Android Robot Documentation

Matti Jokitulppo G8049

Course IIO60320

Version 1

31.03.2014

Software Engineering

JAMK University of Applied Sciences

Table of contents

[1 Project introduction 4](#_Toc405889754)

[2 Objectives 4](#_Toc405889755)

[3 Glossary 4](#_Toc405889756)

[3.1 Arduino Uno 4](#_Toc405889757)

[3.2 I²C 4](#_Toc405889758)

[3.3 UART Serial 5](#_Toc405889759)

[3.4 Raspberry Pi 5](#_Toc405889760)

[3.5 Servo motors 5](#_Toc405889761)

[3.6 DC Motors 6](#_Toc405889762)

[4 The application 6](#_Toc405889763)

[4.1 Overview 6](#_Toc405889764)

[4.2 Activities 7](#_Toc405889765)

[4.2.1 Video feed view 7](#_Toc405889766)

[4.2.2 Arduino pin control view 9](#_Toc405889768)

[4.2.3 Settings view 10](#_Toc405889769)

[4.3 Application use cases 11](#_Toc405889770)

[4.3.1 Case 1 – First-time launching of the application 12](#_Toc405889771)

[4.3.2 Case 2 – Displaying custom messages on the robot 12](#_Toc405889772)

[4.4 Application classes 14](#_Toc405889773)

[4.4.1 MjpegInputStream & MjpegView 14](#_Toc405889774)

[4.4.2 JoyStickView 14](#_Toc405889775)

[4.4.3 ApplicationState 14](#_Toc405889776)

[4.4.4 BluetoothStreamManager 14](#_Toc405889777)

[4.4.5 Feed 15](#_Toc405889778)

[4.4.6 Settings 15](#_Toc405889779)

[4.4.7 PinControl 15](#_Toc405889780)

[5 The robot 15](#_Toc405889781)

[5.1 Introduction 15](#_Toc405889782)

[5.2 Preliminary parts list 16](#_Toc405889783)

[5.3 Deployment diagram 17](#_Toc405889784)

[5.4 Some pictures 19](#_Toc405889785)

[5.5 Functionality 20](#_Toc405889786)

[5.5.1 Power management 20](#_Toc405889787)

[5.5.2 The wiring 20](#_Toc405889788)

[5.5.3 Communicating between Android application and robot 21](#_Toc405889789)

[5.6 The Arduino code 23](#_Toc405889790)

[5.7 How to use the robot 23](#_Toc405889791)

[6 Project assessment 23](#_Toc405889792)

[6.1 Overview 23](#_Toc405889793)

[6.2 Development timeline 23](#_Toc405889794)

[6.3 Issues during development 25](#_Toc405889795)

[6.4 What I’d do differently. 26](#_Toc405889796)

[7 Project workload 27](#_Toc405889797)

# Project introduction

The subject I chose for my final project is a relatively simple robot, that the user can control using an Android phone or other such compatible device. It also comes mounted with a webcam so you can actually see where you’re going. The user can also send messages to display to the robot through the application, and make the robot “talk”. The robot itself will be driven by an Arduino Uno-microcontroller that communicates between the application through Bluetooth, and on-board video processing and webcam server hosting is done by a Raspberry Pi, which has a USB webcam and a Wi-Fi dongle attached to it. You can also toggle the state of Arduino’s digital pins on and off and PWM analog pins from 0 to 255. This can be used, for example, to light up LEDs and control piezos and such.

# Objectives

* Robot can move around on wheels, with pan + tilt functionality for camera.
* Robot can be controlled by an Android device
* Robot can stream video to phone.
* User can send messages the robot can display
* Android application and embedded code should gracefully handle cases where Bluetooth/Wi-Fi signal is abruptly cut out/weakens.
* Robot should have lots of pretty blinking lights and fancy sounds, because why not.
* Arduino’s unused pins should be able to be controlled by the application.

# Glossary

## Arduino Uno

Arduino Uno is a popular open-source microcontroller. Programmed using C or C++, it’s intended to be an easy and inexpensive way for students and hobbyists to ease into electronics. The platform is designed to be very modular, and there exists a large variety of different add-ons and components to enhance the Arduino’s functionality. An example of such a component is the Adafruit Motor Shield, used in this project to drive the motors and servos that allow the robot to move around and interact with the physical world.

## I²C

I²C is a communication protocol originally designed by Phillips Electronics in the 1980s. It’s meant for communication between multiple integrated chips, especially in situations where there exists one master chip and one or more slave chips. It consists of a clock line (SCL) and data line (SDA). The devices connected to the I2C bus are either masters or slaves. In this project, I²C is used to control and animate the eyes of the robot.

## UART Serial

Another method of communication between microchips and other devices is UART serial, although this one is quite a bit older than the aforementioned I²C. Unlike I²C UART communication is two-way, which means there’s no feasible way to communicate between more than two chips at the same time. The rough equivalent of I²C’s SCL and SDA are UART’s TX (transmit) and RX (receive). Since there’s no clock line, the data transfer amount per second must be negotiated in advance, before starting communications. In this project, UART is used to animate the robot’s mouth, and communicate with the Arduino and the HC-05 Bluetooth chip.

## Raspberry Pi

The Raspberry PI is a small, cheap single-board computer. Originally meant as a teaching aid, it has gained much popularity in the maker community as being a portable, cheap solution to adding “intelligence” to embedded projects. Basically it’s a tiny computer that runs Linux, with full support for devices such as keyboards, monitors, mice, webcams and all the other necessities.

## Servo motors

A servo is a special type of motor that’s controlled by electronic pulses of varying lengths. The timing of these pulses tells the servo which position it should move to. Generating these pulses is extremely timing dependent, and therefore using a Raspberry Pi ton control one is not a good idea. Since the Pi is running an operating system with many processes taking up different amounts of resources at times, it’s not guaranteed a length of code is always run at the same exact interval. This can cause the servo to jitter, since the timing may be off by a few tenths of a millisecond, causing the servo to think it should move a couple of degrees.

Most servos have a limited range of movement, usually from 0 to 180 degrees. There’s a ton of different hobby servos on the market, from cheap and tiny ones no bigger than your thumb to models costing up to hundreds of dollars, with features such as acceleration and temperature tracking, not to mention enough torque to literally rip off your arm.

## DC Motors

A DC motor is a type of motor with a magnetic coil inside. When an electric current passes through the coil, the magnetic force generated proceeds to turn the motor. The current is passed through a commutator before entering the coil, which switches the direction of the current at the apex point, so the spinning continues The speed of the motor can be controlled by limiting the current, and the direction is naturally affected by the current of the direction.

# The application

## Overview

The main functionality of the application is to function as a controller, basically. Most of the end user’s time is spent in the video feed view, wherein the user can interact with the robot with an interface resembling modern gamepads with joysticks and buttons and such.

The other main feature of the application is the manipulation of the robot’s Arduino’s pins. The user can switch their state on and off, or set them to pulse at a certain wavelength. This allows the user to add more components to the robot, at their discretion. The user can also possibly read values measured by the Arduino’s analog sensors, but it remains to be seen whether or not such a feature is feasible to implement.

The user can also change options and preferences from a separate view. These include things like the webcam server IP address and used port, and the update speed for the Bluetooth module.

## Activities

### Video feed view

### 



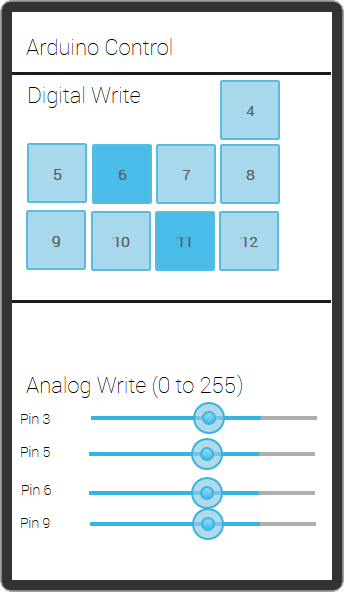
Picture 1 – Video feed layout screenshot – Mockup vs. real thing

This is the part of the application where most of the action happens. The majority of the activity view is taken by the video feed, with some transparent controls located at the bottom of the There are two “joysticks” on the bottom of the screen. The left joystick is used to control the movement of the robot. So for example, if you push the left joystick to the left, the robot will start turning and so on. The right joystick is used to pan and tilt the camera around. The farther away from the center of the joystick you pull the faster the robot will move and react, at least in theory!

The button on the bottom opens up a dialog with a text box. In it, you can type in text for the robot to display. The robot will then scroll this text on a LED matrix, while making a bunch of really adorable beeps and boops.

It came out pretty similar to the mockup, but I decided to scrap the idea of the pin control button on this particular screen, as It didn’t really fit right anywhere

### Arduino pin control view



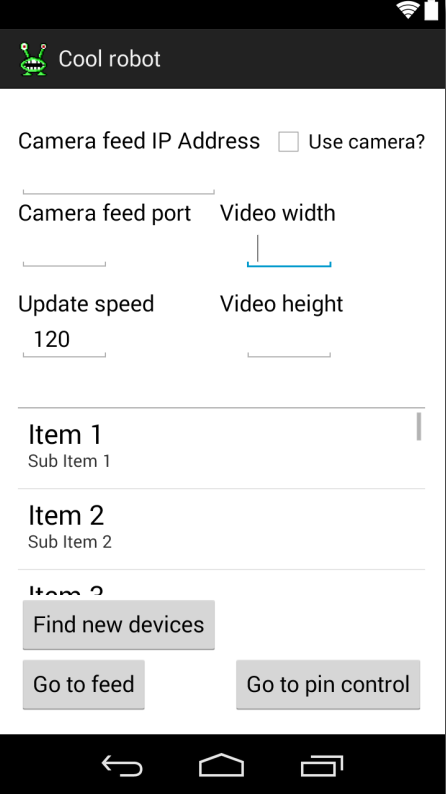
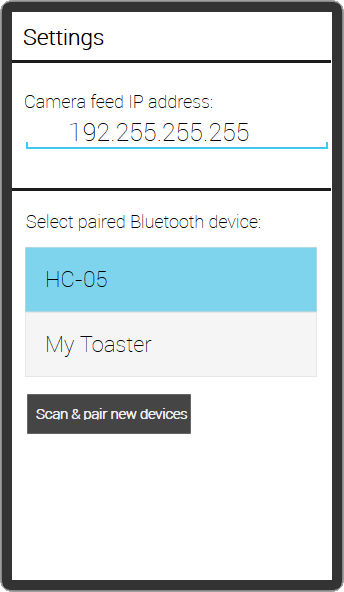
Picture 2 - Arduino pin control layout – Mockup vs. real thing

In this activity, you can change the state of the robot’s different digital pins. The digital pins on the Arduino are either on or off, and this is presented in the application as a grid of togglable buttons. This way you can add new components to the robot, turn on LEDs , emit sound from speakers etc. Some of the pins are reserved for the operation of the robot, and are not available for the user.

On the bottom part of the activity there are a set of sliders you can use to PWM the digital pins on the Arduino. AnalogWrite tells the Arduino to generate a steady square wave with a cycle of 0 to 255. This can be used, for example, to dim a LED at a varying brightness or drive a motor at different speeds.

This particular activity came out pretty much as I expected it to. The only notable difference is that, in the end, I had less pins available to use than I had expected.

### Settings view



Picture 3 - Settings activity – Mockup vs. real thing

This is the view where the application starts in, so in essence you might call it the “main” activity of the whole app. In it, you can for example configure camera settings. Since the feed is sent via Wi-Fi, the Raspberry’s IP address might not be the same in all cases. Therefore it makes the most sense to let the user configure its IP address to whatever they wish. You might also wish to disable the entire feed, if it lags too much on your phone.

You can also change the update speed in milliseconds for the video feed view. This controls how often data is sent to the robot. It’s definitely recommended to keep this well above a 100 milliseconds, the robot has a bad habit of freaking out and acting erratically if the Bluetooth module crashes due to too much data in its buffer! This lead to some fun bugs during the development process.

From the settings you can also configure and pair new Bluetooth devices. The dark gray button at the bottom initiates scanning new Bluetooth devices and appends them to a list of Bluetooth devices, which is loaded from your phone’s memory at startup. From this list you can select devices and attempt to pair and form a Bluetooth connection, after which you can move on to other activities.

This one came out pretty similar to what I had planned out, although the app has a lot more configurable parameters than I originally had planned.

## Application use cases

The following Use case diagram attempts to illustrate the first-time launching of the application, focusing mostly on the necessary user configuring like pairing Bluetooth devices.

1. User launches the application
2. App creates an intent to enable necessary device permissions
3. Having agreed to allowing the application to access the phone’s Bluetooth features, the application moves to the Settings view.
4. The user scans his surroundings for nearby Bluetooth devices, and the Settings view displays these to him/her in list form.
5. The application initiates the Bluetooth connection forming and pairing process.
6. The application switches to the video feed.

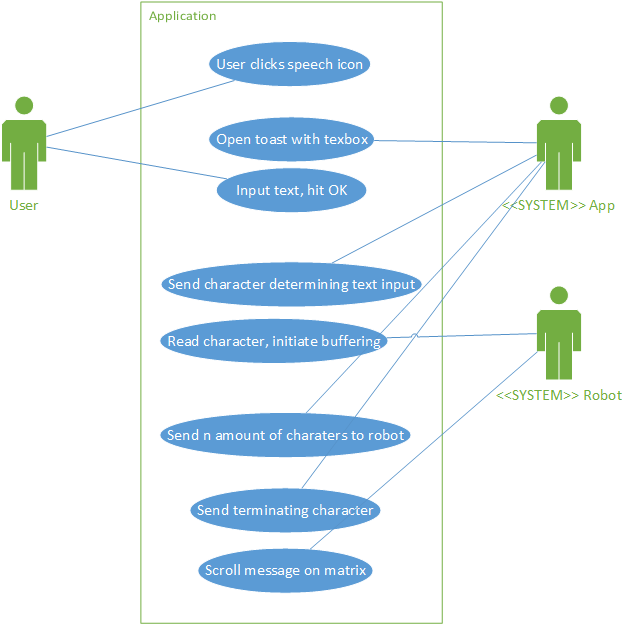
### Case 1 – First-time launching of the application



### Case 2 – Displaying custom messages on the robot

The following use case concerns the user sending messages to the robot, and attempts to sketch out the inner workings of the application-to-robot protocol.

1. User clicks the speech bubble icon on the feed view.
2. The application launches a pop-up with a text box.
3. The user inputs the message and hits ok.
4. The application adds a character indicating the beginning of a new message to the text and sends it to Arduino
5. The Arduino reads the char and initiates buffering of the message, byte by byte.
6. The Arduino sees the terminating character.
7. The Arduino prints the message to the Matrix.



## Application classes

### MjpegInputStream & MjpegView

The camera feed uses the popular MJPEG (Motion JPEG) format to read and handle data from the webcam server hosted by the Raspberry Pi. The Pi is hosting a GStreamer-pipeline, which is a popular open-source framework for handling multimedia. MjpegInputStream is the bytestream that fetches data from the pipeline, and MjpegView is the UI widget which displays it. These two classes aren’t my own work; they’re from a GStreamer-specific fork of the original MJpegViewer-project for Android. See here: <https://bitbucket.org/coisme/simplemjpegview_gst/wiki/Home>

### JoyStickView

This is a nice joystick UI component used on the video feed. Not much to say about this one, it’s got an event-listener that updates on the joystick moving, and you can get it’s angle, power and direction out of it. It’s originally developed by GitHub-user Zerokol, with a few modifications made here and there by me to ensure best fit for my needs (<https://github.com/zerokol/JoystickView>)

### ApplicationState

This one’s a hack job, I’m afraid! As much as I’m averse to using big global singletons in OOP-languages like Java, extending the Android Application class was the best possible solution to pass the Bluetooth stream around between different Activities, since a stream object could not be passed between activities with Parceable or Extras. That’s pretty much the sole job of this class.

### BluetoothStreamManager

This class is responsible for sending data to the Arduino. It utilizes its own thread, which runs continuously in the background during the whole lifetime of the application. Other activities can then use this class to send commands to the Arduino, which are held in a ConcurrentLinkedQueue data structure and written to the paired device’s output. It also holds a reference to the current activity, so it can display an error dialog if it loses connection to the Bluetooth module, which prompts the user to return to Settings and try to repair devices. The whole queue system is a little over-engineered for this particular app, in the sense that there’s not going to be a whole lot data being sent to the module, so concurrency doesn’t really matter.

### Feed

This is the class corresponding to the Feed activity. Its main function is to maintain a motor command byte array, which is sent to the Arduino at a set interval. This byte array contains values that tell the robot what it should do in terms of movement. Two joysticks on the screen are used to get user input, which these motor valued are interpreted from. More on the different kinds of bytes and their meaning on the Arduino side later. The activity also contains a button that opens up a dialog where the user can send things for the robot to say, and an implementation of the AsyncTask class, which is used to fetch the video feed asynchronously, to name a few things.

### Settings

This one’s pretty boring to be honest! It just holds some input fields used for configuration for the rest of the app, and keeps them saved with SharedPreferences. Besides the usual Activity jazz, the only thing worth mentioning is the pairing of Bluetooth devices, which is handled in this class. There’s a list of Bluetooth devices loaded from device memory at the bottom of the activity, and you can press a button to initiate scanning for new, unknown devices. If any are found they’re then appended to the end of the list. Clicking on an item in the list initiates pairing with a Bluetooth device. If the device is new and never been paired with, it opens up the default Android pairing request dialog, which asks for a device pin code. The Bluetooth device is then paired with the phone and the output stream is passed on to BluetoothStreamManager, and the user can proceed to other activities.

### PinControl

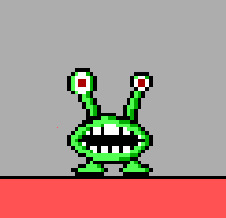
This activity/class holds a big table of buttons, and some seekbars. Pressing any of the buttons changes the state on the Arduino, from 0 volts to 5 volts. Dragging the sliders on the seekbars changes the pins states from 0 volts to 5 volts gradually, depending on how far you push the slider. You can use this Activity to, for example, gradually light up LEDs. It’s pretty similar to the Feed activity in a way, it too works by interpreting user input, translating them to bytes and sending them off to the Arduino.

# The robot

## Introduction

Being a nerd, I’ve always been somewhat interested in science-fiction. Naturally, building an actual functioning robot has always been a dream of mine! The robot I built for this course is essentially a two-wheeled rover complete with live-streaming video camera, although with some nice lights and SFX to make it extra special.

By the way, the look of the robot was heavily inspired by an enemy from the classic video game series “Commander Keen”, developed by id Software in the 90’s. It was one of my favorite game series as a child, and I consider this project an honorable tribute to one of the best game developers of their time.



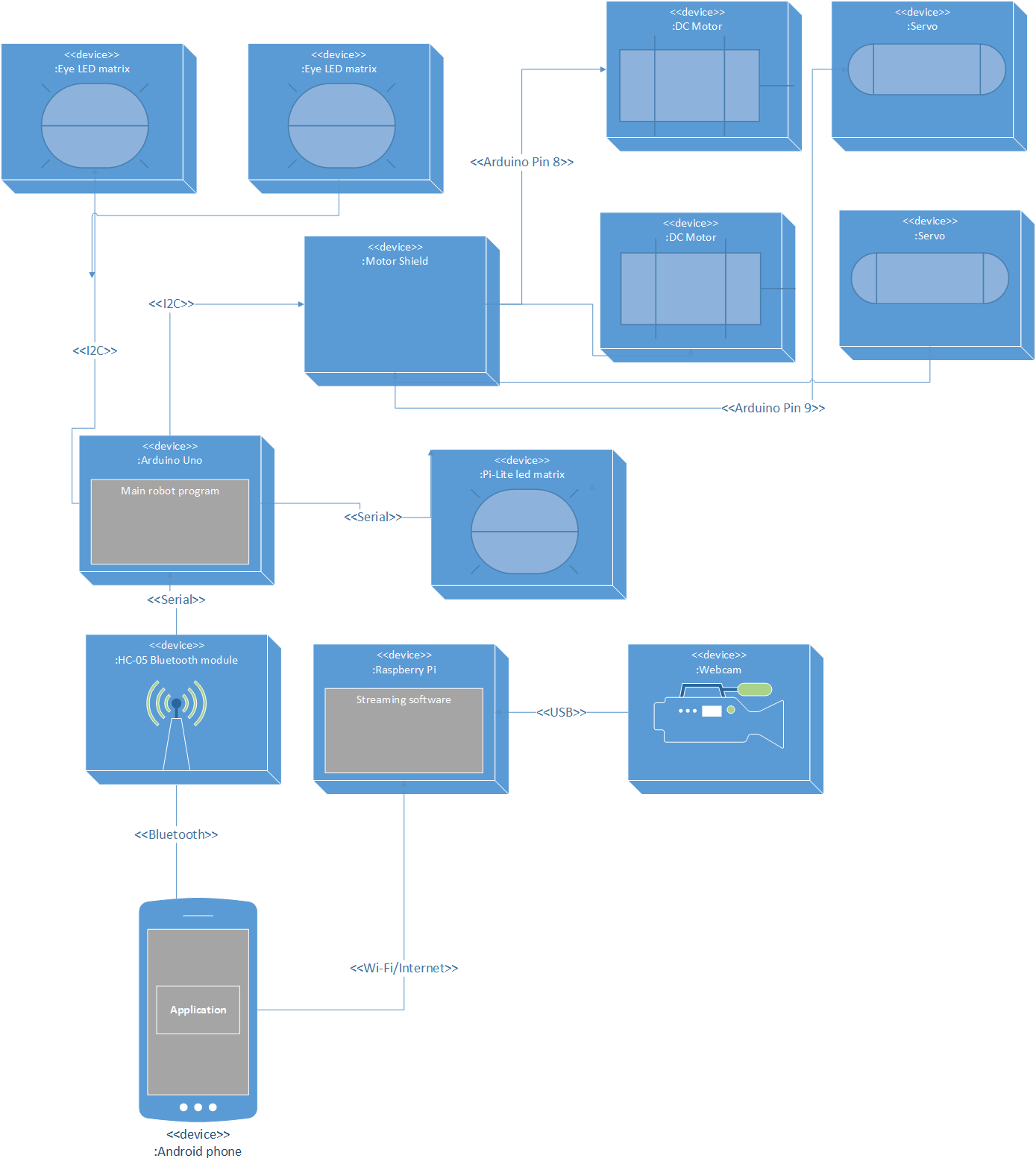
Picture 3 – a Garg, from Commander Keen 1

## Preliminary parts list

|  |  |  |
| --- | --- | --- |
| Part needed | Link to item purchased (if applicable) | Cost |
| Robot chassis | <http://tinyurl.com/nnwl7rl> | € 33,17 |
| Arduino Uno (Chinese knock-off) | <http://tinyurl.com/mart4dh> | € 11,05 |
| Pan + tilt servo rig for camera | <http://tinyurl.com/qj9q3rc> | € 24,36 |
| Bluetooth module | <http://tinyurl.com/lzcq7as> | € 9,95 |
| Motor shield | <http://tinyurl.com/nychgsp> | € 17,56 |
| 2x LED Matrix for eyes | <http://tinyurl.com/kllbes7> | € 20,00 |
| Pi-lite LED shield for mouth | <http://tinyurl.com/pxtm2mm> | € 25,38 |
| Raspberry Pi model B | N/A | € 30,00 |
| Webcam | N/A (borrowed from school) | N/A |
| Wi-Fi dongle | N/A (borrowed from school) | N/A |
| 5x AA+ batteries | N/A | € 10,00 |
| 5V rechargeable battery bank | <http://tinyurl.com/lhsavfx> | € 30,00 |
| Assortment of screws, nuts, wires etc. | N/A | € 15,00 |
|  |  |  |
| Total: €226,47 |  |  |

## Deployment diagram

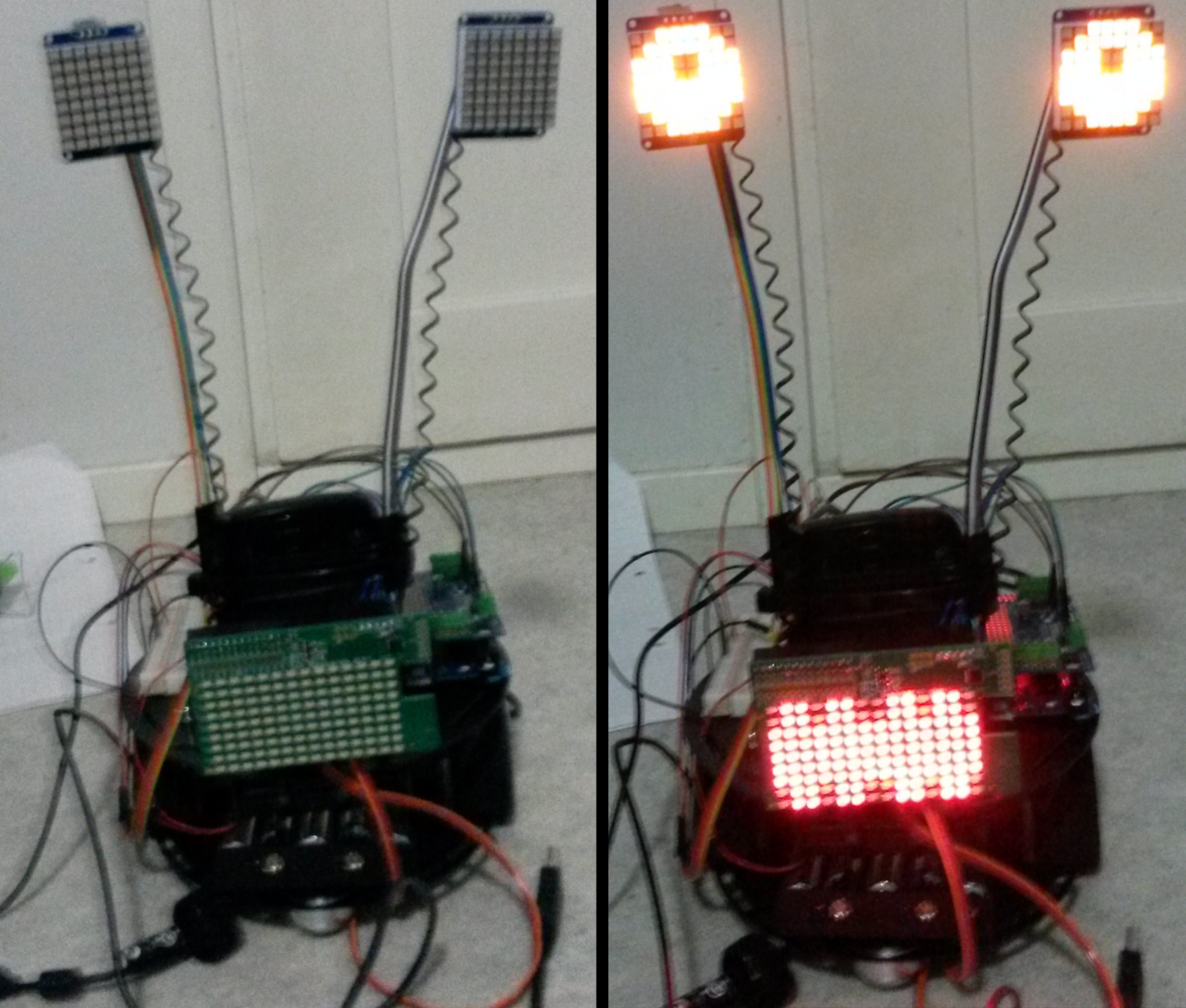
Here’s a diagram on how all the bits and pieces in the whole ensemble communicate and interact together. Power sources are not included in the picture, more on those later.



Picture 4 - Project deployment diagram

## Some pictures

Here are a couple of quick shots of the finished robot.



Picture 5 – First prototype of robot, turned off and on

Isn’t he/she a beauty? The eyes are animated, and they move around quite convincingly, if I say so myself. Originally, the mouth was also animated, but it proved to slow down the code too much, so that idea was unfortunately scrapped. I didn’t take any pictures during the build process itself, which is a bit of a shame in my opinion. The LED matrix and web camera are attached with Blu-tack, since the camera is loaned from the school, and I don’t want to permanently attach anything to the robot. Besides, some of the components used are rather pricy, and I might want to reuse some for future projects.

## Functionality

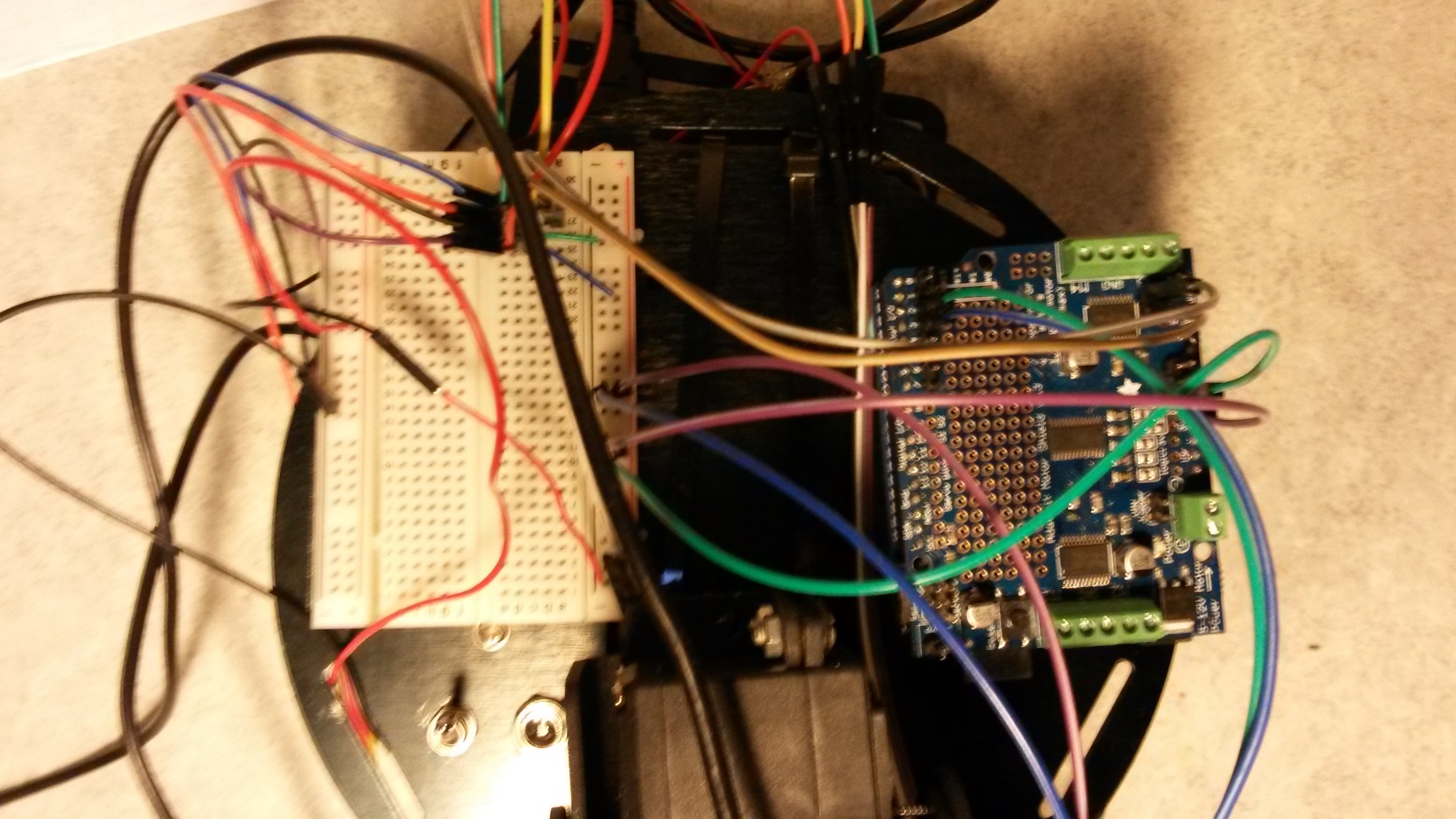
### Power management

Most of the components are powered by a portable USB power bank zip tied to the bottom of the upper plate of the robot. This power bank can output up to 3 amperes at 5 volts. The power bank has two outputs, one of which is used strictly to power the Raspberry Pi, while the other powers up everything else. As for power consumption, the Arduino takes about 40 mA, the Raspberry Pi ~700mA, the mouth ~50mA, both eyes together ~350 mA and the Bluetooth module ~8 mA. This adds up to around 1200 mA/H. The power bank is rated at 10000 mA, which means at full charge the “brains” of the robot can function for about 8 hours, give or take.

While one is planning out a project that incorporates microcontrollers interacting with servos or motors or other “heavy-duty” components, one must always remember to separate their power supplies. This avoids unnecessary noise on the signal line, sudden drops in voltage that can mess up the microcontroller and other nasty issues. In this project, the motors and servos are powered by a 5x AA battery pack. It outputs about 7, 5 volts. The battery pack is hooked up to the Arduino’s motor shield, which is an add-on for the Arduino specifically meant to control and power motors and servos.

### The wiring

For quick prototyping, I’ve glued on a breadboard on the top plate of the robot. Wiring in this project isn’t all that exciting to be honest! Here’s a messy and not-so-helpful top-bottom picture of the wiring.



Picture 6 - Top-down view of wiring

### Communicating between Android application and robot

Every message send to the microcontroller is sent and read as byte arrays. As a side-note, java bytes are from -128 to 127, while C bytes are from 0 to 255. Henceforth, only values from 0 – 127 are used to talk to the Arduino from the application, since the Arduino interprets negative bytes as unsigned. The values from 0 to 127 are multiplied by 2 on the Arduino side to get the correct numbers.

Here’s an example of sending a simple movement command. The brackets represent the cells in the array:

[[123], [70], [64], [66], [127], [1]]

The first character bracket tells the microcontroller that the following five characters should be interpreted as a movement command, and not as a text message or anything else. The next byte tells the direction the leftmost motor should move in. It is either the ASCII value F (70) for forward, B (66) for reverse or R (82) for staying put. The third byte is the speed the leftmost motor should exert, from 0 to 127. In this case, the leftmost motor is told to spin forward at max speed. The next two characters are the same, except for the right motor. In this case, it tells the motor to start moving backward at half-speed. The last character is related to positioning the two servos. It is a byte from 0 to 9, which tells the servo, which direction it should move to. You can think of these as the eight cardinal directions, plus an extra value for implicating staying still. There’s no need for a delimiting byte, since the Arduino code knows the motor command is always six bytes in length.

Here’s a method of displaying text on the robot’s “mouth”. This time, let’s use ASCII values.

z …. \n

This one is pretty simple. The z-symbol (or the byte corresponding to it, whichever way you look at it) marks the start of a new displayable message, followed by an undefined amount of characters to scroll on the matrix, and finally the newline character ends it.

Finally, I have to implement a way to adjust the state of the Arduino’s pins. Handling digital pins is very simple, since their state is either on or off. We can just send a single byte corresponding to the pin we want to toggle.

[[120], [5], [60]]

The first byte implicates that the command should be interpreted as AnalogWriting a pin. The following value tells the microcontroller which pin it should set the state of. The next character tells the microcontroller the value it should write. In this case, it’s 60, which means full wave length. The command is then terminated with a closing character.

You can also toggle the pins on and off, from 0 to 5 volts. It works in sort of the same way as the AnalogWrite.

[[121], [5], [0]]

The first byte is, as always, the delimiter. The second byte is the number of the pin in question, and the third value is either 0 or 1, for off or on, respectively. Like the motor commands, AnalogWrite and Toggle neither require an ending delimiter.

## The Arduino code

In essence, the embedded code is essentially one big state machine, reading bytes from the Bluetooth module and reacting accordingly. The robot has an enum of different states it can exist in, which are “NOTHING”, “MOTOR”, “MESSAGE”, “TOGGLE”, and “PINPWM”, which are rather self-explanatory in my opinion. The main loop checks for new bytes in the serial buffer of the Bluetooth module in the handleSerialInput()-method which also handles changing robot state. If, for example, the incoming byte is 123 and the robot’s current state is NOTHING, the robot knows to go to MOTOR-mode, and the next 5 bytes are related to moving the wheels and the servo. After it’s gotten the necessary values, the state is reset to NOTHING. The main loop also animates the robot’s eyes and handles making beep sounds on a small speaker if the robot is currently scrolling a message on the LED matrix.

## How to use the robot

1. Plug in Arduino and Raspberry USB cables to power bank
2. Push hard-to-reach button on the bottom of the power bank
3. Turn on engines with the switch on top of the plate
4. Wait for the webcam light to turn on¨
5. Pair Bluetooth module from application
6. You are done!

# Project assessment

## Overview

All in all, I think the project came out really great. I managed to implement all the features I originally had in mind and then some. The robot looks cool, is pretty fun to drive around and work. I’m expecting a 5 from this course, I think my project is innovative, well- implemented and different, and also quite broad in scope for being made by one person. Not to mention, my grade-point average depends on it ;).

## Development timeline

I started to work on the project quite early, since I had a pretty clear idea in my head as to what I wanted to achieve from the beginning. First, I started to work on the robot, because I still wasn’t confident enough to start working on the android application as the course was still at such an early stage.

After assembling the chassis and attaching the servos to the top plate, the very first thing I started to work on was the eyes and mouth. My original plan was to buy acrylic fake teeth from eBay and make a creepy-looking mouth from molding putty. However, during a run I had the idea to use a LED matrix for the mouth, so I can scroll messages on it. Originally I planned for the mouth to be animated, too, but that idea had to be scrapped as it slowed down the Arduino program too much for my taste.

After figuring out the mouth, I started to work on the eyes. I had already used the Adafruit LED backpacks in an earlier project to great success, so I decided to use them for the eyes. I searched high and low in many hardware stores for springs, to make some eye stalks for the robot, and to get that bouncy lifelike look during movement. After declaring my search a failure, inspiration struck while I was at an arts & craft shop, and a girl’s headband caught my eye. I bought one for 1€, I think, sawed it in half, drilled a couple of holes in it and voila! I had my eye stalks.

I would’ve started work on the Android-to-Arduino communication sooner, but I had some issues with the Bluetooth module. I decided to save a couple of bucks, and ordered an unsoldered HC-05 module off eBay. Boy, was that a mistake! The module was so tiny soldering on the correct wires proved to be a nigh-impossible task for clumsy old me. A couple of attempts and some burnt fingers later, I swallowed my pride and decided to buy a pre-soldered one.

After a couple of weeks, the new module arrived and I was ready to start working on the Android side of the project. Before week 44 I had made some research on existing implementations on Android Bluetooth communication, but nothing that major. I had already picked out the joystick widget I found on GitHub, and decided to use it instead of spending my time working on an already-solved problem. First thing I did was start working on the settings activity. It didn’t take me long to get my phone to scan for available Bluetooth devices, and pretty soon I had working listing and connecting done. There were some issues with pairing the devices, as the HC-05 didn’t seem to accept the UUID I had generated for my app, but after a brief Google search I found a working UUID, and the pairing was a success.

After pairing was done, it was time to work on the Feed activity, and actually sending bytes to the Arduino! Debugging the embedded application proved to be quite difficult. Normally, you just stick an USB cable into the Arduino, and you can write stuff into the serial monitor on the Arduino IDE, and debug that way. However, I had taken up the Uno’s only serial port with the Bluetooth module! I solved the issue by scrolling text on the mouth matrix, which proved to be an okay solution. I finally got some bytes from the Android to the Arduino.

It was at this time that I found, to my displeasure, that Java considers bytes to be from -128 to 127, and C considers bytes to be unsigned like God intended. Oddly, the bytes would lose their signed status while being transferred from Android to Arduino (-128 became 128, for example). This makes no sense to me, as a byte should be send out as a eight bits, nothing more. After cursing out Ryan Gosling, I decided to only use bytes from 0 to 127 on the Android side. These values would then be doubled on the Arduino side, so that 127 becomes 254 etc.

After that specific ordeal, it was time to wire the motors and servos and let her rip! After some embarrassing mix-ups on the communication protocol, I finally got the damn thing to move. And by God, did it move. Way too fast in fact. At this point in time the Arduino application was still prone to unexpected crashes, which would send the robot straight into the nearest wall/chair/open container of liquid. After toning down the speed on the Arduino side, and lowering the amount of data being sent to the Bluetooth module to stop crashes, the worst of the crashing stopped, and the robot started to work pretty nicely. There are still some issues with the Bluetooth module crashing, but at least now the robot just stays put instead of careening off into new adventures on its own whenever it feels like it.

When I got the robot moving around (at about week 48) I started to work on controlling the pins. This didn’t take all that long; I spent most of my time mucking around with the activity layout, so it looked pleasing to my eye. Sending the messages to turn pins on and off, or PWM them, turned out to be pretty easy after implementing the movement. Originally there were some issues with the seek bars sending too much information to the Bluetooth module, causing some glorious crashes. Random characters being displayed on the mouth, lots of beeping, driving around with reckless abandon etc.

After I got the Pin Control activity up and going, I started to work on the Pièce de résistance, the webcam feed. I had my eyes on a particular Android project, which implemented a Gstreamer pipeline from Android to Raspberry Pi, which seemed to suit my needs perfectly. A Git clone and some configuring later, I got my webcam feed to work! The hardest part about this particular portion of the project was to get the Raspberry to connect to the internet with Wi-Fi. It seems WLAN on Linux is still a nightmare, at least using the default Network-Manager daemon that came pre-installed with Raspbian. As an unlikely solution to connection problems, I found that I could host a Wi-Fi hotspot on my phone, which allowed me to stream video from virtually anywhere. The only problem is, the webcam feed seems to have quite a bit of latency when doing this.

So that about wraps things for this project. It’s been a great learning experience, as I haven’t done anything quite like this before (Android app or a robot). Like I said before, I’m very pleased with the end results.

## Issues during development

Here’s a haphazard listing of some of the problems I encountered during the project, in no particular order:

* Regular run-on-the-mill hardware stores don’t sell screws/nuts with diameter under 3mm, had to buy expensive deluxe nuts from an RC shop for 5€ a pack!
* Too much data being sent to the Bluetooth module caused it to crash, sending the robot into a what seemed like a killing frenzy! Fixed it by configuring the baud rate of the module and toning down the update speed on the app side.
* Java bytes are from -127 to 128 for some reason, while C bytes are unsigned. Transferring negative numbers as integers would’ve required too much overhead (4 bytes vs 1 byte), so I decided to only uses value from 0 to 128 on the Java side. These values are then multiplied by 2 on the Arduino side to get the right numbers.
* LED matrices caused the motors to stop functioning, due to undocumented conflicts in I2C addresses. Solved it by changing the I2C addresses on the matrices after a brief back-and-forth with the manufacturer.
* Discovered a strange kink in the Arduino Servo library. The Arduino has only one actual serial port (which I used to talk to the Bluetooth module), and the mouth matrix also uses serial to communicate with the Arduino. I had to use a technique called bit banging to emulate serial communication using the Arduino’s analog pins, and this caused odd issues with the Servo library, jerkiness and sudden twitching when sending stuff for the mouth to display, stuff like that. Solved it by finding an older version of the Servo library, where the issue seems to be absent.

## What I’d do differently.

Should I redo this project, I would probably use a Bluetooth Low Energy-based adapter. Bluetooth LE is a newer technology than regular Bluetooth (called Bluetooth Classic) used in this project. The main thing I’m interested about it is the promise of reduced latency. Classic Bluetooth has a latency of about 100~ ms, while LE promises speeds up to 3~ ms. The main reason I didn’t use LE technology in this project is that my own phone was using the 4.2.2 firmware, and LE support for Android starts at 4.3!

# Project workload