Automated Process for Unmanned Systems Controller Implementation USING MATLAB

Daniel Ernst, Jeff Craighead, Kimon Valavanis
Department Computer Science and Engineering
University of South Florida
deernst@csee.usf.edu

Abstract – **Unmanned aerial vehicles, or UAVs, currently fly a large variety of missions usually centered around reconnaissance. Because the flight patterns vary for the particular type of mission, and a large variety of UAV platforms exist—everything from small unmanned airplanes to large helicopters such as the Yamaha R-MAX —flight controllers must be changed to allow proper control of the aircraft for the mission. Currently, controllers are designed in a design package such as MATLAB or SIMULINK and then implemented seperately in code but these design methodologies cause problems. 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, but no automated method for converting a simple controller, such as a PID, from MATLAB to implementation on a microcontroller has been presented. To implement current in-house controllers designed in MATLAB/SIMULINK, a system consisting of Real-Time Workshop and a C to Assembly compiler has been used to produce assembly code for a target microcontroller. To aid in verification of the MATLAB controllers and C code produced by Real-Time Workshop, an interface for the controllers in SIMULINK and a flight simulator (X-Plane) was created. While the current conversion system has been developed for small unmanned aerial vehicles with limited power, payload, and processing capability, the process could be extended to other platforms.**

Index Terms – Controller Conversion, MATLAB, SIMULINK, X-Plane, Unmanned System, Autopilot.

I. Introduction

MATLAB and SIMULINK are powerful system design packages used by a wide variety of companies. The powerful variety of toolboxes incorporated within these packages provide an easy to use interface that allow rapid design—a perfect method for quick and easy design of controls for unmanned systems. By combining both easy and quick design of controls within MATLAB/SIMULINK with a system utilizing microcontrollers, plug-in/plug-out capabilities are created allowing for easy reconfiguration of the unmanned system for various mission tasks. Unfortunately, there is no standard process for implementation of the controls. The purpose of this paper is to describe a methodology that may be used to implement three types of controllers designed in MATLAB/SIMULINK on any type of unmanned system. To ensure that individuals wanting to use this methodology may do so with little or no background in programming controllers, the steps for conversion will be kept as automated as possible. Also, to provide an example implementation, an autopilot PCB containing three different Microchip microcontrollers is utilized. On the example PCB, 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. Because the three microcontrollers are made by Microchip, a respected company that produces a wide variety of microcontrollers and microcontroller tools, a PIC-C compiler generates the assembly code [1].

The three controllers developed in-house for implementation on a small unmanned aerial vehicle include PID controllers, fuzzy logic controllers, and LQR controllers. To develop the controllers MATLAB was utilized in conjunction with Simulink and the fuzzy logic toolbox. The 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. While the PID controllers could be converted, problems were encountered when converting the fuzzy inference system because the compiler used to convert the code could not interpret the fuzzy inference system files.

Thus, a set of fuzzy controllers developed in C will be used for the fuzzy helicopter controls.

II. The Design Process

Ideally, a complete design process would incorporate selection of hardware before the controllers are designed in MATLAB. This would aid in reducing the amount of code written in C to convert units from sensor readings. Thus, many hardware choices should be considered and chosen based on the types of controllers to be implemented, cost, size, necessary sensors, and processing capabilities. In addition, it is imperative that the hardware chosen has good support base, and the company that produces the hardware provides a compiler to convert from C to assembly. Once the hardware has been determined, controllers are designed using MATLAB which cause the unmanned system to perform certain actions —this allows for the plug-in/plug-out capabilities using the automated process. Afterwards, the controllers are tested with Simulink and X-plane for design verification. Next, the controllers are converted to C using MATLAB's Real-Time Workshop, altered for the current hardware, tested again using X-plane, converted to assembly, and implemented in the hardware—these steps can be viewed in Figure 1.

Figure 1: Steps for Conversion

III. 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. The makefile option allows for further customization for the processor such as conversion for microcontroller enabled floating point or integer operations (Figure 2). The PID controllers contain floating point operations, but the main microcontroller does not have floating point capabilities, so the default makefile is 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.

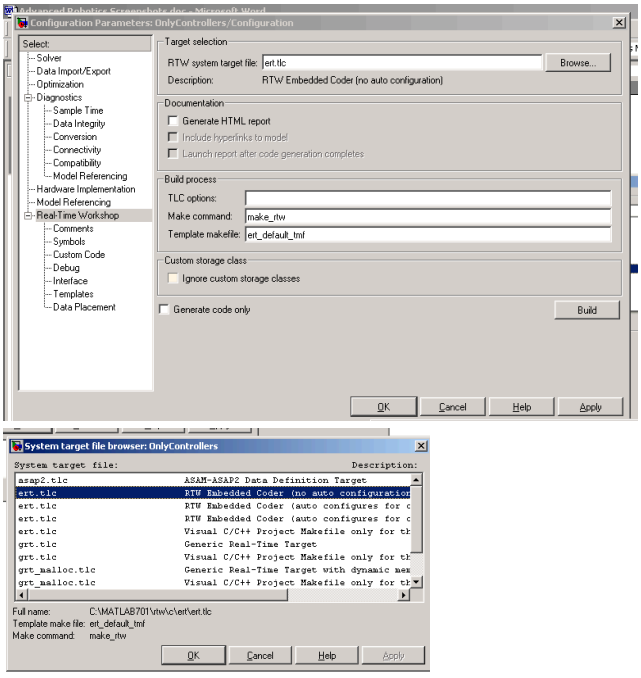

Figure 2: Real-Time Workshop Setup

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. For the sample board, no Microchip microcontrollers exist in the list, so the 8-bit generic processor is selected (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 current controllers to be converted and the steps necessary for each of these types of controllers.

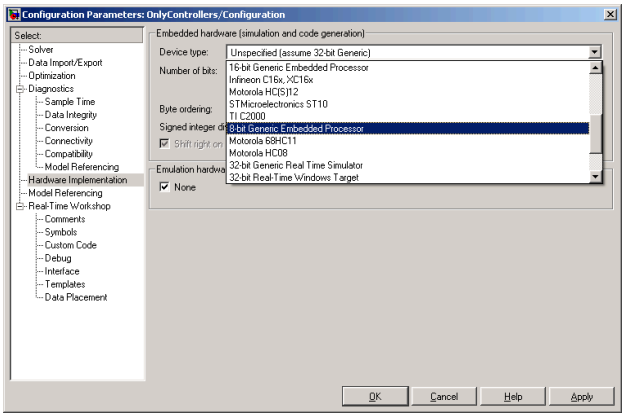

Figure 3: Microcontroller Selection

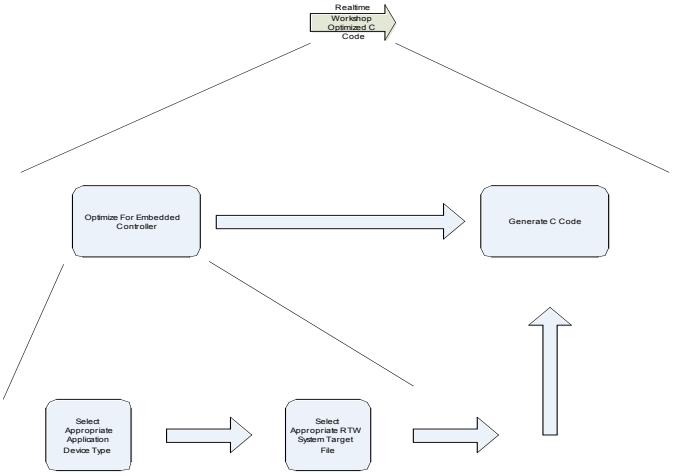

Figure 4: Overall Real-Time Workshop Conversion

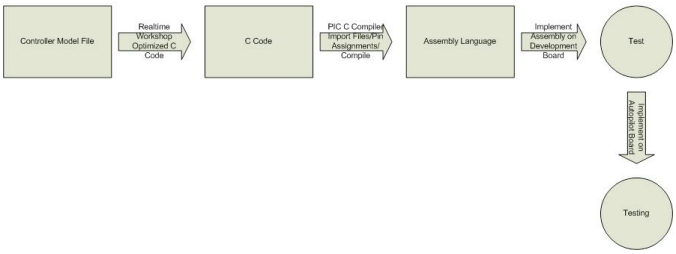

Figure 5: PID Controllers

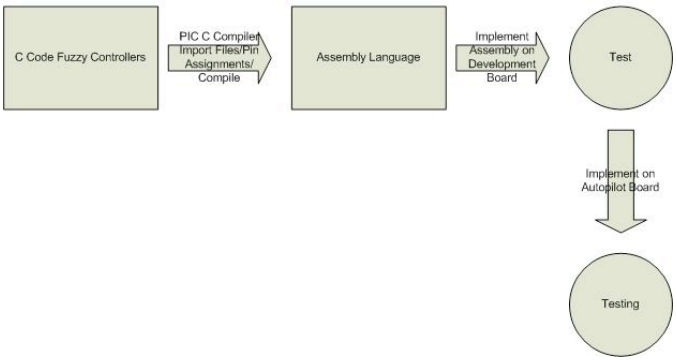

Figure 6: Fuzzy Logic Controllers

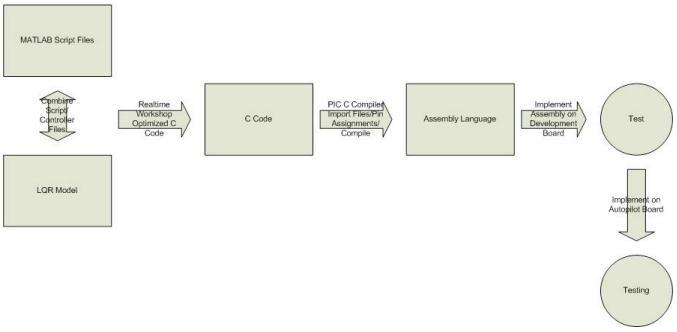

Figure 7: LQR Controllers

IV. ASSEMBLY CODE GENERATION

Once the build has completed, Real-Time Workshop may be closed and the PIC-C compiler 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 PIC-C compiler for proper handling. Figure 4 shows the files that were imported to the 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). 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 PIC-C compiler—Figure 9 shows the flow control conversion. Now, all files are present for compilation.

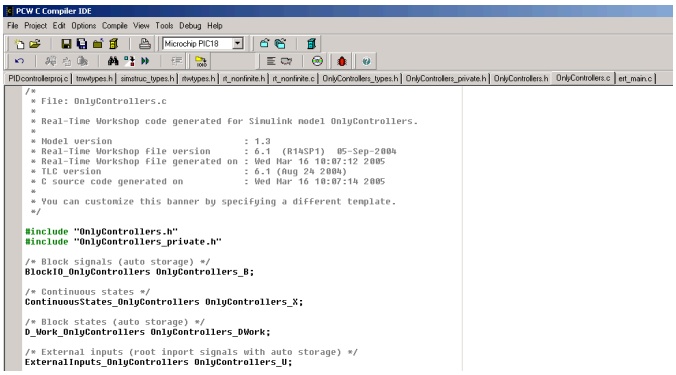

Figure 8: Files Needed for Conversion

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. Figure 10 shows the variables needed for the PID controllers. 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, most data conversion will include a GPS parser, and conversion functions for the IMU and barometric pressure sensor (If used). 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. In addition, any alterations to sampling time should be handled. Once these conversions are complete, the assembly files can be generated and the chip programmed.

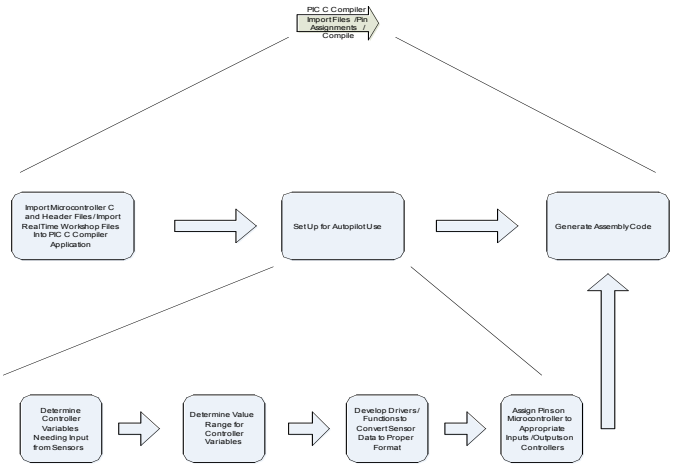

Figure 9: Overall Assembly Generation

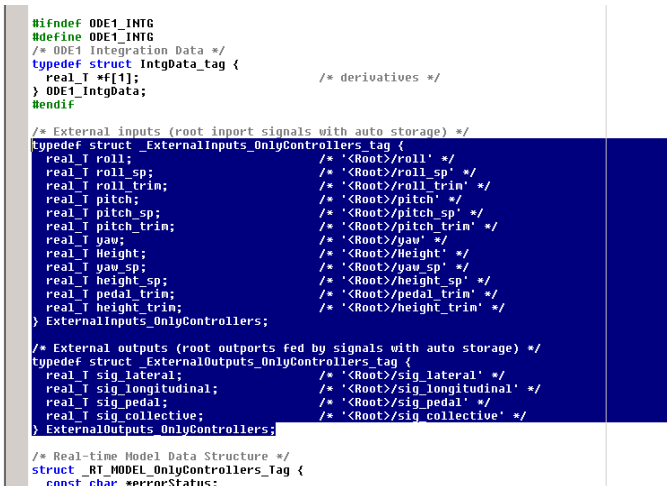

Figure 10: Exported Controller Variables

V. X-PLANE SIMULATION

To enable testing of controllers developed in MATLAB and ensure proper conversion from MATLAB to C code, a 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—and also allows external communication as well as airfoil design. Because the sample PCB is an autopilot board to be used on small unmanned helicopters, a model for a Yahmaha R-Max and a Raptor 90 was created. Creation of these models were based on specifications from actual helicopters, and then verified to be accurate by an individual that flies these helicopters. 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.

X-Plane uses UDP communication to send and receive data packets which allows changes to various values within X-Plane. A large variety of values can be changed including control of the aircraft as well as causing in flight failures. Two different communication programs were created: a Java program to allow for portability and a C program to implement into Simulink with the controllers. Because the inputs and outputs for the controllers will change depending the type of the controller and for what it is used for, the needed variables will be 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. The Java send and receive functions are incorporated into the same file, but the C files are split into send and receive functionality. The send and receive design is shown in Figure 11.

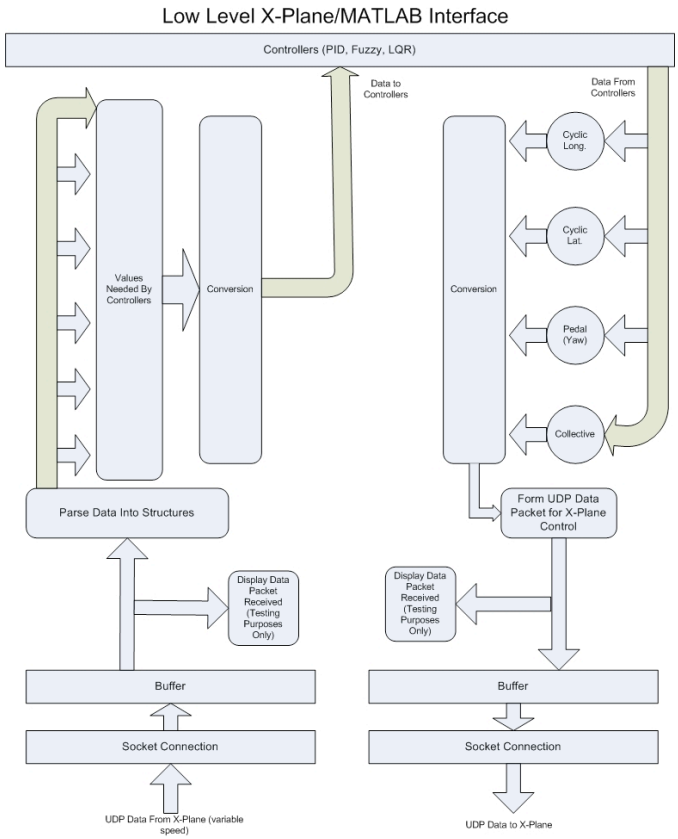

Figure 11: X-Plane/SIMULINK Communication

VI. Results and Future Work

Steps previously described to convert PID controllers from MATLAB to assembly were successful although actual testing using X-Plane has not been performed at this point. Current, C code allows data to be received from X-plane, but code for sending to X-plane is still under development. Experiments with X-Plane and Java communication show data can be received and sent. A sample script demonstrated remote take-off and banking of a Cessna aircraft within X-Plane. Future work will include finishing of the C code for X-plane communication and testing of the converted PID controllers as well as the PID controllers located within MATLAB. Figure 12 shows the overall system view under development.

References

[1] Incorporated, C. C. S. (July 2005). C Compiler Reference Manual. Brookfield. 2005.

[2] A. E. Fisher, D. W. E., and S. M. Ross (2001). Applied C: An Introduction and More. New York, McGraw-Hill, 2001.

[3] Walker, I. M. a. R. (Aug. 16-19, 2004). Simulation for the Next Generation of Civilian Airspace Integrated UAV Platforms. AIAA Modeling and Simulation Technologies Conference and Exhibit, Providence, Rhode Island.

[4] Meyer, A. (2004). UDP Reference. 2005: X-Plane UDP Reference Manual.

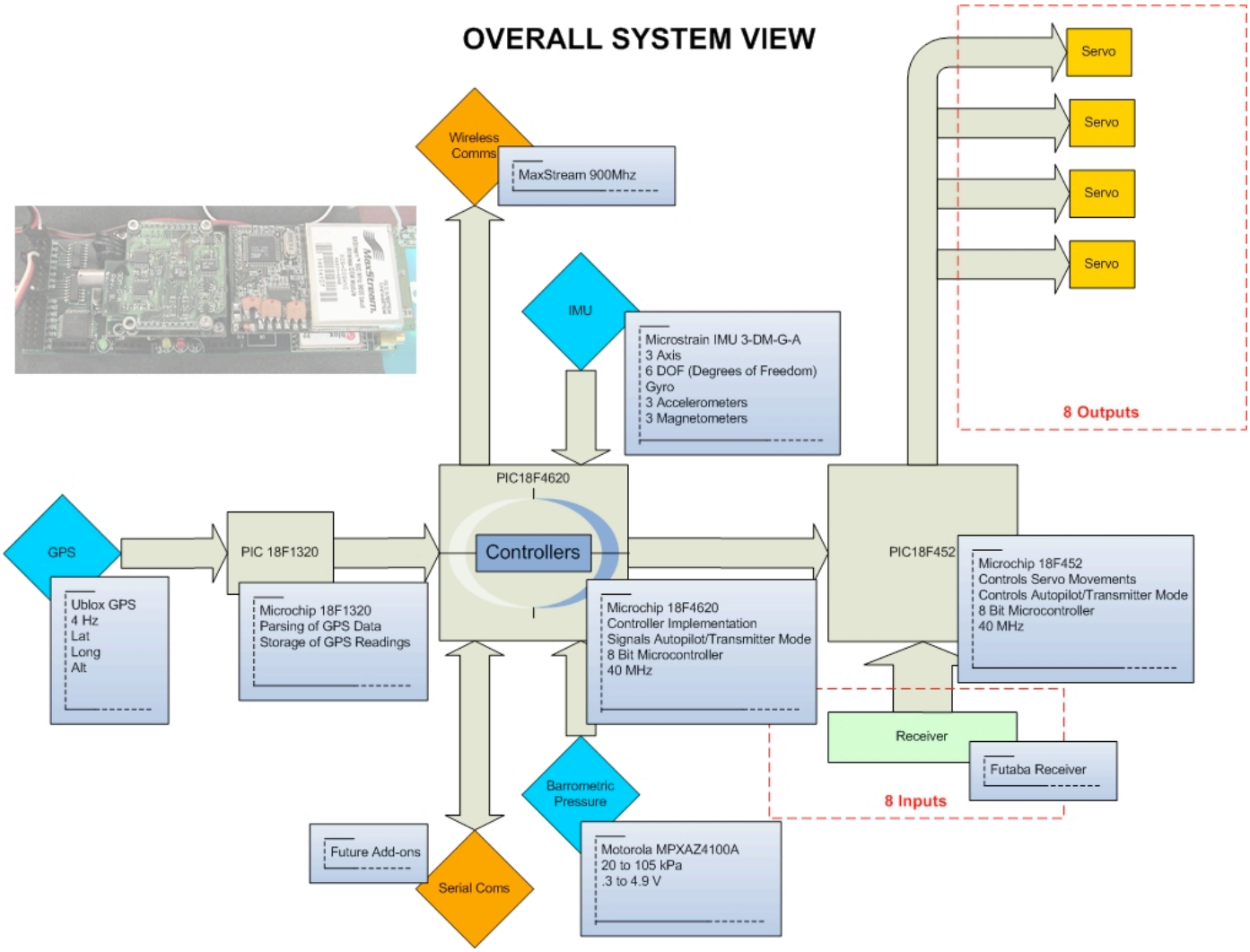

Figure 12: Overall System View