Simulator Centered Design: Abstracting the Operating Environment on Radio Controlled Airplane Autopilot Development

David A. VanKampen
Marquette University

SIMULATOR CENTERED DESIGN: ABSTRACTING THE OPERATING ENVIRONMENT ON RADIO CONTROLLED AIRPLANE AUTOPILOT DEVELOPMENT

by

David A. VanKampen. B.S.

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, Computing

Milwaukee, Wisconsin

December 2012

ABSTRACT

SIMULATOR CENTERED DESIGN: ABSTRACTING THE OPERATING ENVIRONMENT ON RADIO CONTROLLED AIRPLANE AUTOPILOT DEVELOPMENT

David A. VanKampen. B.S.

Marquette University, 2012

Software development for applications used in high-risk and high-reliability environments is a difficult task. Testing in areas where failure could mean loss of equipment or damage to the environment becomes an implausible option. Testing in a simulated environment presents itself as a viable solution to this problem. One issue found when using simulators is they often have complex communication interfaces. After the simulator is no longer needed, the time spent developing the interface is lost. Simulator Centered Design addresses this by re-using simulator interfaces to make the real environment appear the same to the application as the simulator makes the environment appear. This thesis tests this theory by supplying an interface platform for a radio controlled airplane based around the X-Plane simulator interface.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the individuals that helped guide me through the process of writing this thesis and completing the exercise. I would like to express thanks to Dr. Thomas Kaczmarek, for chairing the thesis committee, providing weekly guidance, and being my main Marquette point of contact. I would also like to thank Dr. Richard Garside, my General Electric point of contact, and another main source of input and review on this document, as well as the corresponding GE document. Also, thanks to Dr. Sheikh Iqbal, Mike Francois, the other members of my thesis committee, for their help and input. Thanks to fellow MU graduate student Aaron Pittenger. Aaron developed the corresponding r/c airplane control project, upon which the ideas in this thesis were tested and vetted. And finally, heartfelt thanks to my fiancée Kristin Englund. Kristin's support throughout the weeks and months of working developing this document was priceless.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. Introduction

Today's smartphone platforms have much of the necessary hardware for this application, and the processing power to support it. However, there are a few key challenges associated with doing this.

1. Control of the UAV in flight by a smartphone requires definition of an architecture that enables real-time control. The real-time for a UAV is orders of magnitude faster than real-time for most of the sensors and controls built into a smartphone.
2. It is prudent to perform extensive testing of a control system before launching the UAV on its flight. One approach to supporting the development of the control system through a simulator to perform testing with hardware in the loop and using an approach that is called "Simulator Centered Design" (SCD).

This thesis will focus on the latter challenge – simulator centered design, and the key benefits and challenges associated with that methodology. This thesis is coupled with another thesis, which focuses more on the real-time control problem on a smartphone.

Below, Figure 1 shows the typical operating context for an application like this. Consider a specific example – an autonomous r/c airplane for crop-dusting a small field. In this application, the user could to draw out a flight plan on a satellite image of the field, with a map application. This will generate a navigable flight plan that is passed to the autopilot control program. This software uses the information created by the user to guide the plane over the crop to complete the mission. The controller guides the plane using interface software interacting with the operating environment, which consists of motors and sensors. Now, depending on the situation, this environment could be real or simulated. Either way, software needs to exist to interact with the environment, in order to get information from it and send control signals to it.

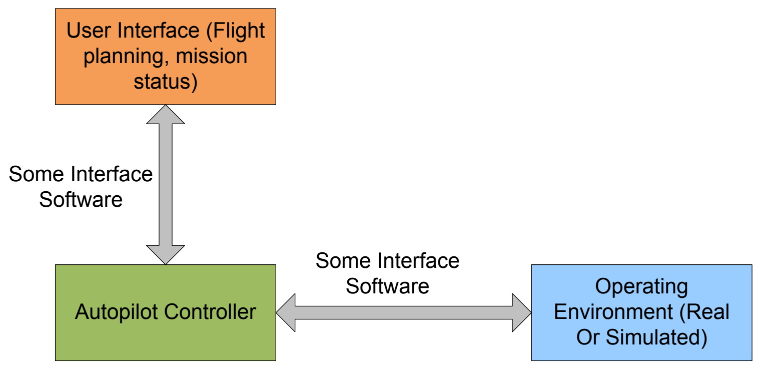
User Interface (Flight planning, mission status)
Some Interface Software
Autopilot Controller
Some Interface Software
Operating Environment (Real Or Simulated)

Figure 1: Autopilot Operational Context

Simulator centered design will focus on the horizontal arrow, abstracting the operating environment, and using the same interface for each operating environment.

Consider the r/c airplane hobbyist. Developing an autopilot to control the airplane can be an appealing idea. You are guaranteed to learn quite a bit about aerospace controls, you will get to have some fun working in software, and definitely get to experiment with electronics.

However, the result can appear daunting as well. It means handing over the reins, so to speak, for the airplane, from a human to a computer! You are giving up control from someone who is capable of making snap decisions, reacting to the environment, to a device, running on transistors, that only responds how it was programmed to respond, and does not display the same level of intelligence a human could.

Also, adding and removing hardware sensors and controls is a common occurrence on projects like this. As more information is needed by the autopilot software, a sensor that provides that information is added. For instance, airspeed is read from a differential pressure sensor. Every time hardware like this is added, more software is necessary to interface with it, parse the data from it, and respond accordingly in controller software.

In addition, the testing of the autopilot poses a nearly insurmountable difficulty. Even if the hobbyist was a skilled software developer, well versed on airplane control mechanics, the likeliness of “getting it right” the first time is extremely low. Testing is very important in any non-trivial software development task. Furthermore, testing must occur at several levels through the development process, from unit testing specific functions or code modules, up to integration testing on the r/c airplane in its environment. This means testing in flight.

One thing that happens during testing is failed tests. This is the main issue that stands in the way of hobbyists. Testing in flight is far too expensive and therefore, most users steer clear of creating an autopilot to turn their r/c plane into an unmanned aerial vehicle (UAV) – it is just too expensive. Beyond the concerns about expense, safety is a key issue as well. Putting a powered machine up in the air and giving it the ability to control itself is a definite risk, and the dangers must be understood and managed.

Flight simulators are an option for these obstacles. They are useful for many reasons. One well known benefit of flight simulators is pilot training. Airline and private pilots can become familiar with the look and feel of a cockpit, the response of the airplane to control inputs, and dealing with weather, all using a flight simulator. However, flight simulators have other benefits. A good flight simulator, with a highly accurate model of fluid dynamics, models of airplane, and weather, can be used to test and debug flight controllers, such as autopilot systems. If the flight simulator has an external interface, then other running programs or computers can communicate with the simulator.

Using this interface, testing out an autopilot controller in flight is much more realistic and possible for hobbyists with reduced budget and resources. They are no longer required to show the ultimate confidence in their design by strapping it to a plane and throwing it into the air essentially untested. Debugging can occur in a simulated environment. Issues can be discovered and resolved in the lab (whatever that lab may consist of), and expensive hardware does not need to be put at risk until after initial testing.

However, developing a software interface compliant with a flight simulator is no trivial task. In some cases the time spent developing an interface that works with the flight simulator could be comparable to the time spent developing the control software, dependent on complexity of controls. In addition, it requires additional expertise of the developer. If they are an r/c hobbyist, they likely know a good deal about aerodynamics, flight controls, and the like. However, it is possible they know very little about interfacing software programs over the internet, or other methods of interfacing. Even more, once they have developed their simulator interface, and completed testing, they then need to rework the interface from the simulator to work on their model airplane. This effectively doubles the amount of time spent in writing interface code, which increases difficulty and introduces a new source of potential error.

Because of all of this, the barriers to entering into this testing environment might be perceived as so great that it is simply not worth the time. In this case, the hobbyist may abandon the idea of developing their own UAV, because it is too expensive to test in the real world and too complex to test in a simulated environment.

This thesis aims to present a solution to this issue, and serve as an example for addressing the issue in the larger context of software-development. The solution is called "Simulator-centered design". SCD is about abstracting the operating environment from the controller software. In SCD, the developer produces a single interface, as if they were interacting with only one external environment - either the simulator, or the real environment. SCD vastly shrinks the development process, by essentially halving the time required for coding and testing interfaces. It gives that developer an opportunity to test their software in a safe environment. By building the “real” interface to look exactly like the simulator interface, the developer is prompted to test in the simulator, because there is no additional work required.

This approach works for the r/c hobbyist discussed earlier, and will also working the larger field of software engineering. One area SCD excels in is when there is no defined standard for an interface. By defining that standard, and re-using it for the simulator and real platform, an r/c hobbyist can develop their autopilot controller and immediately transition to real flight, without any rework required.

The rest of this thesis will focus on a few key items. First, it will look at the software engineering discipline as a whole, to see how simulators are used and incorporated into the development process today. One result of SCD is a reduced interface size. Because of this, this paper will address the influence of interface size and complexity on a project. Section III will provide more detail on implementing SCD on a software project, explaining the stages of an SCD project. Section III also addresses current industries that could benefit from SCD projects. Section V is the heart of the thesis, discussing the application of the SCD practice to a specific project – the r/c airplane autopilot project. For this project, an android smartphone application served as the autopilot controller, and an Arduino board with some hardware peripherals provided the external interfaces for the controller software. Section V will cover the implementation of the design described in V, describing the design decisions that were made throughout the design process. Finally, sections VI and VII will wrap things up, discussing the next steps for SCD, and concluding remarks.

II. CURRENT STATE OF THE ART

In software engineering a large amount of time, relative to the duration of the project, is spent on testing. Another large percentage of time is spent on interface development and interface compliance-checking. Software is developed in a modular fashion, allowing multiple developers to work on the same project. However, time and effort still is spent to ensure all the modules will fit together. This is done through interface control documents (ICDs), informal Wikis, work instructions, and many other means of interface documentation.

However, controlled and well-documented interfaces are a vast improvement over the alternative – a very large, spaghetti-like mass of code, intertwined upon itself in an unreadable and, more importantly, un-maintainable fashion. Software interfaces and APIs (Application Programmer Interfaces) allow developers to use functionality of other code modules without having to develop that functionality themselves. This isolates that function in one spot, providing an abstract interface, and allows the particular implementation to evolve, without change to the interface.

Though interfaces are a good step towards better, cheaper, and more maintainable code, they do not come without some work. A good deal of research has been done investigating the implications on the project from API complexity (Cataldo, de Souza, Bentolila, Miranda, & Nambiar, 2010). Typical measurements for API complexity are based on the number of functions, number of parameters for each function, and the complexity of each parameter (Booleans and integers being less complex than complex custom type definitions). Through the research of Cataldo et al., there is a direct relationship between the complexity of interfaces and the effort to maintain a system. The more complex the interfaces used in an application, the more difficult that application is to maintain and it becomes more prone to failure.

Thus, it is good practice to use only the interfaces you absolutely need, and nothing more. Equally important is the use of existing reliable interfaces before developing your own. In software engineering, there is always some "re-inventing the wheel" required, but it is preferable to use existing and tested functionality.

The remainder of this section will discuss these ideas and the current state of the art for each of them in the software engineering industry today. The implications of interface complexity on project scope will be discussed, followed by a breakdown of projects and industries that currently use simulators as an integral part of the development process (known as simulator based design). Finally simulator centered design will be defined and discussed.

I. RESEARCH

a. Interface Complexity

Interface complexity is an emerging field of research (Cataldo, de Souza, Bentolila, Miranda, & Nambiar, 2010). Cataldo et. al. discuss the implications on overall project quality when the external facing interface grows in size or complexity. They reference the correlation between API complexity and error prone software. As APIs become more complex, the cost to maintain and debug the software increases.

Cataldo and others break down the definition of interface complexity in a manner that can be applied to any software project. The developer chooses one base unit of complexity – this can be applied to a basic data type (such as integer or Boolean). As an interface grows in complexity, more and more of these basic data types are included, or used to make up more complex data structures. In addition to globally sharing basic data types, interfaces are also composed of method calls. These methods can receive the aforementioned data types as parameters, further increasing their complexity. Parameter-less methods are assigned a complexity level equivalent to a base data type. Thus, in the same manner, as the number of methods increases, so do the complexity measurement, and the implied maintenance and debugging costs.

In addition to the implications associated with larger interfaces, there are certain implications that additional complexity has on project development. For example, any increased interface means more code. More code implies the development time is increased, but also means that there are more code modules that need to be tested (Yang, He, Li, Wang, & Boehm, 2008).

Code size can be measured by Source lines of code (SLOC) count, but also can be measured by “function points.” Function points are a more general, and widely accepted as more reliable, method for measuring software size. Increased SLOC does not necessarily equate to increased functionality, as equivalent functions could be written in a largely variant number of lines of code. Function points are simply units of measure that provide the level of use or functionality a piece of software provides its user.

Whatever metric is used for measuring software project size, one thing is consistent. Increased size means increased complexity, debugging, and maintenance. As described by Yang et. al., different development methodologies tend to distribute the time across project phases (definition, design, implementation, testing) differently, but all require every phase. Testing typically takes between 15-30% of the project’s expended effort. Since it has been proven that interface complexity has a direct impact on debugging and maintenance of a software project, it has a direct impact on the size of this 15-30%, as well as which end of that percentage spectrum the project trends toward.

One of the impacts of simulator centered design is the inherent reduction in interface size. These studies on the impact of interface size on software projects serve to show that interface reduction is a key benefit. Simpler software interfaces require less debugging, maintenance, and testing. In addition, since the interface serves as the “external facing” portion of software, and the piece that most other developers are likely to interact with, the more compact and efficient it can be made, the more likely it is that it will be used and appreciated by other developers.

The next step for research, then, is looking into current systems and projects that use the interface of the simulator at some stage during development or testing. This practice is called Simulator Based Design. Simulation Based Design is closely related to Simulator Centered Design.

b. *Simulator Based Design*

Simulator based design (SBD) is the practice of using a simulator throughout the development lifecycle: requirements, design, code, and test. This has several advantages in many different fields.

There are many advantages to designing with SBD. SBD is useful for prototyping complex control algorithms on robotics platforms (Yoo, Ahmed, Kirchner, & Roemmermann, 2009). As described by Yoo et. al., with an acceptable simulator performance, algorithms can be tested and refined in the simulator, and then quickly transferred to the real platform. The simulator is used throughout the process, from ideation, through design, to implementation, and finally in test. The advantage here is it allows for the quick prototyping and testing that it fosters. The drawback is that the interface to the simulator is not necessarily representative of the interface that will be used in the real environment.

One aspect of SBD systems is that they are typically built for a specific application. This means that when a simulator will help in development, by shortening testing time, or decreasing testing cost, then the custom simulator is developed for the application. Or, it is at least re-used from a similar previously completed application. This typically is done in the automotive and aerospace fields, where testing in real environments is far too cost-prohibitive.

Industries currently using SBD are likely good candidates for the SCD process as well. The major players in the SBD fields are:

- Automotive
    - Crash Testing
    - Brake testing
    - Aerodynamic testing
- Aerospace
    - Zero gravity testing
    - Flight Guidance and Control System testing

These applications hold potential for SCD. One potential factor preventing them benefiting from SCD is the complexity of the control algorithms within the code. If it is significantly more work to maintain the control software than to maintain the interface software, then any benefit from reducing the interface portion will be reduced, comparatively.

c. Simulator Interface Centered Design

Simulator interface centered design is SBD taken to another level. It uses the simulator not only as the basis for testing throughout each stage of development, but also uses the interface defined by the simulator as the sole interface to the software. It re-uses that interface for real platform operations.

There is little work currently done in the field of simulator interface centered design. For more complex cases the simulator is designed in parallel to the application, or designed entirely based on the application.

One organization working in a field related to the SCD topic is the Simulator Interoperability Standards Organization (SISO) (SISO, 2012). SISO is an international group working to standardize the interfaces simulators themselves define. This is done more for distributed architecture systems. That is, it is done so simulators can talk to other simulators. This is useful in large-scale simulators, like war-game simulations, large scale/global weather pattern simulation, disease/viral contagion simulations or any other case where there are multiple players interacting over some sort of physical network. SISO focuses on systems scalability. Though they focus on the design of simulator interfaces, their focus is not on a single controller to simulator interface, but rather on many simulators being able to work with one another on a network to simulate larger distributed systems.

Similar to SISO is the High Level Architecture (HLA) (Moller, 2012). Another distributed simulator architecture model, HLA systems have been used for flight combat training in simulators – where multiple flight simulators need to work together, while pilots are controlling the multiple simulated air-vehicles.

The key attribute with the SISO based projects, and systems using HLA is interoperability. Though they are indeed focused on a carefully designed simulator interface, it is not about using that interface for the real product. It is about simulators interacting with other simulators.

Little to no work or research is available in the field of simulator centered design. For most projects, due to project scale, the simulators are developed for the application. The scale of the projects makes the interface size and simulator development task relatively small in comparison to the development of control algorithms.

The focus of SCD is not on simulator interoperability, but rather interfaces interoperability. That is, making sure the interface of the application operates in multiple environments, and makes that transition from environment to environment seamless. There seems to be no current work or research done in this topic. There are a few potential reasons why, but the rest of this thesis will document the validity of this idea by testing it in a real application, and describing its applications to the larger field of software engineering. Also, throughout, the potential pit-falls and issues associated with SCD will be discussed.

III. POTENTIAL FOR GROWTH – SIMULATOR CENTERED DESIGN

Based on the discussion in the above section, a few questions naturally arise:

What opportunities are there for interface re-use?

Will this work for my project?

Should I use the simulator interface, or work on creating my own?

These questions are guided by the above conclusions; if I want to increase external-facing functionality without increasing the required interfaces, how can I re-use those external-facing interfaces?

The answer to those questions, as introduced in section I, is Simulator Centered Design. SCD leverages the lessons learned from the interface complexity studies. It reduces the number of required interfaces to the controller program. Without SCD, if a developer wishes to use a simulator, they need to develop an interface for the simulator, and then an interface for the real platform. There would always be two separate paths out of the program, two different modules of software to maintain. With SCD, the developer need only maintain the single path out of their application, thus essentially halving the amount of interface software that must be maintained.

To achieve simulator centered design, there are two main approaches. Each approach funnels all input and output from an application down to a single path, regardless of environment (simulated or real).

The first approach is to re-use the existing interface the developer is working towards. This maintains the single path "out of" the application code. The second approach is to develop a custom interface for the external world, thus abstracting both the simulator and the real environment from the application code. In this case as well, there is still just a single path out of the application.

There are, of course, benefits and drawbacks to each approach, which will be outlined in the following sections. The first section, Alternate Approaches, will break down the basics of each approach. Section II will describe the benefits and drawbacks of each approach, so that data is summarized in a single place. Section III will review the main benefit of SCD – the reduction in interface complexity. Finally, section IV covers industries and software development practices that could benefit from SCD.

I. Alternate Approaches

Simulator Centered Design can be implemented in two ways. The first way is more obvious – build the interface around the existing simulators interface. For this approach, documentation, support, and wide-spread use already exists. The second approach is less obvious. Truly, this method is less simulator *centered* design than it is simulator *hiding* design. It defines a custom interface and handles the specifics of interfacing with the simulator(s) and the real platform itself behind the scenes.

a. Approach One – Existing Interface

The first approach is using an existing simulator interface as the real interface. This is shown below in Figure 2. Note that in these diagrams, ICD stands for "Interface Control Document." An ICD is the documentation associated with the API.

For approach one, the fundamental aspect of the design is the single path into and out of the application. This path is based on an existing and documented interface. If the application were developed *solely* for the simulator, the interface would look exactly the same. In fact, this is the case that simulator based design operates under (see section II.I.b). That would be the case if nothing on the “Real” side of Figure 1 were present. The format of data sets transferred across that interface was defined when the simulator was developed, and the application developers make their side compliant to that. The physical medium across which the data is transferred is also the same as it would be if it were only the application and the simulator. During application development, before the hardware platform and interface board are available, this is the operating environment.

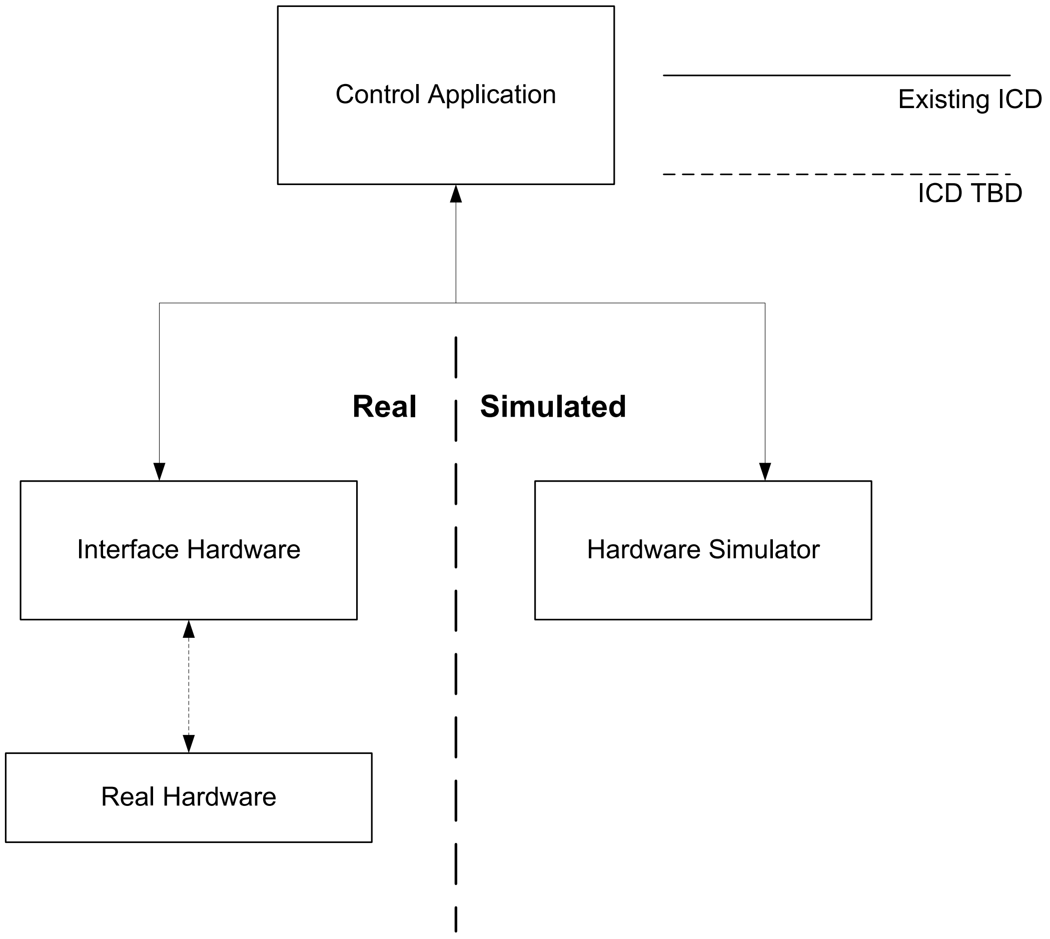

Figure 2: Existing Interface Driven Solution

Once hardware is available, or once hardware in the loop simulation testing is ready to be started, the left half of the diagram can be added. The key to this transition is that there is no requirement that an additional path be added into or out of the application. The same interface that is used for the simulator is used again for the real platform. In fact, the simulator and real platform can be run in parallel, both using the same commands coming out of the application.

Some consideration will need to be given to ensure there are no conflicts about which platform is providing the feedback information. That is why this needs to be constructed as a true hardware in the loop simulation (Bullock, Johnson, Wells, Kyte, & Li). There needs to be only one platform providing feedback signals – either the simulator or the real hardware. Different applications will have different needs, but the loop will need to be constructed to ensure there is only one controller. As development continues, however, different combinations of controls can be used to test the different threads of software and hardware execution.

b. Future State Two – Custom Interface

Transitioning from approach one to approach two is easiest if one compares the two key diagrams. Figure 3 shows the next possible method after the one shown in Figure 2.

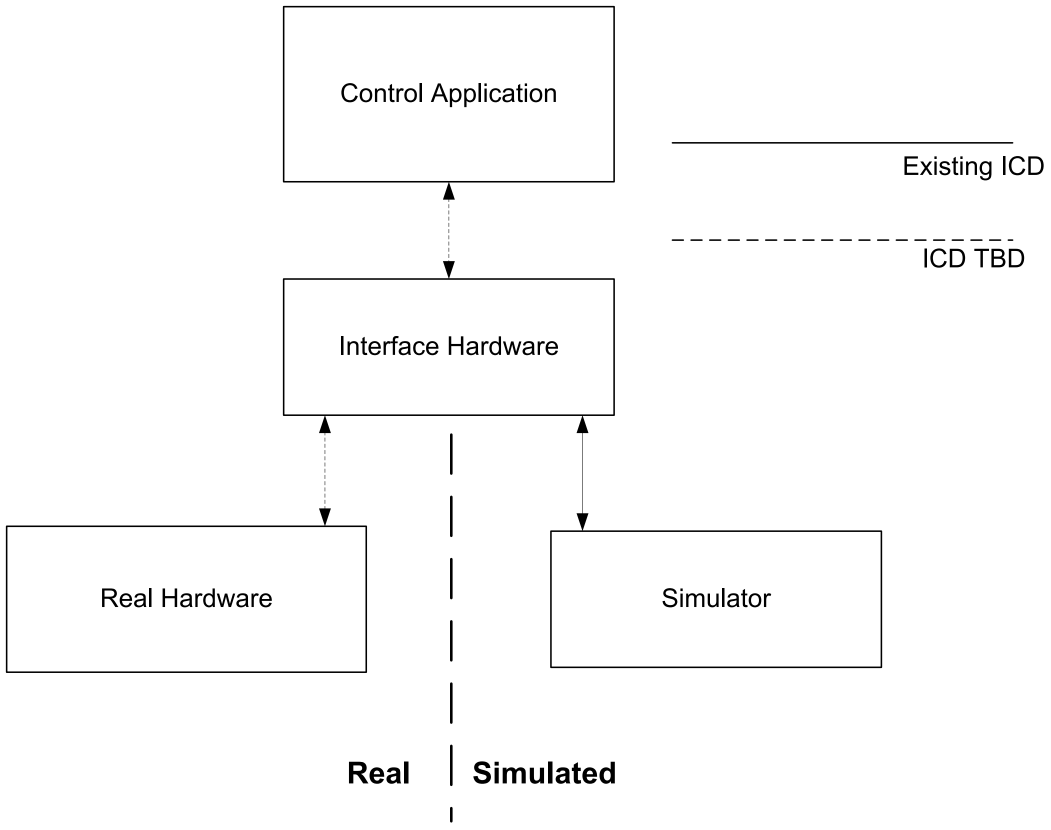

Figure 3: Custom Interface Driven Solution

The first thing to notice in this diagram is that the single path into and out of the application is maintained. As far as the application developers are concerned, only one interface needs to be maintained. The key difference here is that the line out of the application is no longer a solid line, representing an existing ICD, but a dashed line, representing an ICD and API yet to be developed. This interface is developed specifically for this application, or domain. The custom interface is offered by the interface hardware (with software running on it). It hides any specifics or nuances of the simulator from the application, handling those issues internally. Thus, the interface offered to the application is likely simpler and more compact than the simulator's interface.

One drawback of this method is that the interface hardware is required just to interact with the simulator. This means that the interface needs to be defined, tested, and deployed before application development can begin. If the interface hardware and software are supplied before the start of a project as a development platform, this becomes a non-issue. It only needs to be taken into account if the interface is developed in parallel to the application software.

II. Benefits & Drawbacks

One of the key benefits of using an existing interface is that applications have been tested and most of the bugs have been worked out. In addition, it is likely the interface will have all the features, parameters, and methods a typical application being simulated will need to have. In fact, it will likely have a superset of the functionality needed by any given application. This is a key benefit. This interface is well tested, documented, and most likely has all of the features that any given specific application could need.

However, there are some drawbacks to using an existing interface that are closely related to the benefits. The superset of interface features could require extra work to implement. Also, the simulated features offered by the simulated environment do not necessarily represent the features that would be available in the real environment. For example, in the simulated environment you may have sensors available that will not be available on the real platform.

Another drawback is that neither the developer of the application, nor the developer of the interface board, owns the interface. The protocol, the physical connection, data rates, and all other interface attributes are defined and owned by the simulator developers. If the simulator releases a new version the application needs to change to stay in sync. If the application remains with the older version it may become unsupported. For many applications and simulators, this may not be an issue at all. But, if the simulator is newer, or has a tendency of being very dynamic, this could be a real problem. Most versions of simulators work to include backward compatibility, especially with regard to the simulator's interface. Nonetheless, the developer does not control this aspect of the project, and it has to be considered a risk at the very least, and taken into account when taking on an SCD project.

When working with a custom interface, the benefits and drawbacks are basically mirror images of those when using an existing interface. For example, one of the drawbacks of approach one is that the application developer and the interface board developer do not own, nor can they define any features of the interface. Approach two allows the interface to be developed to perfectly match the specific application it will be used in. As shown in Figure 3, the software on the interface board is what is offering up the API to the application under development. Thus, it can be specifically shaped for that application. This means that whoever owns that interface board will be able to own all key interfaces for the project. It abstracts the simulator's possibly more complex interface from the application. Even more, it could equip multiple simulators. The custom interface will be valid as long as the functionality implemented by the custom interface is an intersection of functions offered by all chosen simulators, as shown below in Figure 4. This removes the responsibility of staying in sync and up to date with simulator interfaces from the application developer, and puts it on the developer of the custom interface and interface board.

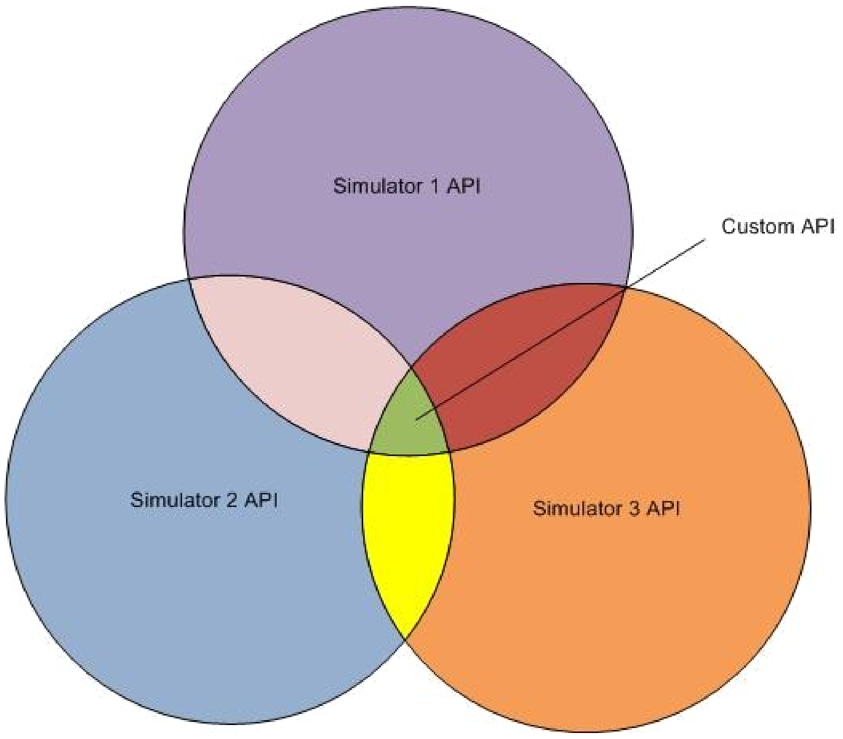

Figure 4: Example Custom Interface API Venn diagram

While a benefit, this can also be a limiting factor. The interface developer needs to take care to only add another simulator if it is proven to add value. As more simulators are added, the intersection of all their features inevitably shrinks. Thus, if one simulation platform is enough, then there is no reason to add more, because of the implications of shrinking which desired features are offered.

Another drawback of the custom interface implementation is the lack of pedigree. Re-using a tried and true simulator interface is a safe choice, because that interface has been proven to work by all developers who have worked with that simulator in the past. The more support a simulator has, the more reasonably confident a developer can be in the quality of the interface. When developing a custom interface, the developer or developing team becomes the testers as well. The benefits of user testing and usage history are no longer part of the situation, as they were with the existing simulator interface.

One final drawback to the custom interface is the increased level of domain knowledge it mandates. The simulator interface was most likely created by developers that understand the industry or domain being simulated. The data included, methods implemented, data rates, and even physical method of transmission, have been designed for a specific reason. Implementing something functionally similar or equivalent will require a non-novice level of understanding in the functional domain.

One final benefit of the custom interface still needs to be addressed. Unlike the simulator interface, which is heavier than necessary, the custom interface offers only what is needed. It can be shaped to fit the specific needs of the application. Because of this, the method names, parameter names, and functions provided, can be well-documented. The simulator interface may have obfuscated certain functionality or made it more complex than necessary, while the custom interface can be implemented exactly as the developer would like it to be, using a preferred physical transport medium.

Table 1 includes a summary of the benefits and drawbacks covered in this section. Either state could be the better choice depending on the developer, on the application, or on the platform capabilities. Approach one matches with the typical development lifecycle better, especially for projects following a simulator based design lifecycle, as it does not need new interface software and documentation created.

Table 1: SCD Benefits and Drawbacks

State	**Benefits**	**Drawbacks**
Existing Interface	• Tested • Documented • Currently in use	• Often heavier than necessary • Not necessarily representative of real platform • Owned by simulator
Custom Interface	• Minimal design(lower implementation costs) • Application-specific • Complete ownership • Allows multiple simulators	• Less tested • Requires more domain knowledge

For the specific application outlined in Section V, both approaches will be implemented, to show an example of both approaches. However, depending on the situation of another project implementing SCD, one approach will likely be preferable over the other.

III. REDUCTION IN COMPLEXITY – A SINGLE API FOR MULTIPLE ENVIRONMENTS

One of the key goals of this research is achieving the benefits that come from requiring less code. Though more software usually means increased capability as well as more features it usually means increased chances for exceptions or failures, and increased maintenance and debugging. If an interface was made available that reduced (or at least held constant) the necessary code while allowing the developer to use their system in both a simulated and real environment, the benefits in reduced development time, reduced debug time, and reduced maintenance time would far outweigh the cost to develop and test that API. This advantage becomes greater as the SCD platform is re-used from project to project. Since the simulator interface is re-usable on multiple projects, so would the SCD platform be.

IV. Potential Applications

As mentioned in section II.I.b, most domains currently doing simulator based design are likely candidates for simulator *centered* design. Any platform or development project that currently creates or employs a custom simulator that was written for the project is a potential candidate for becoming an SCD project. This is because all of the time spent by the developers in creating the simulator is time spent *only* on the simulator – the code and effort that goes into that portion of the project does not provide functionality when deployed in the real environment.

a. Automotive

Hardware in the Loop (HIL) testing and SBD currently play a large role in the automotive mechanical engineering development field. Real world experimentation and testing quickly becomes a cost-prohibitive activity, and alternatives are necessary.

For example, consider an anti-lock brake system. Testing the validity of an electronic brake controller is a very expensive proposition. Crashing even one vehicle when you expect the system to work is an impact on budget. This is not how these tests are done, at least during development. The wheel system, environment, and entire car are modeled in a computer simulator, as are the brakes themselves. The system is run through a variety of tests, all on the simulation computer. Bugs are found and fixed, and all without losing expensive and hard-to-produce hardware.

b. Aerospace

Aerospace is one of the more obvious areas of application for SCD. Aside from the example cited in the introduction, and the basis for this thesis, there are applications in larger-scale aerospace projects as well. The rigor imposed by the highly regulated nature of the aerospace industry presents the challenge that almost everything is remarkably expensive. Developing software for any part of the airplane is a much more rigorous and expensive exercise than in other industries. Though this is a very good for public interest, it places a challenge and constraint on software development projects.

c. *Hobbyist Platforms*

Often, simulators are viewed as a luxury for hobbyists – either because the simulator itself is too expensive, or the time it takes to get the controller compliant with the simulator is close to being the same amount of time it would take to develop all of the control software itself. In these projects, the interface could take up as much or more lines of code than the controller, depending on the complexity of the interface. The implications of interface complexity are detailed in section 2.1.a. Because of this, SCD is a great option for hobbyist platforms.

V. Metrics and Definition of Success

Implementing the two previously described approaches is not a simple task. There are timing considerations (frequency at which the data is sent), data set sizes, and data formats to consider. And that is all just in the specific implementation. The time it takes to develop these interfaces is one of the most important, if not the most important, aspect of this SCD project. One of the key benefits of SCD is the reduction in development time and code size. If implementing the simulator-based interface or custom interface does not equate to a reduction in code size, development time, and software bugs, then the time spent developing the SCD platform itself will have been a loss in the scope of the project.

Therefore, a point must be decided past which SCD becomes ineffective. Once this point is passed, the potential benefits from an SCD platform are no longer outweighing the associated issues and rework. At this point, SCD platforms are no longer outweighing the associated issues and rework, no longer making it a successful methodology. This defines the criteria for a successful project.

There are multiple ways to measure success, for each approach. Interface complexity, number of "bugs" reported and time spent in development are all options. The latter two are harder to track on such a small project. Measuring complexity and code size has an added benefit of being instantly calculable at any point in time, and doesn't require real time tracking. A developer can also go back in time in their version control system and re-calculate any specific branch or tag if so desired.

For approach one, the complexity of the simulator interface should be measured (before transitioning the project to the real environment). Then, when that operating mode is deemed implemented, the same interface will be evaluated. Ideally, it will not have changed at all (one could even perform a code comparison on the 2 code bases). If it did change, the versions will be compared, and the changes noted and explained in the thesis. Note that the changes will only count as induced by the SCD platform if it is determined that the SCD platform is the origin of the problem, and not deficiencies in the application itself, or the simulator's model of the flight dynamics used in development and debugging.

The complexity will be measured by Source Lines of Code (SLOC), and the number of variables being controlled. If there is less than a 10% increase in complexity, or code size, for approach one, following the transition from simulated to real, that will be considered a successful implementation.

This number was chosen because this is the approximation for what percent of the project's complexity the interface itself is responsible for. If the project is 100 lines of code, then 10 of those are for handling the external interface. If there is an added, yet equivalent, interface, it would require an extra 10 lines of code. Thus, if the proposed solution requires less code than an added interface, it is worth the time and effort it takes to implement and maintain. This measure is unique to the idea of SCD, but is based on previous experience with software, and the aforementioned research in interface complexity. Added SLOC and added interfaces inherently increases complexity, which is the measure that SCD works to reduce. Even if the specific measure chosen does not prove accurate to a given project, the theory behind it should stand – any reduction in code development and rework that SCD gives makes it worthwhile. Even more, every project the SCD platform can be re-used on effectively shrinks the cost of any rework.

For approach two, defining success is more difficult. This implementation really represents working in the opposite order as the first – going from how the device would be if we went straight to the real world, and then making that work on the simulator. Here, success is measured by the comparison of the approach two complexity to the complexity of the approach one solution – if the controller operates as expected when in the real environment, and the code is no more complex than approach one, then it was successful. Again a 10% buffer will be used – if it is within 10% the size of approach one, then it was a successful solution.

IV. AUTOPILOT DESIGN CENTERED ON X-PLANE INTERFACE

Taking the ideas outlined in the previous sections, this section will describe the experiment used to explore the question "can an external device that abstracts the environment from the processing code, allowing a single interface out of the controller, reduce development time?" The theories outlined above were applied to a specific project to test out the validity of the SCD approach, and if the benefits associated with reducing interface size, re-using simulator interfaces for real deployment, and the work associated therein outweigh any challenges that arise in doing so.

This section will start with a brief description of the project and describe the operating context for the autopilot developer. Following that, the specific requirements of the application are outlined, so the intersection can be determined between the project's needs and what features the simulator interface offers. That is, it is important to ensure that the simulator meets at least the basic input and output needs of the application. After that, the application requirements and, the requirements levied by the chosen simulator will be outlined. The description of the simulator is pertinent, because the simulator defines the interface being re-used. Based on the simulator and project requirements, the specific design decisions made while creating the SCD system are outlined. These include which hardware and software platforms were selected for the interface board, as well as the physical transfer mediums and protocols for the custom interface. After that the final product will be described, starting with the current state of operation for the developer, and the two potential approaches offered by the SCD platform that were made available to that application developer.

I. PROJECT DESCRIPTION

This project was conducted in conjunction with another project involved in using an Android (Google, Inc., 2012) smartphone as the controller for an autopilot platform on a radio-controlled plane. One of the main challenges in this project was getting the real time computation needs for controlling an aircraft out of the non-real-time Android platform.

The Android controller developer agreed to use the proposed SCD platform for the project. The Android controller project would be provided with an external hardware device that made the real world appear exactly the same way the X-Plane flight simulator presented it. Thus, the development plan was based solely on the X-Plane simulator – there was no need to interface with an actual r/c airplane. The Android application used X-Plane as part of their development environment, interfacing directly with it over the Android's Wi-Fi connection. The specifics of the X-Plane interface will be described later in this thesis.

This approach and architecture matches the previously described development pattern for approach one, where the interface hardware is essentially invisible to the application, since it looks and communicates exactly like the simulator would. Note that, for the sake of comparison, both approaches will be developed for this application – both the simulator centered approach, and the custom interface approach. These are further documented later.

One unique challenge with this particular implementation was that the Android platform has many internal sensors that provide the same readings that are typically provided externally to a controller. For example, GPS (position) and accelerometer (rotation) data are provided by an Android from sensors internal to the device, whereas most autopilot controllers, and any device interacting with a simulator, receive these input sensor readings from external sources. This was addressed by adding a preference to the Android application – use internal or external sensors. If the user selected external, then all sensor data was retrieved from the outside world (simulator or real platform). If they selected internal, the on-board sensors were used. Using this methodology, the developers were able to “trick” the Android, using Mock Location settings, into thinking it was in whatever physical position was being reported by the simulator, instead of sitting stationary on the desk of the developer. One added benefit here was the built-in expandability for the Android platform. If it was determined that the internal sensors proved insufficient or too slow, the framework is in place for external sensors to be added to the platform, without any added work required of the application developer.

II. Application Requirements

The first issue that comes to mind for most people when this topic is discussed is that of safety. As mentioned in the introduction, one benefit of SCD is the ability to test in a simulated environment before transitioning to the real environment. However, there are always unexpected variables and issues in the real environment that were not anticipated. A backup plan is necessary to prevent loss of control of the airplane.

A flying airplane, no matter the size, has safety implications for both people and property. Though the main benefit of the development environment was that extensive testing and debugging could be performed in the lab environment, one should not expect testing to remove 100% of bugs and defects.

Therefore, a failsafe for the r/c airplane needs to be added. There were multiple approaches considered here. For example, the SCD platform itself is a highly reliable platform. The likeliness of a bug in the application is higher than in the SCD code, both due to difference in SLOC count and complexity introduced by the Java Virtual Machine (JVM) running on the application.

Based on that, one method for implementing a failsafe was a communications watchdog timer on the SCD processor. The watchdog would check to make sure the SCD system was receiving a command message from the application at a specified rate. If that rate was not met (that is, a message was not received within the expected time), or a certain number of messages were not received in the correct amount of time, the failsafe mode could be executed. This would put the onus on the SCD system for deciding how to handle this "emergency" condition. Likely, this would be putting the rudder 100% to one direction and decreasing throttle, which would allow the airplane to spiral down to the ground at whatever location the autopilot was able to last guide it to. If the application reported location to a ground station or something similar, this location would be close to where the SCD system brought the craft to the ground.

The drawback to the above implementation is how little allowance is made for the possibility of human involvement. It places all of the need for correct decision making on the software. The purpose of a failsafe mode is to account for the possibility of failing or faulty software. If the backup mode was more software, it is possible that code could fail as well. Therefore, it would be preferable if a human-controlled element was in the loop.

Based on that need, the r/c controller itself was considered. Standard radio controllers come in two channel configurations – four and seven. A four channel controller typically has a channel each for throttle, elevator, rudder, and aileron. A seven channel controller has three extra channels for controlling other items (nose-wheel, camera, etc.). Some airplanes use one channel to determine which controller the control signal should be received from; this is very useful for trainer airplanes.

In this application, an additional r/c channel would be useful for a failsafe mode. If the human controller was within line of sight and determined that it was operating improperly, they could flip a switch on their controller, which would send a signal over the fifth channel. This signal would be received by a failsafe mux board (DIYDrones Failsafe Mux, 2012) which would then switch from using control signals from the Arduino to using control signals from the r/c controller. The human monitoring the airplane is required to make the determination if control needs to be retaken from the application.

The final option for safety builds on top of the previous idea. The Arduino device would still contain the watchdog timer that checks to make sure communication is maintained. In addition, a “failure mode” message could be added that the application could send the SCD system to put it into failure mode. This mode would be the throttle down, hard rudder spiral. This would then be a visual indicator for the human controller. They would be able to recognize the unusual flight path, and activate controls on their r/c transmitter to take over control. Or, they could choose to let the airplane continue spiraling down, if they determined that was the safest choice. Note that this is limited to being a line of sight (LOS) solution. If this SCD platform were to be expanded into other applications, like the crop-dusting application, where the airplane moves over the horizon, a different ground station solution would be necessary. However, the fundamentals of defaulting to human control under any failure conditions would remain the same.

This was determined to be the best option. It combines the best of both previous options. It allows both the application and SCD system to make their own determinations as to when a failure or fault worthy of aborting the mission has occurred. It provides a visual indication to the users, so even if no air to ground digital communication has been established, some indicator of change of flight state is provided. Finally, it keeps the connection the plane was originally built for – the r/c control connection. Because of these failsafe modes, this reduces the risk to being equivalent to the same risk associated with flying a simple r/c airplane, if not less.

After the safety issue was addressed, the input and output needs of the application were developed. These are independent of the operating mode (simulated or real). This section outlines what sensor and status inputs would be required for a simple autopilot controller, as well as the control surface outputs. This was a minimal list, but it was sufficient for purposes of investigating SCD.

First, the inputs were covered. Following a basic navigation, guidance and control approach, the system needs to know current position (latitude, longitude, and altitude), current orientation (pitch, roll, and yaw), current velocities and accelerations, and position of control surfaces.

A table containing the list of inputs and outputs required by the application, and their corresponding units, can be found in Appendix A.

To modify those values (i.e. – change heading), a minimum of three control surfaces are necessary. Those three are elevators for controlling pitch; a rudder for roll/yaw and the engine throttle is also required for controlling velocity. A fourth additional option is the ailerons for controlling roll. In this specific project, the ailerons were not necessary, because the airplane had dihedral wings.

The basic approach employed by the application developer using these inputs and outputs is as follows. At a preset execution rate, the main loop of the Android function will poll the input sensor values. Based on readings regarding current position and velocities, the autopilot controller software will provide new deflection settings for the airplane control surfaces. By deflecting the different surfaces up and down or left and right, the control software will be able to adjust and manage the airplane's position in three-dimensional space. Every time a control surface is deflected, the position will update correspondingly, providing feedback for the next time through the loop. The frequency at which the loop executes can determine the granularity of controls and the airplanes ability to respond to change. For example, if it only updates commands once every five seconds, it likely will be much more susceptible to wind and other air anomalies. If the loop executes faster, the control system will respond better to disturbances, but require more processor time, and faster input and output mechanisms. This will likely play into the interface design decisions, especially for the custom interface, approach two.

For approach one, in this particular application, the specifics of the interface are defined by the chosen simulator, X-Plane (Meyer, 2012). X-Plane offers many methods for getting data into and out of the simulator engine. For one, it has a plug-in interface. Developers can create plug-in software that can be run from within the simulator. This is useful for controlling air traffic, changing scenery, weather conditions, recording flight, and many other purposes.

X-Plane also has a network interface. By providing X-Plane with an IP address and port address for a remote machine, X-Plane begins to send information to that machine, and listens for response messages. This interface can be used for multiple purposes. One is to provide two machines, to simulate a pilot and co-pilot station for pilot flight management system training. However, the network interface makes X-Plane capable of more. Almost any flight control or environmental element can be modified over the network connection. X-Plane does these using elements called “data sets” and “data refs” (DataRefs, 2012).

Data sets are simple data structures, built as arrays of floating point numbers. Data sets contain a large variety of information. For example, current set deflections for control surfaces, current positions of control surfaces, throttle settings, airspeed readings, and GPS position are all included in data sets. These pieces of information are basic to almost any flight, regardless of mission. In addition, less universal settings, such as landing gear, weapon system status, and airplane lights are represented by data sets. X-Plane transmits selected data sets to the computer specified by IP address and port. This is done to report current state of the airplane being simulated. However, the data set interface also allows the remote computer to respond with similar data set messages. If they do respond, X-Plane takes the data in the response as the new state of that setting. For example, if the remote machine responds with a data set that indicates a new control surface deflection, X-Plane moves the control surface of the plane being simulated to that set deflection. Not all data sets are writable. For example, latitude and longitude cannot be set, as they are attributes of the flight, and not elements that can directly be controlled. Data sets are capable of more than this, but these are the features pertinent to this problem. Using data sets, a developer can control flight of the airplane in X-Plane, using the simulators model of flight to test out their autopilot control algorithms.

The other main network interface offered by X-Plane is the data ref interface. This is a string based interface, as opposed to the arrays of floating point numbers used in the data set interface. Data refs are useful for more abstract settings than data sets. For example, a user can set plane fuel levels in multiple tanks, report status of airplane doors, control many different types of cockpit radios, cause equipment failures, and much more. X-plane uses data refs differently than it does data sets. It does not send any messages out, like it does with data sets. Data refs are also aperiodic – they can be set once at any point, and do not have to constantly be reset, like data sets do.

In an SCD project, the quality of the simulator is integral to the success of the project. X-Plane's model of the airplane being simulated is very important. The more accurate the flight model, the more successful SCD will be. This is because the simulator is used for almost all debugging of flight control algorithms. The more accurate this flight model is, and then the better the SCD platform is on making the real world look like the simulator, then the more likely the project will be successful. The airplane used on this project is a PT-E, an electric trainer r/c airplane. The PT-E has 3 control surfaces – throttle, elevator, and rudder. It does not have ailerons because it has dihedral wings, and adjusting the rudder causes the airplane to bank automatically. X-Plane does not have a PT-E model, but does have a PT-60. These are similar airplanes. In addition, X-Plane allows the user to adjust the dihedral angle of the wings. Doing this makes the PT-60 even more like the PT-E.

III. Design Decisions

a. Interface Board

There are many options on the market today targeted at hobbyists for hardware prototyping projects. The needs of this specific project dictate the choice for which option to select. This project will need to interface with an Android device, as well as the simulator on a desktop computer, and also the airplane itself (via servos). Thus, it will need to be very expandable to accommodate all the potential interfaces. In addition, it will need to be fast enough to keep up with the data rates provided by the simulator (though X-Plane's dataset frequency is configurable). Finally, the device should support rapid prototyping, either through available software libraries, community support, or technical support.

System Options:

- BeagleBoard / BeagleBone (Linux based)
- Raspberry Pi (link)
- Texas Instruments MSP430 Launchpad Kit
- Arduino (many different flavors)

The BeagleBoard (BB) system is a compact, yet high-powered computing platform. Basically a full traditional computer on a board (with I/O for keyboard, speakers, monitor and mouse), this platform is typically used running a version of Linux. So, it hosts a full-up operating system. This is powerful, but likely undesirable for this particular project, as the benefits of running an operating system do not outweigh the drawbacks and added work. (Beagle Board Quick Start Guide, 2012)

The Raspberry Pi (RP) is a newer board on the hobbyist market. The RP is similar to the BB, in that it is basically a single board computer. It runs a version of Linux, but is much lower cost. For the same reasons as the BB, it is undesirable for this application. The quick-start guide calls out the need for a keyboard, mouse and monitor – making it a full computer. Though these would not be necessary for all operating modes, it demonstrates that the platform is likely a bit heavy for this application, and another option would be preferable. (Raspberry Pi Quick Start Guide, 2012)

The TI kit is meant to be a very low cost option for developers and hobbyists. It comes with two microcontrollers, which can be swapped out for one another. It comes with its own software development suite, like several other platforms. In addition, TI has made some plug-in hardware modules that help expand functionality. However, since it is such a low-cost platform, its capabilities are limited accordingly. RAM is 512B, and flash is 16k. In addition, the processor has a very limited number of IO pins compared to other platforms. (MSP430 LaunchPad Value Line Development kit, 2012)

The final option is the Arduino. Arduino has the strongest and largest developer support community, with over 100,000 developers actively participating and working on projects. Though it comes in many different shapes and sizes, almost all Arduino boards run an Atmel microcontroller, and have a high number of IO pins available. The Arduino development environment uses its own custom language, but it is an extension of C/C++ (compiled with avr-g++). (Arduino, 2012)

Based on the above options, the Arduino option was chosen largely because of the support community surrounding the Arduino, and the extensibility using libraries. The Arduino community is also the environment used for the DIY-Drones project Ardupilot (DIY Drones - Getting Started, 2012). This project uses the Arduino as the entirety of the platform, both the brains of the autopilot and the interface to the airplane. The Arduino has many hardware and software developers working on shields (printed circuit boards (PCB) used to extend the capabilities of the platform), which allow things like network connectivity, GPS location sensing, servo and DC motor control, and much more. Thus, it was determined that Arduino is a reliable choice to satisfy sensor and network connectivity requirements. Furthermore, it is most likely to have well-documented support for that sensor or network connection.

b. Protocols

The decision for the custom interface was once again predicated by the use of Android. There are physical interfaces into and out of the android device and all of them - except USB – are wireless.

- Bluetooth
- Camera (visual)
- Microphone (audio)
- NFC
- Touchscreen (capacitive sensing)
- Volume Control & Power buttons
- Phone-provided Network (E, 3G, 4G)
- Wireless (802.11b/g/n)
- USB

There are, like in any design decision, benefits and drawbacks to each of these interface protocols. Each can be the best choice for certain applications. For example – the microphone is the obvious best choice for making phone calls – it is much easier to say what words you want transferred over the phone instead of having to type them and have the device turn them into speech. Likewise, the phone network is the best means for connecting to a global network, as opposed to NFC, which would require much added infrastructure.

For this application, several interfaces were chosen, based on the two approaches. For approach one, the decision followed directly from the choice of the X-Plane simulator. As documented before, X-Plane interfaces with external machines using the host computer's network connection, sending and receiving UDP data packets. Fortunately, the wireless network connection on the android device enables the network connectivity required for this interface. If a network adapter is available for all devices that would be sufficient for implementing approach one.

Approach two is slightly more complex. Here, the definition of both the interface and the protocol are up to the developers. Several of protocol choices listed previously can be ruled out easily. For example, controlling via audio or visual signals (using the speaker, headphone jack, or signaling with the screen) is impractical, due to limited baud (or refresh) rates and limited bandwidth.

The practical interfaces are:

- USB
- NFC
- Bluetooth
- Phone-provided Network (E, 3G, 4G)

USB is a very high throughput and reliable protocol. However, it requires a wired connection. Though this does increase reliability, it also increases on-airplane weight and decreases variability in the arrangement of devices on the plane. In addition, for Android, it requires an Android equipped with the “Open Accessory” library. Only a small fraction of active Android devices on the market today are equipped with this library. Finally, it requires USB hosting capability for the interface board (narrowing down the list of valid Arduino devices).

NFC (near field communication) is a relatively new technology for mobile devices. Most often used for services like Google Wallet (Google, 2012), NFC is built upon RFID technology, and requires two devices to be within a few centimeters of one another. The devices communicate via radio, usually limited to short, low data interactions. One common application is “contactless payments,” where a mobile device can be waved near a NFC receiver, communicating payment data (like a credit card number) for a transaction.

While NFC is an exciting new technology, it comes with all of the risks and shortcomings of new technology. Like the Open Accessory protocol, very few Android devices are equipped with the necessary hardware for NFC. Another drawback is the somewhat low bandwidth – data rates are limited to 424 Kbits/second. Also, while the physical range may not present an issue on this application, requiring a distance of 4 centimeters or fewer limits physical arrangements more than is desirable.

The phone network requires a data plan. This means the Arduino would need both a shield that contains a valid SIM card, and the SIM card itself, from a network provider. A SIM card is available for around $10 a month, but requiring a user to subscribe to a service in order to communicate on a network of two devices operating within inches of one another is overkill. Also, the phone network option requires that the system always be used within service range of a cell tower. While this will likely not be a problem for most users, mandating it seems unnecessarily restrictive.

Based on all of the factors discussed, the Bluetooth interface was selected as the best option for implementing approach two. Bluetooth is a very well-established technology (Bluetooth - How it Works, 2012). It works wirelessly, constructing a WPAN (Wireless Personal Area network). This means that more than two devices can communicate using Bluetooth. Bluetooth range goes up to three feet for some devices, even up to 33 or 300 feet for certain classes. This is more than enough for a network around an r/c airplane. Bluetooth has data rates around 2.1 Mbit/s, which should be more than enough for the refresh rates of the simulator, as well as the control loop execution rates of Android functions.

IV. Current State

The team began development without an interface board or other hardware involved. The controller application was connected directly to the simulator, using the simulator to simulate the real world to the controller. Below, Figure 5 shows the current state of this project, as any autopilot developer would see it while developing in their own lab.

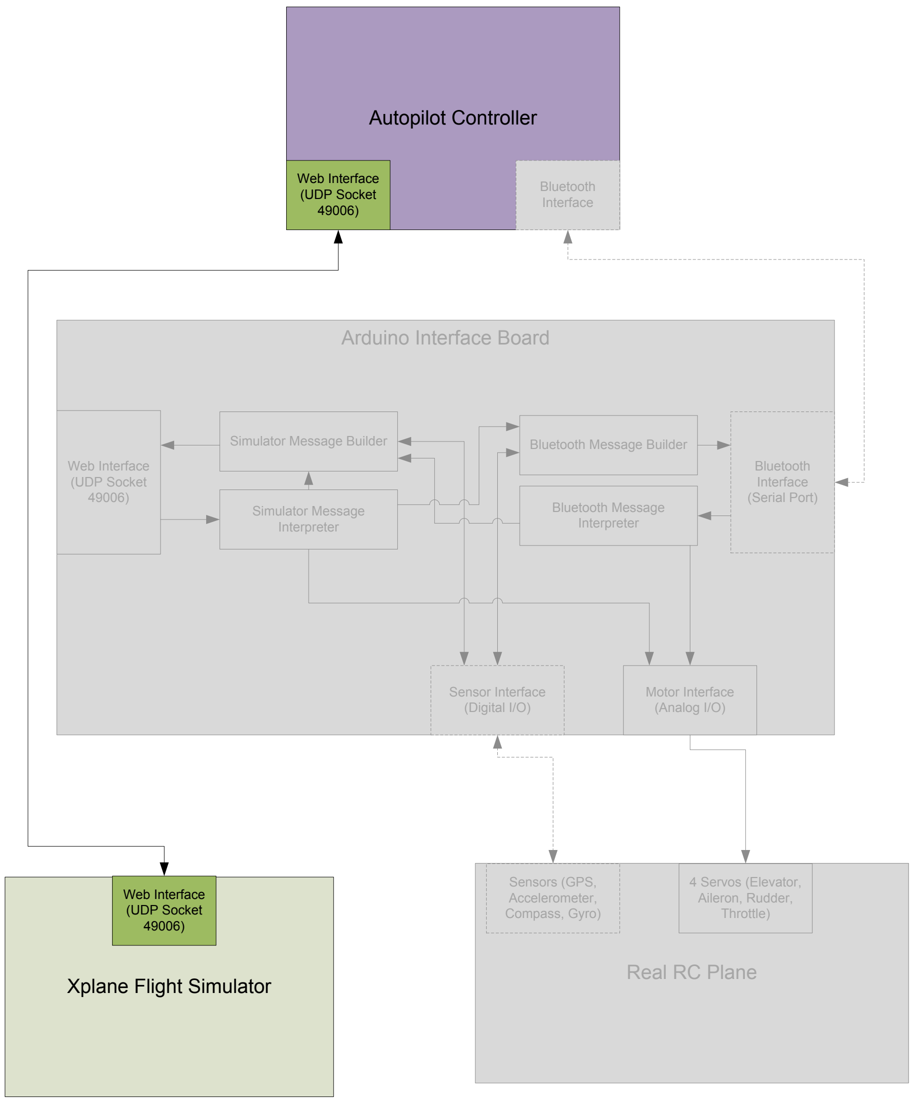
Autopilot Controller
Web Interface (UDP Socket 49006)
Bluetooth Interface
Arduino Interface Board
Web Interface (UDP Socket 49006)
Simulator Message Builder
Simulator Message Interpreter
Bluetooth Message Builder
Bluetooth Message Interpreter
Bluetooth Interface (Serial Port)
Sensor Interface (Digital I/O)
Motor Interface (Analog I/O)
Web Interface (UDP Socket 49006)
Xplane Flight Simulator
Sensors (GPS, Accelerometer, Compass, Gyro)
4 Servos (Elevator, Aileron, Rudder, Throttle)
Real RC Plane

Figure 5: Autopilot Current State

The autopilot communicates directly with the X-Plane flight simulator. There is a portion (shown in green) of code to interface with the simulator. This module translates control messages from the controller into simulator format, and forwards the result to the simulator. In return, the module receives airplane status messages, sensor information, environment information, and other data from the simulator, and translates them into a format prescribed for the autopilot controller. The interface module within the simulator is "closed-source" for this project, meaning that changing this particular implementation lies outside the bounds of the project. In this development configuration, all interfaces are made compliant with the simulator's specification.

As mentioned before, this project will be implemented to mirror the X-Plane data-set interface. The newest version of X-Plane (version 10) has 106 different data sets. Each data set is structured following a standard – a four byte integer containing data set identifier (1-106), followed by 8 four-byte floating point numbers. Not all eight numbers are used in every data set – but the X-Plane documentation makes clear which data set contains which piece of data. So, the X-Plane interface on the controller, as well as the software on the SCD platform that imitates this interface, will need to be able to build and parse data sets that match this format.

V. Approach One – Existing Interface

X-Plane's external interface is centered on the Ethernet connection. Approach one was built around this Ethernet interface. The Arduino board was used for connecting the simulator and the application. It serves as a go-between, mediating the exchange between the controller and the simulator (Figure 5). This sets the stage for more processing as well. Since the Arduino now has access to control signals and sensor information, it can react accordingly. For instance, if the application was commanding the plane in the simulator to bank down, the Arduino software forwards that message, but also interprets it and can perform whatever actions it is programmed to execute. In fact, since it is the go-between, it could modify the data. This could be useful for a filtering function, if the control software required the data to be filtered and damped before being sent on, or returned to the controller.

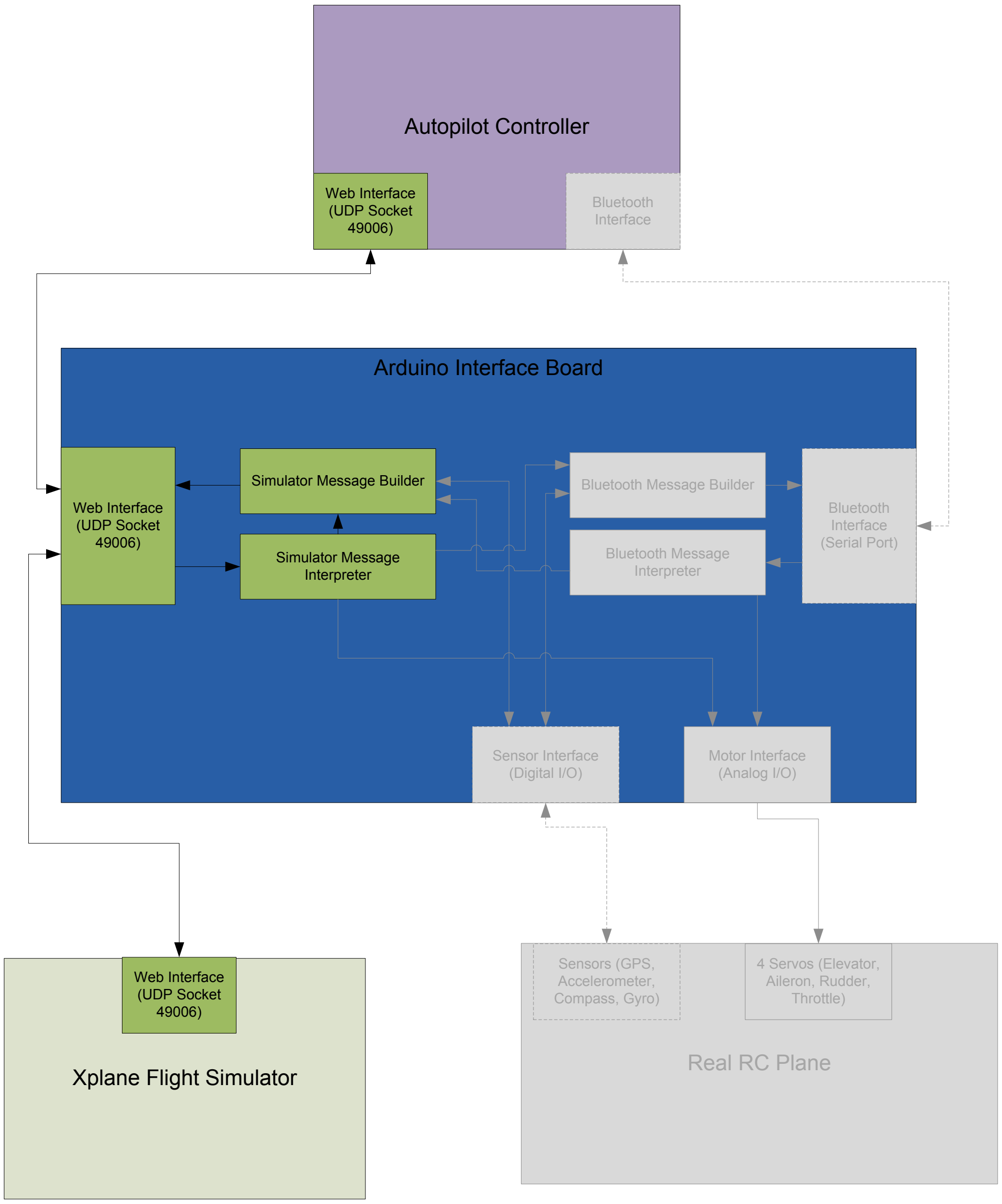
Autopilot Controller
Web Interface (UDP Socket 49006)
Bluetooth Interface
Arduino Interface Board
Web Interface (UDP Socket 49006)
Simulator Message Builder
Simulator Message Interpreter
Bluetooth Message Builder
Bluetooth Message Interpreter
Bluetooth Interface (Serial Port)
Sensor Interface (Digital I/O)
Motor Interface (Analog I/O)
Web Interface (UDP Socket 49006)
Xplane Flight Simulator
Sensors (GPS, Accelerometer, Compass, Gyro)
4 Servos (Elevator, Aileron, Rudder, Throttle)
Real RC Plane

Figure 6: Autopilot Future State One

An alternative configuration of this state with more hardware involved is shown in Figure 7. In this configuration, the r/c airplanes hardware becomes involved. This is a version of Hardware-in-the-loop (HWITL) testing. This type of testing is done to validate the airplanes response to control data. As mentioned before, in this configuration, the Arduino now has visibility into the control commands. So, if the sends a message to move control surfaces, the

Arduino software can interpret the packet, move the surfaces of the r/c airplane, and also send that control command on to X-Plane where the simulated control surfaces will be moved. X-Plane will use this information to simulate the condition in its model of the environment.

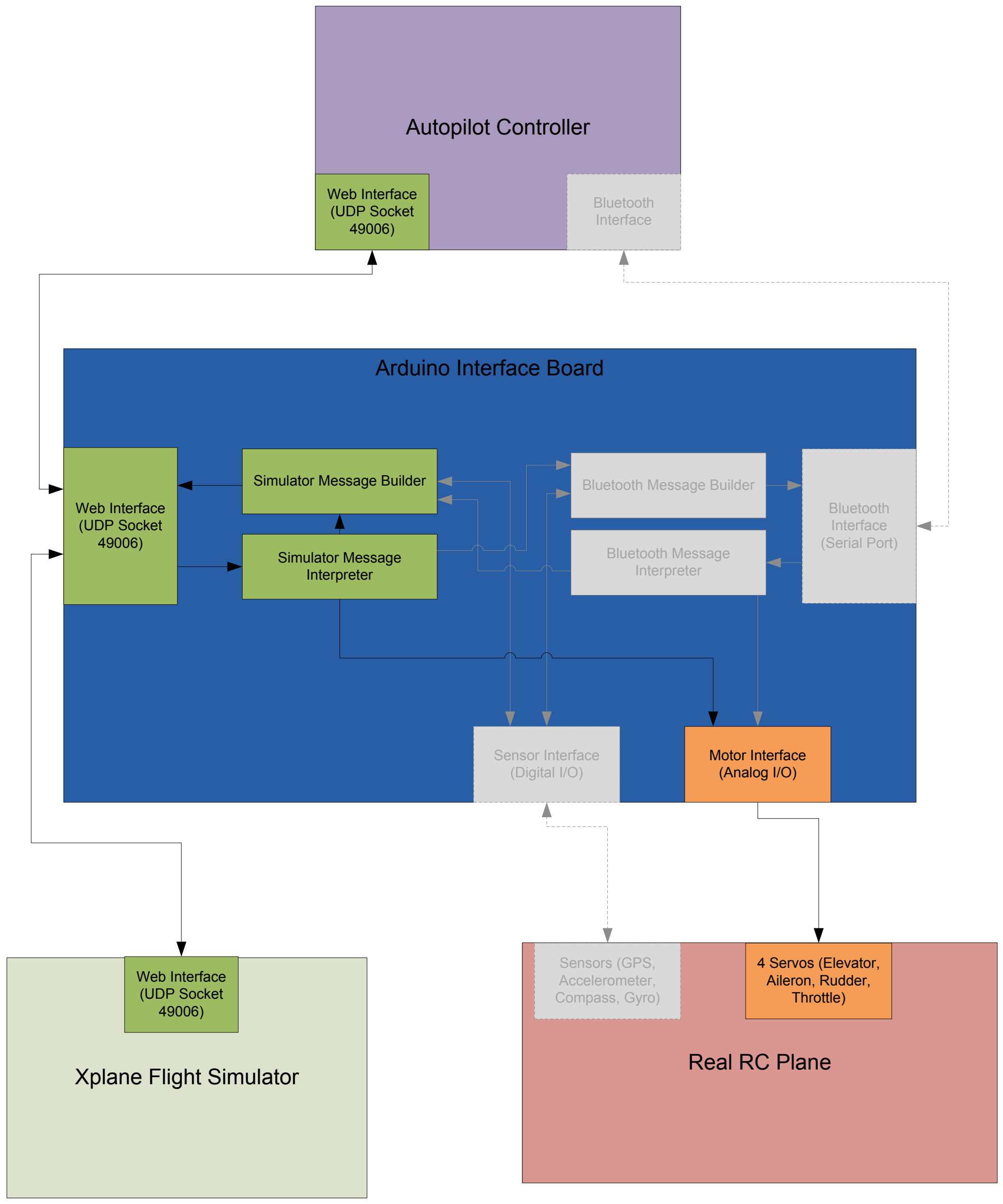
Autopilot Controller
Web Interface (UDP Socket 49006)
Bluetooth Interface
Arduino Interface Board
Web Interface (UDP Socket 49006)
Simulator Message Builder
Simulator Message Interpreter
Bluetooth Message Builder
Bluetooth Message Interpreter
Bluetooth Interface (Serial Port)
Sensor Interface (Digital I/O)
Motor Interface (Analog I/O)
Web Interface (UDP Socket 49006)
Xplane Flight Simulator
Sensors (GPS, Accelerometer, Compass, Gyro)
4 Servos (Elevator, Aileron, Rudder, Throttle)
Real RC Plane

Figure 7: HWITL Future State One

Note that in Figure 7, there are blocks on the real r/c airplane that are still shaded. These are the blocks that would provide information to the application for controller feedback. Any sensor the application would need, like GPS, airspeed, acceleration, would be included in this block. However, as mentioned earlier, one of the particulars of this application is the smartphone platform. This device has all the sensors necessary for flight control internally, and can access their readings internally. Thus, once operating in a real flight mode, the application will not look to get sensor information from an external source, but will use native sensors. However, if this SCD platform were to be used by another application, without those sensors internally available, this hardware would be necessary. The Arduino software would need to read from these sensors and compile their readings into X-Plane formatted data sets.

VI. Approach Two – Custom Interface

For this particular application, approach number two was also implemented; in parallel to the effort for implementing approach number one. This was done to verify that this was a viable alternative to approach one. Approach two consisted of defining a custom ICD with the application developer – the information requirements were listed, and prioritized.

The implementation of approach two is based on the smartphone's Bluetooth controller. As can be seen by the different shading on the diagram in Figure 8, the autopilot controller no longer has a web interface based on X-Plane. However, on the Arduino, the structure of the Bluetooth software is built to be very similar to the web interface on the left side of the block. There is an interface, a message builder, and a message interpreter. These structures, coupled with the web interface (which is still present for use with the simulator), turn the data coming in through the Bluetooth interface into X-Plane compliant messages, formatted as they were in approach one.

Not shown here is the hardware-in-the-loop version of state two. It is identical to the changes for the HWITL diagram for approach one, where the hardware on the bottom left portion of the image becomes un-shaded. This means the Bluetooth message builder is responsible for taking readings from the sensors and putting the data in the defined message format. This does not require more work, as it would do the same for sensor information coming back from the simulator as well.

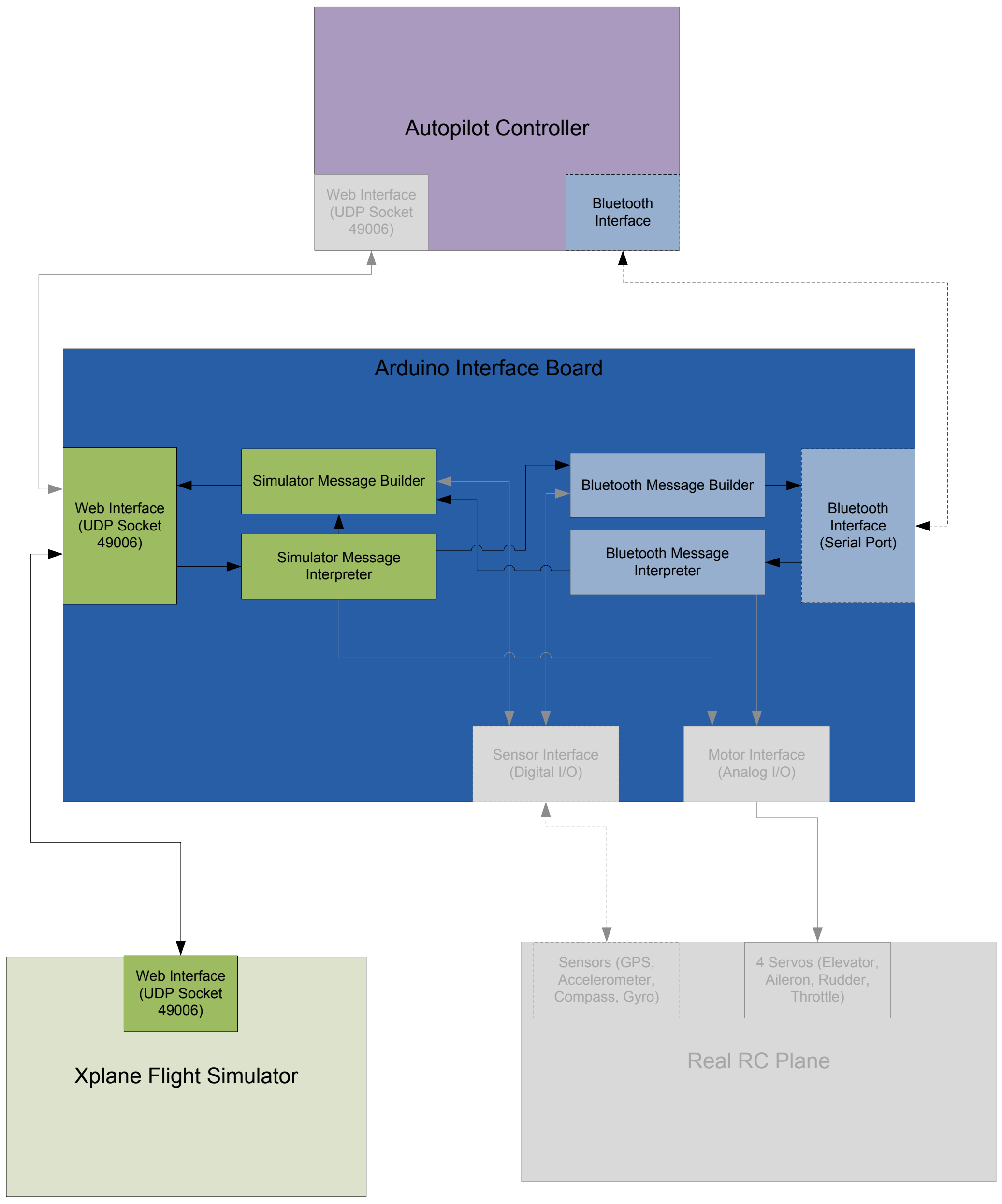
Autopilot Controller
Web Interface (UDP Socket 49006)
Bluetooth Interface
Arduino Interface Board
Web Interface (UDP Socket 49006)
Simulator Message Builder
Simulator Message Interpreter
Bluetooth Message Builder
Bluetooth Message Interpreter
Bluetooth Interface (Serial Port)
Sensor Interface (Digital I/O)
Motor Interface (Analog I/O)
Web Interface (UDP Socket 49006)
Xplane Flight Simulator
Sensors (GPS, Accelerometer, Compass, Gyro)
4 Servos (Elevator, Aileron, Rudder, Throttle)
Real RC Plane

Figure 8: Autopilot Future State Two

Several difficult obstacles arose during the implementation of this option. The first, as outlined in section II.I.a, is the added interface added complexity. More code meant a more complex project. This is because the added code adds functionality, and as functionality increases, so do paths of code execution, and ways the execution can go awry.

Another obstacle that arose here was the need for increased communication with the application developer. Using the existing simulator interface meant that both the platform developer and the application developer had a third party documentation set to reference, which neither party was the author. This gave each developer a common list of specifications, with neither responsible for editing, fixing, or updating the list.

With the custom interface, this is no longer the case. The list of interface items in Appendix A – Application I/O Requirements drove the architecture of the custom interface. Since the list is very short, and the data types are all basic (integers or floating point numbers), a simple get/set interface was deemed sufficient. That is, the application would call a function like “getLatitude” which would then use the custom interface over Bluetooth to retrieve the floating point value for current latitude from the simulator, or external sensor. Similarly, a function like “setElevator” would accept an offset for the elevator control surface. This would be transferred over the Bluetooth interface, and the message interpreter on the Arduino would respond accordingly, by sending the correct corresponding message to X-Plane, and/or deflecting the control surface on the actual r/c airplane.

V. Results of SCD Experiment

The final design of this SCD platform for an r/c airplane ended up very similar to the intended design described in the above sections. Often, when implementing a design, lessons are learned and changes must be made to the design based on issues that arise during that implementation. That was true to an extent with this project, but not as much as typically occurs during testing. That serves as an indicator of the validity of the design. One main benefit of SCD is the reduction of any rework required when transitioning from testing to deployment. Since this was the first exercise using SCD, the rework required would likely be on two platforms: both the SCD platform, and the application platform itself.

I. Hardware Design

The hardware design of the project is based around the Arduino, and any necessary peripherals that are compatible with the Arduino. Below, Figure 9 shows a diagram representing all of the parts that were included in this project. This diagram was drawn using the Fritzing tool (Fritzing - About, 2012).

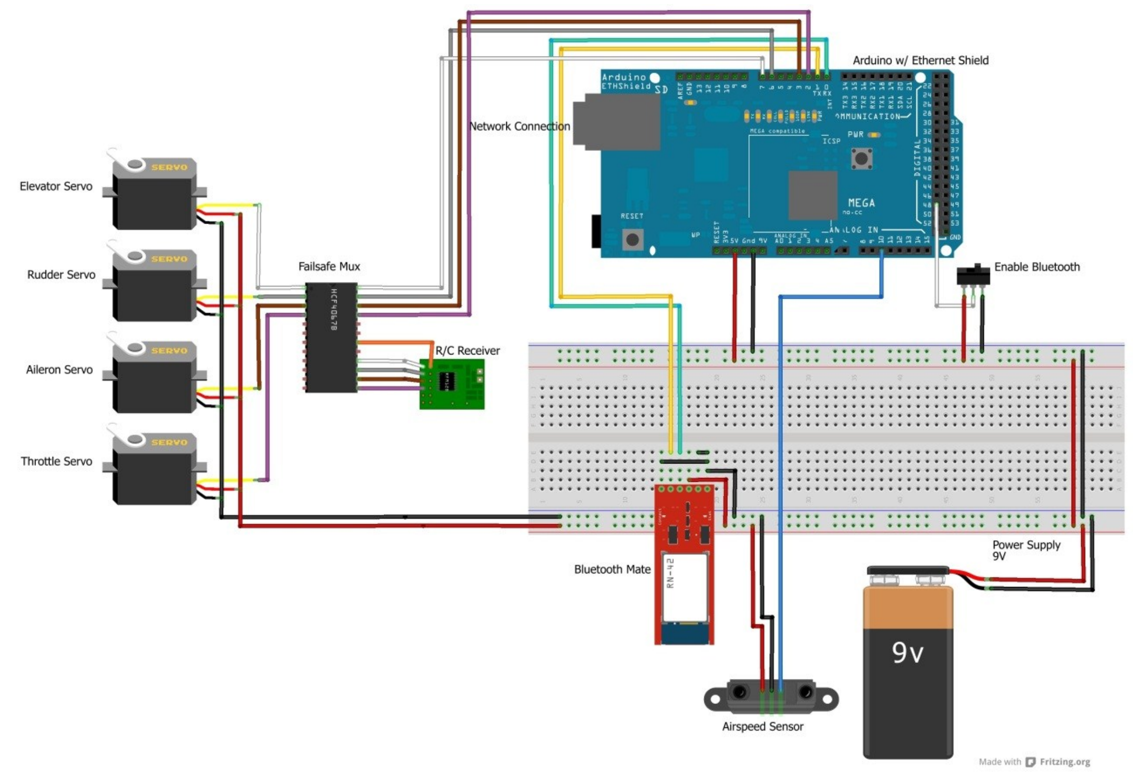

Figure 9: Arduino Configuration

The large blue rectangle is the Arduino Mega. This is the central processing unit for the SCD platform. The software running on the Arduino is described in the next section. On top of the Arduino is an Ethernet shield. This shield is used for enabling standard network connectivity. Not shown attached to this shield is a wireless adapter. The adapter used is a mobile Wi-Fi router that is capable of creating new wireless networks, or joining existing networks. Using this configuration, the Arduino platform can join any existing Wi-Fi network. The four boxes on the left side represent the four servo motors controlling the airplane’s control surfaces. Each is controlled by a single input from the Arduino, or the r/c receiver, depending on the control signal (the orange line from the r/c receiver). The red rectangle is the Bluetooth Mate Silver, a Bluetooth module configured to work on a 2 wire serial connection, which works well with the Arduino.

One positive aspect of the hardware design is that all hardware can function properly off a power supply that is standard for the Arduino, as well as the battery that powers the r/c airplane already. The Arduino has on-board power regulation, so any sensors that need the Arduino power can be hooked up directly to that power supply, and the Arduino can run off the airplane power supply. This helps keep down on-airplane weight.

As described in Section IV.II, one key consideration of this project was safety. The DIYDrones failsafe multiplexer part was installed. It was configured so default operation (even if no signal came through on the control line) was the r/c controller. This made it so the controller had to be switched on, and communicating with the mux, for it to be able to transition into autonomous mode, where the autopilot application would be given control of the airplane surfaces.

Another feature of the hardware design is the various sensors. As previously mentioned, one unique feature of using the Android smartphone as the autopilot platform is how many sensors it already has internally. Accelerometers, pressure sensors, and GPS sensors are all within the phone itself. However, if another autopilot controller were to use this SCD platform, or the Android's sensor suite proved insufficient in some way, the sensors would need to be added to this SCD system. For example, this system did in fact need one additional sensor – airspeed. This was used to calculate true airspeed (by comparing to groundspeed from GPS). This airspeed sensor is shown on the bottom center of the image. Since the Arduino has so many I/O pins and is so easily expandable, adding more sensors is very easy with this configuration.

II. Software Design

The previous section outlined the hardware of the final design. The hardware organization is a key feature of the SCD platform, as it serves to make the physical hardware look exactly like the world created in the simulator, which is entirely software based. However, the gathering of information from hardware sensors, and proper manipulation of the servo motors, would not be possible without software. More so, making the real platform appear exactly the same way the simulator makes the world appear would not be possible without software.

The software designed for this application was isolated into separate modules for each of the approaches. A digital discrete line into the Arduino configured which operating mode the software entered on startup. A high signal meant the Bluetooth module was powered, and the Arduino should be running in Bluetooth mode. This operating mode corresponds to approach two, as documented in section V.VI.

The first attempt at getting the Bluetooth interface working involved using the basic libraries provided on both the Android and Arduino platforms. However, this proved to be very problematic – the Android Bluetooth socket libraries are not well documented, and establishing a reliable connection with the Arduino Bluetooth module proved prohibitively difficult. Getting this up and running on the project's short schedule would not be an option.

To implement this state rapidly, the Amarino toolkit was used (Kaufmann, 2010). This toolkit was used as an alternative after attempting to open a simple Bluetooth socket and transfer data directly like an Ethernet socket. Bluetooth connections on the Android are complex to establish, and require a great deal of handshaking to establish the connection. In addition, the Arduino Bluetooth module is used as a serial port, and needs several commands to configure it to work properly with the Arduino. The Amarino toolkit removed all the overhead of implementing both sides of this interface, by providing a standardized library for both platforms. The Amarino library was added to the Android and Arduino projects, and specific functions were made available for transferring data back and forth. This library worked well due to its ease of integration and implementation, but it did have its limitations. For instance, all data needed to be transferred as strings. This meant that simple data types like integers and floats needed to be converted to strings, transferred via Bluetooth, and then converted back on the other side. In addition, the Arduino serial port library has limits on buffer sizes, due to hardware limitations on the microcontroller itself. This became apparent when trying to take full packets of information of X-Plane, reformatting them, and trying to send them all at once over the Bluetooth interface. The data would quickly become garbled, and the parts that came through correctly were very stale - a few seconds behind X-Plane, with the delay increasing as execution continued.

The software was designed in several distinct modules, to increase readability, and make reconfiguring operation that much easier. The modules were broken down as follows:

- Main Loop
- Bluetooth functions
- Ethernet functions
- Sensor functions
- Control Surface functions
- General Debug functions

Implementing the X-Plane interface (approach one) was done in the Ethernet module, to keep that interface entirely separate from the Bluetooth functions. To enable as fast of operation as possible, the system was designed so every time a packet was received, it was sent directly to the intended target (X-Plane or autopilot), and the parsed to see if any response was necessary from the Arduino, like moving a control surface.

The sensor functions implemented the ability to read from the airspeed sensor, and any other sensors that would need to be added for another version of this SCD platform, like GPS. Within the main loop, a call to sensor read functions is added for each sensor needed. This data is then maintained internally, and made available for the interface functions (Bluetooth and Ethernet). The interface functions can request sensor data, and then package that data appropriately to send to the controller. One key aspect of the design here is data type – the sensor data needed to be stored in the same type of number (integer, float, double) that the state defines – be it X-Plane's implementation of that sensor, or the custom Bluetooth implementation.

A similar method was followed to implement the control surface functionality. Each control surface was given a “get” and “set” function. Using these, either interface (Bluetooth or Ethernet) could set the control surfaces to a specific degree offset, and get the last degree written to each surface. Once again, this sets this SCD platform up to expand for more control options. For example, in X-Plane, a user can control aircraft lights over the Ethernet interface. This SCD platform could expand to control lights on the r/c airplane by adding in a get/set code block for each light, if desired.

III. Platform Testing, Code Metrics & SCD Viability

Once the design was implemented, it was time to test on both the simulated platform and the real platform. The stages of SCD projects match well to stepping up to deployment. First, the platform can be tested in the “current state”, where the application is communicating directly with the simulated environment. This is how the application was developed. The system could then be transitioned to approach one, simply by changing a few IP addresses in configuration settings for X-Plane and the application. Then, X-Plane could be removed from the loop, and the application could be configured to receive sensor information from internal sensors instead of externally, from the simulator.

During deployment on the real platform, it became apparent that the data rates provided by X-Plane could not be matched by the Arduino. In addition, other attributes of the lab environment proved more difficult to replicate on the real platform than anticipated. For instance, basing the platform around an Ethernet connection in the lab was a simple problem, since there is so much network infrastructure available in typical home networks. However, replicating that infrastructure on the mobile and compact environment on an r/c airplane provided more obstacles than were anticipated.

Referring back to the metrics section, all success and expected issues were based around software design. Issues with hardware design were not considered as being a potential problem. It turns out that some of the assumptions made when starting this implementation were false.

One key assumption made was that it would be little to no work to replicate the lab environment on the real platform. However, the restraints associated with embedded environments proved greater than expected. For example, consider network connectivity. In a lab environment, in the typical home network, connecting devices is a trivial task. There is a wireless router capable of assigning IP addresses, and reporting connected devices. In addition, the communication paths it provides are high speed and reliable. Also in this lab is the simulation computer. A typical desktop computer today is powerful. They are mostly running faster than 2 GHz and with several GB of RAM. If the simulator is running on this computer, the representation of the environment is one in which information can update extremely rapidly.

Outside of the results from the platform test itself, success must be measured against the metrics established in Section III.V to determine how close the SCD implementation ended up being to its projected goals. As mentioned, the developer began work interfacing directly with the simulator, with no Arduino interface board in between. This meant they had to spend some time developing an X-Plane interface package for their Android project. The intent, then, is that this same interface can be used for the real platform, as documented in the description of approach one. No rework should be required to get the Arduino in the loop, other than changing IP addresses (see Figure 6). However, as mentioned above, the simulator provided an environment that was assumed to be realistic and in fact operated better than a realistic platform. Thus, some rework will be necessary in the future on the autopilot platform, to correspond to the changes made to simulator configuration. Data will refresh less frequently, to match the real world.

Please see Appendix B for all SLOC counts of this application. Note that all SLOC metrics were produced with the open-source tool Code Analyzer (Code Analyzer, 2012). Although there will need to be some rework, due to inaccuracies and unrealistic elements that were found in the simulator's flight model, the software part of the project can be deemed a success, based on the pre-defined criteria. Most of the rework and redesign will need to be done in the hardware design, by reconfiguring a network in the real environment to closer match the simulated one, or having the simulated environment degraded to a level of functionality similar to the real platform.

In light of all of these issues, approach two is likely the best candidate for the r/c application. Since it was designed custom to the application, it meets the real platform requirements, as it is defined by the real platform. Hiding the simulator behind this custom interface removes the performance levels of the computer from driving development in any direction, since it requires that the SCD system always be in the loop during application development. It also enables more simulators to be added, in case one has certain advantages during development not offered by another. However, in this application, the Android developer did not have the time to implement it in the real-time control framework they were using. Their system was built around the X-Plane interface, so they planned on transitioning to the real platform using a UDP socket connection, not Bluetooth.

Overall, it is apparent that the viability of an SCD platform is extremely dependent on the quality and usability of the selected simulator(s). In addition, it must be very clear upon project start-up that the hardware and software of the lab environment match as closely as possible to the real environment. If possible, start hardware in the loop testing immediately. For issues to be attributable to the autopilot controller design, or the SCD platform, it has to be very apparent that the issues were not related to the flight model used throughout development.

VI. AREAS FOR FUTURE DEVELOPMENT

I. PROJECT SPECIFIC

The first - and possibly most prominent - area for growth lies in this particular implementation of approach two, the custom interface. This is where the most challenges arose during implementation, and where the most design and schedule tradeoffs were made. However, as mentioned, it is likely the most plausible option for enabling rapid transition from development to flight.

For an example of the tradeoffs made, due to the tight schedule for implementation and testing, the Amarino library was used (Kaufmann, 2010). Though this allowed rapid prototyping and had extensive documentation and user support, it is not the best option for large datasets, or faster data exchange rates. It is limited to exchanging and processing strings only (null characters cause end of transmissions), and uses Arduino serial buffers for transmission, which are limited to only 64 or 128 bytes, depending on the particular board (Arduino, 2012).

However, the Bluetooth connection protocol is capable of much more. Amarino uses the EasyBluetooth library. If this was imported directly into the android project (creating a Bluetooth socket object), the full Bluetooth packet could be exploited, as could full transmit and receive rates. This would still have the buffer size limitations on the Arduino, but at least raw binary data (non-strings) could be transmitted, and at a faster rate than currently is the limit.

Another additional improvement would be making the Arduino software more configurable. One benefit of doing this would be the ability to add in another simulator, or multiple simulators. As discussed before, the approach one was architected entirely around the X-Plane interface. This led to shaping the structure of all of the Arduino code. It made the Arduino software architecture very X-Plane specific, and less general. The next step would be to make the simulator interface portion more configurable, so the simulator being used could be easily swapped in and out, with little rework. This would make for more universally viable development platform, less tied to one specific simulator.

If the project were to stay with using only the X-Plane interface, there is a lot of functionality there that was not exploited or reproduced. For instance, the data ref interface, which was explained earlier, implements a great deal of airplane functionality, none of which this SCD platform replicated. If more of the X-Plane interface was offered in approach one, it could be applicable to more projects that currently use the data ref portion of the X-Plane interface for their testing. The lesson learned here is that more of an interface the SCD platform offers, the more potential users that SCD platform will have. It exposes the SCD platform to a larger potential customer base.

Another area for future development lies in the basic architecture of the program running on the Arduino. The standard method for most Arduino programs was used for this project: setup followed by a single infinite looping function. However, code in this formation becomes difficult to track and maintain quickly as complexity grows. In addition, every added line of code means that many added microcontroller instructions, which increases the time it takes to run through the loop. Though one of the benefits of using the Arduino is its simplicity and quick implementation capabilities, a drawback associated therein is the lack of real-time scheduling features.

In light of that, an option for expansion into increased and higher speed functionality for the Arduino is the use of the “Concurrency” software (Jacobsen). Concurrency is a multi-threading platform for Arduino, allowing “simultaneous” operation of multiple threads of execution. This would be useful as the datasets included from X-Plane increase – if the developer required more information, the time to parse through and act on each dataset packet would inherently increase. If this communication message parsing could be done in a thread parallel to the one controlling servos, for example, a more regular schedule for the loop could be achieved, and the developers could be guaranteed a response rate from the Arduino. That is, the execution frequency of the loop could be more precisely controlled and regulated. This would be a big step in the direction of making the Arduino more like X-Plane, which does allow the user to specify UDP dataset frequency.

II. General Applications

As mentioned in section III.IV, there are several possible applications of SCD, beyond the specific exercise that was carried out for this report. Automotive, aerospace, and other general hobbyist projects were all mentioned. Now that a proposed process for implementing the approaches of SCD has been followed, and a sample project implemented, the possibility of applying it to these applications can be revisited.

In response to the lessons discussed in section V.III, the application to these industries is still possible, but the potential issues are more apparent. As mentioned, the key issue is the accuracy and realism of the simulator. The model of the environment needs to be extremely accurate, as does the rates at which the simulated data updates. In the case of the application outlined in this thesis, the simulator actually presented a model of the world that was beyond realistic – it presented a model that behaved at a higher performance level than any traditional real hardware could perform.

The main applications that SCD will benefit are those that do not currently have any defined or standardized interface. For instance, consider this r/c application. There is no “industry standard” for how an r/c airplane needs to communicate from controller to control surfaces, or receive information from sensors. Both approaches of SCD provide that standard. As long as the application developer begins their project without an ICD to develop to, SCD would be a good option for defining and providing that definition.

VII. CONCLUSION

I. CONTRIBUTIONS

This thesis focused on a couple key ideas. Simulator centered design is the umbrella they fall under, but that boils down to two key concepts. These are the main contributions of the thesis. These concepts are the two approaches to simulator centered design: a simulator-based interface, and a custom interface. The framework and prototypes for these approaches were designed, developed, and tested. Advantages and disadvantages to each were outlined before implementation, and difficulties that arose during implementation were outlined and documented, with mitigation plans suggested as well.

Someone looking to implement SCD on their own project will be best served by using one of these two approaches, weighing the decision for which is better for their particular project. This thesis outlined the items that need to be considered in a decision like that.

II. IMPACT

Beyond the specific contribution of the two SCD approaches, the larger and more general impact of SCD is in standardization. If no interface for a system has yet been defined, SCD is likely a good design approach. SCD defines that interface, providing a standard for both developers and test engineers to work towards. It streamlines the transition from development to test, and from test to product deployment.

The issue here is that this implies that any project, application or industry looking to implement SCD must first look at their current interfaces. If there is already an existing standard that their application will be working to, for deployment in the real environment, then SCD will likely have fewer benefits for them. SCD is good when it can both define the interface, and implement part of it for the user. If the interface is already defined, and would have been implemented anyways, it loses some of its value. This is likely the case in some of the automotive or aerospace applications, where the CAN bus, and all of the Avionics bus definitions likely provide most if not all of the interface definitions.

III. Final Comments

The exploration of simulator centered design has panned out well in the small-scale application of an r/c airplane autopilot controller, aside from a few inaccurate assumptions. Though there were obstacles uncovered with the model of data provided by the simulator, with more time, these issues could be worked out. Other than that, the interface provided by the X-Plane flight simulator was very usable and easy to implement, and it was documented well enough that its functionality could be replicated by the custom-built interface software and hardware on the Arduino platform.

Another key feature of this project was that the scale of the work was controlled and defined so that the time spent developing the interface(s) represented a significant portion of the development effort. If the control software became more complex, there might be a point at which the time spent developing the simulator interface becomes negligible compared to the control algorithm software. At this point, the benefits of simulator centered design start to diminish. This is because SCD, especially approach two, the custom interface, does take some time to implement, especially if it is made specific for the project. Therefore, when developing a schedule for software development for the project, the time required to develop an interface must be taken into account.

However, one thing to consider about SCD is its viability as a re-usable platform. After the SCD system is developed once, for any particular airplane, that platform is much easier to re-use for other projects. As the number of projects using one SCD platform increases, Return on

Investment (ROI) from the time spent implementing that SCD platform increases correspondingly. This is because the time spent developing the SCD falls under the category of non-recurring engineering (NRE) – it is a one-time expense. As more projects use the platform, that expense is (indirectly) divided across all of those projects, decreasing the necessary investment from each project. This is true of any NRE project, like the design of a processor core, or an operating system. The more re-use that comes out of that project, the higher the ROI on the time and effort spent on developing it up front.

In addition, one key aspect of the SCD problem was that the benefits came from creating a defined interface. In the application of an r/c airplane, there are no real standards for autopilot control systems. All airplanes use electronic servos to actuate surfaces, but there is not an existing "r/c airplane interface" upon which to build a standard autopilot system. That is where this system gains its value: it defines a re-usable interface specification. Many more r/c autopilot platforms could now be built around this SCD system, because of the work and documentation that went into the interface definition, whether the one based on the simulator, or the custom interface.

Truly, the lessons learned were all about challenges associated with interfaces. This SCD project taught that there is a lot more to consider when defining interfaces than just data that needs to be transferred. Successfully performing SCD on a project ensures the developer team fully defines the single interface they want to re-use. This involves defining physical transport medium, and ensuring they are the same in the lab and real environment. It involves understanding data bandwidth of that medium, and the rates at which data is sent. It involves an exercise ensuring every aspect of the interface in the lab can be totally matched in the real environment; otherwise the issues that came up during this thesis would still be a problem.

Overall, this endeavor was a successful one. The time spent implementing the two approaches for the r/c project taught many lessons, and an SCD approach to a project like this proved to be a viable one. It enabled a speedy and painless transition from development to lab test to real deployment, and allowed the project's developer to use a simulator, when otherwise they might have been inclined not to, or even not to take on the project entirely. SCD has many potential applications in larger markets and industries, and it will be exciting to see where ideas like this will go next.

BIBLIOGRAPHY

Beagle Board Quick Start Guide. (2012). Retrieved from Beagle Board: http://beagleboard.org/static/beaglebone/latest/README.htm

Bluetooth - How it Works. (2012). Retrieved from Bluetooth: http://www.bluetooth.com/Pages/Basics.aspx

Code Analyzer. (2012). Retrieved from Source Forge Project Hosting: http://sourceforge.net/projects/codeanalyze-gpl/

DataRefs. (2012, March 3). Retrieved from X-Plane Data Refs: http://www.xsquawkbox.net/xpsdk/docs/DataRefs.html

DIY Drones - Getting Started. (2012). Retrieved from DIY Drones: http://diydrones.com/profiles/blogs/a-newbies-guide-to-uavs

DIYDrones Failsafe Mux. (2012). Retrieved from DIYDrones: http://store.diydrones.com/product_p/br-0001-10.htm

Fritzing - About. (2012). Retrieved from Fritzing: http://fritzing.org/about/

Google. (2012). Retrieved from Google Wallet: http://www.google.com/wallet/

MSP430 LaunchPad Value Line Development kit. (2012). Retrieved from Texas Instruments, Inc.: http://www.ti.com/tool/msp-exp430g2

Raspberry Pi Quick Start Guide. (2012). Retrieved from Raspberry Pi: http://www.raspberrypi.org/quick-start-guide

Arduino. (2012, October 26). *Arduino - Getting Started*. Retrieved from Arduino: http://arduino.cc

Bullock, D., Johnson, B., Wells, R. B., Kyte, M., & Li, Z. (n.d.). *Hardware in the Loop Simulation.*

Cataldo, M., de Souza, C. R., Bentolila, D. L., Miranda, T. C., & Nambiar, S. (2010). *The Impact of Interface Complexity on Failures: An Empirical Analysis and Implications for Tool Design.* Pittsburgh: Carnegie Mellon University.

Google, Inc. (2012, October 26). *Android Developers Home Page*. Retrieved from Android Developers: developer.android.com

Jacobsen, C. (n.d.). *Multithreading for Arduino*. Retrieved from http://concurrency.cc/.

Kaufmann, B. (2010). *Design and Implementation of a Toolkit for the Rapid Prototyping of Mobile Ubiquitous Computing.* University of Klagenfurt.

Meyer, A. (2012). *X-Plane 10 Manual.* X-Plane.

Moller, B. (2012). *HLA Tutorial.* Pitch Technologies.

SISO. (2012). *SISO Overview*. Retrieved from Simulator Interoperability Standards Organization: http://www.sisostds.org/AboutSISO/Overview.aspx

Yang, Y., He, M., Li, M., Wang, Q., & Boehm, B. (2008). Phase Distribution of Software Development Effort. (pp. 61-69). Kaiserslautern, Germany: ESEM.

Yoo, Y.-H., Ahmed, M., Kirchner, F., & Roemmermann, M. (2009). *A Simulation-Nased Design of Extraterrestrial Six-Legged Robot System.* Bremen, Germany: DFKI Robotics Lab.

APPENDICES

I. Appendix A – Application I/O Requirements

Table 2: Sensor Inputs to the Controller

Status Name	**Units**
1. Latitude	degrees
2. Longitude	degrees
3. Altitude	Feet (mean sea level)
4. Pitch	degrees
5. Roll	degrees
6. Heading	Degrees, magnetic
7. Heading	Degrees, true
8. Magnetic variation	degrees
9. Engine Speed	RPM
10. Ground Speed	knots
11. Control Surface positions	
a. Elevator	Degrees (offset from 0)
b. Aileron	Degrees (offset from 0)
c. Rudder	Degrees (offset from 0)
d. Throttle	Degrees (offset from 0)
12. Pressure	psi
13. *Optional* Temperature	Degrees Fahrenheit
14. Angular Velocities	
a. P	radians/second

b. Q	radians / second
c. R	radians / second

Table 3: Control Outputs from the Controller

Control Name	**Units**
1. Elevator deflection	(+/- Deflection, in percent, where +1 pushes the elevator down, increasing altitude. 0 is neutral.)
2. Aileron deflection	(+/- Deflection, in percent, where +1 pushes the elevator down, increasing altitude. 0 is neutral.)
3. Rudder deflection	(+/- Deflection, in percent, where +1 pushes the elevator down, increasing altitude. 0 is neutral.)
4. Throttle	Percent (0 to 100%)

II. Appendix B – SCD Platform SLOC Counts

Table 4: SLOC Counts

Code Package	**Code Size after Development (SLOC)**
Android X-Plane Interface (approach one)	892
Android Bluetooth Interface (approach two)	211
Arduino Project	943
Arduino Ethernet Interface	145
Arduino Bluetooth Interface	315
Arduino Aircraft Interface (sensors and control surfaces)	208