Real Time Control Framework Using Android

Aaron Pittenger
Marquette University

REAL TIME CONTROL FRAMEWORK USING ANDROID

by

Aaron Pittenger

A Thesis submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, WI

December 2012

ABSTRACT
REAL TIME CONTROL FRAMEWORK USING ANDROID

Aaron Pittenger

Marquette University, 2012

One potential application for a smartphone-type device is a flight management and control computer for an unmanned aerial vehicle (UAV). The hardware employed in most smartphones and tablets has the capabilities necessary to fly an air vehicle without user interaction. The user can pre-program in a flight plan and the smartphone will do the rest. In the past, this real time control application has been done using many separate sensor packages and processors, but never on a single, stand-alone device. Also, capabilities such as the high definition camera present on most smartphones can take photographs and store them on the phone for retrieval later. This opens many potential markets for a device of this nature. Farmers that have large properties could use this to see if their fences are broken. The general public could use the application to take aerial views of their properties. Law enforcement could be an application for this project; to map out house fires or other potentially harmful situations before lives are put at stake.

The real challenge with using a smartphone as a flight management and control computer is the real time control of the aircraft. In order to accomplish real time control, the computer must have the sensors necessary for real-time control, a fast processor, capable of running a periodic process at frequencies greater than 10Hz (the faster the better) and the ability to read the sensor input and act on it during the time slice given for that process. With a multi-threaded, embedded, real-time operating system, this typically is not a problem (given a fast enough processor and enough inputs for all the sensor data). Doing the same type of calculations and control on a consumer product made to run many applications at the same time is difficult. This thesis will demonstrate how a real time control process was implemented on an Android phone.

ACKNOWLEDGEMENTS

Aaron Pittenger

I would first like to thank my wonderful wife, Shari, and all my family for all their love and support during this project. Working on something like this takes valuable time away from what is most important, family. I would also like to thank my committee members for their technical expertise along the way; committee chair, Dr. Tom Kaczmarek, and committee members, Dr. Sheikh Iqbal Ahamed and Dr. Richard Garside. I would also like to thank the many other informal committee members for their expertise in their specific areas, namely Aaron Buehner and Grant Hazard for their help with real-time control and aerodynamics. I would also like to thank those that allowed this project to come to life away from the simulator, namely fellow Marquette Graduate School student David VanKampen for his work on the Simulator Centered Design project as well as Greg Mikowski for his help with the RC aircraft. This project could not have been completed without all of your help. I sincerely thank you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 The Problem

The solution to the on-board autonomy problem should be a low-cost, robust, easily accessible and easily configurable solution for the general public. This allows applications in both civil and military markets. It should be fully autonomous (navigation, guidance, and control) and not exceed the size, weight and power requirements of the vehicle selected.

With these requirements in mind, and with the realization that the smartphone market is increasing rapidly, a smartphone application presents itself as a viable solution to the problem. It is readily accessible (most people already have one). Most new smartphones contain all of the necessary sensors for flight navigation, guidance, and control. It is a low cost solution. Also, the cell phone has a number of other features that would be useful in a number of applications. Some examples of this are recording video, streaming video back to a server via 3G/4G or receiving commands from a ground station for mid-flight flight plan changes. One problem with using a cell phone for this purpose is the phone's inability to control flight surfaces through servos. This can be overcome by making the phone communicate with a separate servo-controller board and send that board the servo commands via a wireless protocol, for example, Wifi, Bluetooth or Wifi-Direct. The separate board can be “dumb” and only control the servos based off the commands of the phone.

1.2 Project Structure

A smartphone based autopilot system for a UAV consists of 3 main components; the smartphone, which contains software that performs flight navigation, guidance, and control, an aircraft to be turned into a UAV and a servo controller, used to control the flight surfaces of the aircraft, as directed by the smartphone. This is displayed pictorially in Figure 1.1.

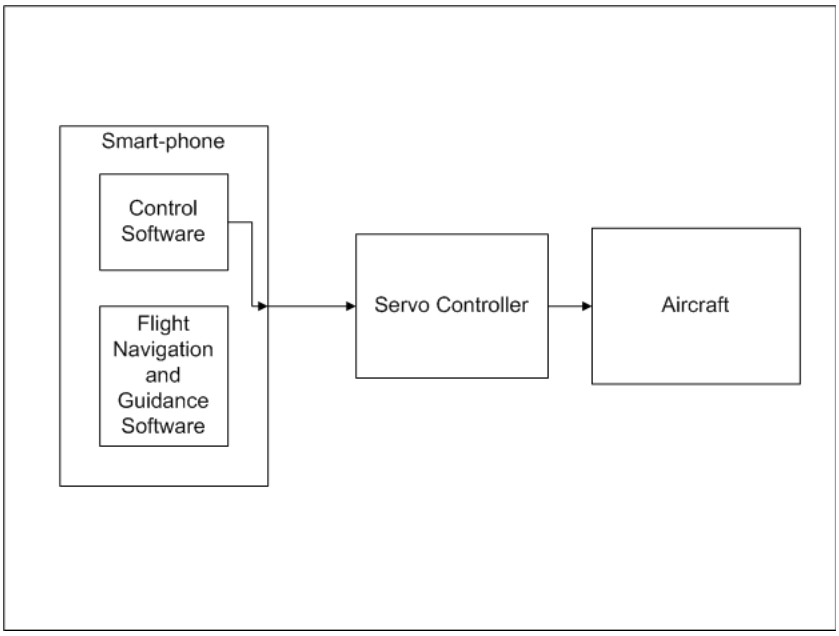

Figure 1.1: Project Architecture

From a software perspective, the development of the entire autopilot suite (including navigation, guidance, and control) is a large task usually completed after years of hard work with large teams. In order to modularize this application, the control framework will be the task of this project and thesis. The benefit of creating a control

framework is that other control applications could re-use the framework easily.

In order to reduce development risk and to keep project costs to a minimum, a simulation of the actual aircraft will be used in order to test the control framework. This will allow development of the servo controller and actual aircraft to be completed externally. A colleague, David VanKampen, is developing the servo controller as well as the interface to the aircraft using a paradigm called “Simulation Centered Design” where using a well designed simulation interface as the real-world interface allows for easy project integration after testing is completed. After both projects are completed, the goal is to integrate the projects together and test the application running in the real world.

Real time software control requires very fast, periodic processing of sensor input in order to dynamically control the aircraft. With a multi-threaded, embedded, real-time operating system, this typically is not a problem (given a fast enough processor and enough inputs for all the sensor data). Doing the same type of calculations and control on a consumer product made to run many applications at the same time is the purpose of this thesis. An Android (created by Google) phone was selected to be used because of the ease of application development and popularity. Android is an open-source platform used by over sixty-five percent (and growing, as of Q2 2012) of the world population [1].

1.3 Criterion for Success

In order to properly evaluate the performance of the framework, criterion for success were established. One of the most critical elements for control loops is the timing of the control loop period. At a 25ms period (40hz frequency) a tolerance of ±2% would keep the period between 24.5ms and 25.5ms. The control framework designed should

also be portable and easily configurable for other control theory applications (outside of aviation). This is the design idea behind a framework. In order to test out the control framework, an aircraft control algorithm will be used. The control algorithm used is tested and proven on other applications and is therefore assumed to be correct. In order to prove that the proper control frequency is established, the aircraft in the simulation must stay aloft and respond to control changes as expected (for example, when directed to change altitude, the aircraft changes altitude accordingly).

2 Current State

Before discussing how an Android device could be used as a flight management and control computer, let's look at the background behind such a computer.

2.1 Background

Typically, a conventional autopilot system is broken up into subsystems. The three main subsystems to control where an airplane is going are navigation (where is the airplane and where is the airplane headed), guidance (using navigation as input, how does the airplane get where it wants to go) and control (what does the airplane need to do in order to accomplish guidance).

In order to better understand the difference between navigation, guidance, and control, take the example of a ground vehicle navigation system (such as a Garmin, TomTom or Google Navigation). If the navigation system were to be running without a destination specified, it would be giving a navigation solution (the current location). After entering in a destination, the navigation system displays the path to get from you current location to the destination. This is considered the guidance aspect of the system. Then, the human driving the vehicle is the control aspect of the system. The user does not necessarily have to follow the guidance solution in order to remain in control, but following the guidance solution will get the user to their destination. Also, the guidance solution cannot perform without knowing the navigation solution.

All three of these systems are closely related and usually distributed around an aircraft. Typically, one company may make a navigation and guidance computer, and

another will make the control computer. This adds another level of complexity to the problem because the data that must be shared between these computers must be done using a highly reliable data bus and is usually accompanied by multiple redundant computers or channels.

2.2 Moving into the UAV Market

As the name implies, an unmanned aerial vehicle (UAV) is an air vehicle that operates without a human on board. The UAV is controlled either by a pilot/operator at a remote ground control station via a communication link, or autonomously through on-board computers. The trend of the UAV market is nothing but upwards. Both the military and civil markets want to move more towards UAVs. The main benefit of a UAV is that no lives need to be put at stake to perform functions that can be automated. This benefits the military because pilots' and flight crews' lives do not need to be put at risk while performing missions in hostile areas. This would also benefit the civil market by limiting pilot's and flight crew's exposure to risk and hazardous situations. Also, using an autonomous UAV would allow people to fly who do not have a license to perform tasks they would normally need to hire a pilot to complete.

As an example, farmers that have large areas of land typically have fences around that area (especially if they have something they want to keep in). King Ranch in South Texas is approximately 825,000 acres, about the same size as the state of Rhode Island. In order to keep their profits high, they should check the fences daily to ensure they are not broken and nothing has gotten out. They could drive around their entire property to ensure the fence is not broken, or they could hire someone who has a plane to fly around

the property and observe the fence. This task his highly repetitive and could easily be automated. If the farmer were to have access to a UAV, they could automate this process daily.

2.3 Challenges Associated with the UAV Market

The biggest challenge associated with the UAV market is typically referred to as size, weight and power (SWaP). UAVs can be much smaller than manned aircraft and therefore size, weight and power become significant factors in UAV design. A typical project management triangle has three points that must be in balance for a project to succeed. These are usually listed as cost, scope and time (or schedule). See Figure 2.1. If one edge of the triangle increases, so do the other two. You cannot increase the time or scope of a project without also increasing the cost.

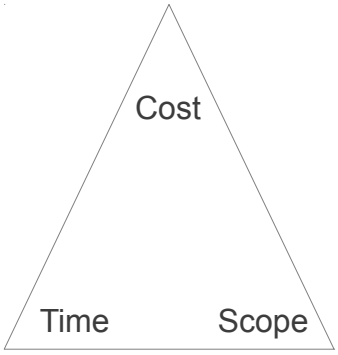

Figure 2.1: Project Management Triangle

The same rule applies for the SWaP problem in UAV design. See Figure 2.2. You cannot increase the size of the aircraft without increasing the weight of the aircraft. If you

increase the weight of the aircraft, you have to increase the amount of power needed to propel the aircraft through the air. If you need to increase propulsion, you have to increase the size of the engine.

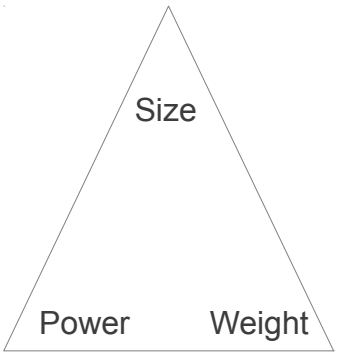

Figure 2.2: SWaP Triangle

Balancing the SwaP triangle is one of the main barriers to more UAV usage. Typically, a UAV doesn't have the size or power on board to take advantage of a conventional autopilot system. Consequently companies are trying to scale down their full size autopilot systems by splitting up the degree of autonomy.

2.4 Degree of Autonomy

UAVs can have different levels of autonomy. Typically, they fall into two categories; ground station control and autonomous control. A UAV that performs with ground station control typically has a host of communication equipment on board and all processing for flight navigation, guidance, and control is done on the ground. This allows for the large processing computers to be kept on the ground and all the weight and power restrictions are fulfilled by the communication equipment on board. Commands are sent

over the communication link to turn and move the plane through the air. This requires a ground based operator or ground based autopilot to be available all times. Currently, most UAV systems employ this strategy.

The other level of autonomy is fully autonomous control. This includes having a flight navigation, guidance, and control computer on board and pre-programming a flight plan into the computer. This option is challenging to deploy because of the SWaP problem coupled with the complexity involved with these systems. Common sensors utilized on and aircraft include Global Positioning Systems (GPS) (for navigation), Inertial Navigation System (typically an accelerometer and gyroscope, for increased navigation performance), Pitot/Static System (for air pressure, altitude and airspeed indicators), thermometer (for air temperature), gyroscope (for attitude indication), compass (for navigation), and motor controllers (for control of flight surfaces). As mentioned earlier, most, if not all, of these capabilities can now be found in a hand held device such as a smartphone.

3 Related Works

Currently, there are several projects focused at solving many of the sub-problems of flight navigation, guidance, and control. Most of them are incomplete, or overly complicated.

3.1 Parrot AR.Drone 2.0

Parrot AR.Drone 2.0 is “a groundbreaking device combining the best of many worlds, including modeling, video gaming and augmented reality.” [20] The idea behind the Parrot AR.Drone 2.0 is a video camera attached to a self-stabilizing quad-copter, controlled by a cell phone's orientation. It then transmits video back to the cell phone so the user can see a drones-eye view. The operator then uses the phone as a joystick to direct the Parrot AR.Drone 2.0. Tilting the phone to the right makes the drone fly to the right. Tilting to the left makes the drone fly left.

This is similar to the proposed solution in the sense that there is a phone communicating with an aircraft, but the practical applications of this are strictly limited to line of sight and aimed predominately at recreational/entertainment purposes. The phone and drone need to be within sight of each other in order to be able to communicate. There is no notion of pre-programming a flight plan and having the drone fly a flight plan. Also, all the entire flight is dynamically controlled by an operator, which is exactly what the proposed solution removes.

3.2 Ardupilot Mega

The Ardupilot Mega is probably the closest in similarity to the proposed solution. It is a “pro-quality IMU autopilot based on the Arduino Mega platform, which can turn any RC vehicle into a fully autonomous Unmanned Aerial (or Ground) Vehicle.” [8] This product is new and completely based on open source hardware and software. It uses the popular Arduino Mega platform as the brains of the flight navigation, guidance, and control for the airplane. At a cost of $200, the package includes a processor board interfaced with the following on-board sensors: 3-axis gyroscope and accelerometer, a barometer, magnetometer, and GPS unit. The systems juggles these sensors between three separate processors and can store up to 4MB of data on board. The only item the user would have to purchase is a battery pack and any other additional sensors they would like to integrate.

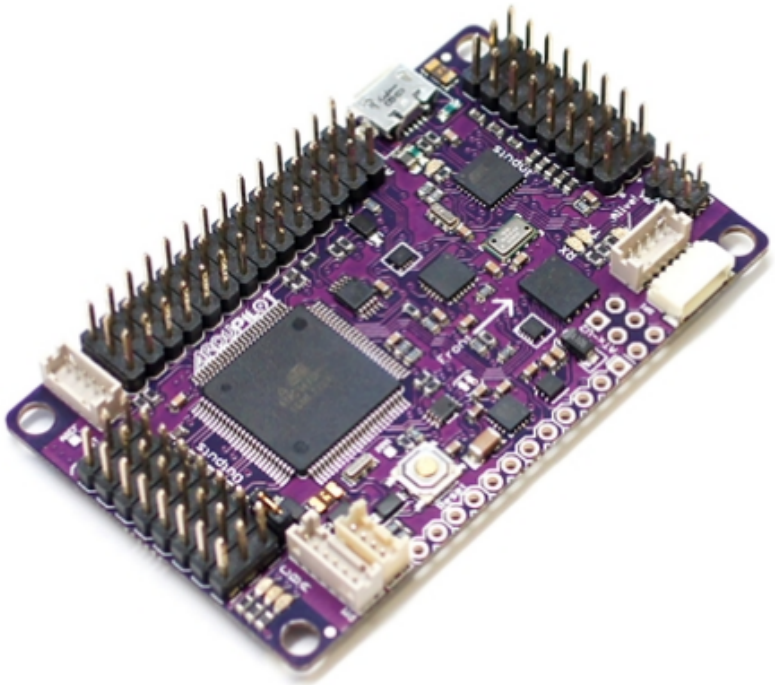

Figure 3.1: Ardupilot Mega 2.0

After the user receives this package, they download the predefined code for their specific application (fixed wing aircraft, helicopter, or land rover) and load the board with the executable application. Then, in order to do mission planning, the user downloads another PC application. After planning the mission, the user then loads the mission onto the APM and the vehicle is ready to be deployed. APM also only offers solutions to a few specific vehicles. This is because different vehicles will perform differently based of the dynamics of the aircraft and the motors involved. In many ways, this is a similar approach to the proposed solution, but lacks the ease of use and fully integrated solution the proposed solution offers. Also, the control aspects of flight are done with a dedicated processor on board and sensors that are made for flight applications, not for commercial use. The proposed solution would remove the necessity for separate processing for the

control of the aircraft and do it in conjunction with all other processing the phone is currently doing. Table 3.1 shows a comparison between APM and the proposed solution.

Table 3.1: Capabilities of APM vs. Proposed Solution

Capability	**APM**	**Proposed Solution**
GPS	On-board	On-board
3-axis Accelerometer	On-board	On-board
3-axis Gyroscope	On-board	On-board
3-axis Magnetometer	On-board	On-board
Pressure Sensor (for altitude)	On-board	On-board
Battery	Extra	On-board
Camera	Optional Extra	On-board
Power	Run off existing RC Airplane power source	On-board
Ground Communication	Through Radio (Line-of-sight only)	Through Cellular Network (world capable)
Flight Software	Extra (download + install)	Extra (install from App Store)
Mission Planning Software	Extra (download + install)	Included with Flight Software

4 Solution Prototype

4.1 High Level Design

In order to prove that real time control using an Android phone is feasible, the aircraft simulation program X-Plane will be used to simulate actual flight. “X-Plane 10 Global is the world’s most comprehensive and powerful flight simulator for personal computers, and it offers the most realistic flight model available.” [17] Using an actual aircraft for testing purposes is cost prohibitive for this project. X-Plane is considered a high accuracy flight model capable of providing flight characteristics suitable for this project. It should be noted that using a flight simulator means that instead of using the sensors provided by the smartphone, the simulated sensors X-Plane provides must be used. In order to keep the simulation as realistic as possible, sensors that are not available on a regular smartphone will not be used.

X-Plane can be manipulated by external programs through its UDP interface. A UDP interface allows X-Plane to communicate with another application by sending datagram packets back and forth over ethernet. This is slightly less reliable than TCP/IP because there is no confirmation of message reception, but X-Plane is sending out data periodically so dropping a single message does not create a significant impact to the data integrity. UDP messages are specified to have a receiver and a port to send to. In order to communicate, both X-Plane and the application must know the IP addresses of each other. The UDP interface allows users to get sensor data and current aircraft variables, (such as elevator position, current airspeed and GPS position), as well as set variables

(such as current throttle position and elevator position).

Figure 4.1 shows a high level drawing of how the autopilot program will interface with the X-Plane simulation.

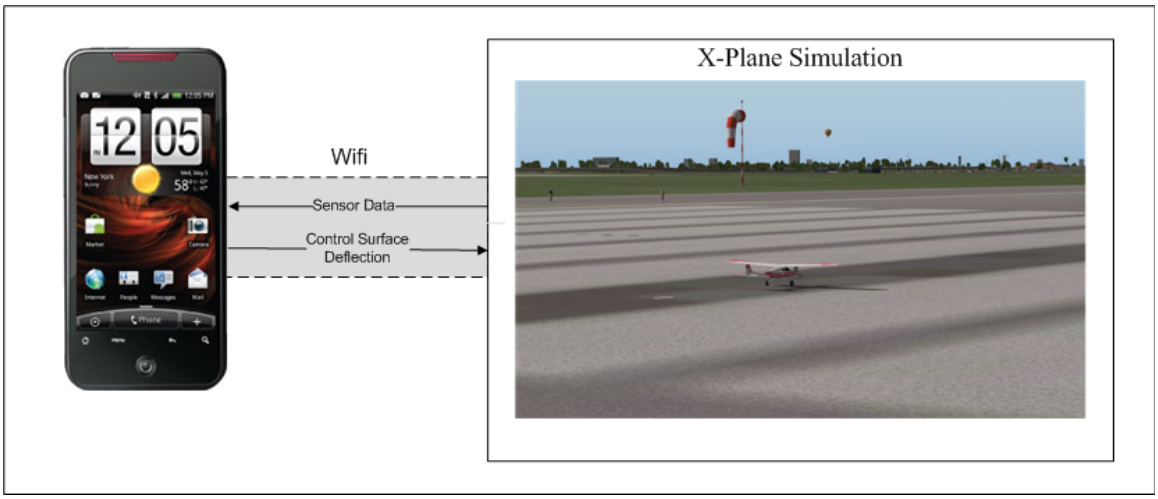

Figure 4.1: High Level Architecture

4.2 Development Environment

The application will be developed on an Ubuntu 12.04 LTS machine with Linux Kernal v3.2.0-29. This environment was chosen because of its compatibility with both the flight simulation (X-Plane) and the Android Development Environment as well as personal familiarity. The Android application will be developed using the Android Developer Tools (ADT) plug-in (provided by Google at https://dl-ssl.google.com/android/eclipse/) for Eclipse 3.7.2 Integrated Development Environment

(IDE). Eclipse comes with the Ubuntu 12.04 LTS installation. The project will use version 20.0.3 of the Android Software Development Kit (SDK). The project will also use the Android Native Development Kit (NDK) version R8b. X-Plane version 10.10r3 will be used as the flight simulator. The test platform for the application will be a Samsung Galaxy Nexus, the specifications of which are show in Table 4.1. The project will not use any input from the simulation that the Samsung Galaxy Nexus does not support.

Table 4.1: Samsung Galaxy Nexus Specifications [16]

Sensor	**Manufacturer/Part Number**
Geomagnetic	Yamaha YAS530
Proximity	GP2A
Barometric Pressure	BOSCH BMP180
Accelerometer	BOSCH BMA250
Gyroscope	InvenSense MPU3050
GPS	SiRF SiRFstarIV GSD4t

5 Detailed Design

The following section details the process of designing the real-time control framework for Android.

5.1 Control Theory

The first step in designing a real-time control framework is to understand a feedback control mechanism. Figure 5.1 shows a generic implementation of a feedback controller.

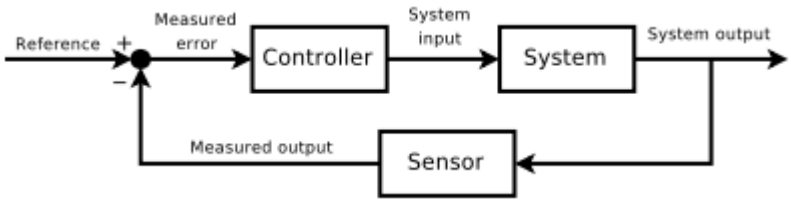

Figure 5.1: Feedback Controller

A feedback controller typically consists of the control output, the system being controlled, and the sensor input which feeds back into the controller. One complete pass through the feedback controller is considered a control loop. A control loop can easily be described through an example. Most common thermostats employ a feedback controller. The temperature of the house is sensed (commonly called the “process value”) and fed into the controller. The controller then calculates the error between the current temperature and the desired temperature of the house (commonly called the “set point”). This difference is referred to as the error of the system. The thermostat then controls the

system (the house) by turning on and off the heater (or air conditioner) for the house inorder to increase or decrease the temperature.

Typical problems seen in control loops include over-damping, under-damping and oscillation. Over-damping happens when the process value returns to the set point too slowly or never reaches the set point. Under-damping happens when the process value surpasses the set point before returning to the set point. Oscillation happens when the process value oscillates above and below the set point. These problems can be remedied by changing tunable gain values built into the controller.

In the thermostat example, the control loop can have a fairly long delay (seconds or even minutes) because the process value changes slowly. The faster the process value changes, the faster the control loop must run in order to keep the variable in control.

5.2 Flight Controller

In order to control flight, three separate process values must be controlled; airspeed, heading, and altitude. Airspeed is controlled by controlling the use of throttle. Increasing the throttle causes the propeller to turn faster, thus increasing the thrust on the aircraft (in propeller based planes). The user sets a speed at which they would like to fly and the computer automatically controls the throttle to keep a constant speed. This is similar to the cruise control in a car. As the vehicle goes up and down hills, the throttle must adjust to maintain a constant speed. Similarly, in an airplane, if a gust of wind blows, the throttle must adjust to maintain a constant airspeed. Also, just like a car, when the plane increases or decreases altitude, the airspeed is impacted and the throttle must be adjusted to maintain airspeed.

Heading can be controlled many ways. In an airplane, the heading of the airplane can be manipulated by both rudder and ailerons. Figure 5.2 shows an aircraft and the resulting motion when the rudder is deflected. Deflection of the rudder causes the airplane to rotate about its center of gravity (called yaw) and change heading.

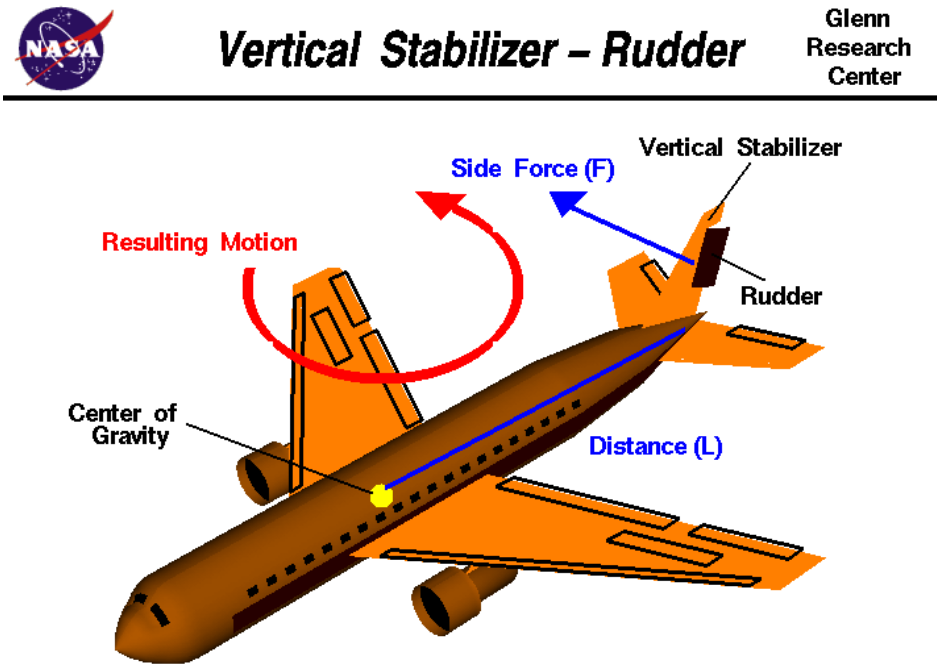

Figure 5.2: Aircraft Rudder Control [6]

Turing using ailerons causes a different motion of the aircraft but a similar resulting location displacement. Figure 5.3 shows the an aircraft with aileron deflection and the resulting motion. Deflection of the ailerons in opposite directions causes the plane to bank (called roll). Using ailerons to turn is typically referred to as a "banking turn".

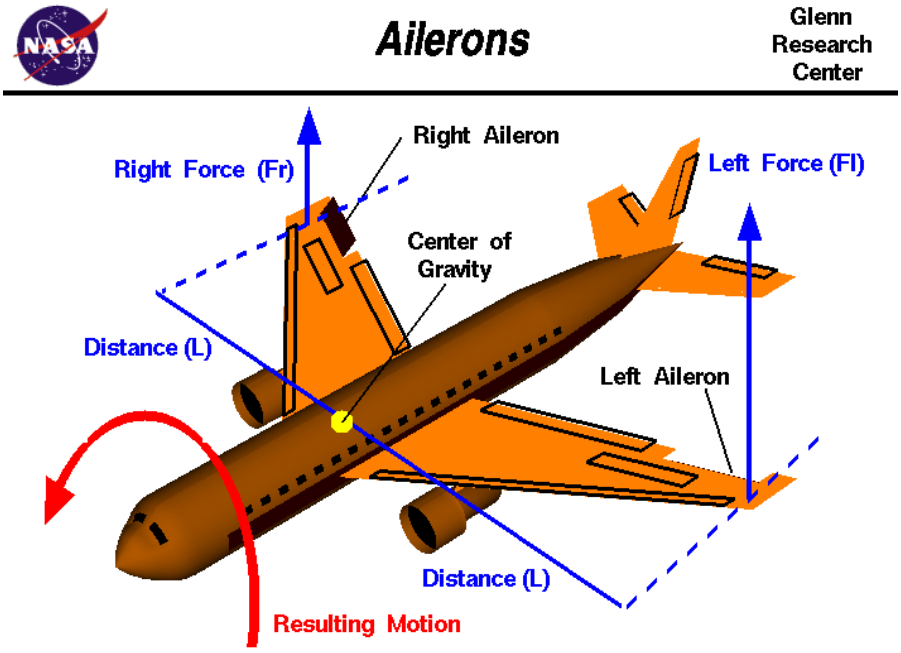

Figure 5.3: Aircraft Aileron Control [3]

As the airplane is banking, the lift force of the aircraft remains perpendicular to the aircraft. This creates a small side force moving the plane in that direction. This is displayed in Figure 5.4.Typically, a banked turn is preferred to a turn using rudder.

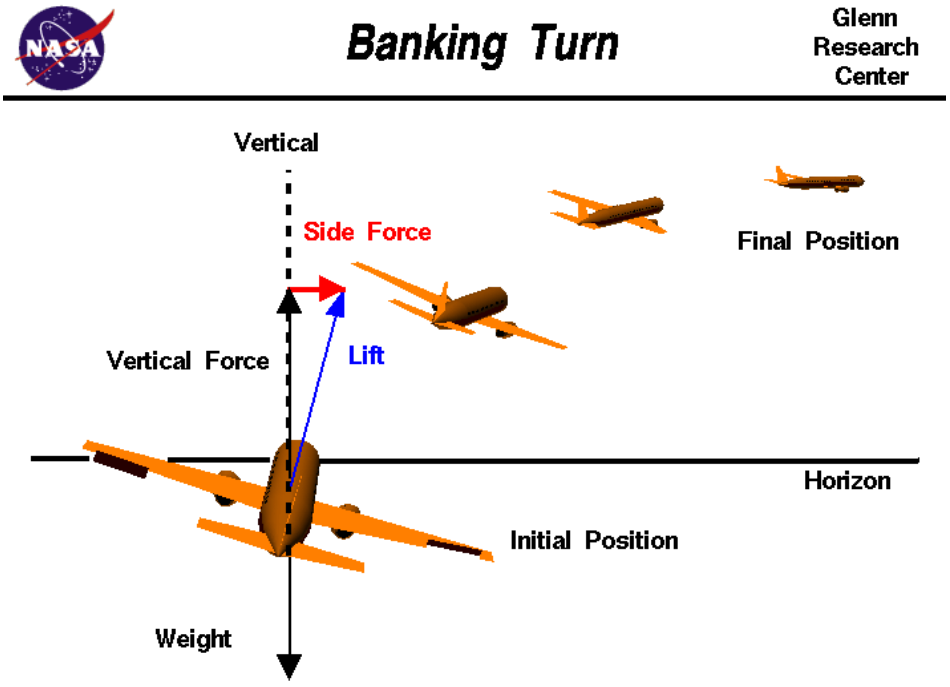

Figure 5.4: Banked Turn Dynamics [4]

The last of the process value that must be controlled is altitude. This is controlled by controlling the elevators on an aircraft. An upward deflection of the elevators creates a downward force on the tail of the aircraft, therefore increasing the pitch of the aircraft. An increase of the pitch of the aircraft, while maintaining a constant airspeed will increase the altitude. A decrease in pitch will result in a decrease in altitude.

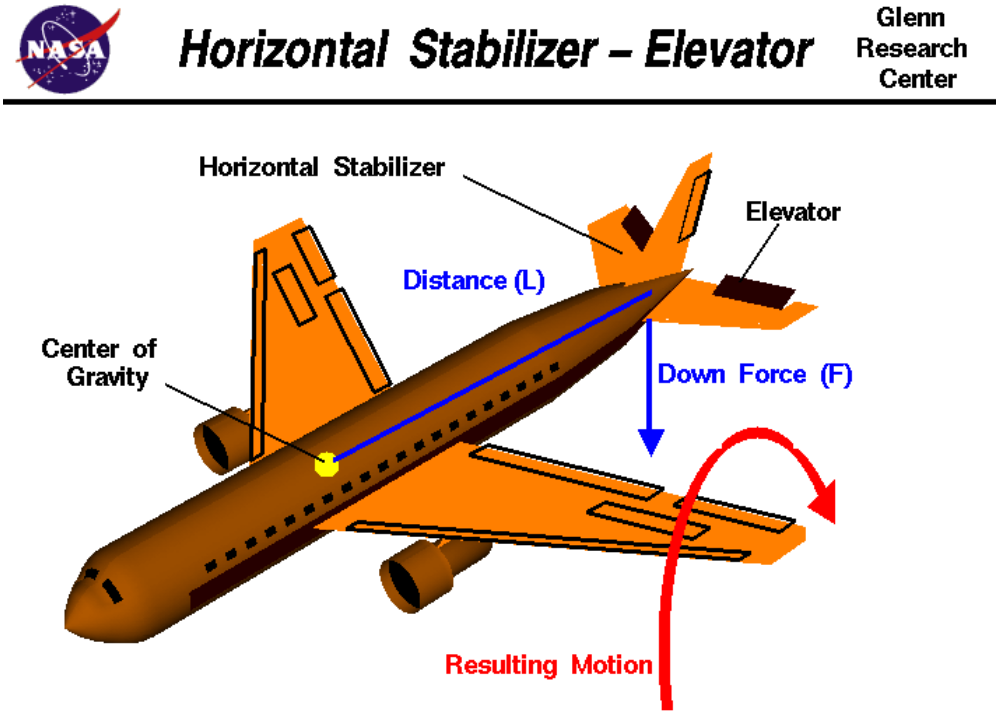

Figure 5.5: Aircraft Elevator Control [5]

Maintaining altitude can be difficult because changing any of these three process values may adversely effect the others. All values are coupled to one another. For example, increasing airspeed without compensating with elevator will also increase altitude. An increase in airspeed causes a greater lifting force on the wings and therefore increases in altitude. Conversely, rolling the aircraft without compensating with elevator will decrease the altitude of the aircraft. Banking the aircraft causes the lifting force on the aircraft to decrease, therefore causing the airplane to pitch down and lose altitude. There are many other relationships between all three of these control surfaces.

In order to keep all three of these flight variables in control, the loops must be run at an extremely fast frequency (compared to the thermostat example given earlier). The

process values can change quickly and the control loops must account for that by running at faster frequencies. The frequency selected for this application is 40Hz. The faster the frequency, the tighter the control.

5.3 High Level Software Architecture

From a high level, the software architecture closely mimics that of the generic feedback controller show in Figure 5.1. Inputs are read in from the sensors (in this case, simulated sensors distributed from X-Plane over UDP), the control loops are stepped, and the flight control surfaces are set to correspond to the commanded process values. This process is repeated at a 40Hz rate.

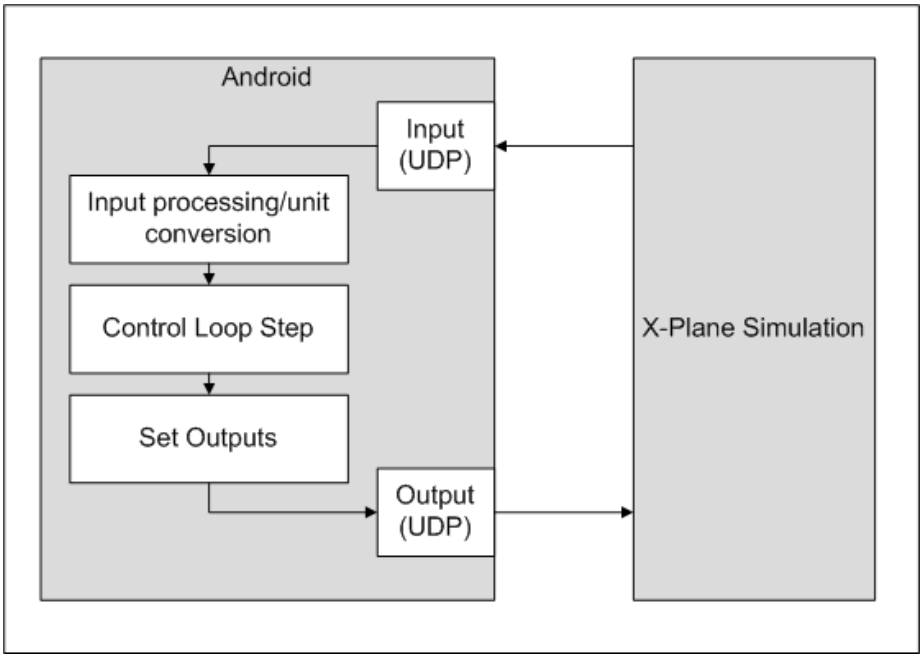

Figure 5.6: High Level Software Architecture

Each of the respective control loops are stepped based off the corresponding sensor input. For airspeed, the throttle output is set based off the current airspeed input. For altitude, the elevator deflection as well as throttle are set based off the current indicated altitude. For heading, the aileron deflection as well as throttle and elevator deflection are set based off current heading. As shown in Figure 5.1, every control step compares the current process value against the set point and performs control surface deflection based off the error.

5.4 Android Architecture

In order to fit these control loops into Android, the architecture of the Android OS must first be understood. Figure 5.7 shows the architecture as presented by Google [9]. It displays the Android Runtime Environment that typical Android applications run in. This includes the Dalvik Virtual Machine. This is similar to how Java applications run in the Java Virtual Machine (JVM).

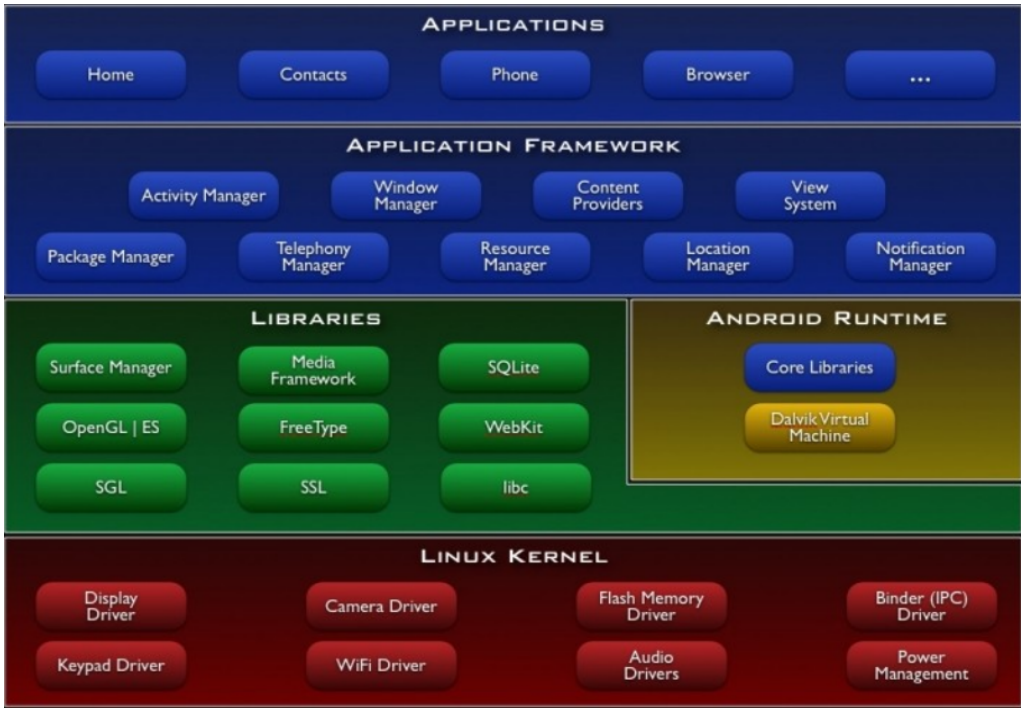

Figure 5.7: Android Architecture [9]

Underneath the Android runtime is the Linux Kernel. The Android Native Development Kit (NDK) allows the Linux Kernel applications to run and interact with applications running on top of the Dalvik Virtual Machine. It uses the Java Native Interface (JNI) to communicate between Java and lower level code. It does so by creating a shared static library that the Java application can make calls to. Also, the lower level code can make calls to Java functions [19]. Oracle (the makers of Java) list a few purposes for JNI including "You want to implement a small portion of time-critical code in a lower-level language such as assembly." [18] The Android documentation claims:

> *Typical good candidates for the NDK are self-contained, CPU-intensive operations that don't allocate much memory, such as signal processing, physics simulation, and so on....Before downloading the NDK, you should understand that*

> ***the NDK will not benefit most apps***. *As a developer, you need to balance its benefits against its drawbacks. Notably, using native code on Android generally does not result in a noticeable performance improvement, but it always increases your app complexity.[10]*

5.5 Low Level Software Architecture

In order to implement this control loop properly, a 40Hz periodic process must be scheduled. Android provides an API for scheduling a process to run at a later time. This is done though the `Handler` class. A `Handler` is described as have two main uses "(1) to schedule messages and runnables to be executed as some point in the future; and (2) to enqueue an action to be performed on a different thread than your own" [11]. In order to register a function to be run at a point in the future, the function `postDelayed()` must be called. The amount of wait time in milliseconds is passed in as a parameter to the `postDelayed()` function. Using the `postDelayed()` function "Causes the Runnable to be added to the message queue, to be run after the specified amount of time elapses. The runnable will be run on the thread to which this handler is attached" [11]. The description provided by the Google API shows that the `Handler` simply places the task on a queue and will run the task sometime "after the specified amount of time elapses" [11]. After experimenting with the tolerance provided by the `postDelayed()` function, it was found to be unacceptable for something that needs precise timing.

Another option for scheduling tasks to happen in Android is through the `Timer` API. `Timer` is a class from the core Java elements that are embedded in the Dalvik VM.

The documentation for the `Timer` API claims “Timers schedule one-shot or recurring tasks for execution” [14]. This is what is needed except shortly after that description, the API reads, “This class does not offer guarantees about the real-time nature of task scheduling” [14]. This is also not acceptable for the real-time scheduling that is needed for a control loop.

On a multi-threaded device that the programmer has no control over task priority, there will always be timing issues when precise timing is needed. Other tasks take over at inopportune times and processes are blocked that are time critical. The way Android provides timer-related classes in Java is unreliable. This is mainly because the Android OS runs on top of the Dalvik VM and the Dalvik VM is where Android gets its timing from. The NDK provides a way to create a precise and reliable timer since it has access to the timing mechanism provided by the Linux Kernel.

The Android NDK has access to the Linux Kernel. The Linux Kernel contains the standard C real time library which contains the POSIX Timers API [15]. The `timer_create()` function “creates a new per-process interval timer” [15] and is contained in the POSIX Timers API. This allows the process running (the Android Activity) to have an interval timer which will generate a signal at an interval the user can specify. When the timer expires, the signal `SIGEV_SIGNAL` is generated for the process. The user can register a listener for that signal as a parameter to the `timer_create()` function.

After creating the timer, the user can then start the timer with the function `timer_settime()`. This function is also included in the POSIX Timer API. Passed

into the `timer_settime()` function is the timer (as a `timer_t` type) and the interval at which to produce the signal (in nanoseconds).

Incorporating this functionality into an Android project is simple. Starting a new Android Activity will create a process in the Linux Kernel for the timer to run. After starting the process, the Android Activity will make a call using the JNI to a function that creates and starts the interval timer. This function also registers the signal handler function with the `timer_create()` function. After the process is complete, the function `timer_settime()` is called with a value of zero for the interval time and the timer is stopped.

Making this general framework application specific is just a matter of filling out the signal handler. In the scenario of a real-time controller for a UAV, the first step is to read the sensor inputs and store their values for use later. The controller implemented needs the following values: pitch and roll of the aircraft (from an accelerometer), heading of the aircraft (from a magnetometer), angular velocities of the aircraft (from a gyroscope), acceleration of the aircraft (from an accelerometer), indicated airspeed of the aircraft (from an airspeed sensor), altitude of the aircraft (from a barometric pressure sensor), and the attack angle of the aircraft (from an accelerometer). One thing to note is that GPS is not required to control an aircraft. This is only necessary for navigation and guidance. In this scenario, all the sensor inputs are received from the simulator and then acted upon by the control loop.

After receiving the sensor input, the control loop must be stepped in order to get the new output values for the control surfaces of the aircraft. For the sake of this project,

a software switch is implemented so that the control loops only run when the switch is turned on. This switch was activated through a switch in the simulator. When you set a switch in the simulator, it triggers the switch in the control software. As soon as that switch is activated, the control loop is stepped. Once that software switch is triggered back to the "off" state, the timer is stopped and the application exits. This architecture can be seen in Figure 5.8.

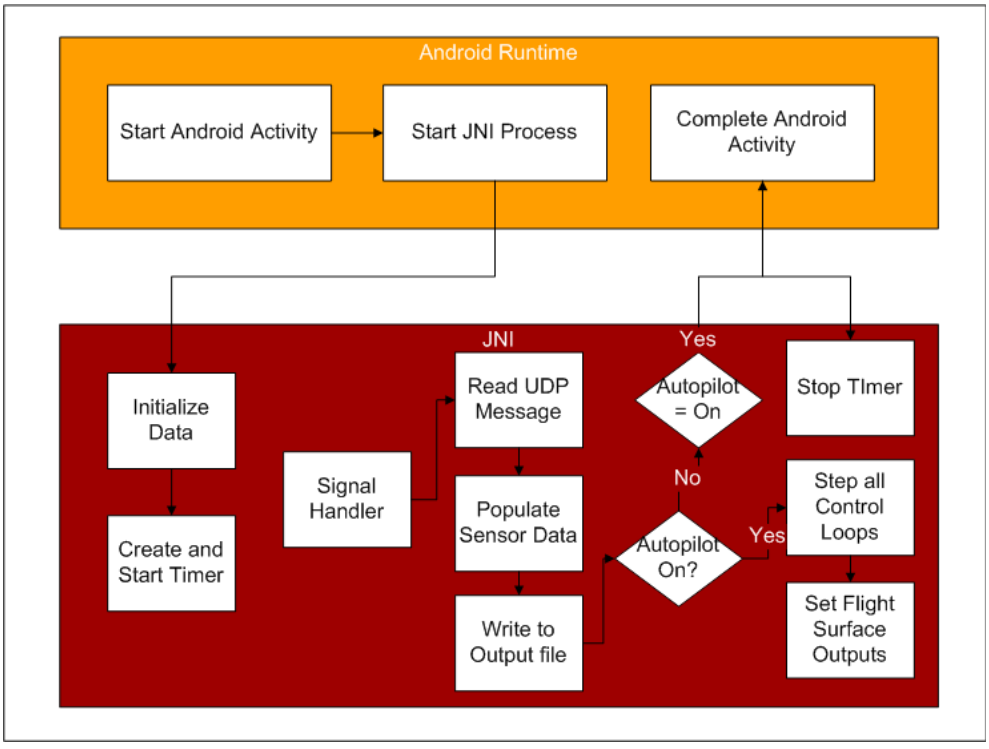

Figure 5.8: Low Level Software Architecture

The software switch allows the plane to take off under user control, fly under autopilot control, then land under user control. This is necessary for testing purposes because takeoff and landing cannot be automated without proper navigation and

guidance.

5.6 Moving to Real-world Flight

Taking this solution for simulated flight and moving it to a solution for a real-world application is relatively straight forward. The framework remains intact, but instead of reading the sensor values via UDP from the simulator, these values need to be read from the sensors on-board the cell phone. The control loops can still be stepped in the same manner, the inputs to the control loops are now just changed from simulated sensor inputs to actual sensor inputs.

The challenge lies with transforming sensor outputs into something useful. For example, the altitude of the phone (and therefore, aircraft) can be calculated from the barometric pressure the phone is currently at. The equation for this can be found in Figure 5.9 [22]. In this equation, y is the altitude; T is standard temperature; P is standard pressure; x is the current barometric pressure; K is a constant calculated from gravity, the Universal Gas Constant, the Molar mass of air, and the standard temperature lapse rate; and L is the standard temperature lapse rate. Once the altitude has been calculated, it can be used in the control loops.

$$y = -\frac{T\left(\frac{x}{p}\right)^{-1/K}\left(\left(\frac{x}{p}\right)^{\frac{1}{K}} - 1\right)}{L}$$

Figure 5.9: Barometric Equation

Some transformations that must take place are more complex. The pitch, roll, and magnetic heading of the aircraft must be calculated from the accelerometer outputs and the magnetic field sensor. The first item that must be attended to is the fact that the accelerometer outputs the acceleration along all three axis of the phone. These axes are labeled *x*, *y* and *z* and are oriented according to Figure 5.10.

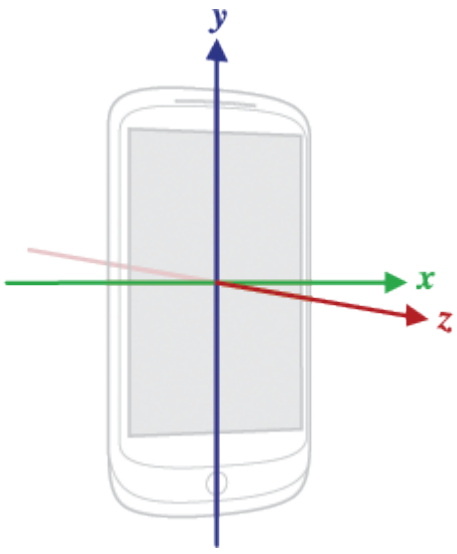

Figure 5.10: Device Axes Orientation

These acceleration readings include the force due to gravity as well as the force due to the true acceleration of the phone. Both the force due to gravity and the true acceleration of the phone are useful. The Android APIs state "It should be apparent that in order to measure the real acceleration of the device, the contribution of the force of gravity must be eliminated. This can be achieved by applying a high-pass filter. Conversely, a low-pass filter can be used to isolate the force of gravity." [12]. In order to

get the pitch and roll of the aircraft, the force of gravity is all that is needed. By knowing the direction of gravity, you can tell the pitch and roll of the phone (and therefore, aircraft). The gravity vector was isolated using the following low pass filter function [24]:

```
float lowPassFilter( float input, float output) {
    const float ALPHA = 0.15f;
    output += ALPHA * (input - output);
    return output;
}
```

The true acceleration of the phone is also necessary and therefore needs to be calculated with a high-pass filter to filter out the acceleration due to gravity. This high pass filter function was used [23]:

```
float highPassFilter( float input, float output) {
    const float ALPHA = 0.8f;
    float out;
    output = ALPHA * output + ((1 - ALPHA) * input);
    out = input - output;
    return out;
}
```

After isolating both the gravity vector and the acceleration vector, the pitch, roll, and magnetic heading of the aircraft can be calculated. In order to do so, the Android API has provided a few functions to do all the calculations for us. The function `getRotationMatrix()` and `getOrientation()` are used in conjunction to get the pitch, roll, and magnetic heading of the aircraft. The `getRotationMatrix()` function takes the gravity vector and magnetic field vector as inputs and returns the rotation matrix necessary to be passed into the `getOrientation()` function. After the rotation matrix is passed into the `getOrientation()` function, the pitch, roll, and

magnetic heading are returned in a three-value float array.

The last item to take into account while calculating the necessary inputs for the control loops is the coordinates of the aircraft. Typically, aircraft refer to the y-axis as the axis whose positive portion points out of the nose of the aircraft, they x-axis's positive portion points over the right wing of the aircraft and the z-axis's positive portion points directly towards the center of the earth. All three of these axis originate at the center of gravity of the aircraft. How the phone is mounted to the aircraft will determine if the coordinates need to be transformed. For this application, the phone will be mounted underneath the aircraft so that the screen is facing the bottom of the aircraft and the top of the phone will be pointed towards the nose of the aircraft. This means the only axis that will need to be transformed is the z-axis. Luckily, the Android API provides a way to transform these axis simply. There is a function called `remapCoordinateSystem()` which allows the user to take the matrices used to calculate the pitch and roll of the aircraft and remap them to a different coordinate system [13].

As stated previously, the sensors are read during the signal handler. Because this is done in the native C library, it is easiest (and quickest) to read the sensors from within that library instead of calling a Java function to read the sensors. The documentation to do this is not well known but can be done. First, it is necessary to include the files `android/sensor.h` and `android/looper.h` when accessing the sensors. The following code shows how to setup a set up a callback for the accelerometer. The same can be done for all the other sensors that need to be monitored.

```
void* accelData;
// Grab the looper. Only needs to be done once.
ALooper* looper = ALooper_forThread();
if(looper == NULL)
    looper = ALooper_prepare(ALOOPER_PREPARE_ALLOW_NON_CALLBACKS);

// Prepare to monitor sensors
sensorManager = ASensorManager_getInstance();

/** Accelerometer **/
accelerometerSensor = ASensorManager_getDefaultSensor(sensorManager,
                                ASENSOR_TYPE_ACCELEROMETER);
if(accelerometerSensor == NULL){
    LOGI("Accel sensor doesn't exist");
}
// Create an event queue for the accelerometer
accelSensorEventQueue = ASensorManager_createEventQueue(sensorManager,
                                looper, LOOPER_ID_USER, &accelerometer_callback,
                        accelData);
// Set the rate at which you would like to receive updates (in microsec)
ASensorEventQueue_setEventRate(accelSensorEventQueue,accelerometerSensor,
                                (1000L/40)*1000);
// enable the sensor (start it)
AsensorEventQueue_enableSensor(accelSensorEventQueue, accelerometerSensor);
```

After using the sensors to try and map all the data necessary to run the control loops, it was found that there were two pieces of data that were missing that could not be sensed using the available sensors on the phone. The first is the angle of attack of the aircraft and the second is the indicated airspeed (IAS) of the aircraft. Both of these pieces of data are important to autonomous control of the aircraft. Both play a part in knowing how much lift the plane is producing. The angle of attack is best described by Figure 5.11. This figure shows that the angle of attack is the pitch of the aircraft minus the flight path angle of the aircraft. These two angles will be different when the wind is blowing the aircraft so the flight path angle is less than the pitch of the aircraft.

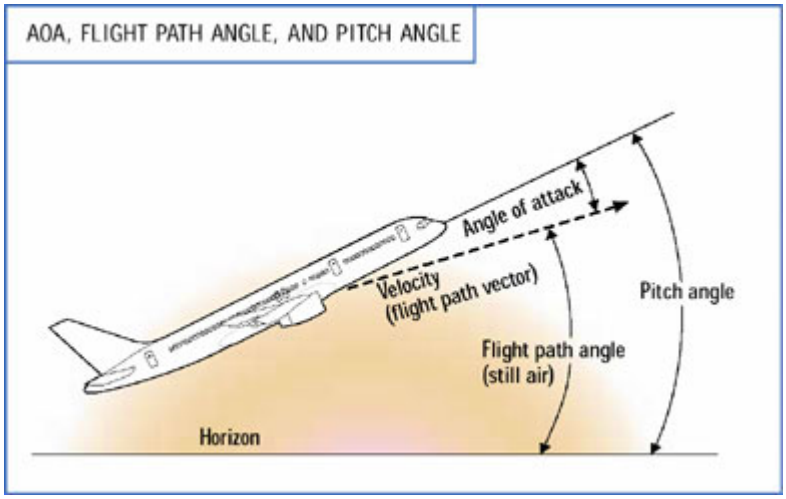

Figure 5.11: Angle Of Attack [7]

There are two ways to calculate the angle of attack without knowing the flight path angle of the aircraft. The first is to calculate the flight path angle of the aircraft using a previous location and altitude of the aircraft and the current location and altitude of the aircraft and calculating the angle change between the two. This is subject to many parameters. The location of the aircraft must be known and precise as well as the altitude of the aircraft. Typical GPS altitude will not be precise enough, as it is only accurate to about 20m. Barometric altitude, on the other hand, is precise enough as it is accurate to about 1m. The other approach to getting the angle of attack is to assume the wind is negligible and therefore, the pitch of the aircraft is equal to the angle of attack. The latter solution was selected for this application because of its simplicity in a proof of concept application. Also, it was selected because the maximum update rate of the GPS module in the cell phone is 1Hz. Although this may be good enough for navigation, it is not good enough to calculate the distance traveled at a 40hz rate. This distance could be calculated using an extrapolation but the error in this type of calculation would be too great to use

reliably.

The other data point missing is the indicated airspeed. Measuring speed of an aircraft is done using two main airspeed readings; ground speed and indicated airspeed. There are also other variants that are used such as true airspeed and calibrated airspeed, but ground speed and indicated airspeed are the most commonly used values. The ground speed of the aircraft is the speed of the aircraft along the ground. This is different than the indicated airspeed of the aircraft because the indicated airspeed measures the speed of the aircraft in relation to the body of air the aircraft is flying in. For example, if the aircraft is flying at 50kts ground speed, but has a 20kts tail wind, it has an airspeed of 70kts. Conversely, if the aircraft is flying at 50kts ground speed but has a 20kts head wind, it has an airspeed of 30kts. The indicated airspeed is more important than ground speed for control because the indicated airspeed plays a factor in calculating the lift of the aircraft. There are a few ways to get the indicated airspeed of the aircraft. The first is to use an airspeed sensor. Although this is not incorporated into the phone, it could be included on the aircraft and its information could be relayed to the phone through the same communication network as the control surface deflection values. Another way to get the indicated airspeed would be to somehow get instantaneous wind values and use vector math to calculate it based off ground speed. This is not feasible for this project so an airspeed sensor was purchased and used.

The airspeed sensor interfaces with the external servo controller board and the airspeed value is passed to the smartphone through the same interface used send servo control commands. All other required data can be calculated from the sensors available

on the phone.

6 Evaluation

In order to properly evaluate the performance of the framework, the following criterion were established to indicate success.

1. Loop timing should be very close to desired frequency (±2%).
2. Framework must be portable and easily useful in other control scenarios.
3. Testing using the simulated aircraft keeps the plane stable under nominal conditions. (Note: The control algorithms used are assumed to be tested and proven stable and therefore, any anomalies in aircraft control are assumed to be problems with the control framework.)

6.1 Performance

All performance calculations were completed using simulated sensor inputs and output to the simulator. In order to record the performance, the time was recorded at the beginning of the signal handler function. That time was then recorded onto a log file, saved to the SD card of the phone. After the test was over, the results were tabulated and graphed.

6.1.1 40Hz

The first test was completed using a 40Hz timer (25ms period). This is the frequency at which the project was designed to run. Table 6.1 Summarizes the data recorded every time the signal handler function was called.

Table 6.1: Statistical Results for 40Hz Frequency

Runtime	67.90s
Average	25.0001ms
Median	24.994ms
Min Time	16.785ms
Max Time	32.593ms
Range	15.808ms
Number of Missed Interrupts	0

This table shows that the total runtime of the application was 67.9 seconds. The signal handler function was called, on average, every 25.0001ms and it never missed an interrupt. One fact to note is that the minimum time between signal handler calls was 16.785ms and the maximum time between calls was 32.593ms. This range of 15.808ms may or may not be acceptable depending on the implementation of the control algorithms. This range may be explained by either a blocking process that does not allow the signal handler to run until it is complete and/or by inaccuracies in the timing mechanism.

Shown in Figure 6.1 is the statistical representation of the time between the signal handlers being called. This shows that 50% of the time between signal handlers being called is within about 24.75ms and 25.25ms. It also shows the average being right at 25.0ms. This makes sense because the timer created is considered an interval timer, calling a signal handler at specific intervals. This is opposed to a regular timer which times a specific amount of time after something occurs (like the signal handler exiting).

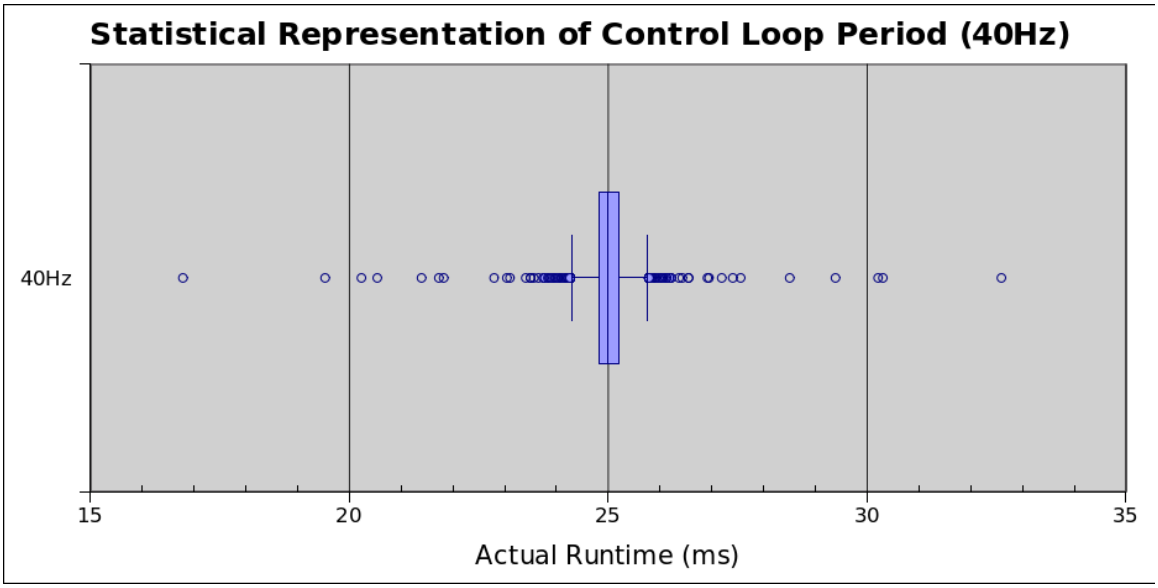

Figure 6.1: Box Plot of Control Loop Period (40Hz)

6.1.2 50Hz

In order to show the preciseness of this timer, faster frequencies were tested. At 50Hz (20ms period), Table 6.2 shows the results.

Table 6.2: Statistical Results for 50Hz Frequency

Runtime	68.84s
Average	20.0000ms
Median	19.989ms
Min Time	10.559ms
Max Time	28.565ms
Range	18.006ms
Number of Missed Interrupts	0

The total runtime of the test is shown as 68.64 seconds. Just as before, the average time between interrupts is almost exactly 20.00ms. Also, just as before, that time between interrupts ranges over 18ms. Also shown in those statistics is the fact that an interrupt was never missed.

Figure 6.2 Shows that statistically, there were a few major outliers and the rest of the times were closer together than when run at 40Hz. This plot shows that over 50% of the times between interrupts were between 19.75ms and 20.25ms. Almost all of the times between interrupts are between 17ms and 23ms.

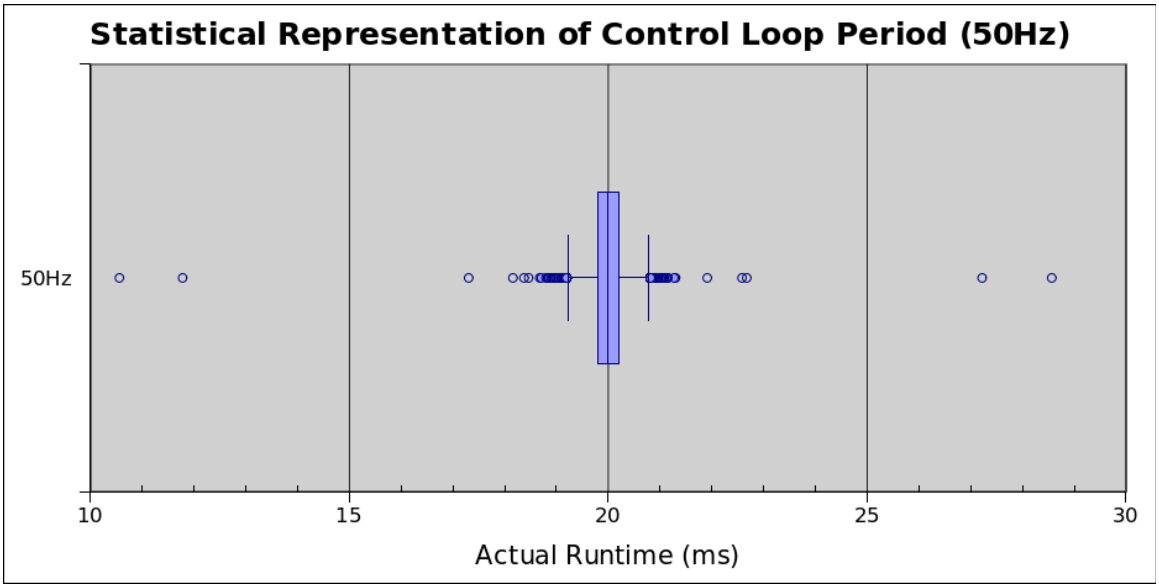

Figure 6.2: Box Plot of Control Loop Period (50Hz)

6.1.3 100Hz

Again, in order to see how far this timer could go, the frequency was increased to 100Hz (10ms period). The results can be seen in Table 6.3. They show that, just like the other two tests, the average runtime is almost exactly the desired period.

Table 6.3: Statistical Results for 100Hz Frequency

Runtime	55.60s
Average	10.0002ms
Median	10.009ms
Min Time	3.662ms
Max Time	17.121ms
Range	13.459ms
Number of Missed Interrrupts	0

This table also shows a range of over 13ms. This is the point where the increasing frequency must stop. Having a range greater than the average means an interrupt could potentially be missed. In this scenario, this was not the case, but the potential is there.

Figure 6.3 shows a similar statistical representation to the other. A wide spread array of outliers, but a tight grouping of values close to the desired period. Again, over 50% of all the time between interrupts is between 9.5ms and 10.5ms.

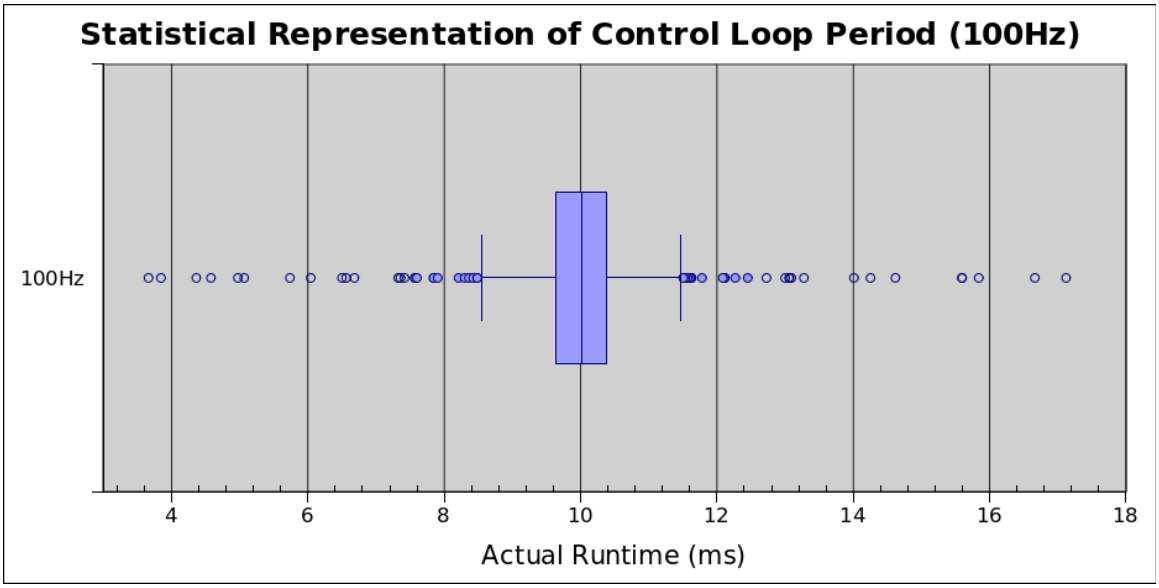

Figure 6.3: Box Plot of Control Loop Period (100Hz)

6.1.4 Dependencies

It is important to keep in mind that these results are dependent on the smartphone running the application. The smartphone used in this situation (Samsung Galaxy Nexus) is a high end phone (1.2GHz Dual Core Processor, 1GB RAM). Running the same application on an older smartphone will not reproduce the same results. Also important to note is that smartphones are only going to get faster and include more memory, therefore, the framework should run with higher accuracy in the future.

6.2 Portability

The package created is easily portable to other applications. As stated previously, it uses the POSIX Timers API (included in the Linux Kernel as well as other systems) so any system that has access to that API will be able to use this package. That means this

package does not necessarily need to be implemented on an Android device. It could be used on an embedded device that contains the Linux Kernel, like a Raspberry Pi [21] or BeagleBoard-xM [2].

6.3 Simulated Aircraft Testing

In order to properly test out the control loop, a timed flight pattern was created. Using a timed flight pattern instead of a location-based flight pattern removes the need for any navigation and guidance to be implemented. After manually taking off from the runway and getting into a stable flight condition, the control code was engaged. After engaging, it sets the desired airspeed to the current airspeed, the desired heading to the current heading, and the desired altitude to the current altitude. It holds this stable flight condition for 20 seconds, then turns 90 degrees to the right. It also increases altitude by 150 ft. After 20 seconds, it then turns another 90 degrees and returns to the initial altitude. It continues this cycle until the autopilot command is removed.

During flight, the aircraft seemed to have no trouble holding a heading, but struggled to hold a stable altitude. Figure 6.4 shows that when commanded, the aircraft turns to the desired heading and holds that heading well.

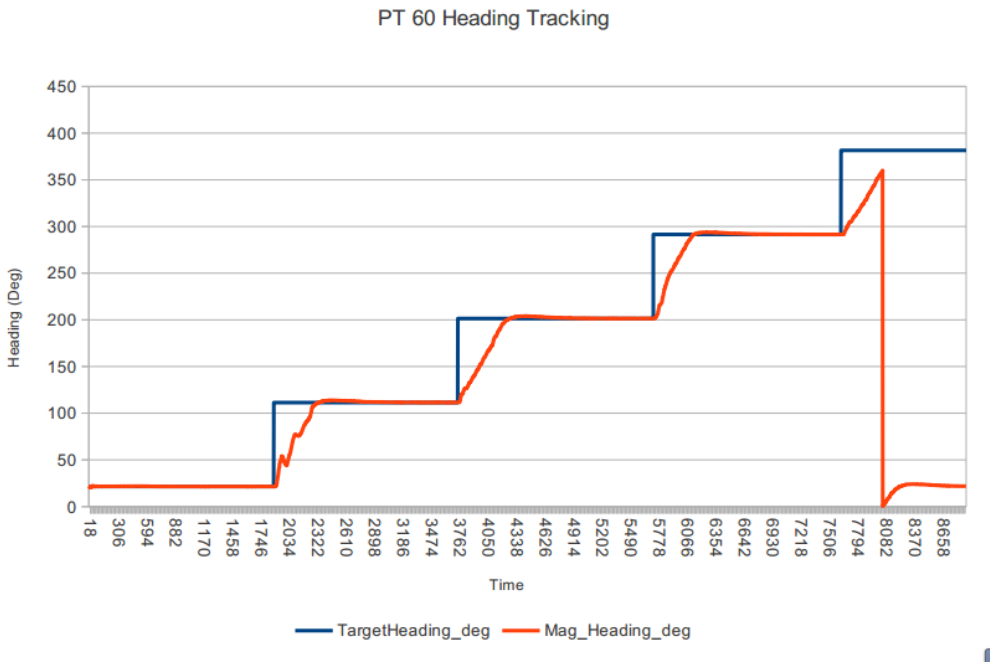

Figure 6.4: Heading Tracking (Desired vs. Actual)

Figure 6.5 shows the altitude tracking during the same flight. It seems to have a hard time holding a constant altitude. It also shows remarkable recovery after over-shooting the target altitude. Aircraft pitch was also included on this graph to show how often the pitch of the aircraft was changing in order to produce the altitude differences.

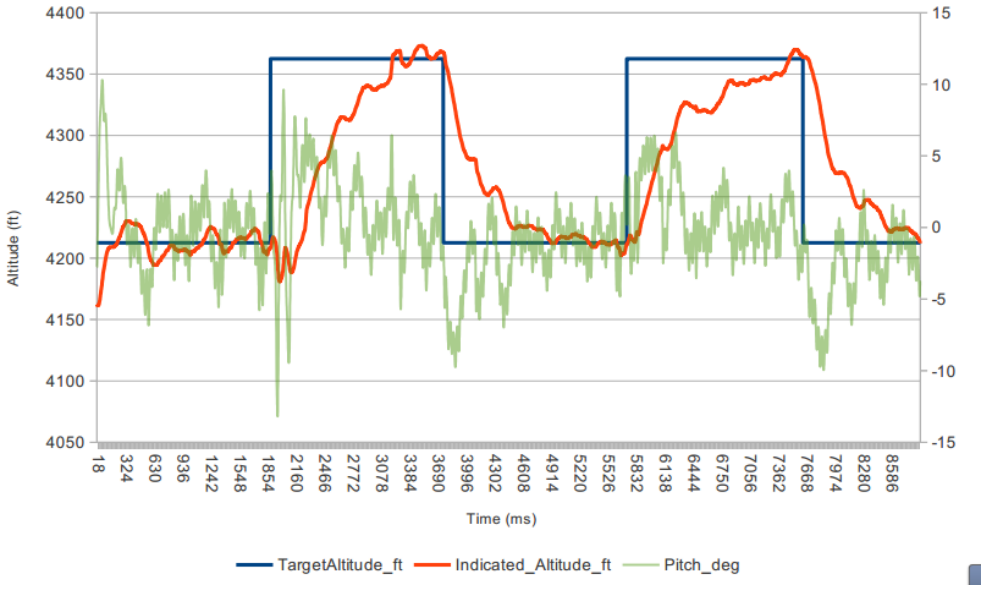

Figure 6.5: Altitude Tracking (Desired vs. Actual) and Aircraft Pitch

7 Future Work

As stated previously, this is one of the more challenging aspects of the on-board autonomy problem. In order to have a complete solution, a flight planning phase would need to be added to the application, as well as flight navigation and guidance. Post processing could also be added which would allow the user to view flight data after the flight is complete. Also, the current parameters implemented in the altitude and speed control loops present some oscillation. Tuning of these parameters would provide better performance of the aircraft.

7.1 Pre-flight: Flight Planning

The pre-flight phase is where the application will get the initial inputs from X-Plane via UDP. This is to establish an initial position using GPS, determine initial heading/orientation and verify that all sensors are working properly. As soon as those items are validated, the user will have the ability to enter a flight plan using an on-screen map. After completing the flight plan, the user can transition to the flight phase by selecting an on-screen button.

User interface mock-ups for the pre-flight phase can be seen in Figure 7.1 through Figure 7.3. The user should initially see a screen that only has a blue dot on it, representing the user's current location (in this case, the simulator's current location). The user could then single tap on the screen to place a waypoint. The waypoints added should automatically be connected in the order they are added. Also, the user's current location should be listed as the starting waypoint and the ending waypoint (depending on how

takeoff/landing is handled).

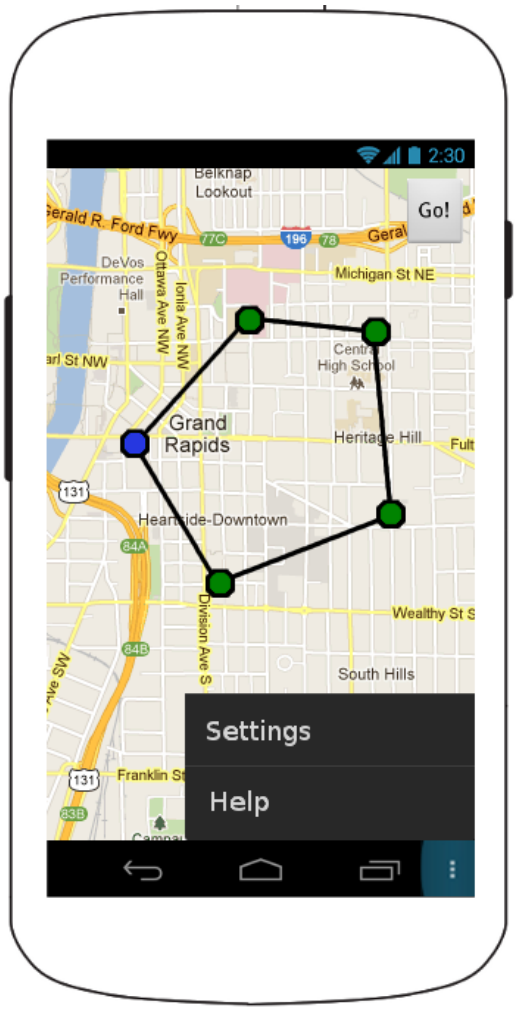

Figure 7.1: Main Screen

After adding waypoints to the flight plan, the user should be able to modify its properties (latitude, longitude and altitude).

Figure 7.2: Modify Screen

The user should also be able to delete waypoints in the flight plan. A confirmation dialog box should show, asking if the waypoint designated would like to be deleted.

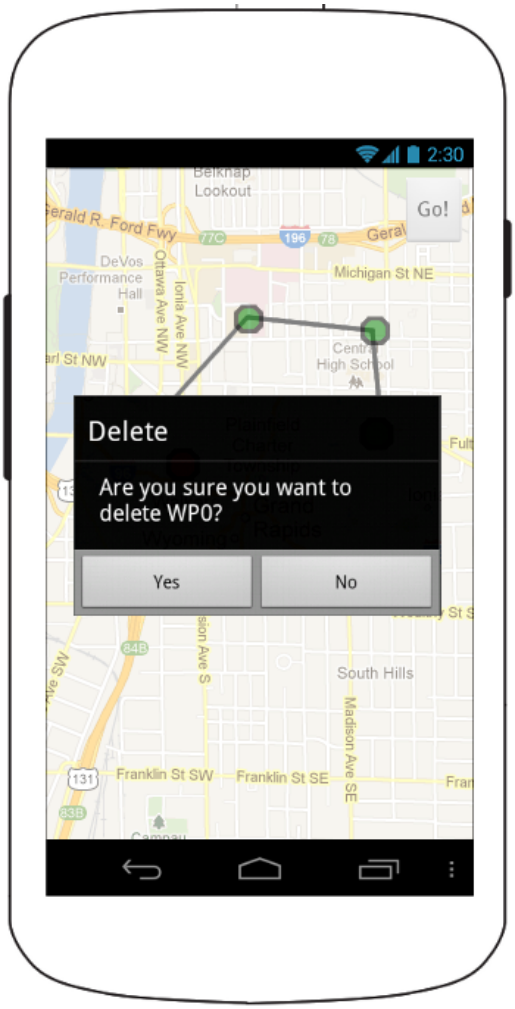

Figure 7.3: Delete Screen

After completing a flight plan, the user should be able to select an on-screen button (labeled “Go!” in the diagrams) which transitions the application into the flight state.

There are many other design decisions that would need to be made for the pre-flight phase. A takeoff and landing strategy would need to be decided on. This would most likely be hardware dependent. The device would need a sensor capable of reporting an extremely accurate altitude reading. In order to land, geographic terrain data would

need to be as precise as typical airport runway information. One potential alternative would be to land in the same location and direction as takeoff. This would create a scenario in which the application could record the flight characteristics of the takeoff phase and use that information for landing.

A design decision would also need to be made for obstacle avoidance. Since the plane would be fully autonomous, the plane would need the ability to avoid obstacles either through human intervention or through an extra on-board sensor. There also may be a terrain database on the internet that data could be pulled from in order to avoid obstacles.

7.2 Flight Navigation and Guidance

After detailing a flight plan, the flight navigation and guidance portion of the application would need to guide the plane in the direction of the waypoints. This includes accounting for generic guidance capabilities such as cross track error correction (when the plane gets blown off course) and dead reckoning (when the primary navigation solution fails). The navigation and guidance solutions could be thesis projects in themselves.

During flight, the current user location on the flight path could be shown, as well as showing a preview of a video recording. This may not be useful to display during flight because the phone will be carried on the aircraft, however, having a video could tell the user an abundance of information. A preview of the video record is required by Android OS in order to prevent people from writing applications that record video without user knowledge. Figure 7.4 shows a potential mock-up of the phone during flight.

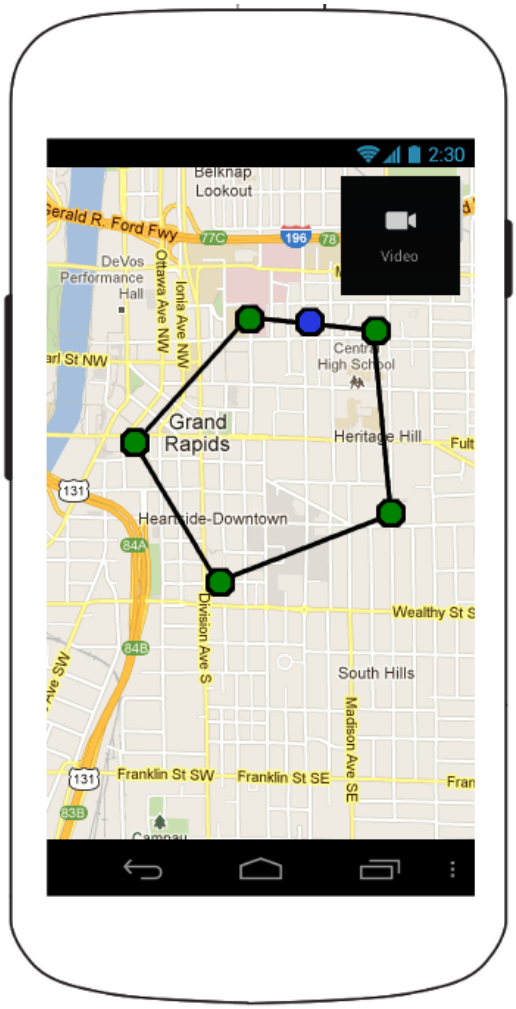

Figure 7.4: During Flight Screen

When originally testing this design, it was found that displaying the flight plan with a map overlay is somewhat infeasible. The map that is displayed is a large bitmap image which is saved in the phones heap space (in memory). The heap space available is limited and the image tends to take up much of that heap space. When other programs and processes need heap space, the garbage collector runs and cleans up the unused items

in the heap. Garbage collection runs frequently and takes too much time for a real-time control application, therefore it is infeasible.

7.3 Control Loop Tuning

As discussed in the results section, the altitude control loop is not tuned properly. Pinpoint control was not a goal of this project and therefore it could use some tuning. There is some oscillation that is currently occurring in the altitude control loop that can be removed through refinement of the control parameters. Currently, the altitude control loop has two stages, altitude capture and altitude hold. Altitude capture performs when the altitude error is less than 50ft. After the airplane is within 50ft of the desired altitude, it goes into an altitude hold mode. From Figure 6.5, the plane looks to have the most trouble during altitude hold. This is a good place to start tweaking the control parameters in order to fine tune the altitude hold.

7.4 Post Processing

Once the flight plan has been fulfilled, the user would want to access the flight data. In order to do this, a post processing screen should be shown. A mock-up has been created and is shown in Figure 7.5. It shows the two files that could be saved during flight (a Flight Data Recorder file and a video of the flight) and allows the user to share those files via whatever methods they see fit.

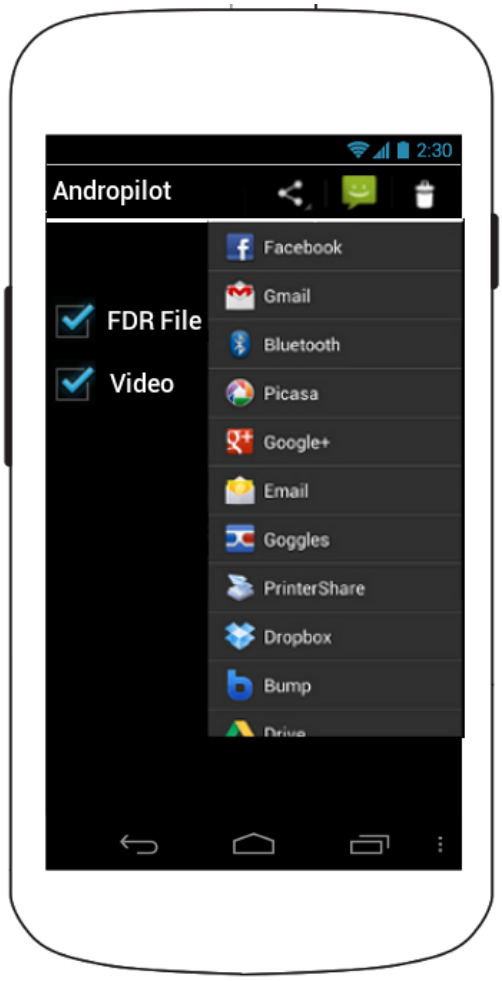

Figure 7.5: Post Processing Screen

7.5 Real-world Flight

A few things could be done in order to improve real-world flight. Currently, the only sensor needed that is not contained within the cell phone is the airspeed sensor. It would be beneficial to this project if there was a way to calculate the airspeed based off some of the other sensors. The airspeed sensor could even be placed on board to condition the plane and then removed after the plane "learned" how to calculate the

airspeed.

Another way of doing this would be to use real-time weather data. Calculating in the weather would change the given ground speed into an airspeed. This weather data would need to be almost instantaneous, possibly from a ground station nearby. This will get closer, but the best way would be to integrate an airspeed sensor directly into the cell phone.

7.6 Conclusion

The control framework developed is a viable solution for the Samsung Galaxy Nexus. The control frequencies necessary to control an aircraft are attainable using the hardware in the Samsung Galaxy Nexus. It seems the control frequency breaking point for this is around 100Hz. The POSIX Timers API allows the framework to be portable between different platforms. As smartphones increase in speed and capability, the framework will get more defined and more accurate.

BIBLIOGRAPHY

1. Associated Press. Worldwide market share for smartphones. 5 September 2012. Web. http://www.businessweek.com/ap/2012-09-05/worldwide-market-share-for-smartphones

2. BeagleBoard. BeagleBoard-xM Product Details. Web. http://beagleboard.org/hardware-xm

3. Benson, Tom. NASA Beginners Guide to Aeronautics. Ailerons. Web. http://www.grc.nasa.gov/WWW/k-12/airplane/alr.html

4. Benson, Tom. NASA Beginners Guide to Aeronautics. Banking Turn. Web. http://www.grc.nasa.gov/WWW/k-12/airplane/turns.html

5. Benson, Tom. NASA Beginners Guide to Aeronautics. Elevator. Web. http://www.grc.nasa.gov/WWW/k-12/airplane/elv.html

6. Benson, Tom. NASA Beginners Guide to Aeronautics. Rudder. Web. http://www.grc.nasa.gov/WWW/k-12/airplane/rud.html

7. Cashman, John E., Brian D. Kelly, Brian N. Nield. What is Angle of Attack? Boeing Aero. Web. http://www.boeing.com/commercial/aeromagazine/aero_12/attack_whatisaoa.html

8. DIY Drones, Ardupilot Mega, Web. http://www.diydrones.com/notes/ArduPilot

9. Google Inc. Android System Architecture. Web. http://developer.android.com/images/system-architecture.jpg

10. Google Inc. Android Developers Guide. Android NDK. Web. http://developer.android.com/tools/sdk/ndk/index.html

11. Google Inc. Android Developers Guide. Handler. Web. http://developer.android.com/reference/android/os/Handler.html

12. Google Inc. Android Developers Guide. SensorEvent. Web. http://developer.android.com/reference/android/hardware/SensorEvent.html

13. Google Inc. Android Developers Guide. SensorManager. Web. http://developer.android.com/reference/android/hardware/SensorManager.html

14. Google Inc. Android Developers Guide. Timer. Web.

http://developer.android.com/reference/java/util/Timer.html

15. Linux Man Pages. timer_create. Web. http://www.kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html

16. Molen, Brad. Behind the glass: A detailed tour inside the Samsung Galaxy Nexus. Engadget. http://www.engadget.com/2011/10/20/behind-the-glass-a-detailed-tour-inside-the-samsung-galaxy-nexu/

17. Morris, Austin, X-Plane Introduction, Web. http://www.x-plane.com/desktop/meet_x-plane/

18. Oracle. Java Native Interface Overview. Web. http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/intro.html#wp725

19. Oracle. Java Native Interface Specification. Web. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

20. Parrot, Parrot AR.Drone FAQ, Web. http://ardrone.parrot.com

21. Raspberry Pi. FAQs. Web. http://www.raspberrypi.org/faq

22. Amar, François G. "Barometric formula." University of Maine. Web. http://chemistry.umeche.maine.edu/~amar/spring2012/barometric.html

23. Wikipedia contributors. "High-pass filter." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 2 Oct. 2012. Web. 31 Oct. 2012.

24. Wikipedia contributors. "Low-pass filter." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 13 Oct. 2012. Web. 31 Oct. 2012.