Final Report
Team Megalodon

Casey O'Rourke (caseyso) ·Jonathan Hirokawa (johiro) · Mike DiNicola (mdinic)

Abstract

Using a Gumstix board as a control hub, we sought to control an RC helicopter through both manual and automatic controls. Manual control inputs were displayed and received via a touch based Qt user interface (LCD touchscreen). Automatic controls operated on the same backend, but included an additional closed loop control system which utilized a Microsoft Kinect to track the helicopter in 3D space. Data from the Kinect was processed on a laptop and then transmitted to the Gumstix via Bluetooth. From there, a user level program performed the necessary calculations to either increase or decrease throttle to maintain altitude. In both cases, communication to the helicopter was achieved by using an Arduino micro-controller to drive a precisely timed infrared LED. Messages were passed to the Arduino from the Gumstix through an interrupt-triggered, parallel, communication system and provided a mechanism to change the helicopter's throttle, pitch and yaw.

Introduction

In recent years, drones (and radio controlled (RC) aircraft more generally) have moved from a fringe hobby to the forefront of both the military and public attention. With potential applications ranging from package delivery, to persistent battlefield tactical capacities, to film and entertainment, drones have changed (and will continue to change) the face of modern aviation [1]. Their use in American airspace has grown to such a degree that the Federal Aviation Administration (FAA) has recently released regulations regarding these new craft and how they may be operated [2]. As this industry continues to grow, drones will increasingly integrate with existing systems to provide new and useful functionality to users.

One of the most popular types of hobbyist drones is the quadrotor helicopter (commonly referred to as a quadcopter). As the name implies, quadcopters make use of four counter-rotating propellers (props) located at the four corners of the craft for both lift and control. This has the advantage of only four moving parts which simplifies construction, but requires careful control as any imbalance in the thrust generated by a prop will result in potentially unstable behavior. While the cost for quad copters has dropped dramatically due to a proliferation of low cost, high power, embedded systems and sensors, most still cost several hundred dollars. A far cheaper alternative currently available for as little as $20 on sites like amazon.com, are coaxial rotor helicopters[3]. These make use of two stacked propellers which rotate in opposite directions. Uncommon at human scale due to mechanical complexity, these are more stable and respond quickly to controls making them popular toys.

With an interest in these RC vehicles, our group set out to integrate the controls of a small helicopter with a Gumstix Verdex Pro (Gumstix) and a Microsoft Kinect . The Gumstix is an embedded Linux system that provides a versatile platform with IO functionality in the form of GPIO pins, an LCD bus, and Bluetooth (BT) radio. The Gumstix communicated with a Syma S107G helicopter through an infrared (IR) communication protocol that was mediated by an Arduino Uno. We initially tried to avoid using an Arduino as an intermediary, but its inclusion proved crucial. It allowed us to divorce the precise timing required by IR communication from the general purpose computing needed to support a touch screen interface and BT communication.

The touchscreen was integrated to allow for manual control and to display the following flight parameters: throttle, pitch, and yaw. After flying the helicopter manually, we quickly noted a high degree of drift and decided to additionally work on developing an autonomous flight system. This system utilized the Kinect to detect the spatial location and send data, which could then be interpreted to counter the drift. It also provided a means to detect unsafe flying conditions such as flying too high or low. Our original plan also called for this autonomous flight system to override the current manual input (via touchscreen), if the user was flying unsafely. Though libraries exist for using a Kinect with Linux directly, we instead opted to perform all Kinect operations on a laptop and then transmit the coordinate data to the Gumstix over Bluetooth [4]. Our reasons for this was twofold: 1) without root access to the lab machines we could not install the necessary tools for the Kinect and 2) this configuration allowed for us to move the Kinect away from lab machine and make use of a more modern communications protocol.

We succeeded in implementing manual user control (all axes) as well as automatic Kinect control, but only for the throttle control. For instance, if the helicopter reached a height which was predetermined to be unsafe (too high), the throttle lowered until the helicopter returned to a safe height. We are, however, only able to run manual or automatic modes separately; the “manual override” functionality proved to be outside the scope of this project, due to interference between the BT and LCD buses.

We initially attempted to complete this implementation without the Arduino Uno using the Gumstix's HRTimer library. HRTimer advertises nanosecond resolution, more than necessary for IR communication, however, due to the limited resources on the Gumstix our attempts to use it for timing the LEDs were unsuccessful. Instead we opted to move all timing responsibilities off board to a dedicated Arduino Uno after discussing the issue as a group and with the lab TA. In summary we integrated the following distinct systems: a Microsoft Kinect, BT radio, a Samsung 4.3 inch LCD touch display, a Gumstix Verdex Pro, an Arduino Uno, and a Syma S107G remote controlled helicopter.

Design Flow

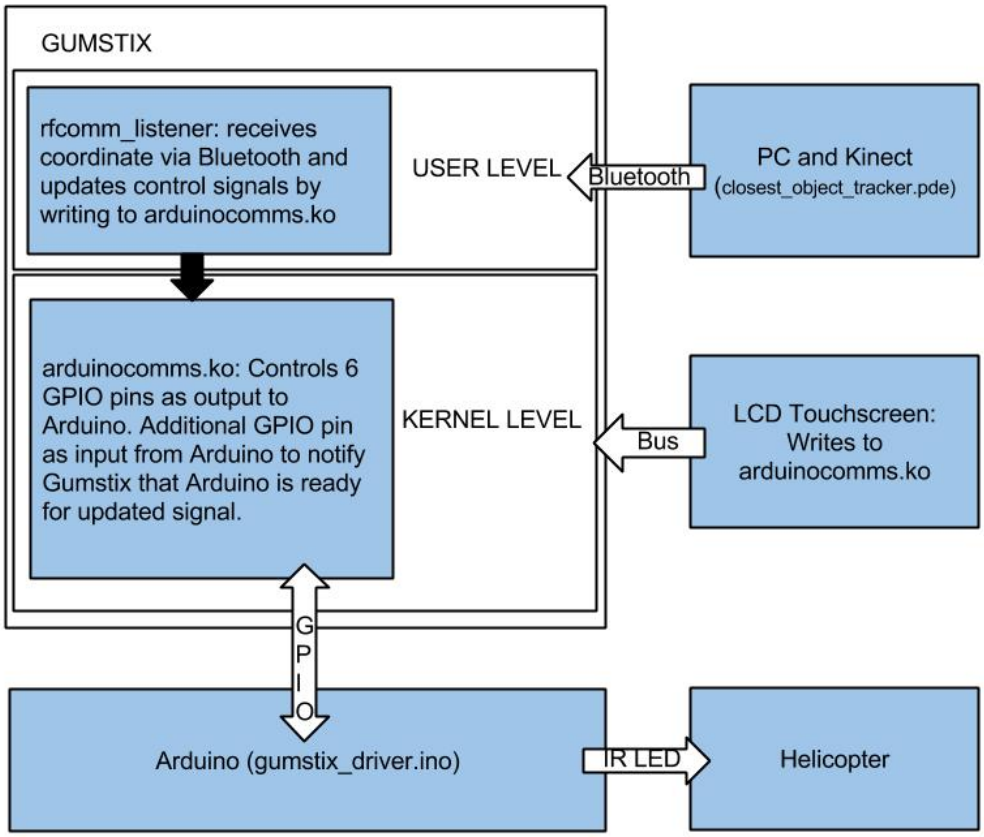

Figure 1. Block diagram detailing design and communication links

The Gumstix Verdex Pro is at the center of our project, as can be seen in the block diagram above (Figure 1). Work started with the development of a kernel module (**Jonathan**) to communicate with the Arduino via 7 GPIO pins. Six were used for sending signals to the Arduino, while one was for receiving from the Arduino. The original Arduino code was found online, but we modified it to communicate with the Gumstix (**Casey and Jonathan)** [5]. The next aspect we worked on was user input via the LCD touchscreen (**Mike)**, which was written in Qt and takes advantage of the kernel modules' write function to pass input on how to change the control signals.

Lastly, a user level program was written to utilize the Gumstix's Bluetooth capabilities **(Casey)**. By binding the Bluetooth connection from the PC to */dev/rfcomm0*, the user level program was able to simply use a system read and grab the x,y,z coordinate data being sent as a string from the PC. The user level program could then write directly to the kernel module with instructions on how to change the control signals. The Kinect code (**Casey and Mike**) was based on code found in *Making Things See* [6], using the example of tracking the nearest object, but was modified to send a signal via Bluetooth to the Gumstix .

We employed a number of group productivity solutions, including Google products (Docs and Drive), Slack, and Github. Google Docs allowed us to collaborate remotely on presentations and reports. Slack is a service which facilitates chat and messaging. All messages are archived and can easily be searched at a later time. This removes the confusion that comes from in-person or phone conversations. Using Git allowed each group member to add or suggest improvements to every piece of the codebase, regardless of who made the original individual contribution.

Project Details

Contents:

a. MS Kinect module (closest_object_tracker.pde)
b. Gumstix
    i. User Level (rfcomm_listener)
    ii. Kernel module (arduinocomms.ko)
    iii. LCD GUI (controller binary; window.h & slidersgroup.h)
c. Arduino
    i. Driver (gumstix_driver.ino)
    ii. Circuit
d. Copter

a. MS Kinect module

To track the helicopter in space, we utilized the Microsoft Kinect. What makes the Kinect such a powerful tool is its depth camera, which allows for depth information to be captured via infrared light. With this data, distinguishing objects becomes a much easier task then when only limited to an RGB camera. To access the Kinect's data, the SimpleOpenNI [7] library was used inside the Processing language and development environment [8]. This can be see in the *closest_object_tracker.pde* file. The Kinect provides a depth map with a resolution of 640 X 480 pixels as an array. Our setup assumed the helicopter would be the closest object and therefore tracking it was relatively straightforward in that we simply looked for the pixel with 'brightest' or closest value to track the helicopter. From there is was simply a matter of writing the coordinates the pixel to the Gumstix via Bluetooth.

b. Gumstix

b.i. User Level:

The user level program *rfcomm_listen.c* was written to handle the Bluetooth connection from the PC/Kinect. By binding the Bluetooth connection to the PC/Kinect to the /dev/rfcomm0, we were

able to interact with the Bluetooth connection via system read call. Once started, the user level program opens up the /dev/arduinocomms file for future writing. The program then waits for /dev/rfcomm0 to open, which occurs when the PC makes a Bluetooth connection. From there it loops constantly reading for Bluetooth input in the format of x,y coordinates (adding the z coordinate would be very straightforward). Once these coordinates are received, a simple controller in the user level program determines whether to send a signal to increase or decrease the throttle (yaw and pitch were ignored). As such, the program simply tries to maintain the altitude of the helicopter.

b.ii Kernel Level:

Communication to the RC helicopter evolved over the course of this project as various solutions were tried and then discarded. Initial plans called for direct control of an IR LED through the use of the GPIO pins and software defined timers. However, after some initial tests of this solution, we decided that a simpler and more robust final product could be achieved by letting a dedicated device handle the precise timing of the IR LEDs. We chose to use an Arduino Uno for the timing as we had one on hand and it allowed us to recycle pre-exising code. The task then changed to communication between the Gumstix and the Arduino.

Two constraints directed our design of Gumstix-Arduino communication. First, timing was a critically important. Since the the Arduino code relied upon delay function calls which suspended activity of the whole board, traditional serial interfaces like I2C were excluded as possible solutions. Second, the Gumstix had relatively few GPIO pins exposed. This meant that we were unable to send absolute values (for example the number 74) in one go as there simply weren't enough open pins to represent each bit. Instead we opted for a hybrid approach. Both the Gumstix and the Arduino had an internal variable for each parameter that was then incremented (+/- 1 or +/- 5) as necessary.

This approach had the advantages of being fast enough to avoid interference with the IR timing, and required only two pins for each parameter (increment/decrement and magnitude). The downside was that without careful management, the internal model of each parameter would drift over time. There was also a delay in response time as we had to increment through a series, but this was minimized by the multipliers. Our first pass of this simply used our existing Gumstix timer code to periodically update the Arduino. Tests of this quickly revealed that many transmissions were missed due to the different clock rates of the Arduino and Gumstix. These results prompted our final version where the Arduino queried the Gumstix for updates thereby trigger an interrupt. The Gumstix then responded by setting its GPIO pins accordingly. User level programs running on the Gumstix interfaced with this kernel module by writing to it's device file. Source code for this kernel module can be found in arduinocomms.c.

b.iii. LCD GUI:

Our controller (figure 3) was designed to mimic the consumer controller for our helicopter (figure 2), with the additional reset button to make up for some of the controllability lost when moving from physical controls. We have not left the trim knob for the user to control, as it is not generally needed mid-flight; this value is hard coded. While the left stick (throttle) is unchanged, the right stick (pitch/yaw) is separated into two sliders. We investigated implementing a joystick in Qt, but this required QML. In the older version used (Qt 4.5.1) QML cannot be used. In order to use the required version (Qt 7.x), we would have had to build our own cross compiler. This was outside the scope of this project. The controller features large sliders, value labels, and buttons, for best usability. This is a customized GUI based on the generic sliders example featured on the Qt Project's 4.8 tutorials page [9]. Colors follow modern flat UI design language.

Figure 2. Syma S107G Controller

Adjusting a slider adjusts its respective value and writes the new values to /dev/arduinocomms. The kernel module only receives an update from the controller when any of the values are updated. This is an improvement over the first design we implemented, which used a timer in the Qt code to continually send to the kernel module. All updates from controller are manual (m) mode updates, as opposed to automatic updates expected from Kinect. This will allow us in a future implementation to include logic which recognizes when both automatic and manual mode updates are being received, at which time manual updates will be ignored.

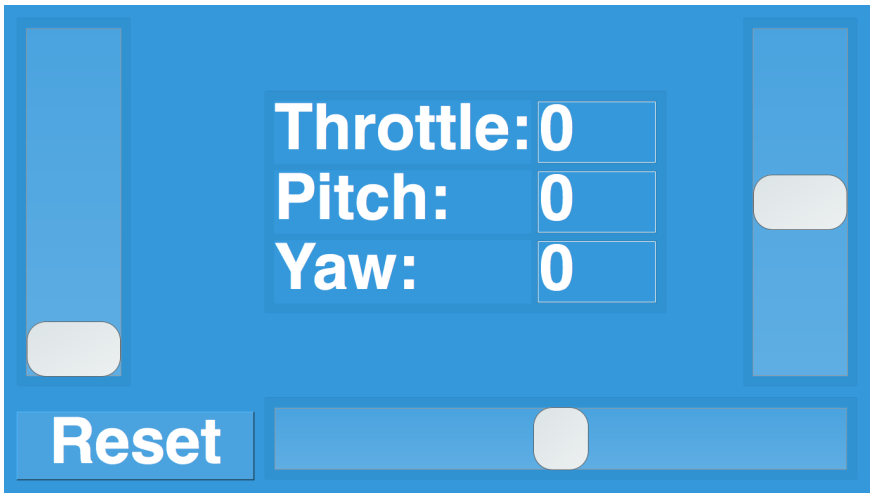

Figure 3. LCD display touchscreen controller

The seven files included in the gumstix_code/controller directory (including Makefile) are required to compile the GUI binary. Source code is written in C, and there is an additional .pro file which is created by Qt and has not been modified. Instructions to install and run are found in README.txt. Versions used are QMake: 2.01a and Qt: 4.5.1.

c. Arduino

c.i Arduino Driver

The helicopter we used was chosen based on the fact that its control signals had been reversed engineered and moreover, Arduino code existed to drive an infrared LED to send those signals [5]. It was originally our plan to port this code to the Gumstix and as mentioned above, while an enormous amount of effort was made to do so, we were unable to get the timing resolution we required. Thus we decided to fall back to the Arduino to drive the infrared LED.

The Arduino *gumstix_driver.ino* makes use of the TimerOne library, which allows for a timer to be initialized and interrupt the Arduino with an attached function *timerISR* every 180ms. This period is simply a parameter of the pre-existing control signal expected by the helicopter and specifies a 50% PWM carrier wave that holds the actual data. Within the *on* phase of each pulse (ie every 180ms), the signal is further subdivided to allow the transmission of a bit code. Broadly, this code consists of a header followed by four bytes (one each for throttle, pitch, yaw, and trim). The *sendComand* function along with the helper *sendZero* and *sendOne* functions handle the timing of each component of transmission.

We ultimately modified this code to read in the state of 6 digital pins to control the helicopter. For each parameter, we have two pins assigned. One pin determines whether we increase or decrease a parameter, while the other pin determines the rate of change (x1 or x5). In order to sync with the Gumstix, the Arduino sends a signal to the Gumstix when the *timerISR* function is called. This lets the Gumstix know the Arduino is ready for new data. Additionally, we built in a "Kinect Mode" which can be turned on and off by the *#define* statement at the beginning of the code. In this mode we lock the yaw and pitch to zero so as to focus controlling the altitude.

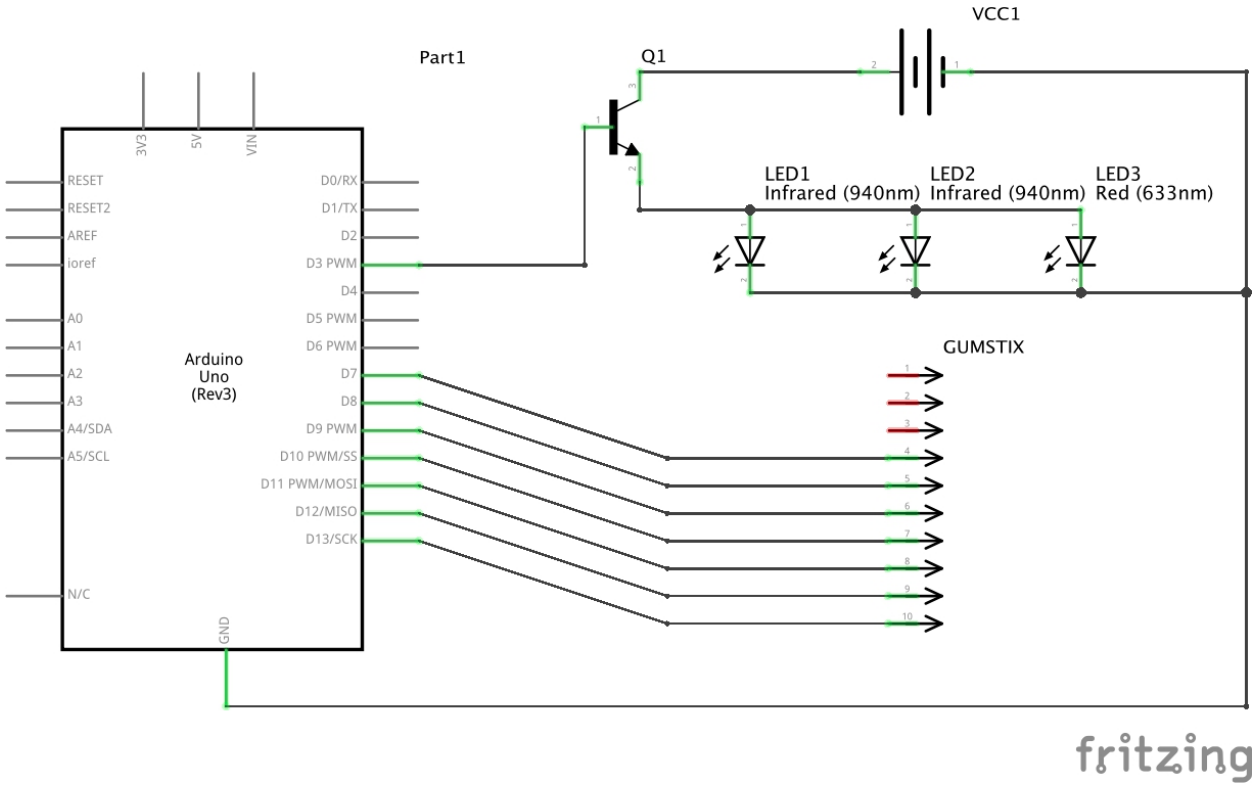

Figure 4. Wiring diagram for connecting the Arduino Uno with the Gumstix and the LEDs.

c.ii Circuit

The circuit above in figure 4, shows the connections between the Gumstix, Arduino and infrared LEDs. Our initial challenge was to drive LEDs with enough power so that we could get a range beyond a couple feet. This involved finding infrared LEDs with a wider degree beam width [10] as well as adding more power via batteries and controlling the power input via an NPN transistor. These additions allowed the Arduino to modulate more power than it would otherwise be available on any given pin.

Summary

We were able to achieve our main goals, albeit with some modifications. We initially set out to have the Gumstix drive an infrared LED to control the RC helicopter all the while receiving input from either a LCD touchscreen or a Kinect. Despite our best efforts, this exact pathway did not happen. Meeting the hard deadlines specified by the communication protocol proved a greater challenge than we anticipated. Instead, we added an extra component to our system, the Arduino. Its addition allowed us to free the Gumstix to better service other requests and better handle its role as a receiver of multiple inputs.

The input via the touchscreen LCD worked very well with kernel module's write function, but with the limitation of tracking only one touch point at a time. This limited the user to only changing one parameter at a time which, felt surprisingly constrained given the current ubiquity of multi-touch devices. More of a challenge, was the communication protocol between the Gumstix and Arduino. We attempt to synchronize the two in terms of that current parameter values for throttle, pitch and yaw via an interrupt from the Arduino to the Gumstix. Unfortunately, even after moving to a request-recieve system, we still encountered some drift between what the Arduino's values actually were and what the Gumstix believed them to be. This was not majorly detrimental to flight controls, but would be one of the first things we would invest more time into understanding and improving.

Our control system via the Kinect and user level program could be rewritten to be more sophisticated in terms of the tracking and decision making. We were able to implement a basic control scheme as a proof of concept, but a better working implementation would require a better filtering system for the Kinect data and a smarter controller for determining the proper action the helicopter should take for flight stabilization. Ultimately we would work towards stabilization in three dimensions.

Overall, we were satisfied with our results, not only with our final product but about the project implementation process in general. We were able to divide up the work in a manner that allowed each of us to work in parallel before ultimately bringing all of the pieces together. This was possible due to the team communicating often on their progress, explaining how their components worked and having a centralized repository for the codebase so each member could view what other members were working on.

References

[1] http://www.amazon.com/b?node=8037720011
[2] https://www.faa.gov/uas/
[3] Syma S107G 3.5 Channel RC Helicopter
[4] http://openkinect.org/wiki/Getting_Started#Linux
[5] http://www.kerrywong.com/2012/08/27/reverse-engineering-the-syma-s107g-ir-protocol/
[6] Borenstein, G. (2012). *Making things see: 3D vision with Kinect, Processing, Arduino, and MakerBot*. Sebastopol: O'Reilly Media.
[7] http://code.google.com/p/simple-openni/
[8] https://processing.org
[9] http://qt-project.org/doc/qt-4.8/widgets-sliders.html
[10] https://www.adafruit.com/products/388