J Intell Robot Syst (2007) 49:85–108
DOI 10.1007/s10846-007-9130-4

UNMANNED SYSTEMS PAPER

Unmanned Vehicle Controller Design, Evaluation and Implementation: From MATLAB to Printed Circuit Board

Daniel Ernst · Kimon Valavanis · Richard Garcia · Jeff Craighead

Received: 25 May 2006 / Accepted: 10 January 2007 /
Published online: 8 March 2007

Abstract A detailed step-by-step approach is presented to optimize, standardize, and automate the process of unmanned vehicle controller design, evaluation, validation and verification, followed by actual hardware controller implementation on the vehicle. The proposed approach follows the standard practice to utilize *MATLAB/SIMULINK* and related toolboxes as the design framework. 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*, and *C to Assembly* language conversion to produce assembly code for a target microcontroller. Considering Unmanned Aerial Vehicles, fixed or rotary wing ones, *X-Plane* is used to verify, validate and optimize controllers before actual testing on an unmanned vehicle and actual implementation on a chip and printed circuit board. Sample designs demonstrate the applicability of the proposed method.

Keywords Autopilot · Controller design · Implementation · MATLAB/SIMULINK · Microcontroller · Unmanned systems · Validation

1 Introduction

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,

D. Ernst (✉) · K. Valavanis · R. Garcia · J. Craighead
Department of Computer Science and Engineering, Unmanned Systems Laboratory,
University of South Florida, Tampa, FL 33620, USA
e-mail: deernst@cse.usf.edu

K. Valavanis
e-mail: kvalavan@cse.usf.edu

R. Garcia
e-mail: rdgarcia@cse.usf.edu

J. Craighead
e-mail: craighea@eng.usf.edu

validation and verification, followed by actual hardware controller implementation on the vehicle. The presented approach is kept as general and generic as possible, so it is applicable to any unmanned vehicle with minor modifications that depend on the specific microcontroller processor and autopilot chip used. However, this paper focuses on and considers as a testbed, miniature unmanned vertical take off and landing (VTOL) vehicles with very strict payload limitations and power supply restrictions using off the shelf components.

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 in general, and unmanned helicopters in particular, are used for 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. This becomes more important given that, 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. 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 such a method.

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*. This is then followed by a *C to Assembly* conversion to produce assembly code for a target microcontroller. *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 aerial vehicle and actual implementation on a chip and printed circuit board. *X-Plane*, a commercially available package, is chosen because of its extremely accurate flight models, external communication, airfoil design [6], also allowing for input/output from external sources. While *X-Plane* provides UAV testing capabilities to ensure accuracy before implementation for safety purposes, the conversion process will work virtually on any platform. Future versions of *X-Plane* will incorporate integration of accurate ground vehicle simulation with aerial vehicle simulation so that UGVs may be tested as well.

To ensure wide applicability 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.

The rest of the paper is organized as follows: The next Section presents system fundamentals and the overall design process. Subsequent sections describe individual steps of the design process followed by examples that are provided in a separate section as a 'walk through' guide for each step of the process. The last section concludes the paper.

2 The Design Process

In the most general case, the starting point of the design process under consideration is selections of hardware components before controllers are designed using *MATLAB*. As such, appropriate sensors (GPS – DGPS, IMU, compass, cameras, lasers, etc.), on-board computers and processors, microcontrollers and other peripherals are chosen considering processing capabilities, size and cost, payload and power restrictions. It is also imperative that chosen hardware components have a solid support base including a compiler to convert *C* code to assembly language. This assembly compiler must be able to compile the structures produced by the *Real-Time WorkshopEmbedded Coder* toolbox. Generally, a multi-pass compiler is needed to interpret these structures. Assuming completion of this initial step, Fig. 1 presents a block diagram of the proposed method and all separate steps involved.

Observing Fig. 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* code using

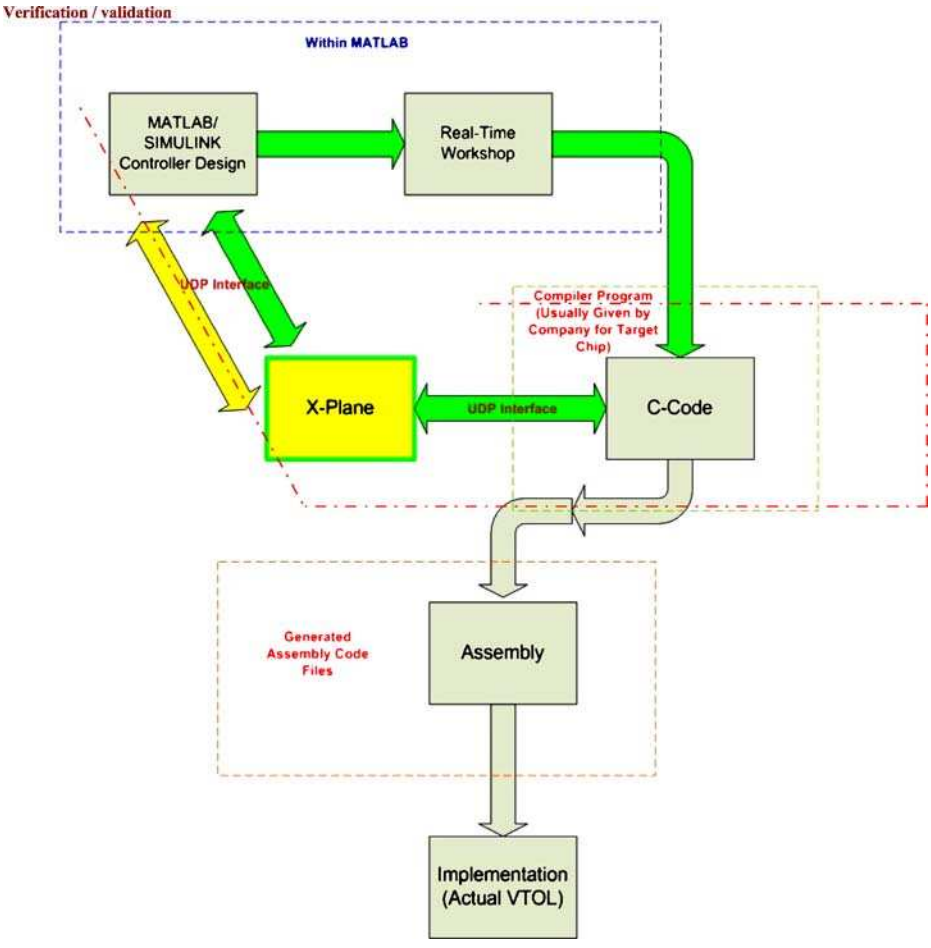

Fig. 1 Block diagram of the automated process

the *MATLABReal-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* conversion before the controller is implemented on the vehicle.

Figure 1, and in particular the validation/verification block illustrates alternative operational steps in which *X-Plane* may be used to check controllers twice, once (for controller validation after the initial design), or never assuming 'perfect initial controller design.' In any case, the importance of controller verification and validation needs no further explanation.

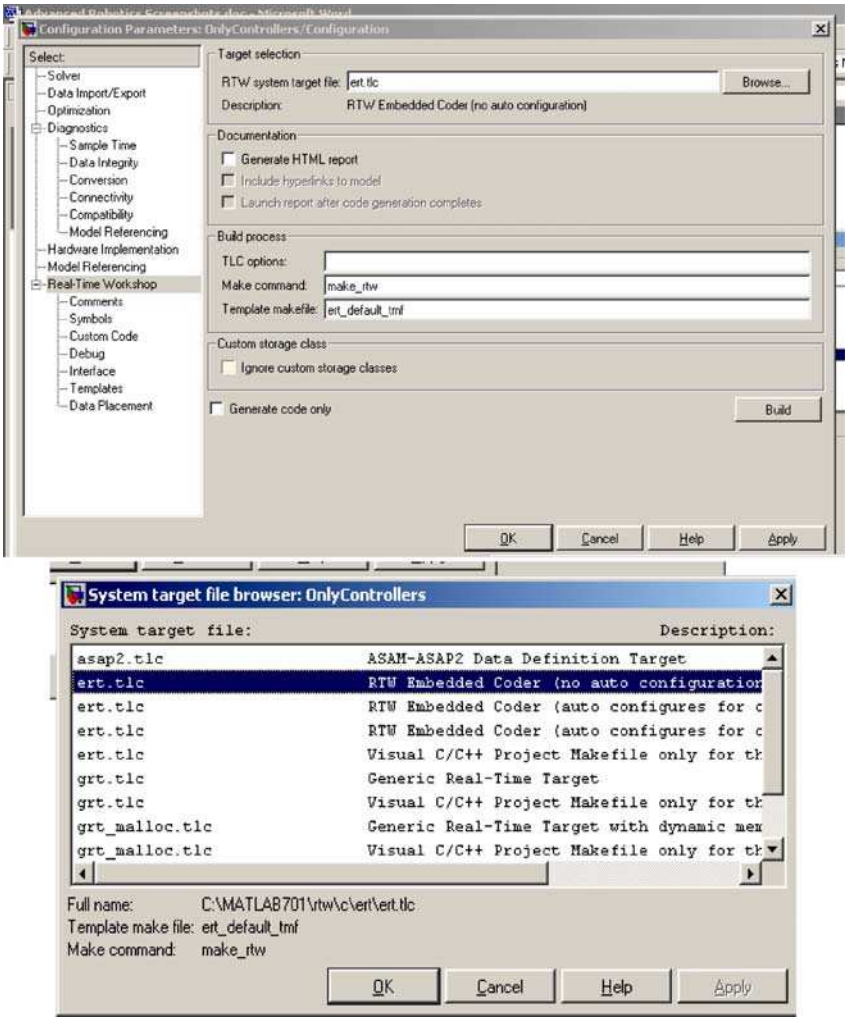

Fig. 2 Real-time workshop setup

3 MATLAB/SIMULINK to *C* Conversion

To convert from *MATLAB* to *C* code, an environment called *Real-Time Workshop* (*RTW*) provides automatic code generation. In addition, *RTW* also provides several ways to optimize controllers for particular types of processors. Once a set of controllers is opened in *SIMULINK*, the file must be "built" using *RTW*. Before building, however, several customizations must be made. First, *RTW* is 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 controllers are implemented on a microcontroller chip, the proper *.tlc* file needs 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 *RTW* 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, *Template makefile* option is changed to *ert_default_tmf*. The *makefile* option allows for further customization such as conversion for microcontroller enabled floating point or integer operations (Fig. 2). For example, 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 designed 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. 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.

Next, the *Configuration* menu must be opened and hardware implementation must be selected. The pull-down menu next to *Device type* contains optimizations for various processors and microcontrollers (Fig. 3). Once this has been selected, *RTW* is selected again from the

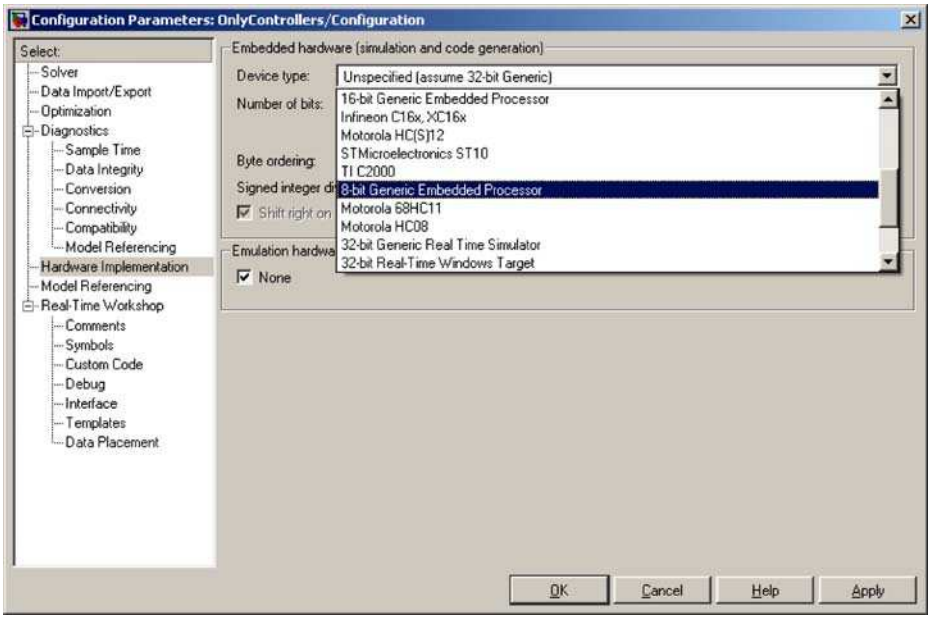

Fig. 3 Microcontroller selection

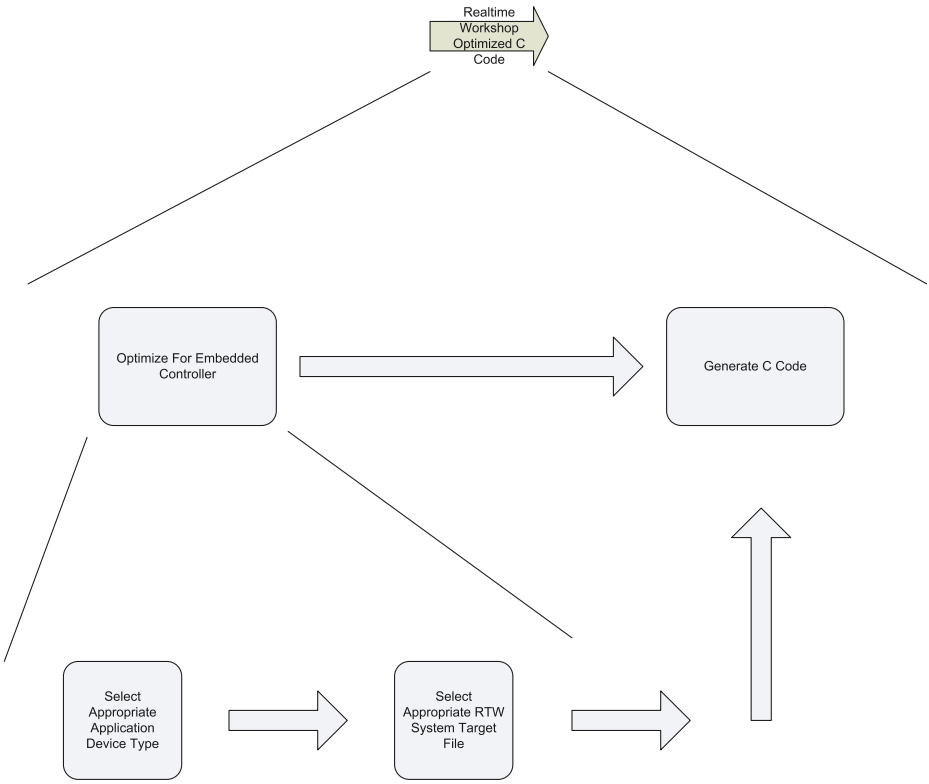

Fig. 4 Overall real-time workshop conversion

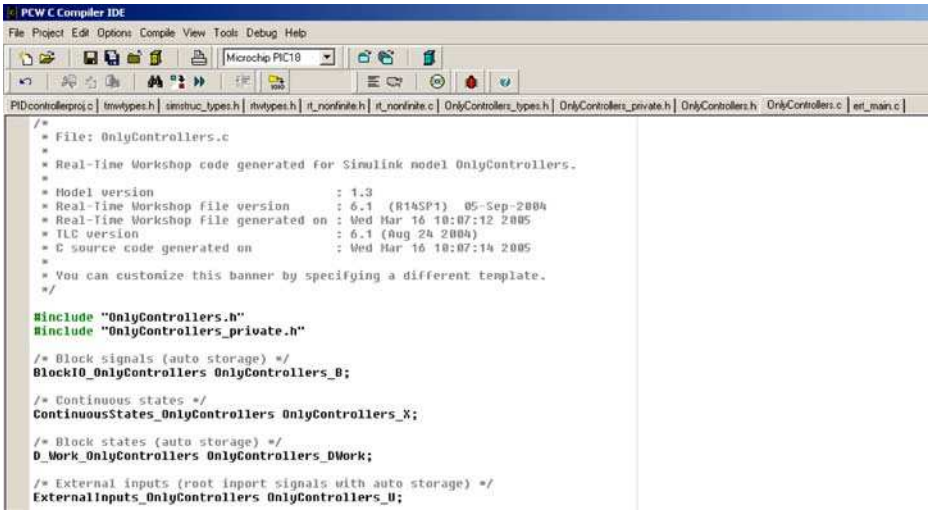

Fig. 5 Files needed for conversion

selection menu and the *Build* button is pressed to start building the *C* files. The only major difference between how the controllers are implemented occurs in the base step: the conversion from *RTW* to *C*. Figure 4 shows a more detailed diagram of the *RTW* conversion to *C* code.

4 Assembly Code Generation

Once files are built, a target compiler is opened to import the files. *RTW* produces different files attempting to tie the *C* files to *MATLAB*; these files must be imported into the target compiler for proper handling. An illustration is shown in Fig. 5; files imported from *RTW* output directory to a compiler for conversion to assembly code. In addition, there exist data structures that must be interpreted through the *tmwtypes.h* and *rtwtypes.h* files (see Fig. 5). While these files are needed by the compiler, *RTW* does not output them to the same directory as the rest of the files. Thus, these files must be located in the *MATLAB* directory, copied into the *RTW* output directory, and then imported into the compiler; Fig. 6 shows the flow control conversion.

Modifications must be made to the *C* files to ensure proper implementation on the microcontroller chip. This is an involved procedure that starts with a detailed schematic of

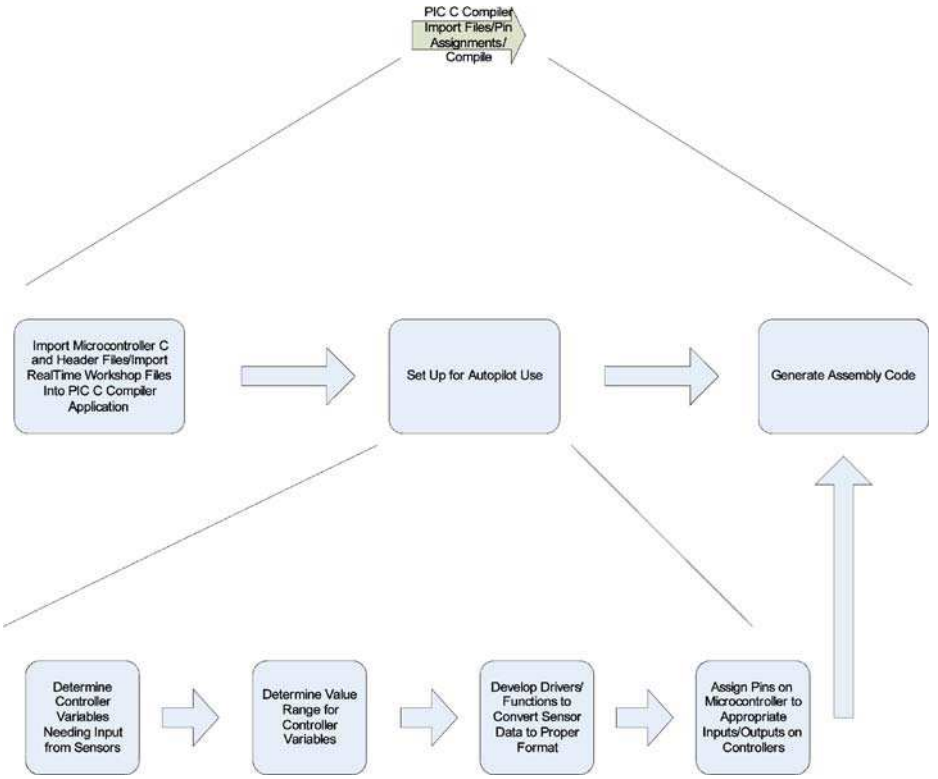

Fig. 6 Assembly code generation

the unmanned vehicle autopilot board that is used. The autopilot schematic includes all pins vehicle sensors are connected to. Once these pins are determined, they are assigned to the variable names on the microcontroller chip. In addition to providing pin assignments, sensor outputs need be converted to the proper format to be handled by the microcontrollers. Timing issues should be dealt with – if this is not handled within the *MATLAB/SIMULINK* controller design – as well as any alterations to sampling times. Rather than programming the pin assignments, initialization, and timing requirements each time the controller is modified, a separate *C* file should be created that makes all of these initializations. Thus, this separate *C* file will be the main *C* file and several function calls will reference the functions produced in the *C* code generated from *Real-Time Workshop*. More compact code may be generated through coding the main *C* file in the compiler native language (usually the syntax is provided in the compiler manual) [1]. Before converting the controllers to assembly code, they should be tested again (once in *SIMULINK* and once with the *C* code) with some type of simulation to ensure proper operation. This is especially true when the controllers are implemented on aerial vehicles and safety is extremely important. Once any needed conversions and testing are complete, the assembly files can be generated and the target chip programmed via the method specified by the chip manufacturer.

This step concludes the controller conversion process, but since these controllers are targeted for unmanned vehicles, they should be verified, validated, tested and refined before actual implementation. This is accomplished using the *X-Plane* simulator.

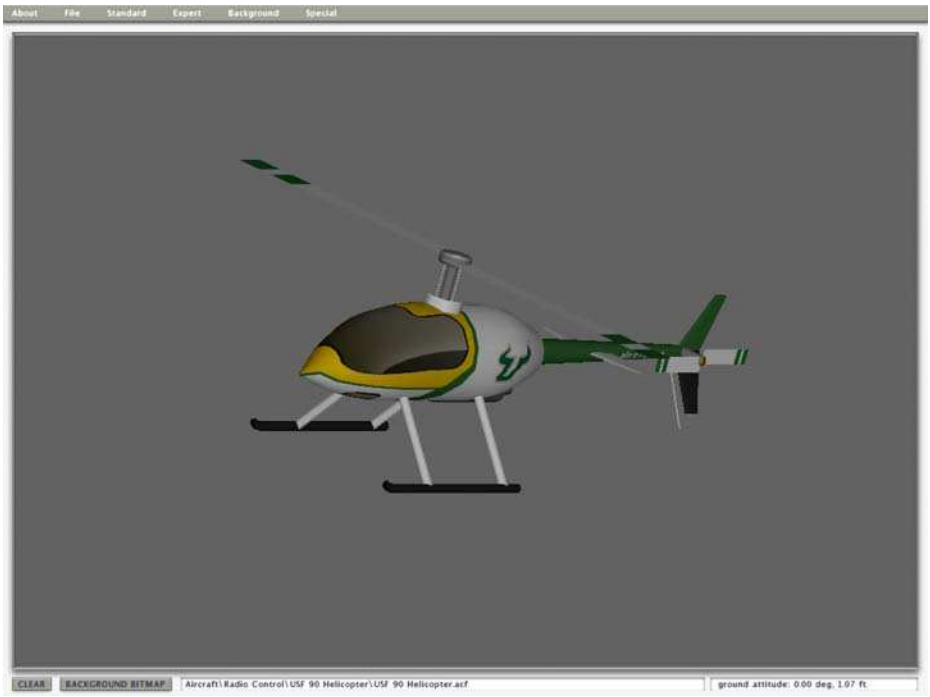

Fig. 7 The raptor 90 model

5 X-Plane Based Controller Simulation

5.1 X-Plane in General

Concentrating in aerial vehicles, *X-Plane* is a closed source, commercially available package used to verify/validate/refine controllers designed in *MATLAB* and ensure proper conversion from *MATLAB* to *C* code (Fig. 1).

Although there exist several simulators like *Microsoft's Flight Simulator*, and *FlightGear*, *X-Plane* provides extremely accurate flight models and allows for external communication as well as airfoil design [6]. It is accurate enough to be used to train pilots [5]. Unlike *Microsoft Flight Simulator*, however, *X-Plane* also allows for input and output from external sources. While *FlightGear* has I/O capabilities similar to *X-Plane*, it is not quite as stable as X-Plane and doesn't provide the level of support. As noted in [3], *X-Plane* provides future capabilities that unmanned aerial vehicles will need, including navigation markers, changing weather conditions, and air traffic control communication. Figures 7, 8, 9, 10 show a *Raptor 90 SE* modified from a Raptor 70, obtained from air.c47.net, and a *YAMAHA R-max* model created using the *X-Plane* plane maker. While *X-Plane's* plane maker provides an interface to design a vehicle based on physical dimensions, power, weight, and a host of other specifications, it produces a specialized file that allows the simulator's built in physics to properly interact with the model. Thus, the equations for the simulator model are not available.

Fig. 8 Raptor 90-hovering

Fig. 9 Raptor 90 in slow forward flight

A key aspect of controller verification/validation/refinement is the actual communication and interface between the *X-Plane* and *MATLAB/ SIMULINK*. This is presented in the next subsections, based on the detailed diagram shown in Fig. 11 for sample controllers. The remainder of this section describes a fuzzy logic controller, designed by Richard Garcia, used to test the *X-Plane* communication. In Section 5.2 we will describe the UDP protocol used to communicate with *X-plane* and how this was incorporated into our *MATLAB* controller. Section 5.3 describes how the I/O is processed within the controller to facilitate communication between *SIMULINK* & *X-plane*. Section 5.4 describes the experimental setup within X-plane and the goals of the simulation. Finally Section 5.5 presents the results of the simulation, verifying proper operation of the controller.

Currently, there are several fuzzy logic controllers for small VTOL vehicles that are integrated with a *SIMULINK/X-Plane* communication block allowing for testing of a Raptor 90 model. This integration of the fuzzy logic controllers with the *SIMULINK/X-Plane* communication block can be seen in Fig. 12. Currently, the controllers allow for hovering of the Raptor 90 as well as for navigation via waypoints, both under normal flight and under tail rotor failures. The fuzzy logic controllers designed in *MATLAB* using the fuzzy logic toolbox consist of separate fuzzy blocks for aileron, elevator, collective, and rudder inputs to the Raptor 90 model. The controllers utilize the vehicle's current information such as latitude, longitude, velocities, and angular rates information from *X-Plane* to calculate movement and correct error. By combining error with the vehicle's current attitude and angular rates, the controllers can directly connect the error to a particular actuator movement.

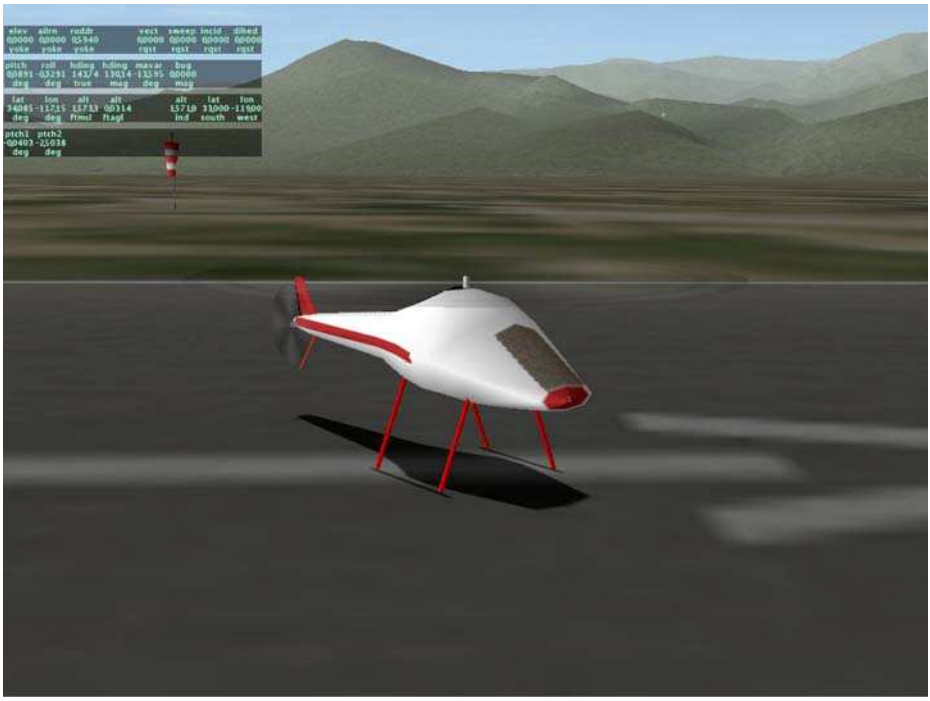

Fig. 10 Yamaha R-max, designed by Jeff Craighead, awaiting take-off

5.2 X-Plane UDP Communication

X-Plane uses UDP communication to send and receive data packets; this allows for changes to be made to various values within *X-Plane*. The UDP protocol has advantages and disadvantages. UDP may be unreliable over any distance due to its protocol, which does not guarantee receipt of any sent packet; on the other hand, UDP is extremely fast and utilizes bandwidth more efficiently [7].

X-Plane is able to dump up to 99.9 data packets per second across a local network; this has an important impact on controller functionality (and simulation) because they require sufficient update speed to operate correctly. *X-Plane* offers several parameters whose values may change, including control of the aircraft, failure introduction, etc. For the fuzzy logic controllers, a rate of 50 Hz was chosen because it realistically reflects the message rates of the Microstrain 3DM-GX1 IMU currently on board the actual Raptor 90 helicopter.

In order to select the appropriate data items to export to *SIMULINK*, *X-Plane* provides an easy to use checkbox interface as shown in Fig. 13. However, *X-Plane* is not open-source, hence, it will be necessary to be familiar with the UDP documentation [4].

5.3 X-Plane Exported UDP Data

X-Plane may send and receive a large number of parameters allowing controllers to be designed using many different sensor readings. When the *Inet2* tab is opened, the IP computer address that includes the controllers is 'written' along with the corresponding (proper) ports as shown in Fig. 14. Next, the *Data Set* tab is chosen and values required by

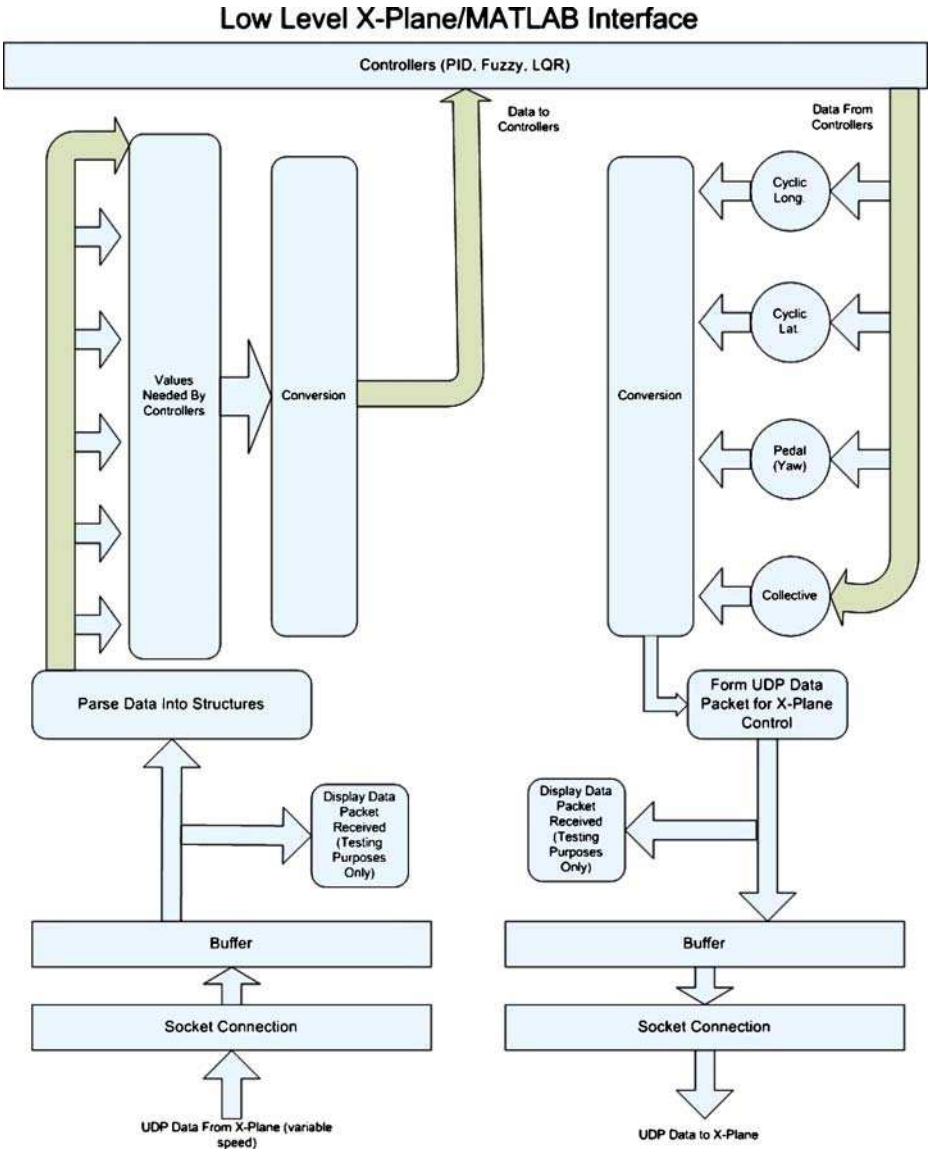

Fig. 11 X-Plane/SIMULINK communication and interface

the controllers are selected. *X-Plane* starts sending the data to the destination IP and port. In addition, *X-Plane* waits to receive packets on another specified IP and port. The speed with which data is exported is controlled by increasing or decreasing the number of data packets per second (a range from 0 to 99.9) in the *Data Tab*. To send the data, a UDP packets are formed consisting of the string of characters "DATA" followed by an integer; then, the data items selected on the screen are attached in increasing order of the data item numbers. Each data item consists of one integer (the number specified in the output screen) and eight float values. The proper index value (begins with 0) for a selected data item may be determined

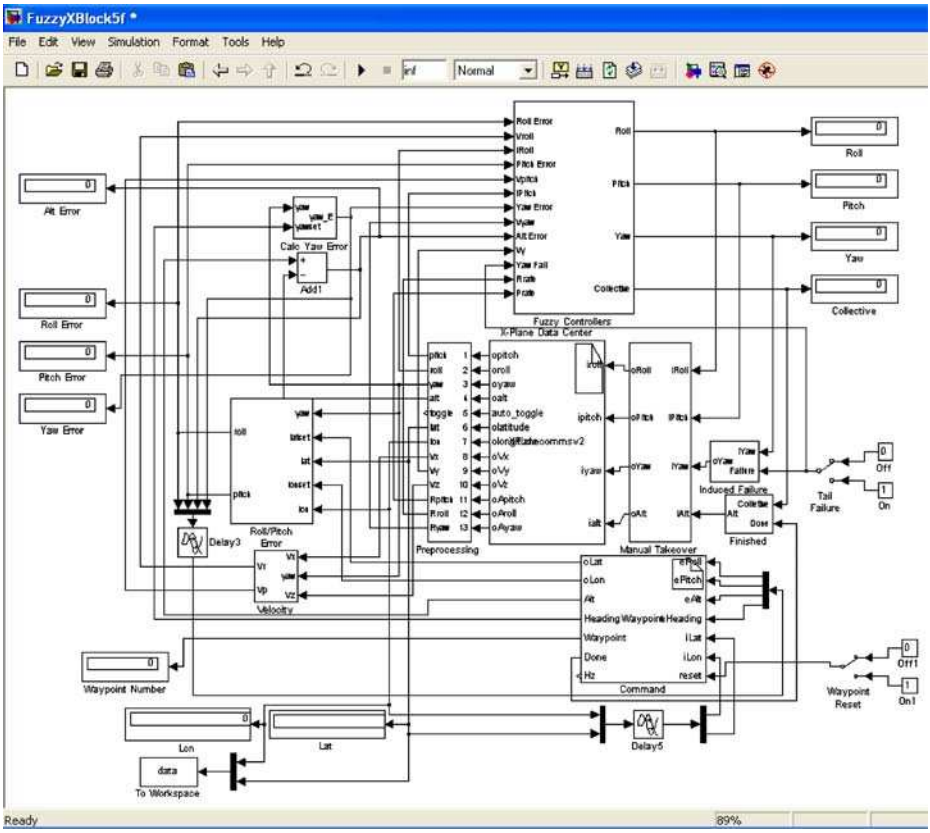

Fig. 12 SIMULINK/X-Plane communication block integration

by displaying the selected values on the screen-selecting one of the four checkboxes in the data output dialog box. For each item, the data structure looks as follows:

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

Once, all the data value selections have been assembled into the packet, the packet is sent to the destination IP and port in network byte order as shown in Fig. 15. Notice that after the header information, each item selected to be output from *X-Plane* contains 36 bytes. Thus for the fuzzy logic controllers, 13 data items are sent from X-Plane to the fuzzy logic controllers, so 416 bytes (36 * 13) plus the five packet header bytes (a total of 421 bytes) are sent inside each UDP packet at a rate of 50 Hz.

5.4 *C* Code Interpretation

Since *X-Plane* runs on a *Windows XP* machine in the lab, *Winsock* is utilized in the *C* code to set up the client/server socket communication. *X-Plane* is considered to be the server

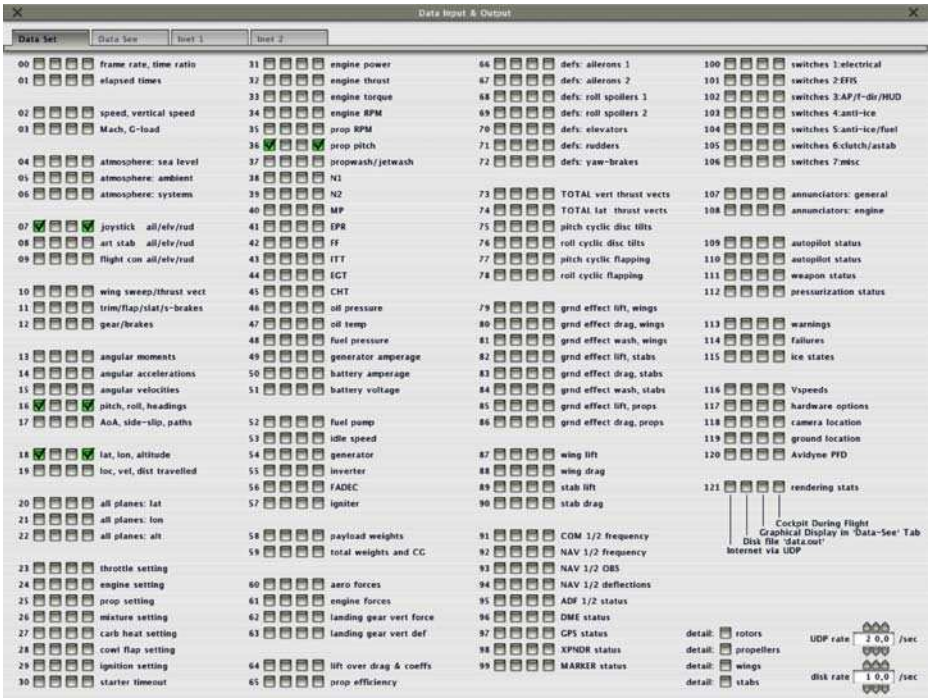

Fig. 13 The X-Plane communication screen

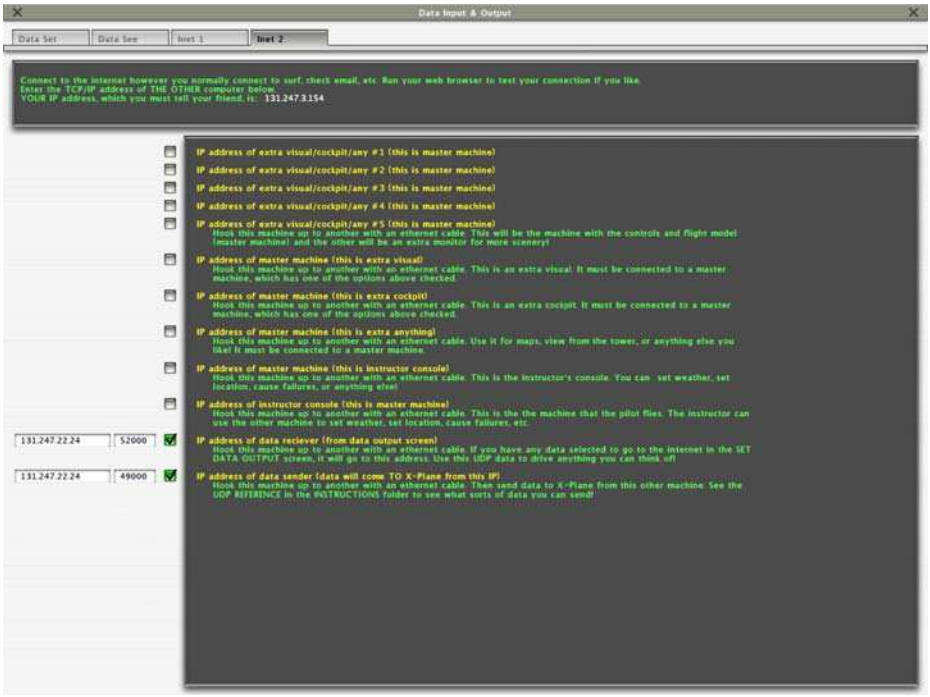

Fig. 14 X-Plane IP and socket interface selection

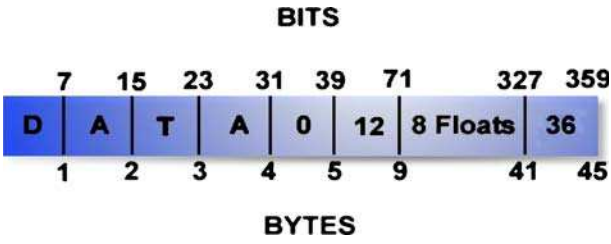

Fig. 15 Example packet sent from X-Plane

while the machine containing the controllers is the client. Thus, socket communication is first implemented in the *C* code to enable and allow communication. Next, the client waits to receive the output from the server, and once the client receives a packet its data is placed in a buffer. The client has an *X-Plane* packet stored, but the data can not be used because it is in network byte order. To overcome this problem, the bytes of each data value are swapped and the readable values are stored in a two dimensional array for easy access. Next, a second two-dimensional array is created to mirror the first, and any data alterations are made in this array. As controllers generate new values for the simulator, variables inside the code receive the data and change the values in the second two-dimensional array; the original data is not 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.

Additional functions may 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, so a global declaration may be necessary along with a normalization function. Thus, when a new helicopter is tested, only the global value is changed.

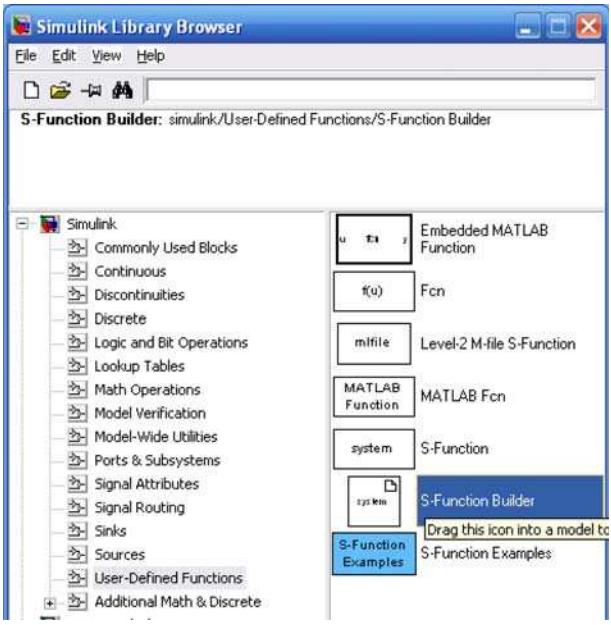

Fig. 16 The SIMULINK library browser

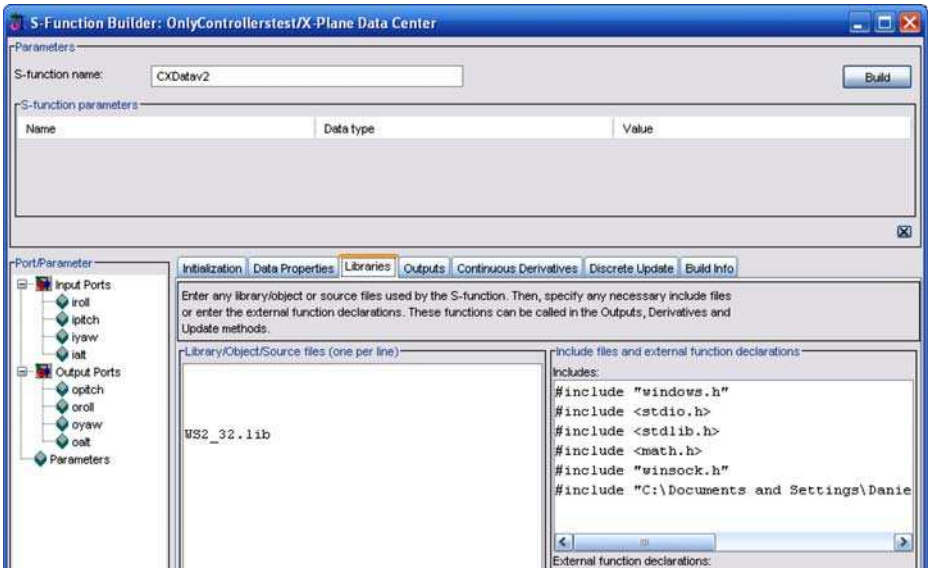

Fig. 17 The S-Function block libraries tab

The *S-Function* builder is used to implement the *C* code into a *SIMULINK* block for interaction with the controllers. The *S-Function* builder is included in *SIMULINK* and it is found in the *User-Defined Functions* category of the *Library Pane* (see Fig. 16). Several items must be added inside this block. 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 as shown in Fig. 17. Any extra functions used by the main function must be included in this pane because *SIMULINK* will look at these files for any function calls. The I/O ports are then created in the *Data Properties* tab (straight forward, it varies depending on the controller built). The Sample mode is set to 'inherited' in the initialization tab to allow the block to sample as quickly as the information is received from *X-Plane*. Figure 18 shows the I/O ports and sampling mode.

The main *C* function is implemented into the *S-Function* block. Updates in *SIMULINK* occur at each time step, thus, any infinite loops must be eliminated or the block will run continuously and the time step within *SIMULINK* will never be updated. The *main()* function encapsulation is removed because *SIMULINK* does not recognize functions within the *Outputs* tab. The code is entered in the *Outputs* tab dialog box consisting only of those items shown in Fig. 19. Once the code is entered, an interface between the *SIMULINK* I/O ports and the internal *C* Code variables is created. It is important to note that *SIMULINK* usually does not run in real time, but because *SIMULINK* is forced to run in real-time because it updates with *X-Plane*, which runs in real-time.

5.5 X-Plane Extra Features

X-Plane has been chosen because of its many features. For example, Fig. 20 illustrates various forces acting on the different control surfaces of a helicopter – the green lines show a 22 knot wind (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

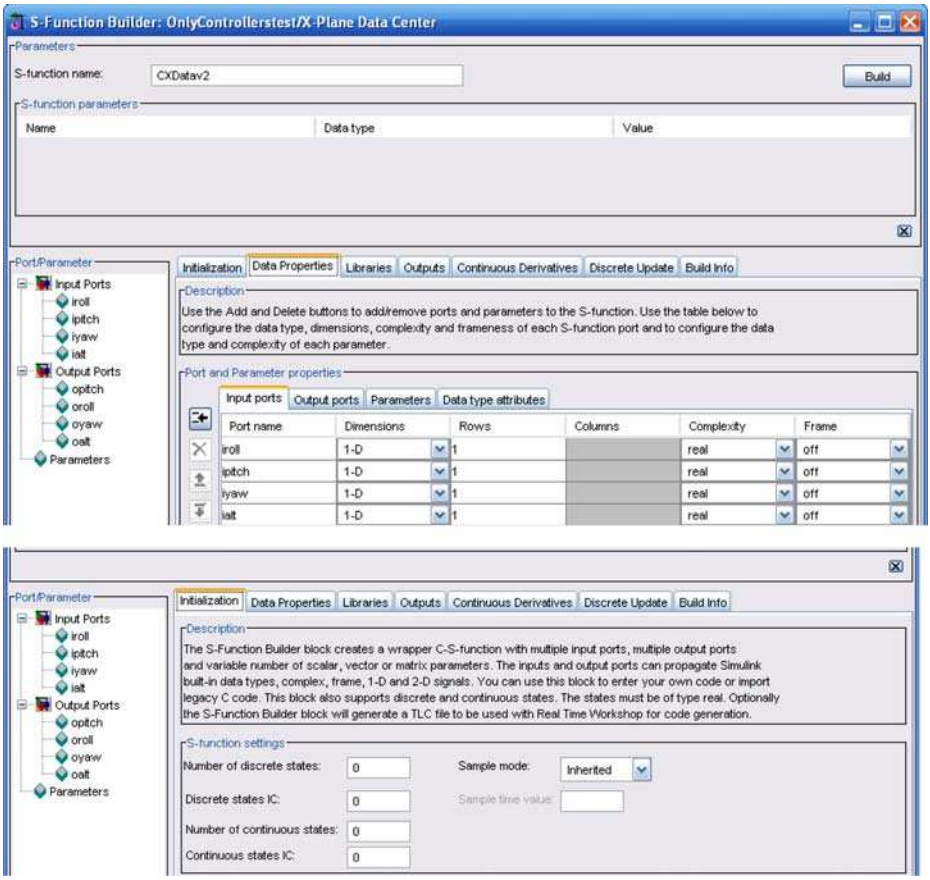

Fig. 18 The S-Function block data properties and initialization tabs

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. To test the fuzzy logic controllers, tail failures were implemented by using the failure panel. In addition, the flight path feature was utilized to determine the vehicle path when in autopilot mode to analyze tightness of the controller for holding a particular position (Fig. 21). *X-plane* has the ability to communicate with multiple aircraft; swarms may be simulated to see how vehicles interact with each other. A host of other features is available and may be viewed in UDP documentation [4]. In addition, future versions of *X-Plane* will incorporate wheel/ground interaction to allow simulation of ground vehicles.

6 Implementation Issues and Examples

As sample implementations, conversion of controllers designed for miniature unmanned helicopters (*RAPTOR* series) with very strict and limited payload, power, and processing capabilities is presented. Controllers designed are PID, Fuzzy Logic and LQR [8–10]. Figure 22 shows a system overview of the involved steps.

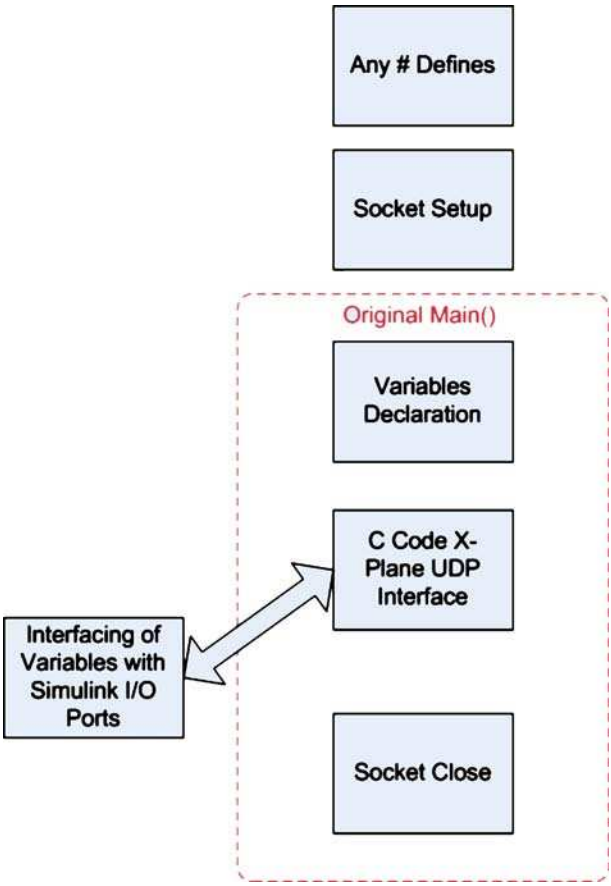

Fig. 19 Items included in the *c* code under the outputs tab

It is known that PID controllers are self contained in one *SIMULINK* model file, while Fuzzy Logic controllers contain a *SIMULINK* model file and three fuzzy inference system files. LQR controllers are implemented in a *SIMULINK* model file with several *MATLAB* script files. It is stated that while 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. Figures 23, 24, 25 show the steps necessary to convert the three controller types mentioned above.

The hardware configuration chosen for implementation includes an autopilot printed circuit board (PCB) with 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]. Once the controllers were designed, each controller was tested within *SIMULINK* to determine correct operation for the target vehicle platform.

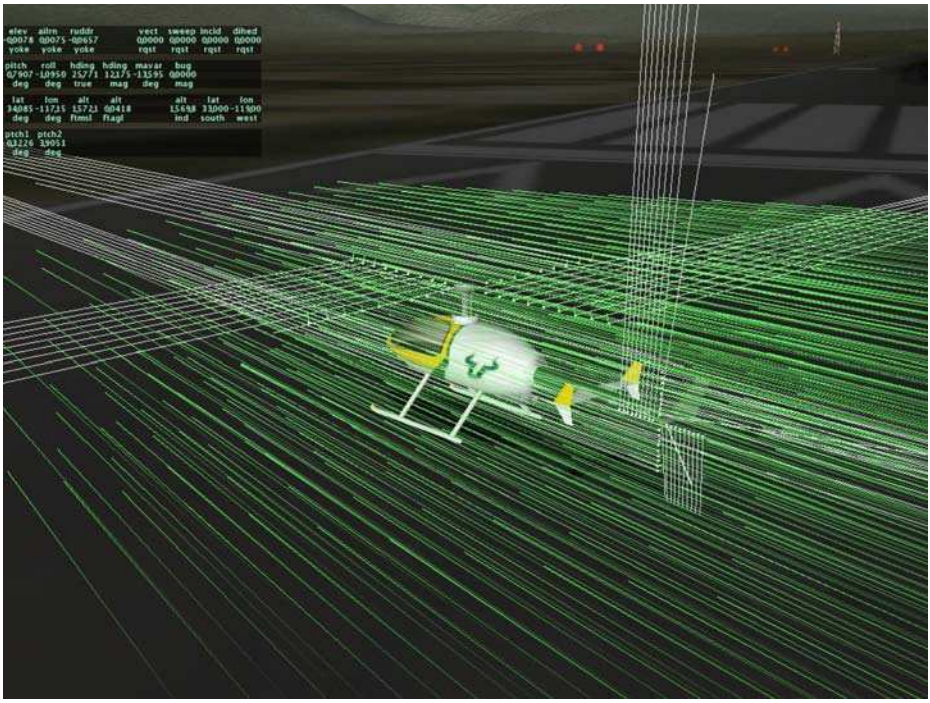

Fig. 20 Forces acting on the raptor 90 control surfaces

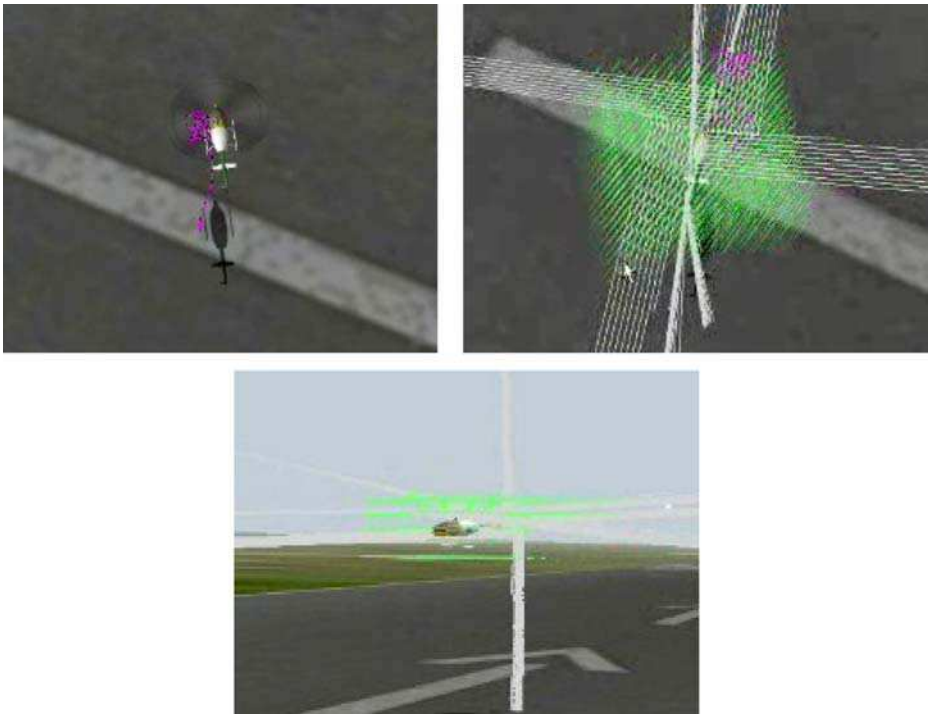

Fig. 21 Raptor 90 hovering with fuzzy logic controller in 11 knot wind

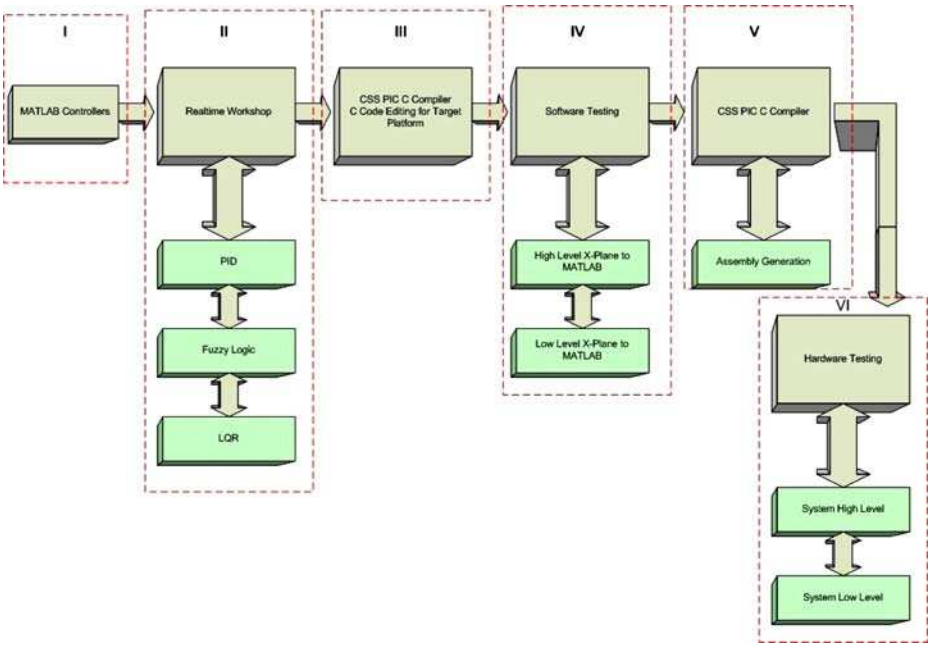

Fig. 22 Detailed steps

During the conversion from *MATLAB* to *C*, procedures are straightforward as presented in Section 3, except that no *Microchip* microcontrollers existed in the target list; so the 8-bit generic processor was selected (see Fig. 3).

The steps outlined in Section 4 for assembly code generation were followed using the *CCS PIC C Compiler* targeted for *Microchip PIC 18XXX* microcontrollers. This allowed for easy generation of the assembly language for specific types of microcontrollers without tediously programming necessary changes in Assembly. Before the assembly code is generated, pins are assigned to the variables as shown in Fig. 26 and all necessary conversion functions are implemented. The assembly code is then generated and implemented on the *PIC 18F4620*.

Concentrating in the RAPTOR 90 helicopter and PID controller design, the *X-Plane* UDP block is placed in *SIMULINK* with the PID controllers as shown in Fig. 27. A model

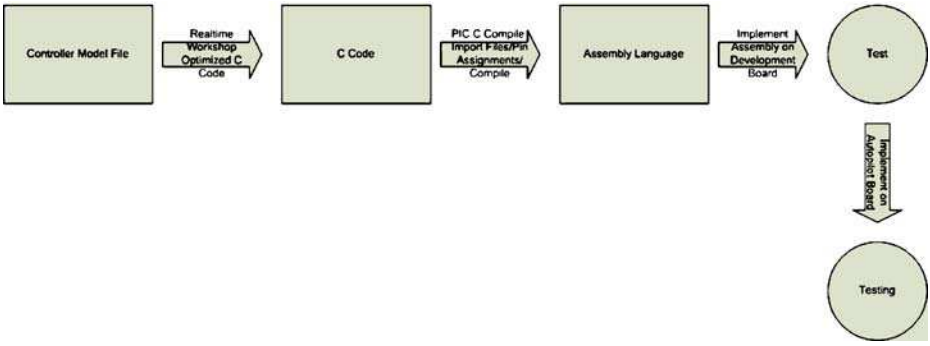

Fig. 23 PID controllers

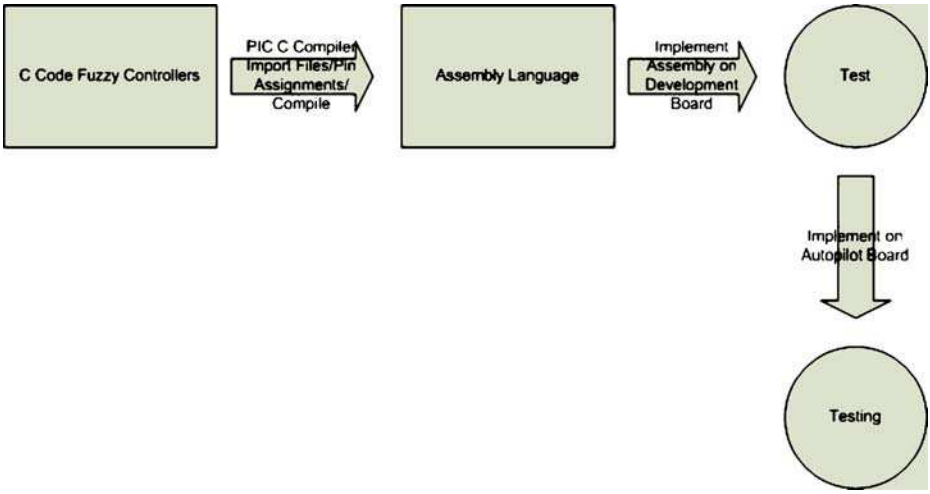

Fig. 24 Fuzzy logic controllers

was created and tested with the controllers until it performed non-aggressive flight scenarios as expected.

In general, controller inputs and outputs may change depending on the controller type and what it is used for. In the *C* code, all necessary variables are listed at the top [2]. Thus, the code can be easily changed to accommodate changes to the controllers.

Controllers utilize radians rather than degrees; controllers output a collective value between −1 and 1 while *X-Plane* accepts a value between 0 and 16 (depending on the helicopter); hence, special functions are created in a separate *C* file to normalize the collective input to *X-Plane* and convert degrees to radians. A global variable was also created for the collective to allow easy switching between types of helicopters.

In addition to implementing UDP communication between *SIMULINK* and *X-Plane* in *C*, a Java interface was created allowing for portability between systems.

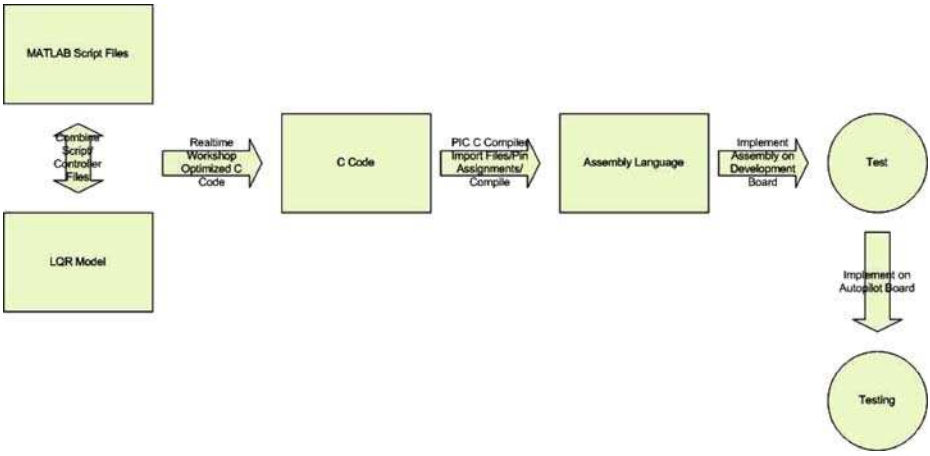

Fig. 25 LQR controllers

```
#ifndef ODE1_INTG
#define ODE1_INTG
/* ODE1 Integration Data */
typedef struct IntgData_tag {
  real_T *f[1];                        /* derivatives */
} ODE1_IntgData;
#endif

/* External inputs (root inport signals with auto storage) */
typedef struct _ExternalInputs_OnlyControllers_tag {
  real_T roll;                         /* '<Root>/roll' */
  real_T roll_sp;                      /* '<Root>/roll_sp' */
  real_T roll_trim;                    /* '<Root>/roll_trim' */
  real_T pitch;                        /* '<Root>/pitch' */
  real_T pitch_sp;                     /* '<Root>/pitch_sp' */
  real_T pitch_trim;                   /* '<Root>/pitch_trim' */
  real_T yaw;                          /* '<Root>/yaw' */
  real_T Height;                       /* '<Root>/Height' */
  real_T yaw_sp;                       /* '<Root>/yaw_sp' */
  real_T height_sp;                    /* '<Root>/height_sp' */
  real_T pedal_trim;                   /* '<Root>/pedal_trim' */
  real_T height_trim;                  /* '<Root>/height_trim' */
} ExternalInputs_OnlyControllers;

/* External outputs (root outports fed by signals with auto storage) */
typedef struct _ExternalOutputs_OnlyControllers_tag {
  real_T sig_lateral;                  /* '<Root>/sig_lateral' */
  real_T sig_longitudinal;             /* '<Root>/sig_longitudinal' */
  real_T sig_pedal;                    /* '<Root>/sig_pedal' */
  real_T sig_collective;               /* '<Root>/sig_collective' */
} ExternalOutputs_OnlyControllers;

/* Real-time Model Data Structure */
struct _RT_MODEL_OnlyControllers_Tag {
  const char *errorStatus;
```

Fig. 26 Exported controller variables

7 Discussion and Conclusions

This paper aimed at providing e a standardized method for controller implementation from *SIMULINK* onto a PCB. As such, it presented a step-by-step approach to covert the *SIMULINK-based* controller block set 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*, 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

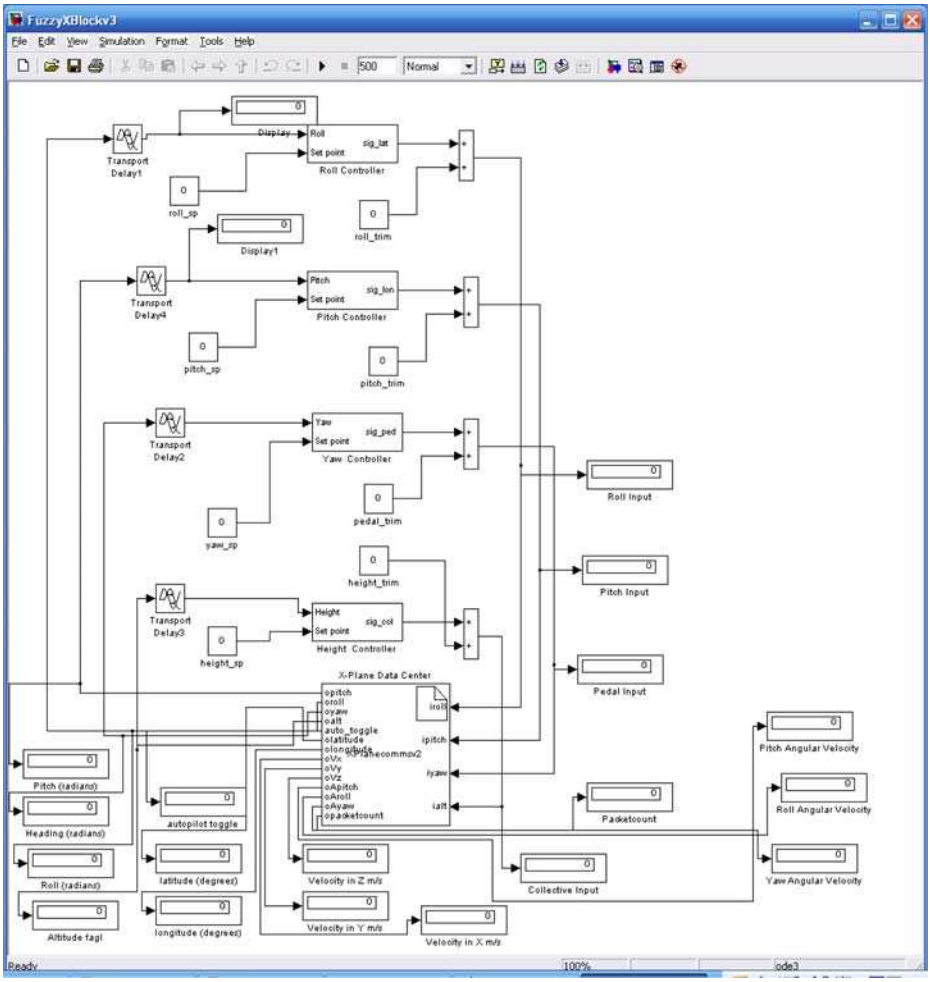

Fig. 27 Actual PID implementation in SIMULINK

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.

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. Future work will include testing of newer Raptor 90 PID controllers as well as implementation of these controllers on the same autopilot board used for the small UGV platform. 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. In addition, interactions between landing a small VTOL on an ATRV Jr. have been tested through *X-Plane* and several of the plug-ins associated with the simulator. While the current version of the *X-Plane/SIMULINK* communication is applicable to non-predictive time based controllers, the *SIMULINK* solver has problems with maintaining real-time when the X-Plane *SIMULINK* block is run with controllers. However, this is not true if the bock is run separately, so the problem lies with the *SIMULINK* solver and socket communication. This problem will be solved in a future version of the *X-Plane/SIMULINK* communication block through working with Mathworks and perhaps implementing the block as two separate blocks.

Acknowledgements This research has been partially supported 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.

References

1. Incorporated, C.C.S.: C Compiler Reference Manual. Brookfield (2005)
2. Fisher, A.E., Eggert, D.W., Ross, S.M.: Applied C: An Introduction and More. McGraw-Hill, New York (2001)
3. Walker, I.M., et al.: Simulation for the next generation of civilian airspace integrated UAV platforms. In: Proceedings, AIAA modeling and simulation technologies conference and exhibit. Rhode Island, (August 2004)
4. Meyer, A.: X-Plane UDP Reference Manual (2005)
5. Kreider, L. http://www.flightmotion.com/docs/faa_approval.htm, FAA Approval Document
6. Fidelity Flight Simulation Homepage, http://www.flightmotion.com
7. Liu, P., Meng, M., Ye, X., Gu, J.: An UDP-based protocol for internet robots. In: Proceedings, 4th World Congress Intelligent Control and Automation. China, June (2002)
8. Alvis, W., Castillo, C., Castillo-Effen, M.: Small scale helicopter control design. Internal report for the Center for Robot Assisted Search and Rescue (CRASAR), USF (2004)
9. Castillo-Effen, M., Valavanis, K.P.: Control of miniature rotorcraft: Linear quadratic methods. Internal report for the Center for Robot Assisted Search and Rescue (CRASAR), USF (2005)
10. Castillo, C., Alvis, W., Castillo-Effen, M., Valavanis, K., Moreno, W.: Small scale helicopter analysis and controller design for non-aggressive flights. In: Proceedings IEEE International Conference on SMC, October (2005)