6-14-2007


Daniel Ernst
University of South Florida

Follow this and additional works at: https://digitalcommons.usf.edu/etd

Part of the American Studies Commons

Scholar Commons Citation
https://digitalcommons.usf.edu/etd/699

This Thesis is brought to you for free and open access by the USF Graduate Theses and Dissertations at Digital Commons @ University of South Florida. It has been accepted for inclusion in USF Tampa Graduate Theses and Dissertations by an authorized administrator of Digital Commons @ University of South Florida. For more information, please contact digitalcommons@usf.edu.
Development of Research Platform for Unmanned Vehicle Controller Design,
Evaluation, and Implementation System: From MATLAB to Hardware Based Embedded System

by

Daniel Ernst

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Kimon Valavanis, Ph.D.
Miguel Labrador, Ph.D.
Wilfrido Moreno, Ph.D.

Date of Approval: June 14, 2007

Keywords: Unmanned systems, SIMULINK, microcontroller, autopilot, automation

© Copyright 2007, Daniel Ernst
Dedication

To my loving family and friends who have all offered constant support.
Acknowledgments

A special thanks to everyone in the USF robotics lab and of course, my major professor Kimon Valavanis, who made writing this thesis possible and provided continuous support. Also, to Jeff Craighead, who built aircraft models in the X-Plane simulation that allowed testing of controllers.

This thesis was also supported in part by an ONR Grant N00014-04-10-487; a U.S. Navy Coastal Systems Station (now called NSWC-Panama City) Grant N61331-04-8-1707; and a U.S. DOT through the USF CUTR Grant 2117-1054-02.
Table of Contents

List of Tables iii

List of Figures iv

Abstract vi

Chapter 1 Motivation 1
1.1 Problem Statement 1
1.2 Proposed Solution 2
1.3 Summary of Contributions 4
1.4 Thesis Outline 5

Chapter 2 The Design Process 6
2.1 MATLAB/SIMULINK to C Conversion 8
2.2 Customizations 8
2.3 Device Selection and Building 10

Chapter 3 Assembly Code Generation 13
3.1 C Code Modification 14
3.2 Programming the Microcontroller 15

Chapter 4 X-Plane Simulation 16
4.1 X-Plane UDP Communication 16
4.2 X-Plane Exported UDP Data 17
4.3 C Code Interpretation 18
4.4 Function Conversions 19
4.5 SIMULINK S-Function Implementation 19
4.6 X-Plane Extra Features 20

Chapter 5 UAV Implementation 28
5.1 Hardware Selection and Controller Design 28
5.2 C Code Generation 29
5.3 Assembly Code Generation 29
5.4 Testing with X-Plane 29

Chapter 6 UGV Implementation 33

Chapter 7 Conclusions and Discussion 35
Chapter 8  Results and Future Work  37

References  38

Appendices  39
   Appendix A: Overall System Board Under Development  40
   Appendix B: System Board Schematic Layout  42
   Appendix C: UGV Controller Design  43
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Flight Simulation Comparison</td>
<td>3</td>
</tr>
<tr>
<td>Table 2</td>
<td>General Hardware Conversion Schemes</td>
<td>5</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1: Block Diagram of the Automated Process 7
Figure 2: Real-Time Workshop Setup 9
Figure 3: Microcontroller Selection 10
Figure 4: Overall Real-Time Workshop Conversion 11
Figure 5: PID Controllers 11
Figure 6: Fuzzy Logic Controllers 12
Figure 7: LQR Controllers 12
Figure 8: Files Needed for Conversion 14
Figure 9: Overall Assembly Generation 15
Figure 10: Raptor 90 Model 21
Figure 11: Raptor 90 Hovering in Simulator 21
Figure 12: Raptor 90 in Slow Forward Flight 22
Figure 13: Yamaha R-Max Model 22
Figure 14: The $X$-Plane Communication Screen 23
Figure 15: $X$-Plane IP and Socket Interface Selection 23
Figure 16: Example Packet Sent from $X$-Plane 24
Figure 17: The $SIMULINK$ Library Browser 24
Figure 18: The S-Function Block Libraries Tab 25
Figure 19: The S-Function Block Data Properties and $Initialization$ Tabs 25
Figure 20: Items Included in the C Code Under the Outputs Tab
Figure 21: Forces Acting on the Raptor 90 Control Surfaces
Figure 22: X-Plane/SIMULINK Communication
Figure 23: Conversion Steps
Figure 24: Exported Controller Variables
Figure 25: The Actual Implementation in SIMULINK
Figure 26: Small UGV for Implementation
Figure 27: Autopilot Block Diagram
Figure 28: Autopilot Board Photo
Figure 29: Autopilot Board Schematic
Figure 30: UGV MATLAB Controller

Daniel Ernst

ABSTRACT

Unmanned aerial vehicles and unmanned ground vehicles, or UAVs and UGVs respectively, currently perform a large variety of missions usually centered around reconnaissance. Because the platforms may vary for a particular type of mission—everything from small unmanned airplanes and remote control vehicles to large vehicles such as the Yamaha R-MAX helicopter and Hummer—flight and navigation controllers must be changed to allow proper control of the selected platform. Currently, controllers are designed and tested in MATLAB/SIMULINK, but then rewritten in C or Assembly for a specific target platform. When designing controllers in a programming language, changes are often tedious, so producing a working controller takes considerable time.

MATLAB/SIMULINK provides a GUI interface and SIMULINK provides excellent testing capabilities, so changes may be quick and easy. However, no automated method for converting a simple controller, such as a PID for example, from MATLAB to implementation on a microcontroller has been presented in literature.
To implement current in-house controllers designed in MATLAB/SIMULINK, a system consisting of Real-Time Workshop and a C compiler has been used to produce assembly code for a target microcontroller. To aid in verification of the controllers and C code produced by Real-Time Workshop targeted toward aerial platforms, an interface for the controllers in SIMULINK and a flight simulator (X-Plane) has been created. Thus the overall system allows for rapid changes and implementation on a variety of platforms as well as plug-in/plug-out capabilities in the field for diverse missions. Functionality and diversity of the system is demonstrated through testing of PID VTOL controllers in SIMULINK with X-Plane as well as implementation of UGV controllers onboard a small radio controlled truck.
Chapter 1

Motivation

This research has been motivated by the challenge to optimize, standardize, and automate as much as possible the process of unmanned vehicle controller design, evaluation, validation and verification, followed by actual hardware controller implementation on the vehicle. The presented approach is kept as nonspecific as possible, so it is applicable to any unmanned vehicle with minor modifications that depend on the specific microcontroller or processor used. To prove aerial vehicle controller testing capabilities a set of PID VTOL controllers are utilized from within SIMULINK in conjunction with a simulator called X-Plane. In addition, the automated code conversion process is proven to work adequately through implementation of a simple waypoint controller on an unmanned ground vehicle (UGV).

1.1 Problem Statement

The rationale behind the attempt to ‘automate’ controller design, evaluation, validation and verification is manyfold; it stems from the central objective to utilize the ‘plug in – plug-out’ concept of mission specific controllers. As such, given that unmanned vehicles are used in a multitude of applications requiring different controllers and mission profiles, rather than hard coding everything a-priori, it is deemed better to use application specific (low level) and (overall) mission controllers. Moreover, depending on a specific mission, flight patterns may change following non-aggressive or aggressive modes of operation that dictate different vehicle models (linear, linearized,
nonlinear and approximations to linearization). For example, for non-aggressive flights, it is customary to follow a ‘small angle approximation’ that results in all sine and cosine functions being 0 and 1, respectively. Further, controllers are designed using mostly MATLAB/SIMULINK and then implemented separately in code. But when designing controllers in a programming language, changes are often tedious, so deriving a working controller requires not only considerable time, but it is also difficult to modify. Additionally, when designing controllers for aerial vehicles, the controllers must be tested before implementation because any failures may be catastrophic. In short, there is not a method that introduces a series of concrete steps to convert a controller (such as a PID, PD, Fuzzy Logic or an LQR) from MATLAB to implementation on a microcontroller chip. This paper presents a safe method allowing for graphical controller design, testing in near real time, automated generation of C code, validation of code generation, and assembly code generation.

1.2 Proposed Solution

MATLAB and SIMULINK, produced by Mathworks, are industry standard tools used for the design of controllers that have proven successful in a wide variety of applications, so these applications were chosen as the design framework of controllers for the target platform. X-Plane, a commercial, closed source, flight simulation package produced by Laminar Research, provides near real time verification of the controller operation in SIMULINK as well as validation testing capabilities once code has been generated through Real-Time Workshop. Table 1 shows the different flight simulation programs compared and why X-Plane was the only program that met all necessary qualifications (an X denotes the qualification was met). X-Plane provides extremely
accurate flight models, system failures, real world weather interaction, airfoil design [6],
and allows for input/output from external sources through UDP communication.

Table 1: Flight Simulation Comparison

<table>
<thead>
<tr>
<th></th>
<th>Ease of Use</th>
<th>Data Manipulation</th>
<th>FAA Certified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Flight Simulator</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-Plane</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Flight Gear</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

The proposed approach follows the standard practice to utilize
MATLAB/SIMULINK and related toolboxes as the design framework. It also takes
advantage of the fact that MATLAB/SIMULINK provides a GUI interface with
SIMULINK offering excellent testing capabilities. Controller design in
MATLAB/SIMULINK is followed by automatic conversion from MATLAB to code
generation and optimization for particular types of processors using Real-Time
Workshop. If the target embedded system is a microcontroller, this is then followed by a
‘C to Assembly’ conversion to produce assembly code. MATLAB/SIMULINK controllers
and C code produced by Real-Time Workshop, are verified, validated and optimized first
using a flight simulator, X-Plane, before actual testing on an unmanned vehicle and
actual implementation on a chip and printed circuit board.

To ensure wide applicability to target hardware as well as utilization by
individuals with limited or no background in programming controllers, conversion steps
have been kept as straightforward and automated as possible. Table 2 shows this
applicability to a wide variety of hardware devices as well as the comminlarity between
the conversion schemes.
1.3 Summary of Contributions

Unlike other research controller testing suites, this system utilizes a \textit{MATLAB/SIMULINK} code conversion process as well as \textit{X-Plane}, to implement code safely on almost any type of platform as long as the hardware can support the controllers. Many systems demonstrate capabilities to design and test controllers within \textit{SIMULINK}, but they usually focus on testing within the \textit{SIMULINK} environment posing two issues: \textit{SIMULINK} timing is not real time and often the environments for controller testing purposes are created for the purpose of testing a specific controller. Often these \textit{SIMULINK} environments are not as accurate as software packages designed for simulation of that particular environment. For example, \textit{X-Plane 8.20} is an FAA approved simulator with an extremely complex physics model to interact with models produced by the \textit{X-Plane} model designer whereas a custom simulation in \textit{SIMULINK} would not have the same level of realism. Furthermore, because \textit{X-Plane} communicates directly with \textit{SIMULINK}, sensor modeling can be implemented within the \textit{SIMULINK} model by modifying incoming \textit{X-Plane} data values to provide proper noise and drift to the controllers. Also, when implementing controllers, this system, unlike other systems, allows a user to validate code generation from Real-Time Workshop to ensure the code produced actually functions properly thereby saving development time. Furthermore, the research testing suite provides capabilities for producing robust controllers for aerial vehicles by providing system failures, real world weather interaction, air traffic controller interaction as well as other vehicle interaction through \textit{X-Plane}. 

4
1.4 Thesis Outline

This thesis is designed to provide a step by step process for implementation of controllers designed in *SIMULINK* on an unmanned vehicle. Thus, the paper assumes a controller has been designed for a target vehicle and starts by describing the process for converting the controller to C code. Next, the generated code is modified for the particular hardware that will be used on board the unmanned vehicle and steps are shown to convert the C code to assembly. If the target system is an aerial platform (or ground vehicle for basic function tests), the simulator X-Plane can be used in conjunction with SIMULINK to test the controllers before implementation, so a description for utilizing X-Plane with SIMULINK is provided. Afterwards, the controllers are implemented on board the unmanned vehicle.

**Table 2: General Hardware Conversion Schemes**

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Design/Verification</th>
<th>C Code Generation</th>
<th>Assembly Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microchip DSPs &amp; Microcontrollers</td>
<td>MATLAB/SIMULINK</td>
<td>Real-Time Workshop</td>
<td>PIC-C Compiler/MPLAB</td>
</tr>
<tr>
<td>Texas Instruments DSPs &amp; Microcontrollers</td>
<td>MATLAB/SIMULINK</td>
<td>Real-Time Workshop</td>
<td>Code Composer</td>
</tr>
<tr>
<td>XILINX DSPs</td>
<td>MATLAB/SIMULINK</td>
<td>Real-Time Workshop</td>
<td>System Generator for DSP</td>
</tr>
<tr>
<td>Computer System Implementation</td>
<td>MATLAB/SIMULINK</td>
<td>Real-Time Workshop</td>
<td>------</td>
</tr>
</tbody>
</table>
Chapter 2

The Design Process

In the most general case, a complete design process starts with selection of hardware before controllers are designed in *MATLAB*. As such, given the class of miniature unmanned VTOL vehicles, appropriate sensors (GPS – DGPS, IMU and cameras) are first chosen as well as on-board computers and processors, considering processing capabilities, size and cost, payload and power restrictions. It is also imperative that chosen hardware components have a good support base including a compiler to convert C to assembly. Assuming the above has been decided, Figure 1 gives a block diagram of the steps of the overall proposed automated process.

Regarding controller design in *MATLAB*, as known, PID controllers are self contained in one *SIMULINK* model file, the Fuzzy Logic controllers contain a *SIMULINK* model file and three fuzzy inference system files, and the LQR controllers are implemented in a *SIMULINK* model file with several *MATLAB* script files.
Observing Figure 1, controllers are first designed using MATLAB; they are tested using SIMULINK and they are initially validated and verified using X-plane. The process is repeated and controllers are refined. Once this step is complete, conversion to C using
MATLAB’s Real-Time Workshop follows. The generated C code is for a target microcontroller or DSP chip. Additional validation and verification using X-plane follows, until the generated C code satisfies set requirements. The next step is C to Assembly before the controller is implemented on the vehicle.

2.1 MATLAB/SIMULINK to C Conversion

To convert from MATLAB to C, an environment called Real-Time Workshop provides automatic code generation. In addition to providing the automatic C code generation, Real-Time Workshop also provides several ways to optimize the controllers for particular types of processors. Once a set of controllers are opened in SIMULINK, the file must be “built” using Real-Time Workshop. Before building, however, several customizations must be made. First, Real-Time Workshop must be selected under the Configuration menu. Next, the Solver option in the left box is chosen, and under Solver Options, the Type box must be changed to Fixed-Step for an embedded target. Because the controllers are implemented on a microcontroller, the proper .tlc file will be selected—information for proper selection can be determined from the designer’s reference [2]. In the RTW system target value, type ert.tlc, which causes Real-Time Workshop to produce code targeted for embedded systems. Once this filename has been entered, the options under Build process should change. However, if they don’t, change the Template makefile option to ert_default_tmf.

2.2 Customizations

The makefile option allows for further customization for the processor such as conversion for microcontroller enabled floating point or integer operations (Figure 2). If the PID controllers contain floating point operations, but the main microcontroller does
not have floating point capabilities, the default makefile should be selected. If the controllers created in MATLAB/SIMULINK are created utilizing hardware not present in the particular microcontroller, errors will occur when trying to generate the C code for that particular controller. Thus when converting controllers with floating point operations for a microcontroller that does not contain a floating point unit, the fixed point tlc file can not be chosen.

Figure 2: Real-Time Workshop Setup
2.3 Device Selection and Building

Next, the Configuration menu must be opened and hardware implementation selected. The pull-down menu next to Device type contains optimizations for various processors and microcontrollers (Figure 3). Once this has been selected, Real-Time Workshop is selected again from the selection menu and the Build button is pressed to start building the C files. The only major difference between the way the controllers are implemented occurs in the base step: the conversion from Real-Time Workshop to C. While the PID controllers present no problem in conversions, more complex designs such as Fuzzy Logic or implementation of MATLAB script files require extra time in getting them to work properly together. Before converting the more advanced controllers, attempt to fully implement the PID controllers as these are the easiest to work with and provide a basis to implementing more complex controllers. Figures 5, 6, and 7 show some common controllers and the steps necessary to convert these types of controllers.

![Figure 3: Microcontroller Selection](image)
Figure 4: Overall Real-Time Workshop Conversion

Figure 5: PID Controllers
Notice how all three systems are similar in the way the conversion process takes place as this is critical to keeping the conversion process automated between different types of controllers. The key element in the entire process is the C code produced by Real-Time Workshop because this toolbox must interpret any separate script files and fuzzy inference system files within MATLAB as well as produce code that can be interpreted by a C to Assembly compiler.
Chapter 3

Assembly Code Generation

Once the build is completed, *Real-Time Workshop* may be closed and a target compiler for the particular hardware may be opened to import the files. *Real-Time Workshop* produces several different files that attempt to tie the C files to *MATLAB*, so these files must be imported into the hardware target compiler for proper handling. Figure 4 shows files that were imported to a compiler for conversion to assembly to provide an idea of which files need importing from the *Real-Time Workshop* output directory. In addition to the files located in the *Real-Time Workshop* output directory, some data structures exist that must be interpreted through the `tmwtypes.h` and `rtwtypes.h` files (Figure 8). In the newest versions of MATLAB—version 7.1—a different file is needed, `rtlibsrc.h`. While these files are needed by the compiler, *Real-Time Workshop* does not output these files to the same directory as the rest of the files. Thus, these files must be located in the *MATLAB* directory and copied into the *Real-Time Workshop* output directory, and then imported into the compiler—Figure 9 shows the flow control conversion. Now, all files are present for compilation. While the number of files and amount of code may seem enormous for a PID controller, modern compilers simplify and remove any unnecessary variables—this is critical for implementation on a microcontroller.
3.1 C Code Modification

Next, modifications must be made to the C files to ensure proper implementation on the microcontroller. First, a proper schematic of the autopilot board should provide the pins that the sensors are connected to. Once these pins are determined, the pins are assigned on the microcontroller to the variable names. In addition to providing pin assignments, the outputs from the sensors may need to be converted to the proper format to be handled by the microcontrollers. While this conversion of data will vary depending on the design of the controllers and the layout of the PCB, most data conversion control of an unmanned vehicle will include a GPS parser, and conversion functions for the IMU and barometric pressure sensor (aerial platforms). If more than one microcontroller is utilized in the hardware, sensory input to the microcontroller may already be properly formatted and only timing should be dealt with (if this isn’t handled in the MATLAB/SIMULINK Controllers.)
3.2 Programming the Microcontroller

Once these conversions are complete, the assembly files can be generated and the chip programmed. This completes the automated controller conversion process, but if the controllers are targeted toward any type of UAV, they can be tested with *X-Plane* using the steps outlined in the next sections.

![Figure 9: Overall Assembly Generation](image)
Chapter 4

X-Plane Simulation

To enable testing of controllers developed in MATLAB and ensure proper conversion from MATLAB to C code, an aircraft simulation environment was utilized. Similar to Microsoft’s Flight Simulator, X-Plane provides extremely accurate flight models—accurate enough to be used to train pilots [5] and [6]—and also allows external communication as well as airfoil design. Unlike options such as Microsoft Flight Simulator and Flightgear, however, X-Plane allows input and output from an external source. As noted in [3], X-Plane also provides future capabilities that UAVs will need including navigation markers, changing weather conditions, and air traffic control communication. Figure 10 shows a Raptor 90 model created in the X-Plane plane maker and figures 11, 12, and 13 show some models—Raptor 90 and Yamaha R-Max—inside of the simulator. The next several sections will discuss the design process (seen in Figure 22) for communicating with X-Plane.

4.1 X-Plane UDP Communication

X-Plane uses UDP communication to send and receive data packets which allows changes to various values within X-Plane, but UDP protocol has its strengths and weaknesses. Over a distant network connection, UDP may be unreliable because no error detection exists in the packets; however, UDP is extremely fast [7]. X-Plane is able to dump up 50 frames per second across a local network—this has an important impact on the operation and simulation of many controllers because they require sufficient update
speed to operate correctly. *X-Plane* offers a large variety of values that can be changed including control of the aircraft as well as causing in-flight failures. To select which data items to export to SIMULINK, *X-Plane* provides an easy to use checkbox interface (Figure 14). The sections below describe the interface setup between *X-Plane* and SIMULINK, but because *X-Plane* is not open-source, it will be necessary to become familiar with the UDP documentation [4].

4.2 *X-Plane* Exported UDP Data

*X-Plane* has the ability to send and receive a large number of parameters allowing controllers to be designed using a variety of sensor readings. When the *Inet 2* tab is opened, the IP address of the computer containing the controllers is entered as well as the proper ports (Figure 15). Afterwards, the *Data Set* tab is chosen and the values required by the controllers are selected. *X-Plane* now begins sending the data to the destination IP and port. In addition, *X-Plane* will wait to receive packets on the other IP and port specified. In the *Data Tab*, the speed at which the data will exported is controlled by increasing or decreasing the number of frames per second—a range from 0 to 50. To send the data, a UDP packet is formed consisting of the string of characters “DATA” followed by an integer, and then the data items selected in the screen are attached in increasing order of the data item numbers to the packet. For ease of use, each data item consists of one integer (the number specified in the output screen) and 8 float values. The proper index value for a particular data item selected may be determined by displaying the selected values to the screen—selecting one of the four checkboxes in the data output dialog box. The index value for each data item begins with 0. For example, *X-Plane* data item number 0 is the frame rate, and the frame rate data consists of 3
different values which are indexed by 0, 1, and 2. For each item, the data structure looks as follows:

```c
{
    int=item number
    float[8]=values within the item
}
```

Once, all the data value selections have been assembled into the packet, it is then sent to the destination IP and port in network byte order (Figure 16). Notice that after the header information, each item selected to be output from X-Plane contains 36 bytes.

4.3 C Code Interpretation

Currently, X-Plane runs on a Windows XP machine, so Winsock was utilized in the C code to set up the client/server socket communication. In this scheme, X-Plane is considered to be the server and the client is the machine containing the controllers. Thus, socket communication must first be implemented in the C code to allow communication.

Next, the client waits to receive the output from the server, and once it receives a packet, the received data will be placed in a buffer. The client now has an X-Plane packet stored, but the data can not be used because it is in network byte order. To fix this, the bytes of each data value were swapped and the readable values stored in a two dimensional array for easy access. Next, another two dimensional array was created to mirror the first, and any data alterations were made to this array. Thus, as controllers generate new values for the simulator, variables inside the code will receive the data and change the values in the second two dimensional array so that the original data will not be modified. Updated values are now ready to be sent to X-Plane, so the new packet is assembled with a header of “DATA,” and the two dimensional array is transformed into a one-dimensional array.
However, before sending the data, each byte must be converted to network byte order by again swapping the individual bytes inside each data value.

4.4 Function Conversions

In addition to the main function, other functions may need to be created to handle conversion of values from X-Plane to a format accepted by the controllers. For example, each helicopter in X-Plane has a different collective range (because of the varying degrees of collective pitch for each particular helicopter), so a global declaration may be necessary and a normalization function created. Thus, when a new helicopter is tested, only the global value must be changed.

4.5 SIMULINK S-Function Implementation

To implement the C code into a SIMULINK block for interaction with the controllers, the S-Function builder was used. The S-Function builder is included with SIMULINK and can be found in the User-Defined Functions (Figure 17) category of the Library Pane. Inside the block, several items must be added. Since socket communication through Winsock is used, the appropriate library must be included in the Library Box under the Libraries Tab. Additionally, any header or C files must also be included in the Includes box (Figure 18). Any extra functions used by the main function must be included in this pane because SIMULINK will look to these files for any function calls. Next, the I/O ports must be created in the Data Properties tab which this is fairly straight forward and will vary depending on the controller built. Be sure to set the Sample mode to inherited in the initialization tab to allow the block to sample as quickly as the information is received from X-Plane. Figure 20 shows the I/O ports and sampling mode.
Afterwards, the main C function should be implemented into the S-Function block. Because updates in *SIMULINK* occur at each time step, any infinite loops must be eliminated or the block will run continuously and the time step within *SIMULINK* will never be updated. Also, the main() function encapsulation must be removed because *SIMULINK* does not recognize functions within the Outputs tab. The code is entered in the Outputs tab dialog box and should consist only of those items shown in Figure 20. Once the code has been entered, an interface between the *SIMULINK* I/O ports and the internal C Code variables must be created.

4.6 *X-Plane* Extra Features

*X-Plane* was chosen because it offers a vast amount of model information and flexibility for future development. For example, Figure 21 shows various forces acting on the different control surfaces of the helicopter—the green lines show a 22 knot wind (the determined maximum wind speed to be able to fly the Raptor 90 model safely). For flight failures, *X-Plane* offers the ability to fail GPS, control surfaces such as left roll, right roll, pitch up, pitch down, yaw left, yaw right, roll trim, pitch trim, yaw trim, control throttle jam minimum, maximum, and current, engine failure, engine fire, and engine mixture. Because *X-plane* has the ability to communicate with multiple aircraft, swarms can be simulated to see how the vehicles will interact with each other. A host of other features are available and can be viewed in UDP documentation [4].
Figure 10: Raptor 90 Model

Figure 11: Raptor 90 Hovering in Simulator
Figure 12: Raptor 90 in Slow Forward Flight

Figure 13: Yamaha R-Max Model
<table>
<thead>
<tr>
<th>Data Set</th>
<th>Data Set 2</th>
<th>Data Set 3</th>
<th>Data Set 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>02</td>
<td>02</td>
<td>02</td>
<td>02</td>
</tr>
<tr>
<td>03</td>
<td>03</td>
<td>03</td>
<td>03</td>
</tr>
<tr>
<td>04</td>
<td>04</td>
<td>04</td>
<td>04</td>
</tr>
<tr>
<td>05</td>
<td>05</td>
<td>05</td>
<td>05</td>
</tr>
<tr>
<td>06</td>
<td>06</td>
<td>06</td>
<td>06</td>
</tr>
<tr>
<td>07</td>
<td>07</td>
<td>07</td>
<td>07</td>
</tr>
<tr>
<td>08</td>
<td>08</td>
<td>08</td>
<td>08</td>
</tr>
<tr>
<td>09</td>
<td>09</td>
<td>09</td>
<td>09</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Figure 14: The X-Plane Communication Screen

Figure 15: X-Plane IP and Socket Interface Selection
Figure 16: Example Packet Sent from X-Plane

Figure 17: The SIMULINK Library Browser
Figure 18: The S-Function Block **Libraries Tab**

Figure 19: The S-Function Block **Data Properties and Initialization Tabs**
Figure 20: Items Included in the C Code Under the Outputs Tab

Figure 21: Forces Acting on the Raptor 90 Control Surfaces
Figure 22: X-Plane/SIMULINK Communication
Chapter 5

UAV Implementation

While not currently implemented on the helicopters due to controller re-design, this chapter will discuss conversion of controllers developed in-house for small autonomous helicopters with limited payload, power, and processing capabilities. The conversion provides proof of concept for code conversion and proves working capabilities of the *X-Plane* simulator. Because controller design for helicopters is extremely complex, a simple waypoint navigation controller was created and implemented on a small electric UGV with limited payload, power, and processing capabilities. Also, for provability, the same PCB and microcontroller used for the VTOL is utilized on the small UGV.

5.1 Hardware Selection and Controller Design

The hardware chosen for the implementation is an autopilot printed circuit board (PCB) that includes three different microcontrollers manufactured by *Microchip*. One microcontroller controls inputs and outputs to servos and provides a safety by allowing control of the helicopter to be switched between the autopilot board and the transmitter. A second microcontroller interfaces with the GPS module on the PCB and the third microcontroller interfaces with the IMU, GPS, and barometric pressure sensor. *Microchip* produces a wide variety of microcontrollers and microcontroller tools; in this case a *PIC-C* compiler generates the assembly code [1]. Figure 23 offers a more detailed system overview of the involved steps considering three controller designs.
5.2 C Code Generation

Once the controllers were designed, each controller was tested within SIMULINK to determine correct operation for the target vehicle platform. During the conversion from MATLAB to C, all techniques were straightforward as described in Section III above except that for the sample board, no Microchip microcontrollers existed in the target list, so the 8-bit generic processor was selected (Figure 3).

5.3 Assembly Code Generation

The steps outlined in Section IV for assembly code generation were followed using the CCS PIC C Compiler targeted for Microchip PIC 18XXX microcontrollers. This allows easy generation of the assembly language for specific types of microcontrollers without tediously programming necessary changes in Assembly. Before the assembly code can be generated, pins are assigned to the variables shown in Figure 24 and all necessary conversion functions are implemented. Afterwards, the assembly code is generated and implemented on the PIC 18F4620. The final size of the assembly code file was approximately 50KB, so the controllers are easily implemented on the PIC microcontroller.

5.4 Testing with X-Plane

For the controllers that were designed for aerial vehicles, the X-Plane simulator was utilized to avoid spending time re-implementing the controllers many times to get the controllers tuned properly. The X-Plane UDP block is placed in SIMULINK with the PID controllers in Figure 25. The target platform for integration of the controllers is a Raptor 90, so a model was created and tested with the controllers until it could perform non-aggressive flight appropriately in the simulator. Later testing will involve aggressive
flight and adapting to flight failures. Inputs and outputs for the controllers will change depending the type of the controller and for what it is used for, so in the C code, the needed variables are listed at the top along with the number of variables needed [4]. Thus, the code can be easily changed to accommodate changes to the controllers. In addition the controllers utilize radians rather than degrees, and the controllers output a collective value between -1 and 1 while X-Plane accepts a value between 0 and 16 (depending on the helicopter), special functions were created in a separate C file to normalize the collective input to X-Plane and convert degrees to radians. In addition, a global variable was created for the collective to allow easy switching between types of helicopters. Through testing, it was discovered that the PID controller could not handle sensor drift (which is simulated in sensory outputs by X-Plane), so the controllers are currently under re-design to be able to handle this. In addition to implementing UDP communication between SIMULINK and X-Plane in C, a Java interface was created allowing for portability between systems. This interface will be used for the C code generated by Real-Time Workshop for correctness.
Figure 23: Conversion Steps

```c
// Model ODE1_INTG
#define ODE1_INTG

/* ODE1 Integration data */
typedef struct IntgData_tag {
    real T = [ ]; /* derivatives */
} ODE1_IntgData;

/* External inputs (root import signals with auto storage) */
typedef struct _ExternalInputs_OprControllers_tag {
    real I roll;   /*<Root>/roll */
    real I roll_sp; /*<Root>/roll_sp */
    real I roll_trin; /*<Root>/roll_trin */
    real I pitch; /*<Root>/pitch */
    real I pitch_sp; /*<Root>/pitch_sp */
    real I pitch_trin; /*<Root>/pitch_trin */
    real I yaw; /*<Root>/yaw */
    real I yaw_sp; /*<Root>/yaw_sp */
    real I height; /*<Root>/height */
    real I height_sp; /*<Root>/height_sp */
    real I pedal_trin; /*<Root>/pedal_trin */
    real I height_trin; /*<Root>/height_trin */
} ExternalInputs_OprControllers;

/* External outputs (root outputs fed by signals with auto storage) */
typedef struct _ExternalOutputs_OprControllers_tag {
    real I sig_lateral; /*<Root>/sig_lateral */
    real I sig_longitudinal; /*<Root>/sig_longitudinal */
    real I sig_pedal; /*<Root>/sig_pedal */
    real I sig_collective; /*<Root>/sig_collective */
} ExternalOutputs_OprControllers;

/* Real-time Model Data Structure */
struct RT_Model_OprControllers_Tag {
    /*<Root>/ModelStatus */
}
```

Figure 24: Exported Controller Variables
Figure 25: The Actual Implementation in SIMULINK
Chapter 6

UGV Implementation

Because helicopter controllers require a large amount of experience and knowledge of control theory, and current VTOL controllers were under re-design, a simple controller was implemented in MATLAB for a small UGV (as seen in Figure 26). By successful implementation two key ideas were proved: the controller conversion does work adequately, and application of the conversion scheme is cross-platform compatible (since it was originally designed for aerial vehicle implementations). The same PCB used for the VTOLs was placed on the ground vehicle because the sensor information required by a ground vehicle is similar to that needed by an aerial vehicle. The board layout can be viewed in Appendices A and B.

Figure 26: Small UGV for Implementation
First, a controller was designed using SIMULINK that allows the UGV to navigate waypoints using a set of pre-defined GPS waypoints. Appendix C contains a picture of the layout and a description of the design. Steps were taken as described in Chapter 2 to convert the controllers to C, and the C code was then imported to the CCS PIC C Compiler. No timing constraints were necessary because the controller simply needed to compare whether the GPS coordinate was close, and GPS updates are slow relative to other sensors. In addition, appropriate variables were created and functions were utilized that captured data and converted it to the proper format. Of course, these functions would be created once and included with any controller created for the target hardware platform. Afterwards, the C code was converted to assembly as outlined in Chapter 3 and implemented on the PIC 18F4620 microcontroller.

Once the programming was completed, tests were performed to ensure that the vehicle could follow the GPS waypoints assigned. The vehicle successfully navigated the five GPS waypoints thereby proving that the controller conversion scheme works adequately. Further descriptions and results can be viewed in Appendix C.
Chapter 7

Conclusions and Discussion

To create a standardized method for controller implementation from SIMULINK onto a PCB, this thesis provided a step-by-step approach to covert the SIMULINK-based controller blockset to C code, and then convert the C code to assembly language for implementation. While this approach is critical because of lack of support for some types of microcontrollers, FPGAs, and DSPs within SIMULINK’s Real-Time Workshop (RTW), some specific chip types are pre-built into RTW, so this step-by-step process may not yield the optimal method for code conversion if the target PCB utilizes one of these built-in chip types. Current architecture specific code optimizations built into Real-Time Workshop include: ARM 7/8/9, Infineon TriCore, C16x, and XC16x series, Motorola 32-bit PowerPC, 68332, 68HC11, and HC08, NEC V850, Renasas SH-2, SH-3, and SH-4, TI C6000 and C2000, STMicroelectronics ST10, and SGI UltraSPARC Iii.

Although the Microchip PIC18F series of microcontrollers are not listed under code optimization techniques, conversion of PID controllers for the target application consumed only 72 KB in the final hexadecimal format. While this is small enough to fit on most microcontrollers, PID controllers are extremely simple when compared to more complex Fuzzy Logic and LQR controllers which will produce much more code. Using these higher level controllers will require more memory and better processing power than what a simple microcontroller can provide.
Testing controllers with X-Plane from inside SIMULINK has proven to be an extremely valuable tool both in time and cost. Initial testing showed a flaw in one of the early PID controllers allowing for redesign before implemented on the actual platform. This saved time because the actual VTOL did not have to be flown (which requires a substantial amount of time on the part of several individuals) and the actual VTOL did not crash. Current testing with Fuzzy Logic controllers is allowing for easy tuning of the rules without having to test on board the vehicle (another valuable time saver). Once the rules are properly configured, noise will be added to simulate the noise data collected from the sensors located on the VTOL. As a result the controllers will implemented on an very accurate simulated flight model before being implemented on the actual model to provide the smoothest transition possible.
Chapter 8
Results and Future Work

Although helicopter controllers are more complex, the successful implementation on board the small UGV shows that the automated conversion process can be used to simplify controller implementation. In addition, no errors were encountered when converting previous versions of PID controllers to assembly. Actual testing using X-Plane and a PID controller has shown success and the PID controllers are under revision to allow for IMU drift (which caused instability in the model). For portability between systems both C code and Java X-Plane communication implementations have been developed. Currently, a set of Fuzzy Logic controllers are undergoing tests between SIMULINK and the X-Plane simulator to validate controller functionality. In addition, to ease implementation of the X-Plane block diagram within SIMULINK, the block will be reworked to allow changing of the imported data items within SIMULINK rather than within the code. Appendix A shows the overall system view currently under development for a Raptor 90.
References


Appendix A: Overall System Board Under Development

Figure 27: Autopilot Block Diagram

Figure 28: Autopilot Board Photo
Appendix A: (Continued)

The autopilot board, developed at Oregon State University, consists of one main board with accepting connectors for two daughter cards: the Microstrain IMU and the MaxStream wireless card. On the main board, three Microchip PIC microcontrollers are utilized for servo functionality, GPS parsing, and controller implementation. The 18F1320 handles GPS parsing, the 18F452 handles servo movement, and the 18F4620 contains necessary functions to handle GPS, IMU, and barometric data as well as the flight controllers. In addition, the 18F4620 contains the safety switch logic allowing control to be switched between autonomous mode and teleoperation mode via a switch on a radio control transmitter. Thus, this information is obtained from the receiver rather than the wireless card. Further information about the components can be obtained from the manufacturer’s websites.
Appendix B: System Board Schematic Layout

Figure 29: Autopilot Board Schematic
Appendix C: UGV Controller Design

Following the outline for the automated controller conversion process, the controllers for driving to a coordinate using GPS waypoints were implemented in SIMULINK. Because the current version of Real-Time Workshop does not have support for algebraic loops and an array counter must be used to traverse an array of GPS waypoints, incrementing of the counter was implemented in C code. Figure 30 shows the layout of the controller in SIMULINK. Once the controls were converted to C through Real-Time Workshop, a set of functions created for the target hardware was utilized to format the data properly for use with the converted controller. MATLAB version 7.1 was used in creation of the controllers, so a newer file provided by Mathworks was required during compilation (this file was rtlibsrc.c). Code generation proceeded from MATLAB without errors or warnings and the code was next imported into the PIC C Compiler.
Appendix C: (Continued)

Unfortunately, the compiler used for assembly generation does not support the “const” declaration, so all “const” declarations were removed from the rtlibsrc.c file through is a simple find/replace. Once this file is modified once, it can be used in other projects without modification because the code contained within the file is not dependant on code generated from Real-Time Workshop. The PCB used for testing controllers on the small UGV came with a driver file that obtained GPS waypoints as well as IMU sensory data (like most autopilot/controller boards). To interface with sensory data variables, the controller function produced by Real-Time Workshop was copied/pasted into the driver file for the PCB and the variables used within generated function were assigned to the sensory variables within the driver file. Several problems were determined within the SIMULINK UGV controller design, but the code generation from Real-Time Workshop was flawless. In addition, the PIC C Compiler performed better than expected, but care must be taken in variable assignment so that there are no register overflows during code execution. Variables were re-assigned from type real32_T to floats within the PIC C Compiler (again a one time assignment) to alleviate this problem.

When utilizing any embedded system on an autonomous platform, it is critical to have live feedback from the system to determine current calculations of the controllers. This was a major issue in initial testing of controllers with the small UGV because the ground vehicle appeared to operate normally with the in circuit debugger attached in an in door environment, but failed to operate as expected when placed in an outdoor environment utilizing live GPS signals.
Appendix C: (Continued)

Once all controls were interfaced and problems were corrected, the UGV was given five GPS points to navigate. It successfully navigated all 5 waypoints with any errors within range of the GPS sensor error. During the waypoint navigation the vehicle did “wobble” along the calculated path, but this may be a result of the ackerman steering or GPS error. The controllers will eventually be revised to eliminate this problem, but code conversion techniques have proven successful for the rapid MATLAB controller implementation.