI E2E Community (2009). CERuntime_exit() and DSPLink. http://e2e.ti.com/support/embedded/linux/f/354/t/6089.aspx.

[23] Sarris, Zak (2001). *SURVEY OF UAV APPLICATIONS IN CIVIL MARKETS.* Technical University of Crete, Greece. http://med.ee.nd.edu/MED9/Papers/Aerial_vehicles/med01-164.pdf.

[24] Wikipedia (2012). *Avro Canada VZ-9 Avrocar.* https://en.wikipedia.org/wiki/Avro_Canada_VZ-9_Avrocar.

[25] Wikipedia (2012). *Quadrotor.* https://en.wikipedia.org/wiki/Quadrocopter.

[26] Corporaal, Henk (2012). *Embedded Visual Control 5HC99.* http://www.es.ele.tue.nl/~heco/courses/EmbeddedVisualControl/index.html.

[27] Parrot (2012). *Ar.Drone.com.* http://ardrone.parrot.com.

[28] Wendel, Andreas et. al. (2012). *Dense Reconstruction On-the-Fly.* YouTube video, http://www.youtube.com/watch?v=N2HpQ3pht7k.

[29] Huang, Albert S. et. al. (2011). *Visual Odometry and Mapping for Autonomous Flight Using an RGB-D Camera.* Int. Symposium on Robotics Research (ISRR), Flagstaff, Arizona, USA, Aug. 2011. http://people.csail.mit.edu/albert/pubs/2011-huang-isrr.pdf.

[30] Rosten, Edward; Drummond, Tom (2006). *Machine learning for high-speed corner detection.* European Conference on Computer Vision. http://www.edwardrosten.com/work/rosten_2006_machine_poster.pdf.

[31] Wikipedia (2012). *Bayer filter*. https://en.wikipedia.org/wiki/Bayer_filter.

[32] Zuliani, Marco (2012). *RANSAC for Dummies*. http://vision.ece.ucsb.edu/~zuliani/Research/RANSAC/docs/RANSAC4Dummies.pdf.

[33] Horn, Berthold K.P. et. al. (1987). Closed-Form Solution of Absolute Orientation Using Orthonormal Matrices. http://people.csail.mit.edu/bkph/papers/Absolute_Orientation.pdf.

[34] Triggs, Bill et. al. (2000). *Bundle Adjustment — A Modern Synthesis*. http://lear.inrialpes.fr/pubs/2000/TMHF00/Triggs-va99.pdf.

[35] OpenCV (2012). *OpenCV*. Official website. http://opencv.org/.

[36] Arduino (2012). *Arduino*. Official website. http://www.arduino.cc/.

[37] Aeroquad (2012). *Downloads - aeroquad - An Arduino based four rotor R/C helicopter or quadrocopter*. http://code.google.com/p/aeroquad/downloads/list.

[38] Fuzesi, Szabolcs (2012). *Static Thrust Calculator - STRC*. http://personal.osi.hu/fuzesisz/strc_eng/index.htm.

[39] Arduino Forum (2012). *Controlling A Brushless Motor*. http://arduino.cc/forum/index.php/topic,20594.0.html.

[40] SparkFun Electronics Forum (2012). *Connecting an ESC to an Arduino*. https://forum.sparkfun.com/viewtopic.php?f=32&t=32759.

[41] EZRover (2012). *Arduino & ArduMotion and ESC Speed Controllers & Servos*. http://ezrover.com/2012/06/01/arduino-ardumotion-and-servos/.

[42] Mirror Image (2010). *How Kinect depth sensor works - stereo triangulation?*. http://mirror2image.wordpress.com/2010/11/30/how-kinect-works-stereo-triangulation/.

[43] BeagleBoard.org (2012). *hardware-xM*. http://beagleboard.org/hardware-xM.

[44] Twigg, Christopher (2003). *Catmull-Rom splines*. http://graphics.cs.cmu.edu/nsp/course/15-462/Fall07/462/assts/assn2/catmullRom.pdf.

[45] Two YouTube videos about casting carbon fibre. http://www.youtube.com/watch?v=Ybyh6Q9MBgE and http://www.youtube.com/watch?v=IAdVO8Rkv6c.

[46] Carbonwinkel.nl. Shop for carbon fibre parts and tools for casting. http://www.carbonwinkel.nl/.