ENVE: AN ENVIRONMENT FOR THE EMULATION OF UNMANNED AERIAL VEHICLES

BY

ANDREW VINCENT LOUIS

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Marco Caccamo
Abstract

Since the dawn of modern computation, computing devices have been steadily increasing in both performance and efficiency while decreasing in cost. This trend has produced incredibly powerful computers in tiny form factors at prices that average consumers can afford. Installing such computational resources on a hobby-grade aircraft outfitted with microelectromechanical sensors results in a compelling UAV platform for both research and education in the field of cyber-physical systems. While such UAVs have great potential, they also pose significant danger to anyone in the vicinity of the UAV operation.

With the extreme risks involved in flight testing, some method of controlled testing is needed prior to live testing. Providing such a method in a lab context would benefit both research and education applications of UAV systems. To that end, this work is focused on the development and use of an emulation environment for UAVs. The environment is designed with flexibility and versatility in mind. As such, it supports a wide range of hardware and software control platforms and numerous fixed-wing and rotary-wing aircraft. By requiring only a PC, some cables, and the custom low-level adapter, it is ideal for use in a lab environment. Early evaluation has shown that EnvE provides invaluable testing opportunities for research and an incredible platform for cyber-physical systems education.
Acknowledgments

The work contained within this thesis could not have been done without the help of a few individuals. I must begin by acknowledging my adviser, Professor Marco Caccamo. He has provided me with three years of opportunity and education that cannot be found in a classroom. Without Professor Caccamo taking a chance by giving me the opportunity to work in his lab three years ago, I would not have had the experience of working on this and many other projects. I must express my gratitude to him for his constant support, both financially and intellectually. I must also acknowledge Professor Caccamo’s pursuit of excellence in education and research, which allowed me to work to improve both in my thesis.

I must also acknowledge the contributions of Stanley Bak. He laid the early ground work that made EnvE possible. His early work included modifying FS One to support an additional custom DLL and a development project for such a DLL. This eased the process of creating a DLL to support hardware-in-the-loop emulation of a feedback control loop.

The material presented in this thesis is based upon work supported by the National Science Foundation (NSF) under grant number CNS-1646383. Any opinions, findings, conclusions, or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of the NSF.
To my parents for instilling within me the insatiable desire to learn.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Abbreviations</td>
<td>vii</td>
</tr>
<tr>
<td>Chapter 1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>2.1 UAV Risks</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Research</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Education</td>
<td>4</td>
</tr>
<tr>
<td>Chapter 3 EnvE</td>
<td>6</td>
</tr>
<tr>
<td>3.1 The Control Loop</td>
<td>6</td>
</tr>
<tr>
<td>3.2 FS One Modifications</td>
<td>6</td>
</tr>
<tr>
<td>3.3 getDllFuncs</td>
<td>7</td>
</tr>
<tr>
<td>3.4 shouldOverrideOutputs</td>
<td>8</td>
</tr>
<tr>
<td>3.5 shouldResetState</td>
<td>8</td>
</tr>
<tr>
<td>3.6 Closing the Loop with Hardware</td>
<td>9</td>
</tr>
<tr>
<td>Chapter 4 Emulation Adapter</td>
<td>10</td>
</tr>
<tr>
<td>4.1 The Revised Control Loop</td>
<td>10</td>
</tr>
<tr>
<td>4.2 Implementation</td>
<td>10</td>
</tr>
<tr>
<td>4.2.1 Setup</td>
<td>11</td>
</tr>
<tr>
<td>4.2.2 Loop</td>
<td>11</td>
</tr>
<tr>
<td>Chapter 5 Development Environment for EnvE</td>
<td>13</td>
</tr>
<tr>
<td>5.1 EnvE Simulator</td>
<td>13</td>
</tr>
<tr>
<td>5.1.1 System Requirements</td>
<td>13</td>
</tr>
<tr>
<td>5.1.2 Install Microsoft Windows 7</td>
<td>14</td>
</tr>
<tr>
<td>5.1.3 Install TortoiseSVN</td>
<td>14</td>
</tr>
<tr>
<td>5.1.4 Install Visual Studio</td>
<td>15</td>
</tr>
</tbody>
</table>
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLL</td>
<td>Dynamically Linked Library</td>
</tr>
<tr>
<td>I²C</td>
<td>Inter-Integrated Circuit</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>RC</td>
<td>Radio Control</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicle</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
</tbody>
</table>
In recent years, computers have continuously gotten smaller and more efficient while increasing in performance. Interest in unmanned aerial vehicles has grown with these changes in computers. These developments have driven the Real-Time and Embedded Systems Laboratory at the University of Illinois at Urbana-Champaign to pursue UAVs as both research and educational platforms.

This thesis focuses on the development of EnvE, a hardware-in-the-loop emulation environment for unmanned aerial vehicles. The environment focuses on providing reliable, flexible, and accurate flight emulation in a manner that requires minimal changes to the flight controller that will be placed in the loop. The goals of this work in the realm of research are to aid the development and testing of flight control software and other related components of a UAV platform. The goal in education is to provide an engaging and hands-on environment in which students can experiment with and learn about automatic flight control. Feedback from students in embedded and cyber-physical systems courses often includes requests for more intriguing labs. It is hoped that EnvE will address this.

The body of this work is divided into a number of different sections. Chapter 2 motivates the need for EnvE in both research and education. The inner workings of EnvE are discussed in Chapter 3. Chapter 4 covers a custom emulation adapter that provides a clean interface for use by low-level flight controllers. The process of setting up a development environment for EnvE is outlined in detail in Chapter 5. Evaluation based on feedback from a number of users is discussed in Chapter 6. Finally, Chapter 7 contains concluding remarks.

After the conclusion of the main body, some appendices can be found. In Appendix A, a 4-bit character codec used in EnvE is described. Code for encoding and decoding strings is also provided. Appendix B contains code
listings for the EnvE adapter firmware. The code for the simulator portion of EnvE is listed in Appendix C. Finally, a thorough and detailed user manual is provided in Appendix D.
Chapter 2
Motivation

Unmanned aerial vehicles are an excellent platform for both research and education in the field of cyber-physical systems. They provide an interesting combination of opportunities and constraints. UAVs provide opportunities in interdisciplinary research, live experimentation and demonstration, application of classroom knowledge, and cyber-physical interactions. Additionally, there are many constraints, such as weight, energy and power, unstable dynamics, and computational resources.

The combinations of opportunities and constraints provided by UAVs make them extremely valuable parts of many projects. The most valuable opportunity afforded by UAVs is their ability to be piloted fully autonomously using relatively inexpensive equipment. The development of an autopilot system is both an inspiring educational project and useful research tool.

2.1 UAV Risks

As is often the case, something of such value comes with many great risks. There are obvious risks to the aircraft system itself. These are monetary and temporal in nature. Improper operation of the aircraft can result in damage to the aircraft, which can cost money for parts and time for rebuilding. There are also risks to the operators and bystanders. Mechanical, electrical, and computational errors can result in bodily harm or even death. Many more issues can arise if communication with the aircraft is lost. These issues may include risks as extreme as the aircraft flying aimlessly into restricted airspace, such as an airport. While some of these issues are fairly minor and will only result in temporary setbacks and budgeting issues, others have the potential to cause irreparable harm and
damage to people, property, and the project itself. These risks must be mitigated as much as possible.

2.2 Research

Development of a UAV system typically occurs in a laboratory. Flight tests, however, cannot occur within a laboratory. Testing within the development environment accelerates the development process by providing quick performance feedback. It is the goal of any project to perform as much testing as possible closest to the development environment for this purpose. This is the primary goal and purpose of EnvE: to allow thorough flight testing within a lab environment. Thorough testing within the laboratory helps developers eliminate bugs and tune settings without risking anything. A hardware-in-the-loop emulation environment allows testing of all systems except for the physical sensors and actuators installed on the aircraft. This can greatly accelerate the development process and make flight tests easier.

A hardware-in-the-loop emulation environment provides another major benefit: real-time testing of the computational platform used in flight. Researchers can verify that the computational tasks do not overutilize the resources because identical or nearly identical (minor modifications may be necessary) tasks and resources are used in both flight and emulation. This is extremely important, especially on platforms such as UAVs in which missed deadlines or processing delays can cause performance issues.

2.3 Education

To maintain the highest quality education at universities, it is important for researchers to evaluate the potential educational benefits of research projects. A hardware-in-the-loop emulation environment for UAVs has great educational value. A challenge in cyber-physical systems education is providing a hands-on development experience to students. This stems from the inherent physical aspect of these systems. Many physical systems have some characteristic that makes them unsuitable for classroom use, whether it be size, operational hazards, cost of purchase and maintenance, issues
providing a development environment, or the system is unable to engage students. EnvE looks to provide a platform for classroom use that is free of such issues. By emulating the aircraft, the purchase cost and operational danger of the aircraft are eliminated. EnvE provides a simple interface that can be used by many computation platforms, so providing a development environment is simple. Finally, students are truly engaged by the realistic emulation of the flight of an aircraft. At the time of development, interest in UAVs has boomed and encouraged the formation of a hobbyist market.
With the clear motivation for a hardware-in-the-loop emulation environment, the explanation of the mechanisms of the environment can proceed. In Section 3.1 the feedback control loop is described. Section 3.2 discusses modifications made to FS One to provide a control loop interface. The three functions of the environment are discussed in Sections 3.3, 3.4, and 3.5. Finally, some closing remarks about hardware are given in Section 3.6.

3.1 The Control Loop

A typical feedback control loop, as shown in Figure 3.1a, consists of four components: sensors, actuators, the controller, and the process. In order to emulate the feedback control loop, the actuators, process, and sensors are replaced by a simulator capable of real-time operation. In the case of EnvE, the FS One Precision RC Flight Simulator is used. The emulated feedback control loop is depicted in Figure 3.1b. With the feedback control loop defined, the challenge of providing an interface between the controller and simulator remains.

3.2 FS One Modifications

FS One is well-suited for use in emulation because it is a realistic simulator with high-fidelity physics and aircraft models. In addition, Professor Caccamo has secured access to the source code of the simulator in order to make modifications for research and education purposes. The most significant modification made has been the ability to provide a dynamically linked library with functions that the simulator calls at specific points.
(a) A typical feedback control loop. (b) The emulated feedback control loop.

Figure 3.1: Comparison of typical and emulated feedback control loops.

during its execution. The function \texttt{DllFuncs * getDllFuncs(int version);} is called upon loading the DLL. The functions \texttt{int shouldOverrideOutputs(float * ch1, float * ch2, float * ch3, float * ch4, float * ch5, float * ch6);} and \texttt{int shouldResetState(TFmResetState2 * currentState, double Udot, double Vdot, double Wdot, double Pdot, double Qdot, double Rdot);} are called before and actuation and after a new state is computed respectively. These functions are described in detail in the following sections and the code can be found in Appendix C.

3.3 getDllFuncs

\texttt{DllFuncs * getDllFuncs(int version);}  

The basic purpose of this function is to provide FS One with the addresses of the two other callback functions. It does this by placing the addresses in a structure and returning a pointer to that structure to the simulator. Before the function returns, it has an opportunity to perform some operations while the simulator is loading for a flight. The function begins by loading and processing a configuration file so that users can configure certain aspects of the environment. If users have enabled logging in the configuration file, a log file is created with the current timestamp as its name. Finally, a serial connection with asynchronous read capabilities is
established. The ability to read asynchronously allows the
**shouldOverrideOutputs** function to operate without blocking execution to
perform a read.

### 3.4 shouldOverrideOutputs

```c
int shouldOverrideOutputs(float * ch1, float * ch2, float * ch3, float * ch4, float * ch5, float * ch6);
```

This function is called just before the simulator performs actuation. It
provides the capability to override the control outputs from the manual
flight controls with those provided via the six channel arguments. It is
important to note that these control outputs are expected to be a floating
point value in the interval \([-1, 1]\). Given a return value of 0, the simulator
ignores the control outputs from this function. Given a non-zero return
value, the simulator overrides the manual controls using the controls
provided by this function. The control outputs and return value used are
the latest values received from the receive callback of the asynchronous
serial connection. Each time data is received, the callback function looks
for a control data sample. If one is found, it is parsed from string format to
binary format. If the parsing is successful, the parsed values are made
available to future calls to **shouldOverrideOutputs**.

### 3.5 shouldResetState

```c
int shouldResetState(_TFmResetState2 * currentState, double U_dot, double V_dot, double W_dot, double P_dot, double Q_dot, double R_dot);
```

This function is called immediately after the simulator computes a new
state for the aircraft. The _TFmResetState2 structure contains the
attitude, position, velocity, and rotation rates of the aircraft. Additionally,
the acceleration and angular acceleration of the aircraft are provided as
arguments to the function. The return value is interpreted as a bit field to
specify which portions of the state to reset. The details of the bit field are
given in Table 3.1.
<table>
<thead>
<tr>
<th>State Component</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotation Rate $(p, q, r)$</td>
<td>0x1</td>
</tr>
<tr>
<td>Attitude $(\phi, \theta, \psi)$</td>
<td>0x2</td>
</tr>
<tr>
<td>Velocity $(u, v, w)$</td>
<td>0x4</td>
</tr>
<tr>
<td>Position $(p_n, p_e, p_d)$</td>
<td>0x8</td>
</tr>
</tbody>
</table>

Table 3.1: The state components and their associated bits in the bit field for state reset.

The function begins by setting the state components that the user has configured as fixed. It proceeds by checking the current time to see if the user configured period has elapsed. If it has, the configured mapping occurs and all state information is converted to metric. The state data is then converted to string format for output. If logging is enabled, it is done at this time. The output string is then optionally encoded using the 4-bit codec described in Appendix A. Finally, the data is written to the serial interface.

3.6 Closing the Loop with Hardware

With the simulator ready to close the feedback control loop, steps must be taken to allow the flight controller hardware to complete the loop. Since RS-232 is a common and simple interface, use of a USB to RS-232 cable allows many flight control computers to interface with the simulator. In addition to an RS-232 interface, a custom emulation adapter can be used. This adapter is described in Chapter 4. For details on using the hardware-in-the-loop interface, see Appendix D.
Chapter 4

Emulation Adapter

While the serial interface to the simulator is very flexible, it is not the easiest interface to program, especially on low-level hardware controllers. Issues can arise from the use of strings to transmit data. String processing is usually done with the help of software libraries. Not all devices have strong support for string processing from libraries. In order to improve the flexibility of the interface, a low-level emulation adapter was developed. The adapter provides a simple way of using a different interface that is not dependent on the flight controller’s ability to process strings. In Section 4.1, the revision to the emulated feedback control loop is discussed. The implementation and functionality of the adapter are covered in Section 4.2.

4.1 The Revised Control Loop

The EnvE adapter positions itself between the flight controller and the simulator, as depicted in Figure 4.1b. This positioning allows it to perform some of the work needed to interface with the simulator. Instead of using the serial interface directly, the flight controller interfaces with the adapter using I²C for state data acquisition and PWM for actuation. Usage details are given in Appendix D.

4.2 Implementation

The EnvE adapter is a firmware developed for and deployed on the APM 2.6 platform. The firmware is implemented using a snapshot of the ArduPilot software libraries. It is implemented as any other ArduPilot firmware, consisting of a setup and a loop function. The significant difference is that the adapter does not perform any control algorithms. It
The emulated feedback control loop simply acts as an interface to the simulator. The mechanisms of the two functions are explained in the following sections.

### 4.2.1 Setup

This function is called once when the device is powered on or reset. Its task is extremely simple. It begins by initializing some data structures for use by the rest of the firmware. It then sets the time at which actuation data should next be sent to the simulator. Finally, the device is configured as an I²C slave at address 2. In addition, a callback function is registered for I²C read request events. This request event function simply writes the latest state data in binary format to the I²C master.

### 4.2.2 Loop

This function is continuously called in an infinite loop after setup is performed. The first task is to read the PWM input channels and convert them from integer microseconds to floating point values in the interval $[-1, 1]$. Next, the time is checked to see if the period between outputs has elapsed. If it has, the actuation data is sent to the simulator using the
serial connection over USB. The adapter then enters a loop of reading and decoding data from the serial connection until there is no more data, the decoded buffer is filled, or the end of a data sample has been found. Upon finding the end of a data sample, the string is then parsed into binary form and the decoded buffer is cleared. Upon successful parsing, the current sample is updated, making it ready for use in future I²C request events.
Chapter 5

Development Environment for EnvE

It is rare for a project to ever be complete while in active use. There will almost always be new bug reports and feature requests from users. Because of this, an active development environment is necessary. Section 5.1 covers the process of setting up a PC for development of the simulator portion of EnvE. Section 5.2 covers the process of setting up a machine for development of the adapter portion of EnvE.

5.1 EnvE Simulator

This section covers, in detail, the process of setting up a new machine for development of the simulator portion of EnvE. The steps include identifying a system that meets requirements (Section 5.1.1), installing Windows 7 (Section 5.1.2), TortoiseSVN (Section 5.1.3), Visual Studio (Section 5.1.4), Boost (Section 5.1.5), and finally FS One (Section 5.1.6).

5.1.1 System Requirements

The main component of EnvE is the FS One Precision RC Flight Simulator [3]. The extensions and modifications made by EnvE have little to no impact on performance. Thus, the system requirements for EnvE are roughly the requirements for FS One, which are provided in Table 5.1. Almost all modern desktop machines greatly exceed the recommended specification. Some effort should be put into acquiring a display for use with EnvE. Large format and high resolution displays provide users with a better view while they develop software on another machine.
<table>
<thead>
<tr>
<th>Component</th>
<th>Minimum</th>
<th>Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Windows XP/Vista/7/8/8.1</td>
<td></td>
</tr>
<tr>
<td>Processor</td>
<td>1.8 GHz</td>
<td>2.8 GHz or better</td>
</tr>
<tr>
<td>Memory</td>
<td>512 MB RAM</td>
<td>1 GB RAM or more for Windows XP; 2 GB RAM or more</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for Windows Vista/7/8/8.1</td>
</tr>
<tr>
<td>Hard Disk Space</td>
<td>4 GB</td>
<td></td>
</tr>
<tr>
<td>Video/ Graphics</td>
<td>64 MB RAM NVIDIA GeForce4 Ti, GeForce5200</td>
<td>128 MB RAM or more NVIDIA Geforce6600 or above,</td>
</tr>
<tr>
<td></td>
<td>or better ATI Radeon 9200 or better</td>
<td>7000-series, or better</td>
</tr>
</tbody>
</table>

Table 5.1: System requirements of FS One V2 from http://www.fsone.com/products/fs-one-v2

5.1.2 Install Microsoft Windows 7

There are no special instructions for installing Microsoft Windows 7. Be sure to allocate enough disk space for Windows, FS One, and Visual Studio. It is best to install Windows 7 and fully update it before proceeding to the next step. Windows 7 is currently the recommended OS. Earlier versions of Windows no longer have any support from Microsoft. Later versions of Windows have not been tested and are unlikely to work with FS One.

5.1.3 Install TortoiseSVN

In order to interact with the Subversion version control system used by the EnvE project, TortoiseSVN [6] must be installed. Begin by downloading the latest version for the correct architecture from https://tortoisesvn.net/downloads.html. Continue by running the downloaded installer. The process of using the TortoiseSVN installer is depicted in Figure 5.1. In the initial window, press the “Next >” button to begin the process. Accept the GNU General Public License by clicking the “Next >” button. In the following window, select all components for installation then proceed by clicking the “Next >” button. Trigger the installation by clicking the “Install” button. Be sure to allow changes to
the system by agreeing to any security prompts. Finally, conclude the installation process by clicking the “Finish” button.

5.1.4 Install Visual Studio

Microsoft Visual Studio 2008 is used as the integrated development environment to build the DLL that is added to FS One to form EnvE. The Real Time Systems Laboratory has access to a licensed version of this software. Begin by inserting the media into your machine. Run the installer from the disc. The installer will appear as the images in Figure 5.2. In the initial window, click “Install Visual Studio 2008”. In the next window, be sure to deselect the checkbox so that information is not sent to Microsoft Corporation. Proceed by clicking the “Next >” button. In the following window, accept the license terms and proceed by clicking the “Next >” button. Select the “Full” option in the following window. Begin the installation by clicking the “Install” button. In the following window, click the “Finish” button after a successful installation. Complete the process by clicking the “Exit” button.

5.1.5 Install Boost

The FS One extension of EnvE relies on portions of the Boost C++ Libraries [2]. To communicate with the flight controller without blocking execution of the flight simulator, the EnvE extension uses the asynchronous serial library included in Boost. To install Boost, begin by downloading the current release from http://www.boost.org/users/download. As of writing this document, the current version is 1.63.0. Proceed by unzipping the archive to a location accessible by the development user, such as C:\local\boost_1_63_0 for version 1.63.0. This example location and version will be used in the remaining instructions. Simply replace the location and version number when necessary. Now that the files are in place, the libraries must be bootstrapped and built. These two steps can be completed using the Visual Studio 2008 Command Prompt, which can be found at Start > All Programs > Microsoft Visual Studio 2008 > Visual Studio Tools > Visual Studio 2008 Command Prompt. With the Command
Prompt open, begin by changing to the correct directory by running `cd C:\local\boost_1_63_0`. Proceed by bootstrapping the build process by running `bootstrap.bat`. Build the libraries by issuing the `.\b2` command. This step will take some time to complete. Upon successful completion of the build process, the build program will display include and linker paths like those shown in Figure 5.3.

Now that Boost has been built and installed, the Visual Studio project must be configured to use the installed libraries. Upon completion, the builder provides two directories: one for includes and one for libraries. In the example case, the includes directory is `C:\local\boost_1_63_0` and the libraries directory is `C:\local\boost_1_63_0\stage\lib`. To configure the project, right click on the project in the solution explorer and click the “Properties” option. Place the includes directory in Configuration Properties > C/C++ > General > Additional Include Directories. Place the libraries directory in Configuration Properties > Linker > General > Additional Library Directories.

### 5.1.6 Install FS One

The process of installing FS One is a simple copy. On a 32-bit installation of Windows, the destination is `C:\Program Files`. On a 64-bit installation of Windows, the destination is `C:\Program Files (x86)`. To perform the installation, simply copy the FS One v2 folder to the correct destination folder for the system. Note that the custom DLL is placed in this FS One v2 folder. It is highly recommended that a shortcut to FS One be placed on the desktop.

### 5.2 EnvE Adapter

The development environment for the EnvE adapter consists of an installation of GNU/Linux with a number of packages installed. In Ubuntu, the necessary packages are arduino, gawk, and subversion. These packages are often available in other distributions, although they may be provided under a different name. It is important to note that the paths to the adapter device and arduino installation can be customized in `config.mk`. 

16
Building is done by running make. The adapter device is programmed by running make upload.
Figure 5.1: The images show the different UIs for each step of installation of TortoiseSVN. Version 1.9.5 is depicted. Order is left to right then top to bottom.
Figure 5.2: The different frames of the Microsoft Visual Studio 2008 installer. The order is from left to right then top to bottom. Note that a product key is normally displayed in the third window. It has been redacted in this image.
Figure 5.3: The final output of the Boost build process. Note that the includes and libraries directories are clearly listed.
Chapter 6

Evaluation

Development of EnvE began in late 2014, coinciding with the start of autopilot development efforts. This rather long development time has allowed for considerable evaluation and testing. The evaluation period revealed the need for new features, exposed a number of bugs, and confirmed the value of a hardware-in-the-loop emulation environment. The information gathered from research and education experiences is covered in Sections 6.1 and 6.2 respectively.

6.1 Research

EnvE has proven to be an invaluable research tool. It has greatly aided and simplified the autopilot development process. EnvE provided a method of testing and tuning different control schemes without risks. EnvE allowed the development process to be much more experimental, delaying the need for extreme caution until the time for live flight tests. Such live flight tests have not encountered issues that EnvE could have exposed. In addition, the process of tuning control by using EnvE results in control that performs well enough in the real world to only need minor tweaking. In addition to the reduced development risks, the speed of development was greatly increased. EnvE eliminated the need to travel to a safe area to perform flight tests in the early development process. Additionally, it allowed developers to work at any time of day during any weather conditions, which drastically increased productivity.

The laboratory environment containing EnvE matured during this development period. Features were added to improve the usability and increase the value of the environment. This lab environment is depicted in Figure 6.1. These additional features include a large format display and a
retired aircraft. The large format display improves productivity by allowing developers to work away from the simulation workstation while still in view of the display. This simple feature gives developers space, which is valuable since development is done on a machine other than the simulation workstation. The retired aircraft provides a set of servos connected to control surfaces, resulting in a visualization of actuation. This helps developers ensure that the controller is actuating correctly on real hardware in addition to in the simulator.

6.2 Education

To date, Professor Caccamo has used EnvE in his cyber-physical systems course twice. The students in these courses have provided valuable feedback to improve the user experience of EnvE. In order to ease the learning curve for using EnvE, the EnvE adapter is used in educational contexts. In addition to the APM running the adapter firmware, an APM is used as the flight controller. Figure 6.2 depicts the arrangement and wiring. During the first course, students revealed weaknesses in the adapter’s custom parsing code. Bugs within that code caused the firmware to fill its data buffers without ever emptying them. This caused the adapter to reach a point where it would no longer acquire new data samples. The solution was to modify the ArduPilot libraries to allow the use of AVR LibC’s implementation of `sscanf` to perform the parsing duties. In addition to this change, the simplification of the data reception process resulted in a highly reliable adapter firmware. During the second course, students did not report any issues with the emulation environment in any way.
Figure 6.1: An example setup for a laboratory.
Figure 6.2: An example setup for use of the EnvE adapter. A: The USB connection to the simulator PC. B: The APM running the EnvE adapter binary. C: PWM cables for actuation data between controller and adapter. D: An APM running a flight control binary. E: I²C connection for transmission of state data from adapter to controller.
Chapter 7

Conclusion

Over the duration of EnvE’s development, it has transformed from a rough proof of concept into a full-featured and reliable environment. The development of EnvE in conjunction with an autopilot provided a very rapid feedback loop. Bugs were eliminated very quickly while useful features were added regularly. The result is an incredibly useful tool to assist the development of flight controllers and other UAV related technologies, including ground based flight monitoring stations.

Almost all of the feature ideas that came up during development were added to EnvE. Unfortunately some late ideas did not make it into EnvE as of this work. It is hoped that through discussing them here, they will make it into EnvE in the future. To aid the development of fully engineered UAV systems, incorporation of a power model would increase the realism of the environment. Transforming the state data produced by the simulator into the coordinate systems used by sensors on the aircraft system would be valuable for testing the transformations performed on the aircraft. One example is the fact that GPS produces position information as latitude, longitude, and altitude. This is then converted into Universal Transverse Mercator (UTM) for use in the flight controller. The simulator produces position information that is already in UTM, causing the flight controller to bypass its transformation process. Modifications to the EnvE adapter to improve flexibility would be valuable. In its current state, the firmware code must be modified if some configuration changes. Ideally, some configuration decisions could be made during setup by reading signals from the numerous unused pins on the APM device. Although many of these features would be nice additions, EnvE is not lacking because of their absence.

It is hoped that EnvE will prove to be valuable for years to come. The great amount of feedback has shown that EnvE can improve the educational process by allowing students to tackle projects that were once
outside their reach. It is through new opportunities such as this that young minds are set free to learn and pursue research interests that have the potential to transform this world.
References

    common-apm25-and-26-overview.html


    source. https://github.com/ArduPilot/ardupilot


Appendix A

Codec

While serial interfaces provide a simple and platform agnostic communication protocol, they do not provide very much data throughput. In order to double the data throughput, a 4-bit encoding scheme was developed. The characters needed to communicate each data sample are contained within a nearly contiguous block in ASCII encoding. The ten arabic numbers, the comma, the period, and the minus sign are the only characters needed. Those thirteen characters are contained with a fourteen character segment of ASCII. By adding two more characters to the start of the block, a simple 4-bit encoding can be produced from the sixteen characters. Table A.1 shows the relation between characters, ASCII codes, and 4-bit codes. By encoding immediately before writing and decoding immediately after reading, each 8-bit serial character can communicate two data characters instead of one.

A.1 Usage

The following two sections explain how to encode and decode strings using the codec code provided in Section A.2. It is important to note that if characters outside the supported range are used, the decoded string will not be the same as the original string before encoding.

A.1.1 Encoding

The `encode` function takes a single `char *` as an argument. The argument is expected to be a pointer to a C-string. The function performs an in-place encoding on the string. Upon completion of this process (a null byte is
<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Code</th>
<th>4-bit Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0x2A</td>
<td>0x0</td>
</tr>
<tr>
<td>+</td>
<td>0x2B</td>
<td>0x1</td>
</tr>
<tr>
<td>,</td>
<td>0x2C</td>
<td>0x2</td>
</tr>
<tr>
<td>-</td>
<td>0x2D</td>
<td>0x3</td>
</tr>
<tr>
<td>.</td>
<td>0x2E</td>
<td>0x4</td>
</tr>
<tr>
<td>/</td>
<td>0x2F</td>
<td>0x5</td>
</tr>
<tr>
<td>0</td>
<td>0x30</td>
<td>0x6</td>
</tr>
<tr>
<td>1</td>
<td>0x31</td>
<td>0x7</td>
</tr>
<tr>
<td>2</td>
<td>0x32</td>
<td>0x8</td>
</tr>
<tr>
<td>3</td>
<td>0x33</td>
<td>0x9</td>
</tr>
<tr>
<td>4</td>
<td>0x34</td>
<td>0xA</td>
</tr>
<tr>
<td>5</td>
<td>0x35</td>
<td>0xB</td>
</tr>
<tr>
<td>6</td>
<td>0x36</td>
<td>0xC</td>
</tr>
<tr>
<td>7</td>
<td>0x37</td>
<td>0xD</td>
</tr>
<tr>
<td>8</td>
<td>0x38</td>
<td>0xE</td>
</tr>
<tr>
<td>9</td>
<td>0x39</td>
<td>0xF</td>
</tr>
</tbody>
</table>

Table A.1: List of supported characters with their ASCII values and 4-bit values.

encountered), **encode** returns the size of the encoded string in bytes. Note that any incomplete byte is padded with 0 bits until it is filled.

### A.1.2 Decoding

The **decode** function takes three arguments. The first is the location of the encoded string, a `const char *`. The second is the location to store the decoded string, a `char *`. The third and final argument is the size of the memory chunk pointed to by the second argument, a `size_t`. The function will continue decoding until it either encounters a null 4-bit character in the encoded string or runs out of room in the destination array. Upon encountering a null 4-bit character in the encoded string, a null byte is appended to the decoded string and the length of the string is returned. Note that the * character is encoded as a null 4-bit character, which means it cannot be recovered by the **decode** function. If the function runs out of room in the destination memory chunk, it will return −1. To avoid this situation, it is highly recommended that the destination memory chunk be at least twice the size of the encoded data.
A.2 Code Listings

Listing A.1: codec.h

```c
#include <stdio.h>

/*
  * 0x2A -> 0x0
  * 0x2B + -> 0x1
  * 0x2C , -> 0x2
  * 0x2D - -> 0x3
  * 0x2E . -> 0x4
  * 0x2F / -> 0x5
  * 0x30 0 -> 0x6
  * 0x31 1 -> 0x7
  * 0x32 2 -> 0x8
  * 0x33 3 -> 0x9
  * 0x34 4 -> 0xA
  * 0x35 5 -> 0xB
  * 0x36 6 -> 0xC
  * 0x37 7 -> 0xD
  * 0x38 8 -> 0xE
  * 0x39 9 -> 0xF
*/

size_t decode(const char * e, char * d, size_t dLen);
size_t encode(char * src);

#endif
```

Listing A.2: codec.c

```c
#include "codec.h"

size_t decode(const char * e, char * d, size_t dLen) {
    size_t count = 0;
    char toggle = 1;
    ```
while (count < dLen) {
    char val = *e;
    if (toggle) {
        val = (val >> 4);
    }
    val &= 0x0F;
    if (val == 0x00) {
        *d = '\0';
        return count;
    }
    val += 0x2A;
    *d = val;
    toggle = !toggle;
    count++; 
    if (toggle) {
        e++;
    } 
    d++;
    toggle = !toggle;
    src++;
} 
return -1;

size_t encode(char * src)
{
    size_t count = 0;
    char * dst = src;
    char toggle = 1;
    while (*src != '\0') {
        char val = *src;
        val -= 0x2A;
        if (toggle) {
            *dst = (val << 4) & 0xF0;
            count++; 
        } else {
            *dst |= (val & 0x0F);
            dst++;
        }
        toggle = !toggle;
        src++;
    }
    return count;
}
Appendix B

Emulation Adapter Code

Listing B.1: Emulation_Adapter.pde

```c
// Some information about scanf use with floats can be found here

#define THISFIRMWARE "UIUC-Emulation-Adapter"

#define DEBUG 0

// The DEBUG flag can be enabled so that the adapter sends debug information over I2C in addition to the normal state data

// Aircraft type selection. This firmware attempts to clean up channel mapping.
#define AIRPLANE 1
#define HELI 0

#include <AP_Common.h>
#include <AP_HAL.h>
#include <AP_HAL_AVR.h>
#include <AP_Math.h>
#include <AP_Param.h>
#include <AP_Progmem.h>
#include <Wire.h>
#include <avr/interrupt.h>
#include <stdio.h>
```
const AP_HAL::HAL& hal = AP_HAL_BOARD_DRIVER;

#define DECODE_BUF_SIZE 4096
char decodeBuf[DECODE_BUF_SIZE];
uint16_t decodeBufWriteIdx;

#define DATAPoints 15

struct sample {
    union {
        float f[DATAPoints];
        uint8_t raw[DATAPoints * sizeof(float)];
    } data;
};

#if DEBUG
struct debug {
    union {
        uint16_t i[5];
        uint8_t raw[5 * sizeof(uint16_t)];
    } indices;
    union {
        bool b[1];
        uint8_t raw[1 * sizeof(bool)];
    } bools;
};
#endif

// Variables for handling data samples
#define SAMPLES_SIZE 2
const uint8_t SAMPLES_MASK = 0x01;
struct sample samples[SAMPLES_SIZE];
uint8_t currentSample;
uint8_t workingSample;

// Number of PWM channels to use
#define CHANNELS 5
// Period of output to simulator, in microseconds
#define PERIOD 10000
// Range of PWM signals, in microseconds
# define SERVO_MIN 1000
# define SERVO_MID 1500
# define SERVO_MAX 2000

uint16_t pwm[CHANNELS];
float range[CHANNELS];
uint32_t nextWrite;
bool consoleAvailable = false;
uint16_t lastParsed = 0;

void setup()
{
    // Initialize indices
    decodeBufWriteIdx = 0;
    currentSample = 0;
    workingSample = 1;

    // Zero out initial data sample so we don't just spew garbage
    for(uint8_t i = 0; i < DATAPoints; i++) {
        samples[currentSample].data.f[i] = 0.0;
    }

    // Compute time for next write to simulator
    nextWrite = hal.scheduler->micros() + PERIOD;

    // Begin I2C communication with slave address 2
    Wire.begin(2);
    // Register callback function for requests
    Wire.onRequest(requestEvent);
}

uint16_t loopCount = 0;

void requestEvent()
{
    // In debug mode, reading state data, debug integers, and ←
    debug booleans is possible
    static int select = 0;
    if(select == 0) {
        Wire.write(samples[currentSample].data.raw, sizeof(←
        samples[currentSample].data));
if (select == 1) {
    struct debug d;
    d.indices.i[0] = decodeBufWriteIdx;
    d.indices.i[1] = currentSample;
    d.indices.i[2] = lastParsed;
    d.indices.i[3] = hal.console->get_flow_control();
    d.indices.i[4] = loopCount;
    Wire.write(d.indices.raw, sizeof(d.indices));
}
if (select == 2) {
    struct debug d;
    d.bools.b[0] = consoleAvailable;
    Wire.write(d.bools.raw, sizeof(d.bools));
}
select = (select + 1) % 3;
#pragma else
    // With debug mode off, reading state data is the only option
    Wire.write(samples[currentSample].data.raw, sizeof(samples[currentSample].data));
#pragma endif

void loop() {
    // Check if it is time to output to simulator
    if (hal.scheduler->micros() >= nextWrite) {
        // Read PWM signals and map them from microseconds to
        // [-1,1] interval that the sim expects
        hal.rcin->read(pwm, CHANNELS);
        for (uint8_t i = 0; i < CHANNELS; i++) {
            range[i] = 2.0*((float)((float)pwm[i]-SERVO_MID))/(SERVO_MAX-SERVO_MIN);
            if (range[i] < -1.0) {
                range[i] = -1.0;
            } else if (range[i] > 1.0) {
                range[i] = 1.0;
            }
        }
        nextWrite += PERIOD;
        loopCount++;
    }
}
// Send actuation data to the simulator. The two classes of aircraft are slightly different to ease channel mapping for the user.

```c
#ifndef HELI
    hal.console->printf("%f,%f,%f,%f,%f,%d\r\n", -1.0*range[2], -1.0*range[0], range[3], -1.0*range[1], range[4], 1);
#endif

#ifndef AIRPLANE
    hal.console->printf("%f,%f,%f,%f,%f,%d\r\n", range[0], range[4], range[1], range[2], range[3], -1.0, 1);
#endif
```

```c
bool foundSample = false;
// Read data from simulator until no more is available, we run out of space, or we find a data sample
while((consoleAvailable = hal.console->available()) && decodeBufWriteIdx < DECODE_BUF_SIZE-1 && !foundSample) {
    // Read a byte
    int16_t c = hal.console->read();
    if(c != -1) { // Check if read was successful
        uint8_t cc = (uint8_t)c; // Cast the byte to the right type
        // Pull out the two 4-bit components
        char v1 = (cc >> 4) & 0x0F;
        char v2 = cc & 0x0F;
        // Decode the first component
        if(v1 == 0x00) {
            foundSample = true;
            decodeBuf[decodeBufWriteIdx] = '\0';
        } else {
            decodeBuf[decodeBufWriteIdx] = (v1 + 0x2A);
        }
        decodeBufWriteIdx++;
        // Decode the second component
        if(!foundSample) {
            if(v2 == 0x00) {
                foundSample = true;
            }
        }
    }
}
```
```c
 decodeBuf[decodeBufWriteIdx] = '\0';
 }
 else {
   decodeBuf[decodeBufWriteIdx] = (v2 + 0x2A);
   decodeBufWriteIdx++;
 }

 if (foundSample) {
   int parsed = 0;
   #if DATAPOINTS == 15
     // Parse out data
     parsed = sscanf(decodeBuf,
         "%f,%f,%f,%f,%f, ",
         "%f,%f,%f,%f,%f, ",
         "%f,%f,%f,%f,%f, ",
         samples[workingSample].data.f,
         samples[workingSample].data.f+1,
         samples[workingSample].data.f+2,
         samples[workingSample].data.f+3,
         samples[workingSample].data.f+4,
         samples[workingSample].data.f+5,
         samples[workingSample].data.f+6,
         samples[workingSample].data.f+7,
         samples[workingSample].data.f+8,
         samples[workingSample].data.f+9,
         samples[workingSample].data.f+10,
         samples[workingSample].data.f+11,
         samples[workingSample].data.f+12,
         samples[workingSample].data.f+13,
         samples[workingSample].data.f+14);
   decodeBufWriteIdx = 0;
   #else
     // If the number of data points changes, the call to sscanf
     // must change
     #error "DATAPOINTS does not match"
   #endif
   lastParsed = (uint16_t)parsed;
   if (parsed == DATAPOINTS) {
```
// Critical section, performing switch from working copy
// to current copy. Disable interrupts, perform switch,
// and enable interrupts. This prevents I2C request
// handler from having a race condition with this switch.

uint8_t oldSREG = SREG;
cli();
workingSample = (workingSample + 1) & SAMPLES_MASK;
currentSample = (currentSample + 1) & SAMPLES_MASK;
SREG = oldSREG;

AP_HAL_MAIN();
Listing C.1: AsyncSerial.h

```cpp
/*
 * File: AsyncSerial.h
 * Author: Terraneo Federico
 * Distributed under the Boost Software License, Version 1.0.
 * Created on September 7, 2009, 10:46 AM
 */

#ifndef ASYNCSERIAL_H
#define ASYNCSERIAL_H

#include <vector>
#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/thread.hpp>
#include <boost/utility.hpp>
#include <boost/function.hpp>
#include <boost/shared_array.hpp>

/**
 * Used internally (pimpl)
 */

class AsyncSerialImpl;

/**
 * Asynchronous serial class.
 * Intended to be a base class.
 */

class AsyncSerial: private boost::noncopyable {

public:
```
AsyncSerial();

/**
 * Constructor. Creates and opens a serial device.
 * \param devname serial device name, example "/dev/ttyS0" or "COM1"
 * \param baud_rate serial baud rate
 * \param opt_parity serial parity, default none
 * \param opt_csize serial character size, default 8bit
 * \param opt_flow serial flow control, default none
 * \param opt_stop serial stop bits, default 1
 * \throws boost::system::system_error if cannot open the serial device
 */
AsyncSerial(const std::string& devname, unsigned int baud_rate,
    boost::asio::serial_port_base::parity opt_parity= boost::asio::serial_port_base::parity( boost::asio::serial_port_base::parity::none),
    boost::asio::serial_port_base::character_size opt_csize= boost::asio::serial_port_base::character_size(8),
    boost::asio::serial_port_base::flow_control opt_flow= boost::asio::serial_port_base::flow_control( boost::asio::serial_port_base::flow_control::none),
    boost::asio::serial_port_base::stop_bits opt_stop= boost::asio::serial_port_base::stop_bits( boost::asio::serial_port_base::stop_bits::one))
;
```cpp
void open(const std::string& devname, unsigned int <-
  baud_rate,
  boost::asio::serial_port_base::parity opt_parity=
    boost::asio::serial_port_base::parity(
      boost::asio::serial_port_base::parity::none),
  boost::asio::serial_port_base::character_size opt_csize<=
    boost::asio::serial_port_base::character_size(8),
  boost::asio::serial_port_base::flow_control opt_flow=
    boost::asio::serial_port_base::flow_control(
      boost::asio::serial_port_base::flow_control::<-none),
  boost::asio::serial_port_base::stop_bits opt_stop=
    boost::asio::serial_port_base::stop_bits(         
      boost::asio::serial_port_base::stop_bits::one))<-;

/**
 * \return true if serial device is open
 */
bool isOpen() const;

/**
 * \return true if error were found
 */
bool errorStatus() const;

/**
 * Close the serial device
 * \throws boost::system::system_error if any error
 */
void close();

/**
 * Write data asynchronously. Returns immediately.
 * \param data array of char to be sent through the serial device
 * \param size array size
 */
void write(const char *data, size_t size);

/**
 * Write data asynchronously. Returns immediately.
 */
* 
\param data to be sent through the serial device
*
void write(const std::vector<char>& data);

/**
* Write a string asynchronously. Returns immediately.
* Can be used to send ASCII data to the serial device.
* To send binary data, use write()
* 
\param s string to send
*/
void writeString(const std::string& s);

virtual ~AsyncSerial() = 0;

/**
* Read buffer maximum size
*/
static const int readBufferSize = 512;

private:

/**
* Callback called to start an asynchronous read operation.
* This callback is called by the io_service in the spawned thread.
*/
void doRead();

/**
* Callback called at the end of the asynchronous operation.
* This callback is called by the io_service in the spawned thread.
*/
void readEnd(const boost::system::error_code& error,
size_t bytes_transferred);

/**
* Callback called to start an asynchronous write operation.
* If it is already in progress, does nothing.
* This callback is called by the io_service in the spawned thread.
*/
void doWrite();

/**
 * Callback called at the end of an asynchronous write operation,
 * if there is more data to write, restarts a new write operation.
 * This callback is called by the i/o service in the spawned thread.
 */
void writeEnd(const boost::system::error_code & error);

/**
 * Callback to close serial port
 */
void doClose();

boost::shared_ptr<AsyncSerialImpl> pimpl;

protected:

/**
 * To allow derived classes to report errors
 * \param error status
 */
void setErrorStatus(bool e);

/**
 * To allow derived classes to set a read callback
 */
void setReadCallback(const
    boost::function<void (const char*, size_t)>& callback);

/**
 * To unregister the read callback in the derived class destructor so it
 * does not get called after the derived class destructor but before the
 * base class destructor
 */
void clearReadCallback();
Asynchronous serial class with read callback. User code can write data from one thread, and read data will be reported through a callback called from a separate thread.

```
class CallbackAsyncSerial: public AsyncSerial
{
  public:
    CallbackAsyncSerial();

  /**
   * Opens a serial device.
   * \param devname serial device name, example "/dev/ttyS0" or "COM1"
   * \param baud_rate serial baud rate
   * \param opt-parity serial parity, default none
   * \param opt-csize serial character size, default 8bit
   * \param opt-flow serial flow control, default none
   * \param opt-stop serial stop bits, default 1
   * \throws boost::system::system_error if cannot open the serial device
   */
    CallbackAsyncSerial(const std::string& devname, unsigned int baud_rate,
                        boost::asio::serial_port_base::parity opt_parity=
                          boost::asio::serial_port_base::parity(boost::asio::serial_port_base::parity::none),
                        boost::asio::serial_port_base::character_size opt_csize=
                          boost::asio::serial_port_base::character_size(8),
                        boost::asio::serial_port_base::flow_control opt_flow=
                          boost::asio::serial_port_base::flow_control(boost::asio::serial_port_base::flow_control::none),
                        boost::asio::serial_port_base::stop_bits opt_stop=
                          boost::asio::serial_port_base::stop_bits(boost::asio::serial_port_base::stop_bits::one))
    {
    }
```
/**
 * Set the read callback, the callback will be called from a thread
 * owned by the CallbackAsyncSerial class when data arrives from the
 * serial port.
 * \param callback the receive callback
 */
void setCallback(const
  boost::function<
    void (const char*, size_t)>& callback);

/**
 * Removes the callback. Any data received after this function call will
 * be lost.
 */
void clearCallback();

virtual ~CallbackAsyncSerial();
};

#endif //ASYNC_SERIAL_H

Listing C.2: AsyncSerial.cpp

/*
 * File: AsyncSerial.cpp
 * Author: Terraneo Federico
 * Distributed under the Boost Software License, Version 1.0.
 * Created on September 7, 2009, 10:46 AM
 * *
 * v1.02: Fixed a bug in BufferedAsyncSerial: Using the default constructor
 * the callback was not set up and reading didn't work.
 * *
 * v1.01: Fixed a bug that did not allow to reopen a closed serial port.
 * *
 * v1.00: First release.
 * *
 * IMPORTANT:
 */
On Mac OS X boost asio's serial ports have bugs, and the usual implementation of this class does not work. So a workaround class was written temporarily, until asio (hopefully) will fix Mac compatibility for serial ports.

Please note that unlike said in the documentation on OS X until asio will be fixed serial port *writes* are *not* asynchronous, but at least asynchronous *read* works.

In addition the serial port open ignores the following options: parity, character size, flow, stop bits, and defaults to 8N1 format.

I know it is bad but at least it's better than nothing.

#include "AsyncSerial.h"

#include <string>
#include <stdio.h>
#include <algorithm>
#include <iostream>
#include <boost/bind.hpp>

using namespace std;
using namespace boost;

//
//Class AsyncSerial
//

#ifndef __APPLE__

class AsyncSerialImpl: private boost::noncopyable
{

public:
    AsyncSerialImpl(): io(), port(io), backgroundThread(), open(false),
    error(false) {}
boost::asio::io_service io; ///< Io service object
boost::asio::serial_port port; ///< Serial port object
boost::thread backgroundThread; ///< Thread that runs read/←write operations
bool open; ///< True if port open
bool error; ///< Error flag
mutable boost::mutex errorMutex; ///< Mutex for access to ←error

/// Data are queued here before they go in writeBuffer
std::vector<char> writeQueue;
boost::shared_array<char> writeBuffer; ///< Data being ←written
size_t writeBufferSize; ///< Size of writeBuffer
boost::mutex writeQueueMutex; ///< Mutex for access to ←writeQueue
char readBuffer[AsyncSerial::readBufferSize]; ///< data ←being read

/// Read complete callback
boost::function<void (const char*, size_t)> callback;

AsyncSerial::AsyncSerial(): pimpl(new AsyncSerialImpl)
{
}

AsyncSerial::AsyncSerial(const std::string& devname, unsigned ←int baud_rate,
    asio::serial_port_base::parity opt_parity,
    asio::serial_port_base::character_size opt_csize,
    asio::serial_port_base::flow_control opt_flow,
    asio::serial_port_base::stop_bits opt_stop)
    : pimpl(new AsyncSerialImpl)
{
    open(devname, baud_rate, opt_parity, opt_csize, opt_flow,←
        opt_stop);
}

void AsyncSerial::open(const std::string& devname, unsigned int←
    baud_rate,
    asio::serial_port_base::parity opt_parity,
asio::serial_port_base::character_size opt_csize,
asio::serial_port_base::flow_control opt_flow,
asio::serial_port_base::stop_bits opt_stop)
{
  if(isOpen()) close();
  setErrorStatus(true);// If an exception is thrown, error_ \rightarrow
  remains true
  pimpl->port.open(devname);
  pimpl->port.set_option(asio::serial_port_base::baud_rate(\rightarrow
    baud_rate));
  pimpl->port.set_option(opt_parity);
  pimpl->port.set_option(opt_csize);
  pimpl->port.set_option(opt_flow);
  pimpl->port.set_option(opt_stop);
  // This gives some work to the io_service before it is \rightarrow
  started
  pimpl->io.post(boost::bind(&AsyncSerial::doRead, this));
  thread t(boost::bind(&asio::io_service::run, &pimpl->io));
  pimpl->backgroundThread.swap(t);
  setErrorStatus(false);// If we get here, no error
  pimpl->open=true; // Port is now open
}

bool AsyncSerial::isOpen() const
{
  return pimpl->open;
}

bool AsyncSerial::errorStatus() const
{
  lock_guard<mutex> l(pimpl->errorMutex);
  return pimpl->error;
}

void AsyncSerial::close()
{
  if(!isOpen()) return;
  pimpl->open=false;
  pimpl->io.post(boost::bind(&AsyncSerial::doClose, this));
pimpl->backgroundThread.join();
pimpl->io.reset();
if(errorStatus())
  {  
    throw(boost::system::system_error(boost::system::←
      error_code(),
      "Error while closing the device");
  }  
}

void AsyncSerial::write(const char *data, size_t size)
  {
    lock_guard<mutex> l(pimpl->writeQueueMutex);
    pimpl->writeQueue.insert(pimpl->writeQueue.end(), data,←
      data+size);
  }
pimpl->io.post(boost::bind(&AsyncSerial::doWrite, this));

void AsyncSerial::write(const std::vector<char>& data)
  {
    lock_guard<mutex> l(pimpl->writeQueueMutex);
    pimpl->writeQueue.insert(pimpl->writeQueue.end(), data.begin(),
      data.end());
  }
pimpl->io.post(boost::bind(&AsyncSerial::doWrite, this));

void AsyncSerial::writeString(const std::string& s)
  {
    lock_guard<mutex> l(pimpl->writeQueueMutex);
    pimpl->writeQueue.insert(pimpl->writeQueue.end(), s.begin(),
      s.end());
  }
pimpl->io.post(boost::bind(&AsyncSerial::doWrite, this));

AsyncSerial::~AsyncSerial()
  {

if(isOpen())
{
    try {
        close();
    } catch(...) {
        //Don't throw from a destructor
    }
}

void AsyncSerial::doRead()
{
pimpl->port.async_read_some(asio::buffer(pimpl->readBuffer, readBufferSize),
    boost::bind(&AsyncSerial::readEnd, this,
        asio::placeholders::error,
        asio::placeholders::bytes_transferred));
}

void AsyncSerial::readEnd(const boost::system::error_code& error,
    size_t bytes_transferred)
{
    if(error)
    {
        #ifdef __APPLE__
        if(error.value() == 45)
        {
            //Bug on OS X, it might be necessary to repeat the setup
            //http://osdir.com/ml/lib.boost.asio.user/2008-08/msg00004.html
doRead();
            return;
        }
        #endif //__APPLE__

        //error can be true even because the serial port was closed.
        //In this case it is not a real error, so ignore
        if(isOpen())
        {
        }
doClose();
setErrorStatus(true);
}
else {
    if (pimpl->callback) pimpl->callback(pimpl->readBuffer, bytes_transferred);
doRead();
}
}

void AsyncSerial::doWrite()
{
    // If a write operation is already in progress, do nothing
    if (pimpl->writeBuffer == 0)
    {
        lock_guard<mutex> l(pimpl->writeQueueMutex);
pimpl->writeBufferSize = pimpl->writeQueue.size();
pimpl->writeBuffer.reset(new char[pimpl->writeQueue.size()]);
copy(pimpl->writeQueue.begin(), pimpl->writeQueue.end(),
    pimpl->writeBuffer.get());
pimpl->writeQueue.clear();
async_write(pimpl->port, asio::buffer(pimpl->writeBuffer.get(),
    pimpl->writeBufferSize),
    boost::bind(&AsyncSerial::writeEnd, this, asio::placeholders::error));
    }
}

void AsyncSerial::writeEnd(const boost::system::error_code& ← error)
{
    if (!error)
    {
        lock_guard<mutex> l(pimpl->writeQueueMutex);
        if (pimpl->writeQueue.empty())
        {
            pimpl->writeBuffer.reset();
pimpl->writeBufferSize = 0;
            return;
        }
    }
pimpl->writeBufferSize=pimpl->writeQueue.size();
pimpl->writeBuffer.reset(new char[pimpl->writeQueue.size()]);
copy(pimpl->writeQueue.begin(), pimpl->writeQueue.end(), pimpl->writeBuffer.get());
pimpl->writeQueue.clear();
async_write(pimpl->port, asio::buffer(pimpl->writeBuffer.get(), pimpl->writeBufferSize),
            boost::bind(&AsyncSerial::writeEnd, this, asio::placeholders::error));
} else {
    setErrorStatus(true);
    doClose();
}

void AsyncSerial::doClose() {
    boost::system::error_code ec;
pimpl->port.cancel(ec);
    if (ec) setErrorStatus(true);
pimpl->port.close(ec);
    if (ec) setErrorStatus(true);
}

void AsyncSerial::setErrorStatus(bool e) {
    lock_guard<mutex> l(pimpl->errorMutex);
pimpl->error=e;
}

void AsyncSerial::setReadCallback(const boost::function<void (const char*, size_t)>& callback)
{
    pimpl->callback=callback;
}

void AsyncSerial::clearReadCallback()
{
    pimpl->callback.clear();
}
```cpp
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <unistd.h>

class AsyncSerialImpl: private boost::noncopyable
{
public:
    AsyncSerialImpl(): backgroundThread(), open(false), error(← false) {}

    boost::thread backgroundThread; ///< Thread that runs read ← operations
    bool open; ///< True if port open
    bool error; ///< Error flag
    mutable boost::mutex errorMutex; ///< Mutex for access to ← error

    int fd; ///< File descriptor for serial port

    char readBuffer[AsyncSerial::readBufferSize]; ///< data ← being read

    // Read complete callback
    boost::function<void (const char*, size_t)> callback;
};

AsyncSerial::AsyncSerial(): pimpl(new AsyncSerialImpl)
{
}

AsyncSerial::AsyncSerial(const std::string& devname, unsigned ←
    int baud_rate,
    asio::serial_port_base::parity opt_parity,
    asio::serial_port_base::character_size opt_csize,
    asio::serial_port_base::flow_control opt_flow,
    asio::serial_port_base::stop_bits opt_stop)
    : pimpl(new AsyncSerialImpl)
{
```
open(devname, baud_rate, opt_parity, opt_csize, opt_flow,←
    opt_stop);
}

void AsyncSerial::open(const std::string& devname, unsigned int←
    baud_rate,
    asio::serial_port_base::parity opt_parity,
    asio::serial_port_base::character_size opt_csize,
    asio::serial_port_base::flow_control opt_flow,
    asio::serial_port_base::stop_bits opt_stop)
{
    if(isOpen()) close();
    setErrorStatus(true);// If an exception is thrown, error ←
    remains true

    struct termios new_attributes;
    speed_t speed;
    int status;

    // Open port
    pimpl->fd=::open(devname.c_str(), O_RDWR | O_NOCTTY | ←
        O_NONBLOCK);
    if (pimpl->fd<0) throw(boost::system::system_error(
        boost::system::error_code(),"Failed to open port")←
        );

    // Set Port parameters.
    status=tcgetattr(pimpl->fd,&new_attributes);
    if(status<0 || !isatty(pimpl->fd))
    {
        ::close(pimpl->fd);
        throw(boost::system::system_error(
            boost::system::error_code(),"Device is not ←
            a tty");
    }
    new_attributes.c_iflag = IGNBRK;
    new_attributes.c_oflag = 0;
    new_attributes.c_lflag = 0;
    new_attributes.c_cflag = (CS8 | CREAD | CLOCAL);//8 data ←
    bit,Enable receiver,Ignore modem
    /* In non canonical mode (Ctrl-C and other disabled, no ←
    echo,...) VMIN and VTIME work this way:
if the function read() has't read at least VMIN chars it ←
waits until has read at least VMIN
chars (even if VTIME timeout expires); once it has read at ←
least vmin chars, if subsequent
chars do not arrive before VTIME expires, it returns error;←
if a char arrives, it resets the
timeout, so the internal timer will again start from zero (←
for the nex char, if any)*

new_attributes.c_cc[VMIN]=1; // Minimum number of characters←
to read before returning error

new_attributes.c_cc[VTIME]=1; // Set timeouts in tenths of ←
second

// Set baud rate
switch(baud_rate)
{
    case 50: speed= B50; break;
    case 75: speed= B75; break;
    case 110: speed= B110; break;
    case 134: speed= B134; break;
    case 150: speed= B150; break;
    case 200: speed= B200; break;
    case 300: speed= B300; break;
    case 600: speed= B600; break;
    case 1200: speed= B1200; break;
    case 1800: speed= B1800; break;
    case 2400: speed= B2400; break;
    case 4800: speed= B4800; break;
    case 9600: speed= B9600; break;
    case 19200: speed= B19200; break;
    case 38400: speed= B38400; break;
    case 57600: speed= B57600; break;
    case 115200: speed= B115200; break;
    case 230400: speed= B230400; break;
    default:
    {
        :close(pimpl→fd);
        throw(boost::system::system_error(
            boost::system::error_code(),"←
            Unsupported baud rate"));
    }
}
cfsetospeed(&new_attributes, speed);
cfsetispeed(&new_attributes, speed);

// Make changes effective
status = tcsetattr(pimpl->fd, TCSANOW, &new_attributes);
if (status < 0)
{
    ::close(pimpl->fd);
    throw (boost::system::system_error(
            boost::system::error_code(), "Can't set port←attributes");
}

// These 3 lines clear the O_NONBLOCK flag
status = fcntl(pimpl->fd, F_GETFL, 0);
if (status != -1)
    fcntl(pimpl->fd, F_SETFL, status & ~←O_NONBLOCK);

setErrorStatus(false); // If we get here, no error
pimpl->open = true; // Port is now open

thread t(bind(&AsyncSerial::doRead, this));
pimpl->backgroundThread.swap(t);

bool AsyncSerial::isOpen() const
{
    return pimpl->open;
}

bool AsyncSerial::errorStatus() const
{
    lock_guard<mutex> l(pimpl->errorMutex);
    return pimpl->error;
}

void AsyncSerial::close()
{
    if (!isOpen()) return;
    pimpl->open = false;
::close(pimpl->fd); // The thread waiting on I/O should return

pimpl->backgroundThread.join();
if(errorStatus())
{
    throw( boost::system::system_error( boost::system::error_code(),
        "Error while closing the device");
}

void AsyncSerial::write(const char *data, size_t size)
{
    if(::write(pimpl->fd, data, size)!=size) setErrorStatus(true);
}

void AsyncSerial::write(const std::vector<char>& data)
{
    if(::write(pimpl->fd,&data[0],data.size())!=data.size())
        setErrorStatus(true);
}

void AsyncSerial::writeString(const std::string&s)
{
    if(::write(pimpl->fd,&s[0],s.size())!=s.size())
        setErrorStatus(true);
}

AsyncSerial::~AsyncSerial()
{
    if(isOpen())
    {
        try {
            close();
        } catch(...) {
            // Don't throw from a destructor
        }
    }
```cpp
void AsyncSerial::doRead()
{
    // Read loop in spawned thread
    for(;;)
    {
        int received = ::read(pimpl->fd, pimpl->readBuffer, readBufferSize);
        if (received < 0)
        {
            if (isOpen() == false) return; // Thread interrupted because port closed
            else {
                setErrorStatus(true);
                continue;
            }
        }
        if (pimpl->callback) pimpl->callback(pimpl->readBuffer, received);
    }
}

void AsyncSerial::readEnd(const boost::system::error_code& error,
                          size_t bytes_transferred)
{
    // Not used
}

void AsyncSerial::doWrite()
{
    // Not used
}

void AsyncSerial::writeEnd(const boost::system::error_code& error)
{
    // Not used
}

void AsyncSerial::doClose()
{
    // Not used
}
```
void AsyncSerial::setErrorStatus(bool e)
{
    lock_guard<mutex> l(pimpl->errorMutex);
    pimpl->error = e;
}

void AsyncSerial::setReadCallback(const function<void (const char*, size_t)>>& callback)
{
    pimpl->callback = callback;
}

void AsyncSerial::clearReadCallback()
{
    pimpl->callback.clear();
}

#endif // _APPLE_

//
// Class CallbackAsyncSerial
//

CallbackAsyncSerial::CallbackAsyncSerial(): AsyncSerial()
{
}

CallbackAsyncSerial::CallbackAsyncSerial(const std::string& devname,
    unsigned int baud_rate,
    asio::serial_port_base::parity opt_parity,
    asio::serial_port_base::character_size opt_csize,
    asio::serial_port_base::flow_control opt_flow,
    asio::serial_port_base::stop_bits opt_stop)
: AsyncSerial(devname, baud_rate, opt_parity, opt_csize, opt_flow, opt_stop)
{
}

void CallbackAsyncSerial::setCallback(const

boost::function<void (const char*, size_t)>& callback
{
  setReadCallback(callback);
}

void CallbackAsyncSerial::clearCallback()
{
  clearReadCallback();
}

CallbackAsyncSerial::~CallbackAsyncSerial()
{
  clearReadCallback();
}

Listing C.3: codec.h

#pragma once

#include <stdio.h>

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

/*
  * 0x2A  * -> 0x0
  * 0x2B + -> 0x1
  * 0x2C ,  -> 0x2
  * 0x2D -  -> 0x3
  * 0x2E .  -> 0x4
  * 0x2F /  -> 0x5
  * 0x30 0  -> 0x6
  * 0x31 1  -> 0x7
  * 0x32 2  -> 0x8
  * 0x33 3  -> 0x9
  * 0x34 4  -> 0xA
  * 0x35 5  -> 0xB
  * 0x36 6  -> 0xC
  * 0x37 7  -> 0xD
  * 0x38 8  -> 0xE
  * 0x39 9  -> 0xF

60
size_t decode(const char * e, char * d, size_t dLen);
size_t encode(char * src);

#define __cplusplus
}
#endif

size_t decode(const char * e, char * d, size_t dLen) {
    size_t count = 0;
    char toggle = 1;
    while (count < dLen) {
        char val = *e;
        if (toggle) {
            val = (val >> 4);
        }
        val &= 0x0F;
        if (val == 0x00) {
            *d = '\0';
            return count;
        }
        val += 0x2A;
        *d = val;
        toggle = !toggle;
        count++;
    }
    return -1;

Listing C.4: codec.cpp
#include <string>
#include "TFmResetState2.h"

class Configuration
{
public:
    Configuration();

    int load(const char * path);
    void print() const;
}
```cpp
int fixed() const;
void fixedAttitude(float *phi, float *theta, float *psi) const;
void fixedPosition(float *pn, float *pe, float *pd) const;
void fixedRotation(float *p, float *q, float *r) const;
void fixedVelocity(float *u, float *v, float *w) const;
bool isEncodingEnabled() const;
bool isLoggingEnabled() const;
std::string logDirectory() const;
void mapAndConvert(const TFmResetState2 *state, TFmResetState2 *mapped) const;
long outputPeriod() const;
int serialBaud() const;
std::string serialPath() const;

private:
int _baud;
int _encoding;
int _fixed;
float _fxdAtt[3];
float _fxdPos[3];
float _fxdRot[3];
float _fxdVel[3];
std::string _logdir;
int _logging;
float _map[4];
long _period;
std::string _serial;
void readLogLine(std::string line);
```

Listing C.6: configuration.cpp

```cpp
#include "configuration.h"

#include <fstream>
#include <math.h>

#define M_PI 3.14159265358979323846
#define M_PER FOOT 0.3048
```
Constructor simply sets all of the default settings.

Configuration() {
    _baud = 115200;
    _encoding = 1;
    _fixed = 0;
    _fxdAtt[0] = 0;
    _fxdAtt[1] = 0;
    _fxdAtt[2] = 0;
    _fxdPos[0] = -365 * M_PER_FOOT;
    _fxdPos[1] = -400 * M_PER_FOOT;
    _fxdPos[2] = -10;
    _fxdRot[0] = 0;
    _fxdRot[1] = 0;
    _fxdRot[2] = 0;
    _fxdVel[0] = 0;
    _fxdVel[1] = 0;
    _fxdVel[2] = 0;
    _logdir = "/";
    _logging = 0;
    _map[0] = -365 * M_PER_FOOT;
    _map[1] = -400 * M_PER_FOOT;
    _map[2] = 0;
    _map[3] = 0;
    _period = 100000; // This is 100Hz. Units are 100ns
    _serial = "COM1";
}

/*
 * Load the file from the specified path.
 * The file is processed line by line.
 */
int Configuration::load(const char * path) {
    std::ifstream config(path);
    std::string s;
    int l = 0;
    while (std::getline(config, s)) {
        readLogLine(s);
        l++;
    }
Parse out data from log lines.
* Basically just a series of cases, one for each variable.

```cpp
void Configuration::readLogLine(std::string line)
{
    if (line.find("baud") == 0) {
        sscanf(line.c_str(), "baud=%d", &_baud);
    } else if (line.find("encoding") == 0) {
        sscanf(line.c_str(), "encoding=%d", &_encoding);
    } else if (line.find("fixed") == 0) {
        sscanf(line.c_str(), "fixed=%d", &_fixed);
    } else if (line.find("fxdAtt") == 0) {
        sscanf(line.c_str(), "fxdAtt= %f, %f, %f", _fxdAtt, _fxdAtt + 1, _fxdAtt + 2);
    } else if (line.find("fxdPos") == 0) {
        sscanf(line.c_str(), "fxdPos= %f, %f, %f", _fxdPos, _fxdPos + 1, _fxdPos + 2);
    } else if (line.find("fxdRot") == 0) {
        sscanf(line.c_str(), "fxdRot= %f, %f, %f", _fxdRot, _fxdRot + 1, _fxdRot + 2);
    } else if (line.find("fxdVel") == 0) {
        sscanf(line.c_str(), "fxdVel= %f, %f, %f", _fxdVel, _fxdVel + 1, _fxdVel + 2);
    } else if (line.find("logdir") == 0) {
        _logdir = line.substr(line.find("=") + 1);
    } else if (line.find("logging") == 0) {
        sscanf(line.c_str(), "logging=%d", &_logging);
    } else if (line.find("map") == 0) {
        return 1;
    }
    /*
    */
}
```
sscanf(line.c_str(), "map=%f,%f,%f,%f", _map, _map+1, _map+2, _map+3);
}
else if (line.find("period") == 0) {
    sscanf(line.c_str(), "period=%ld", &_period);
}
else if (line.find("serial") == 0) {
    _serial = line.substr(line.find("=") + 1);
}
} /*
 * Print out the configuration for debugging purposes.
 */
void Configuration::print() const
{
    printf("*** CONFIGURATION ***\n");
    printf("baud=%d\n", _baud);
    printf("encoding=%d\n", _encoding);
    printf("fixed=%d\n", _fixed);
    printf("fxdAtt=%f,%f,%f\n", _fxdAtt[0], _fxdAtt[1], _fxdAtt[2]);
    printf("fxdPos=%f,%f,%f\n", _fxdPos[0], _fxdPos[1], _fxdPos[2]);
    printf("fxdRot=%f,%f,%f\n", _fxdRot[0], _fxdRot[1], _fxdRot[2]);
    printf("fxdVel=%f,%f,%f\n", _fxdVel[0], _fxdVel[1], _fxdVel[2]);
    printf("logdir=%s\n", _logdir.c_str());
    printf("logging=%d\n", _logging);
    printf("map=%f,%f,%f,%f\n", _map[0], _map[1], _map[2], _map[3]);
    printf("period=%ld\n", _period);
    printf("serial=%s\n", _serial.c_str());
}
/*
 * Returns the value of fixed.
 * fixed is a bit field
 * 0x1 fix Rotation Rates
 * 0x2 fix Attitude
 * 0x4 fix Velocity
 * 0x8 fix Position
 */
int Configuration::fixed() const
{
    return _fixed;
}

/*
 * Places the attitude fix point in the memory addressed by the argument pointers
*/
void Configuration::fixedAttitude(float *phi, float *theta, float *psi) const
{
    *phi = _fxdAtt[0];
    *theta = _fxdAtt[1];
    *psi = _fxdAtt[2];
}

/*
 * Places the position fix point in the memory addressed by the argument pointers
 * Note that the data is stored in meters, but given in feet since the sim uses feet
*/
void Configuration::fixedPosition(float *pn, float *pe, float *pd) const
{
    *pn = _fxdPos[0] / M_PER_FOOT;
    *pe = _fxdPos[1] / M_PER_FOOT;
    *pd = _fxdPos[2] / M_PER_FOOT;
}

/*
 * Places the rotation fix point in the memory addressed by the argument pointers
*/
void Configuration::fixedRotation(float *p, float *q, float *r) const
{
    *p = _fxdRot[0];
    *q = _fxdRot[1];
    *r = _fxdRot[2];
}
void Configuration::fixedVelocity(float *u, float *v, float *w) const
{
    *u = _fxdVel[0] / M_PER_FOOT;
    *v = _fxdVel[1] / M_PER_FOOT;
    *w = _fxdVel[2] / M_PER_FOOT;
}

bool Configuration::isEncodingEnabled() const
{
    return _encoding != 0;
}

bool Configuration::isLoggingEnabled() const
{
    return _logging != 0;
}

std::string Configuration::logDirectory() const
{
    return _logdir;
}

/*
 * Accepts a pointer to the current state as an argument.
*/
* Performs conversion from Imperial to metric.
* Also performs the transformations necessary to map the simulator
  coordinate system to a real-world location.
* The mapped and converted state information is placed in the structure
  that is pointed to by mapped.

```cpp
void Configuration::mapAndConvert(const TFmResetState2 * state, TFmResetState2 * mapped) const
{
    if (mapped != NULL && state != NULL) {
        // These are body frame, so they are unchanged
        mapped->U = state->U * M_PER_FOOT;
        mapped->V = state->V * M_PER_FOOT;
        mapped->W = state->W * M_PER_FOOT;
        mapped->P = state->P;
        mapped->Q = state->Q;
        mapped->R = state->R;
        mapped->phi = state->phi;
        mapped->theta = state->theta;

        // Roll and Pitch are unaffected
        if (mapped->psi < -1 * M_PI) {
            mapped->psi += (2 * M_PI);
        } else if (mapped->psi > M_PI) {
            mapped->psi -= (2 * M_PI);
        }

        mapped->z = state->z * M_PER_FOOT + _map[2];
        float xDist = (state->x + 365) * M_PER_FOOT;
        float yDist = (state->y + 400) * M_PER_FOOT;
        mapped->x = _map[0] + xDist*cos(-1*map[3]) + yDist*sin(-1*map[3]);
        mapped->y = _map[1] - xDist*sin(-1*map[3]) + yDist*cos(-1*map[3]);
    }
}```
long Configuration::outputPeriod() const
{
    return _period;
}

int Configuration::serialBaud() const
{
    return _baud;
}

std::string Configuration::serialPath() const
{
    return _serial;
}

Listing C.7: main.cpp

// Simulator External Control Example
#define WIN32_LEAN_AND_MEAN
#define _WIN32_WINNT 0x601
#include <windows.h>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <math.h>
#include <iostream>
#include <iomanip>
#include <ctime>
```cpp
#include "AsyncSerial.h"
#include "codec.h"
#include "configuration.h"
#include "TFmResetState2.h"
#include "toString.h"

#define M_PI 3.14159265358979323846
#define M_PER_FOOT 0.3048

Configuration config;
bool COM_init = false;
FILE * logFile = NULL;
CallbackAsyncSerial * serial;
float currentControl[] = {-1, 0, 0, 0, 0, 0};
std::string input;
int overrideOutputs = 1;
int serialCount = 0;
int outCount = 0;
ULARGE_INTEGER lastWrite;
bool firstWrite = false;

/*
 * This function is registered as the callback when data is received by the AsyncSerial object.
 * The data is added to an input string.
 * The input string is checked for a full sample.
 * If a sample is found, it is parsed.
 * If the sample parses correctly, it is made available in a global control array.
 * The full sample is removed from the input string.
 */
void receiveCallback(const char *data, unsigned int len)
{
    float readBufF[6];
    int readBufI;
    std::string i = "";
    i.append(data, len);
    printf("%s", i.c_str());
    input.append(data, len); // Add sample to input data
    size_t end = input.rfind("\r\n"); // Find end of most recent data sample
```
size_t begin = input.rfind("\r\n", end-1); // Find start of most recent data sample
if (end != std::string::npos) { // If end is found
    std::string sample = input.substr((begin == std::string::npos) ? 0 : begin+2, end+2); // Get most recent sample
    input.erase(0, end+2); // Delete everything up to and including most recent sample
    int sscanfRes = sscanf(sample.c_str(), "%f,%f,%f,%f,%f,%f,%d\r\n", readBufF, readBufF+1, readBufF+2, readBufF+3, readBufF+4, readBufF+5, &readBufI); // Parse sample
    if (sscanfRes == 7) { // If sample was correctly parsed
        // Store parsed data sample
        currentControl[0] = readBufF[0];
        currentControl[1] = readBufF[1];
        currentControl[2] = readBufF[2];
        currentControl[3] = readBufF[3];
        currentControl[4] = readBufF[4];
        currentControl[5] = readBufF[5];
        overrideOutputs = readBufI;
        // printf("%f,%f,%f,%f,%f,%f,%d\r\n", currentControl[0], currentControl[1], currentControl[2],
        currentControl[3], currentControl[4],
        currentControl[5], overrideOutputs);
    }
}
}

static int dllFuncsVersion = 100; // if signature is changed, change this number as well

typedef struct {
    int (*shouldOverrideOutputs)(float * ch1, float * ch2, float * ch3,
        float * ch4, float * ch5, float * ch6);
    int (*shouldResetState)(_TFmResetState2* currentState,
        double U_dot, double V_dot, double W_dot,
        double P_dot, double Q_dot, double R_dot);
}

DllFuncs toExport;
/*
 * Used to override the manual control with automatic control.
 * Called before actuation in the physics loop.
 * Arguments are pointers to floats for each channel.
 * Actuation data is expected to be from -1 to 1.
 * The channel to actuator mapping is aircraft dependent.
 * Return value of 0 uses manual control.
 * Return value other than 0 uses specified control (automatic).
 */

int envShouldOverrideOutputs(float* ch1, float* ch2, float* ch3, float* ch4, float* ch5, float* ch6)
{
    *ch1 = currentControl[0];
    *ch2 = currentControl[1];
    *ch3 = currentControl[2];
    *ch4 = currentControl[3];
    *ch5 = currentControl[4];
    *ch6 = currentControl[5];
    return overrideOutputs;
}

/*
 * Used to acquire state data.
 * Called after computing the next state in the physics loop.
 * Arguments provide access to the current state of the aircraft.
 * The function begins by fixing state information.
 * Then it enforces a write period.
 * If the period has elapsed, the data is prepared for output and written to the serial device.
 * Return value is a bit field.
 * Setting 0x1 will reset p,q,r to values set in currentState
 * Setting 0x2 will reset phi,theta,psi to values set in currentState
 * Setting 0x4 will reset u,v,w to values set in currentState
 * Setting 0x8 will reset x,y,z to values set in currentState
 */
int enveShouldResetState(_TFmResetState2* currentState, double U_dot, double V_dot, double W_dot, double P_dot, double Q_dot, double R_dot)
{
    // Check which fixation modes are enabled and update the current state accordingly.
    if (config.fixed() & 0x1) {
        config.fixedRotation(&currentState->P, &currentState->Q, &currentState->R);
    }
    if (config.fixed() & 0x2) {
        config.fixedAttitude(&currentState->phi, &currentState->theta, &currentState->psi);
    }
    if (config.fixed() & 0x4) {
        config.fixedVelocity(&currentState->U, &currentState->V, &currentState->W);
    }
    if (config.fixed() & 0x8) {
        config.fixedPosition(&currentState->x, &currentState->y, &currentState->z);
    }

    // Get current time and enforce periodic behavior
    FILETIME curr;
    GetSystemTimeAsFileTime(&curr);
    ULARGE_INTEGER currInt;
    currInt.LowPart = curr.dwLowDateTime;
    currInt.HighPart = curr.dwHighDateTime;
    if (!firstWrite || (currInt.QuadPart - lastWrite.QuadPart) >= config.outputPeriod()) {
        // A minimum of config.outputPeriod() 100ns have passed since last write
        // Update start of next period
        if (!firstWrite) {
            firstWrite = true;
            lastWrite.QuadPart = currInt.QuadPart;
        } else {
            lastWrite.QuadPart += config.outputPeriod();
        }
    }
}
// Perform unit conversions and mapping from simulator ←
// coordinate system to a configurable real world ←
// location
TFmResetState2 mapped;
config.mapAndConvert(currentState, &mapped);

// Create data sample string
std::string fUdot = to_string((float)(U_dot * M_PER_FOOT)←
);
std::string fVdot = to_string((float)(V_dot * M_PER_FOOT)←
);
std::string fWdot = to_string((float)(W_dot * M_PER_FOOT)←
);
std::string fP = to_string(mapped.P);
std::string fQ = to_string(mapped.Q);
std::string fR = to_string(mapped.R);
std::string fPhi = to_string(mapped.phi);
std::string fTheta = to_string(mapped.theta);
std::string fPsi = to_string(mapped.psi);
std::string fx = to_string(mapped.x);
std::string fy = to_string(mapped.y);
std::string fz = to_string(mapped.z);
std::string fU = to_string(mapped.U);
std::string fV = to_string(mapped.V);
std::string fW = to_string(mapped.W);
std::string wind1 = to_string(mapped.junk1);
std::string wind2 = to_string(mapped.junk2);
std::string Data = fUdot+" , " +fVdot+" , " +fWdot+" , " +
fP+" , " +fQ+" , " +fR+" , " +
fPhi+" , " +fTheta+" , " +fPsi+" , " +
fX+" , " +fY+" , " +fZ+" , " +
fU+" , " +fV+" , " +fW);

if (config.isLoggingEnabled()) {
    // Log time, state data, and actuation data if logging←
    // is enabled by user
    std::stringstream logLineSS;
    // time is in 100 nanosecond units
    logLineSS << currInt.QuadPart;
    logLineSS << "; ";
    logLineSS << Data;
    for (int i = 0; i < 6; i++) {
        logLineSS << currentControl[i];
        if (i < 5) {
            logLineSS << " ";
    }
logLineSS <<= ",";
}
}
fprintf(logFile, "%s\n", logLineSS.str().c_str);
}
if(config.isEncodingEnabled()) {
    // Encode data sample to compress
    Data.append("*");
    char * encodedData = new char[Data.size() + 1];
    std::copy(Data.begin(), Data.end(), encodedData);
    encodedData[Data.size()] = '\0';
    size_t enCount = encode(encodedData);
    if(encodedData[enCount - 1] != '\0') {
        enCount++;
        encodedData[enCount - 1] = '\0';
    }
    if(COM_init) {
        // Perform output
        serial->write(encodedData, enCount);
        outCount+=enCount;
        // printf("Output Count: %d\n", outCount);
    }
    delete[] encodedData; // Clean memory
} else {
    Data.append("\r\n");
    if(COM_init) {
        // Perform output
        serial->write(Data.c_str(), Data.length());
        outCount+=Data.length();
    }
}
}
return config.fixed();

// ///////////////////////////////////////////////////////////////////////////////////// EXTERNAL /////////////////////////////////////////////////////////////////////////////////////

#endif __cplusplus
extern "C" {
#endif
This function is called by the simulator when it starts loading for flight.

Its main purpose is to provide the addresses of the shouldResetState and shouldOverrideOutputs functions.

It also provides the opportunity to perform some initialization.

First, the user configuration file is loaded and processed. It is printed for verification purposes.

Next, a log file is created and opened if logging has been enabled by the user.

Finally, the serial device is opened as an AsyncSerial object. The receive callback function is registered at this time.

Returns a pointer to a DllFuncs struct, which contains function pointers to the enveShouldResetState and enveShouldOverrideOutputs functions.

>Returns a pointer to a DllFuncs struct, which contains function pointers to the enveShouldResetState and enveShouldOverrideOutputs functions.

__declspec(dllexport) DllFuncs* _cdecl getDllFuncs(int version)
{
    DllFuncs* rv = 0;
    printf("Inside Dll\n");
    int linesLoaded = config.load("\fsonesimexternalcontrol.config");
    printf("Loaded %d config lines\n", linesLoaded);
    config.print();
    if(config.isLoggingEnabled()) {
        char logpath[512];
        logpath[0] = '\0';
        strcat(logpath, config.logDirectory().c_str(), 511);
        char datetime[128];
        time_t rawtime;
        struct tm * timeinfo;
        time(&rawtime);
        timeinfo = localtime(&rawtime);
        strftime(datetime, 128, "%Y-%m-%d_%H-%M-%S", timeinfo);
        strcat(logpath, datetime);
        logFile = fopen(logpath, "w");
```cpp
int err = errno;
if (logFile == NULL) {
    printf("Error opening log file. %d\n", err);
}

serial = new CallbackAsyncSerial(config.serialPath(), config←
    .serialBaud());
serial->setCallback(boost::bind(&receiveCallback,_1,_2));
COM_init = true;

if (version != dllFuncsVersion)
{
    printf("Inside Dll, error, version doesn't match\n");
}
else
{
    toExport.shouldOverrideOutputs = <->
        enveShouldOverrideOutputs;
    toExport.shouldResetState = enveShouldResetState;
    rv = &toExport;
}
return rv;
```

# Listing C.8: TFmResetState2.h

```cpp
#pragma once
typedef struct _TFmResetState2
{
    // angular position
    float theta;
    float phi;
    float psi;
}
    // absolute location
```
```cpp
float x;
float y;
float z;

// velocity (body frame)
float U;
float V;
float W;

// ignore
float junk1;
float junk2;

// angular velocity rates
float P;
float Q;
float R;

} TFmResetState2;
```

---

Listing C.9: tostring.h

```cpp
#pragma once

#include <sstream>
#include <string>

std::string to_string(float val) {
    std::ostringstream os;
    os << std::setprecision(5);
    os << std::fixed;
    os << val;
    return os.str();
}

std::string to_string(int val) {
    std::ostringstream os;
    os << val;
    return os.str();
}
```
# Appendix D

User Manual

## EnvE

User Manual

April 18, 2017

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1 Introduction</td>
<td>81</td>
</tr>
<tr>
<td>D.2 Direct Connection</td>
<td>81</td>
</tr>
<tr>
<td>D.2.1 Receiving State Data</td>
<td>81</td>
</tr>
<tr>
<td>D.2.2 Sending Actuation Data</td>
<td>83</td>
</tr>
<tr>
<td>D.3 Adapter</td>
<td>84</td>
</tr>
<tr>
<td>D.3.1 Receiving State Data</td>
<td>85</td>
</tr>
<tr>
<td>D.3.2 Sending Actuation Data</td>
<td>85</td>
</tr>
<tr>
<td>D.4 Logs</td>
<td>86</td>
</tr>
<tr>
<td>D.5 Configuration</td>
<td>86</td>
</tr>
<tr>
<td>D.5.1 baud</td>
<td>87</td>
</tr>
<tr>
<td>D.5.2 encoding</td>
<td>87</td>
</tr>
<tr>
<td>D.5.3 fixed</td>
<td>87</td>
</tr>
<tr>
<td>D.5.4 fxdAtt</td>
<td>88</td>
</tr>
<tr>
<td>D.5.5 fxdPos</td>
<td>88</td>
</tr>
<tr>
<td>D.5.6 fxdRot</td>
<td>88</td>
</tr>
<tr>
<td>D.5.7 fxdVel</td>
<td>88</td>
</tr>
<tr>
<td>D.5.8 logdir</td>
<td>89</td>
</tr>
<tr>
<td>D.5.9 logging</td>
<td>89</td>
</tr>
<tr>
<td>D.5.10 map</td>
<td>89</td>
</tr>
<tr>
<td>D.5.11 period</td>
<td>89</td>
</tr>
<tr>
<td>D.5.12 serial</td>
<td>90</td>
</tr>
</tbody>
</table>
Table D.1: The default serial parameters for the simulator interface.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baud Rate</td>
<td>B115200</td>
</tr>
<tr>
<td>Character Size</td>
<td>CS8</td>
</tr>
<tr>
<td>Enable Receiver</td>
<td>Yes</td>
</tr>
<tr>
<td>Modem Control Lines</td>
<td>Ignore</td>
</tr>
</tbody>
</table>

D.1 Introduction

EnvE is a complete environment for the emulation of unmanned aerial vehicles. The heart of EnvE is the FS One Precision RC Flight Simulator. The flight simulator provides realistic and accurate modeling of the flight dynamics of numerous aircraft. The version of FS One used in EnvE has been modified to include a custom interface for use with a standalone controller, allowing for hardware-in-the-loop emulation.

D.2 Direct Connection

This section addresses the issue of connecting to the simulator directly. For information about connecting with the adapter, see Section D.3.

To close the feedback loop, your controller must begin by establishing a connection with the flight simulator. The controller must open the correct serial port and set its parameters, which vary depending on the configuration. The default parameters can be found in Table D.1. Contained in Listing D.1 is an example C code to open /dev/ttyUSB0 at a baud rate of 115200 on GNU/Linux.

D.2.1 Receiving State Data

Due to the character nature of serial communication, the simulator transmits state data as a simple string. While serial connections allow for simple communication between a number of diverse devices, they do not support high bandwidths. Because of this and the desire to support sensor streams at rates of at least 100 Hz, a simple codec is used to reduce the size of the data strings. The encoded string is sent to the controller. The controller must receive the data, decode it, and parse the floating point
Listing D.1: Example C code for connecting to the simulator.

```c
int fd;
struct termios oldtio, tio;
FILE * sim;

fd = open("/dev/ttyUSB0", O_RDWR | O_NOCTTY);
tcgetattr(fd, &oldtio);
tio.c_cflag = B115200 | CS8 | CLOCAL | CREAD;
tio.c_iflag = 0;
tio.c_oflag = 0;
tio.c_lflag = 0;
tio.c_cc[VMIN] = 1;
tio.c_cc[VTIME] = 0;
tcsetattr(fd, TCSANOW, &tio);
sim = fdopen(fd, "w+");
```

Figure D.1: Data flow control loops with and without the EnvE adapter.

(a) Data flow in basic control loop. (b) Data flow in control loop with EnvE adapter.

data from the string. It is important to note that each data sample is
separated by a null (zero) byte.

The raw message is constructed using the following printf-style format:
%.f,%.f,%.f,%.f,%.f,%.f,%.f,%.f,%.f,%.f,%.f,%.f,%.f,%.f. These fields are filled
in order with the state information described in Table D.2. The process of
receiving state data is quite simple. Example code can be found in
Listing D.2. Begin by receiving a message by using getdelim with a ’\0’
delimiter. If unbounded blocking is a concern, use select to set a timeout.
Upon receiving a message, it must be decoded using the custom decode
<table>
<thead>
<tr>
<th>State Variable</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\dot{u}$</td>
<td>Body Frame Acceleration of CG (i Axis)</td>
<td>m/s²</td>
</tr>
<tr>
<td>$\dot{v}$</td>
<td>Body Frame Acceleration of CG (j Axis)</td>
<td>m/s²</td>
</tr>
<tr>
<td>$\dot{w}$</td>
<td>Body Frame Acceleration of CG (k Axis)</td>
<td>m/s²</td>
</tr>
<tr>
<td>$p$</td>
<td>Body Frame Rotation Rate (i Axis)</td>
<td>rad/s</td>
</tr>
<tr>
<td>$q$</td>
<td>Body Frame Rotation Rate (j Axis)</td>
<td>rad/s</td>
</tr>
<tr>
<td>$r$</td>
<td>Body Frame Rotation Rate (k Axis)</td>
<td>rad/s</td>
</tr>
<tr>
<td>$\phi$</td>
<td>Roll Angle</td>
<td>rad</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Pitch Angle</td>
<td>rad</td>
</tr>
<tr>
<td>$\psi$</td>
<td>Yaw Angle</td>
<td>rad</td>
</tr>
<tr>
<td>$p_n$</td>
<td>North Position of CG (i Axis)</td>
<td>m</td>
</tr>
<tr>
<td>$p_e$</td>
<td>East Position of CG (j Axis)</td>
<td>m</td>
</tr>
<tr>
<td>$p_d$</td>
<td>Down Position of CG (k Axis)</td>
<td>m</td>
</tr>
<tr>
<td>$u$</td>
<td>Body Frame Velocity of CG (i Axis)</td>
<td>m/s</td>
</tr>
<tr>
<td>$v$</td>
<td>Body Frame Velocity of CG (j Axis)</td>
<td>m/s</td>
</tr>
<tr>
<td>$w$</td>
<td>Body Frame Velocity of CG (k Axis)</td>
<td>m/s</td>
</tr>
</tbody>
</table>

Table D.2: State data provided by EnvE.

function. Once the original message is decoded, the data points can be easily parsed using `sscanf`. Check the man pages for descriptions of functionality and behavior of these functions.

### D.2.2 Sending Actuation Data

Control is divided amongst six channels and one mode. The channel mapping is dependent on the selected aircraft. While some aircraft use all six channels, others do not. The simulator does expect a standard actuation range. An actuation value is expected to be a float in the set $[-1, 1]$. In this scheme, $-1$ and $1$ are actuation extremes. Depending on the control channel, these extremes are interpreted differently. For example, the extremes of an aileron are fully up and fully down while the extremes of a flap are even with the wing or fully down. The mode has definite behavior. Specifying a mode of 0 tells the simulator to not use the accompanying actuation data. The simulator will instead use the wired controller pad. Alternatively, specifying a mode of 1 tells the simulator to override all controls with the accompanying actuation data.
Listing D.2: Example C code for receiving state data from the simulator.

```c
// FILE * sim; From opening connection

#define DECODEDBUF_SIZE 512

char decodedBuf[DECODEDBUF_SIZE];
char * inbuf = NULL;
size_t inbufSize = 0;
size_t len;
int parsed;
ssize_t readSize;

struct sensor_data sd;

readSize = getdelim(&inbuf, &inbufSize, '\0', sim);
len = decode(inbuf, decodedBuf, DECODEDBUF_SIZE);
parsed = sscanf(decodedBuf,
"%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f",
&sd.Udot, &sd.Vdot, &sd.Wdot,
&sd.P, &sd.Q, &sd.R,
&sd.Phi, &sd.Theta, &sd.Psi,
&sd.Pn, &sd.Pe, &sd.Pd,
);
```

Once the proper outputs and mode have been computed, they can be sent to the simulator as a string. This can be done using a simple `fprintf` call. The data starts with the first channel and is followed sequentially up to and including the sixth channel. The last item to be sent is the mode followed by a carriage return and a line feed. An example of this can be seen in Listing D.3.

D.3 Adapter

As seen in Section D.2, the direct connection technique relies heavily on string parsing and generation. This can be problematic for low level controllers that have constrained resources and minimal libraries. To alleviate these issues, an emulation adapter is provided. This adapter takes on the responsibility of parsing and decoding sensor data and generating actuation data strings.
Listing D.3: Example C code for sending actuation data to the simulator.

```c
// FILE ∗ sim; From opening connection
int mode;
float output[6];
int printed;

// Compute output
printed = fprintf(sim,
    "%f,%f,%f,%f,%f,%f,%d\r\n",
    output[0],
    output[1],
    output[2],
    output[3],
    output[4],
    output[5],
    mode);
```

D.3.1 Receiving State Data

The emulation adapter is programmed to act as an I²C slave at address 0x2. Upon receiving a read request, the adapter replies with 60 bytes of data. The data is a set of 15 sequential floats ordered as in Table D.2.

D.3.2 Sending Actuation Data

Sending actuation data is incredibly simple using the emulation adapter. Low-level controllers can simply output PWM signals as they would normally. Instead of connecting the PWM lines to servos, simply connect them to the adapter’s input ports. At a rate of 100 Hz, the adapter reads PWM input channels 1 through 6 inclusive. The adapter immediately maps these inputs to the appropriate range, produces a formatted string, and sends it to the simulator. The adapter expects PWM signals in the range of 1000 to 2000 µs. Any signals outside of this range will be modified to be within the range.
D.4 Logs

The flight simulator can be configured to log state and actuation data to a file. To enable logging, see Section D.5.9. To configure the directory in which logs are saved, see Section D.5.8. When the simulator starts flight mode, a log file is created in the log directory with the name \%Y-%m-%d-%H-%M-%S, where \%Y is the four digit year, \%m is the two digit month, \%d is the two digit day of the month, \%H is the two digit hour using a 24-hour clock, \%M is the two digit minute, and \%S is the two digit second. Data is logged at the rate configured in Section D.5.11. Each log entry is on its own line. The first element of the log entry is the current time in 100 nanosecond units. The time is followed by a semicolon. Then a comma separated list of the data points in Table D.2 is given with a terminating asterisk. Finally, the six output channels are listed in a sequentially increasing, comma separated list.

D.5 Configuration

EnvE has a number of configuration options to tweak interaction with the flight simulator. These options can be modified in the plain text file found at C:\fsone_simexternalcontrol_config. The file is interpreted by some very basic code, which has the unfortunate effect of forcing a strict format. The file is processed line by line from the beginning to end. The start of each line is checked for the existence of a variable name. If one is found, the line is parsed. The expected format after the variable name is an immediate equals sign followed by the value. In the case of vector types, the components are comma-separated. No whitespace is expected within the line. Since the lines are processed sequentially, the last value in the file is used if a variable is set multiple times. After the file is read, the simulator prints the configuration to the console so the user can verify proper configuration. The following is an example configuration with default values. Note that the order of variables does not matter.

```
baud=115200
encoding=1
fixed=0
```
fxdAtt=0,0,0
fxdPos=-111.252,-121.920,-10
fxdRot=0,0,0
fxdVel=0,0,0
logdir=C:\
logging=0
map=-111.252,-121.920,0,0
period=100000
serial=COM1

D.5.1 baud
Type Integer
Default Value 115200
Description This variable allows the user to specify the baud rate for the serial connection between the simulator and the flight controller.

D.5.2 encoding
Type Integer
Default Value 1
Description This variable allows the user to enable and disable use of 4-bit encoding. Disabling encoding decreases possible bandwidth, but increases readability.

D.5.3 fixed
Type Integer
Default Value 0
Description This variable is a bit field that allows certain portions of the aircraft’s state to be fixed. Setting bit 0 (0x1) fixes the rotation rates \((p, q, r)\). Setting bit 1 (0x2) fixes the attitude \((\phi, \theta, \psi)\). Setting bit 2
(0x4) fixes the velocity \((u, v, w)\). Setting bit 3 (0x8) fixes the position \((p_n, p_e, p_d)\).

**D.5.4 fxdAtt**

**Type** Float[3]

**Default Value** \{0, 0, 0\}

**Description** This variable allows the user to set a fixed attitude \((\phi, \theta, \psi)\) for the aircraft. Note that this attitude is raw and represents a state before the map is applied.

**D.5.5 fxdPos**

**Type** Float[3]

**Default Value** \{-111.252, -121.920, -10\}

**Description** This variable allows the user to set a fixed position \((p_n, p_e, p_d)\) for the aircraft. Note that this position is raw and represents a state before the map is applied.

**D.5.6 fxdRot**

**Type** Float[3]

**Default Value** \{0, 0, 0\}

**Description** This variable allows the user to set a fixed rotation rate \((p, q, r)\) for the aircraft.

**D.5.7 fxdVel**

**Type** Float[3]

**Default Value** \{0, 0, 0\}

**Description** This variable allows the user to set a fixed velocity \((u, v, w)\) for the aircraft.
D.5.8 logdir

Type String

Default Value C:\

Description This variable allows the user to specify a path to a directory in which to store logs. For more information on logging in general, see Section D.4.

D.5.9 logging

Type Integer

Default Value 0

Description This variable allows the user to enable and disable logging. Setting this variable to a non-zero integer enables logging. For more information on logging in general, see Section D.4.

D.5.10 map

Type Float[4]

Default Value \{-111.252, -121.920, 0, 0\}

Description This variable allows the user to modify the origin and orientation of the simulator world. This is most useful for mapping the simulator world to the real world. The values are north, east, down, and rotation. An example to map to Monticello is \{4435503.52, 367686.51, -207.064, 2.333851\}. The default values cause no changes.

D.5.11 period

Type Long

Default Value 100000
Description This variable allows the user to modify the period between state data outputs. The units of this variable are 100ns.

D.5.12 serial

Type String

Default Value COM1

Description This variable allows the user to specify the path to the serial device used for communication with the controller. This is most likely a COM port.