REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 7/26/2006	3. REPORT TYPE AND DATES COVERED FINAL; 6/20/2005 - 3/19/2006

4. TITLE AND SUBTITLE Development of a Novel, Two-Processor Architecture for a Small UAV Autopilot System	5. FUNDING NUMBERS W911NF-05-1-0324
6. AUTHOR(S) Robert Klenke, Ph.D.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Virginia Commonwealth University, School of Engineering, Electrical Engineering, Box 843072, 601 W. Main St., Richmond, VA 23284	8. PERFORMING ORGANIZATION REPORT NUMBER 529864
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) U. S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709-2211	10. SPONSORING / MONITORING AGENCY REPORT NUMBER 48763.1-CI-II

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The potential uses for small UAVs are quite varied and include such applications as targeting and even small weapons delivery to sensing, surveying, and search and rescue. This class of UAVs typically have a gross weight of less than 100 lbs and their limited payload capacity, interior volume, and power supply, place significant constraints on on-board systems, including the autopilot system used to control them. On the other hand, the capabilities of the aircraft themselves continue to grow which imposes larger computational requirements on the flight control system portion of the autopilot – that portion of the autopilot system that actually controls the vehicle's dynamic behavior.

This project developed a new autopilot architecture that utilizes a state-of-the-art processor to perform the flight control function for small UAVs. This system provides a light-weight, low-cost autopilot system that is flexible and powerful enough to be used for a wide variety of UAV platforms and because of its open architecture and basis on COTS processors, is easily upgradeable to track the state-of-the-art in commercial processor design.

14. SUBJECT TERMS Unmanned aerial vehicle, autopilot, flight control	15. NUMBER OF PAGES 46
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT **UNCLASSIFIED**	18. SECURITY CLASSIFICATION ON THIS PAGE **UNCLASSIFIED**	19. SECURITY CLASSIFICATION OF ABSTRACT **UNCLASSIFIED**	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev.2-89)
Prescribed by ANSI Std. 239-18
298-102

Research Agreement No. W911NF-05-1-0324 Final Report

July 5, 2006

Prepared for:

Army Research Laboratory

By:

Virginia Commonwealth University

I. Background

The potential uses for small UAVs are quite varied and ever expanding. These range from targeting and even small weapons delivery in the military arena to sensing, surveying, and search, applications in the civilian arena. This class of UAVs typically have a gross weight of less than 100 lbs and their limited payload capacity, interior volume, and power supply, place significant constraints on on-board systems, including the autopilot system used to control them. On the other hand, the capabilities of the aircraft themselves continue to grow. This class of UAVs includes turbine powered aircraft capable of speeds approaching 300 knots, helicopters and other vertical takeoff vehicles that are dynamically unstable, and aircraft designed to be carried in unique locations on larger launch vehicles that have less than ideal aerodynamic characteristics. This in-turn, imposes larger computational requirements on the flight control system portion of the autopilot – that portion of the autopilot system that actually controls the vehicle's dynamic behavior. At the same time, the computational requirements of the mission software side of the autopilot system – that portion of the autopilot system that controls the vehicle's actions in response to mission requirements - are also increasing. Mission requirements such as adaptive behaviors, swarming, and in-flight aircraft reconfiguration, all in concert with increasing requirements for vehicle autonomy, require large computational resources. Furthermore, the nature of these mission-related computations, in terms of such things as precision (fixed vs. floating point), memory requirements, latency, etc., are very different that that of the computations required for flight control.

In addition to using 8- or 16-bit microcontrollers and associated technology that is becoming increasingly dated, most commercial, and even experimental, autopilot systems designed for this class of UAVs utilize a single processor architecture. In this architecture, the single processor must execute both the flight control algorithm software, as well as the mission application software. This compromise can significantly degrade the performance of the autopilot system in both areas.

This proposed project will develop a new autopilot architecture that will utilize two, state-of-the-art processors to perform the flight control and mission application functions. Each processor will be optimized to perform its specific function. The two processors will communicate via a dedicated bus. A software API will be provided to facilitate development or porting of flight control and mission software to the new hardware platform. Several example applications will be developed on the platform to demonstrate its effectiveness. The architecture of the developed system will be open and completely documented. This will facilitate adding additional functionality to the system, or even porting it completely to new processor architectures as they become available. Development of the proposed system will provide a light-weight, low-cost autopilot system that is flexible and powerful enough to be used for a wide variety of UAV missions and because of its open architecture and basis on COTS processors, and is easily upgradeable to track the state-of-the-art in commercial processor design.

II. Proposed System Architecture

A block diagram of the proposed architecture for the UAV autopilot system is shown in Figure 1. The proposed system consists of two processor boards, one for the flight control system, and the other for the mission processor. Each of the two processors will likely be housed on its own board, and in fact, the processor board for each system will likely be a pre-built COTS board.
The flight control system will be responsible for executing the algorithms necessary to actually control the vehicle itself. Typically, these algorithms are a form of proportional, integral, derivative (PID) control

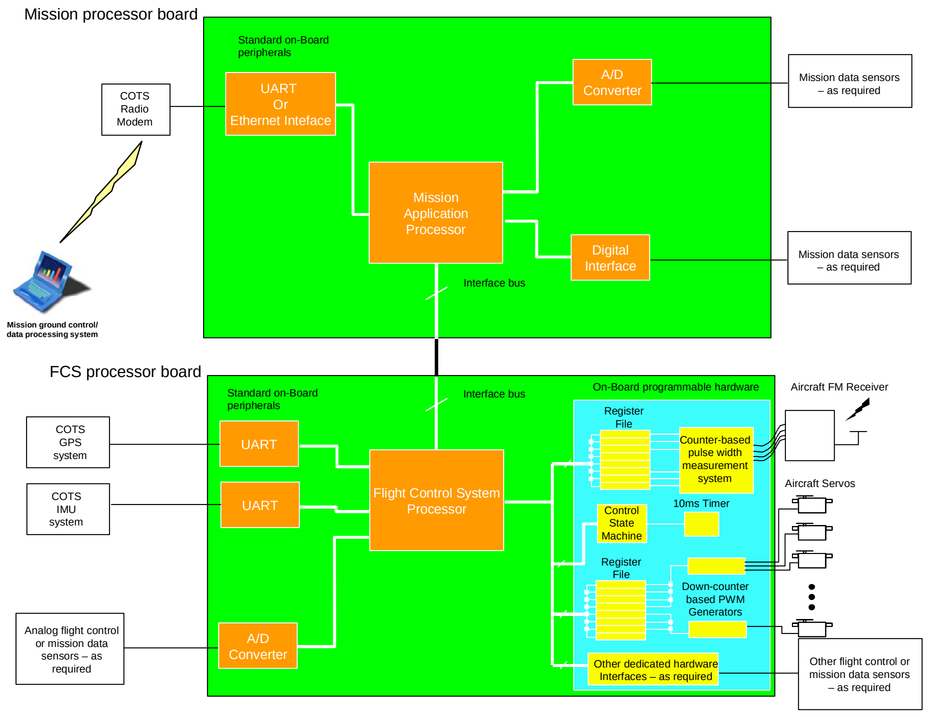

Mission processor board
Standard on-Board peripherals
UART Or Ethernet Inteface
A/D Converter
Mission data sensors – as required
COTS Radio Modem
Mission Application Processor
Digital Interface
Mission data sensors – as required
Interface bus
Mission ground control/ data processing system
FCS processor board
Standard on-Board peripherals
Interface bus
On-Board programmable hardware
Aircraft FM Receiver
COTS GPS system
UART
Register File
Counter-based pulse width measurement system
COTS IMU system
UART
Flight Control System Processor
10ms Timer
Aircraft Servos
Control State Machine
Register File
Down-counter based PWM Generators
Analog flight control or mission data sensors – as required
A/D Converter
Other dedicated hardware Interfaces – as required
Other flight control or mission data sensors – as required

Figure 1. Proposed autopilot system architecture.

loops. These control loops measure the roll, pitch, yaw, position, speed, and altitude of the aircraft along with their rates of change, and determine the required position of the aircraft's control surfaces necessary to achieve the desired aircraft state in terms of heading, speed, position, and altitude. The complexity and required processing power for these algorithms will vary depending on the type of vehicle the UAV is, and the control laws the user implements to control it.

The flight control system board will contain the processor selected for this system along with its associated memory and I/O system. Most likely this processor will be a microcontroller intended for real-time control systems. Included on this board will be user-programmable custom hardware in the form of a Field Programmable Gate Array (FPGA). The FPGA will be used to efficiently implement custom interfaces to other systems and sensors on the UAV that the FCS needs to communicate with. This includes interfaces to the COTS remote control (RC) receiver that is typically used to control the UAV when it is in manual mode, and the COTS RC servos that are typically used to move the flight control systems in a small UAV. As shown in Figure 2 standard RC servos are controlled by a pulse width modulated (PWM) signal with a nominal pulse period of 20 ms. The pulse period can vary between approximately 10 ms and 25 ms and the servos will still perform normally. The pulse width varies from a

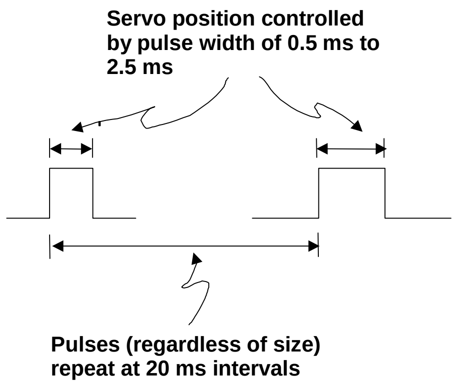

Servo position controlled by pulse width of 0.5 ms to 2.5 ms
Pulses (regardless of size) repeat at 20 ms intervals

Figure 2. Standard RC servo PWM control signals

minimum of 0.5 ms to a maximum of 2.5 ms. A pulse width of 1.5 ms corresponds to the servo arm being in the center of its travel, and on most servos, a pulse width of 1.0 ms and 2.0 ms corresponds to the servo arm being 60 degrees off-center in either direction. The position of the servo arm is proportional to the width of pulse and there is some variability among servos as to the correspondence between pulse width and servo arm position.

In a standard model aircraft control system, the airborne receiver generates the PWM signal at the proper repetition rate with the pulse width as commanded by the stick position on the transmitter. In the UAV control system, the width of the PWM pulses generated by the receiver must be measured to provide the control system with the command inputs from the transmitter. In addition, the control system must generate the proper PWM signals on the outputs to the servos with the pulse width as required by the application. Although the measurement and generation of these control pulses can be done directly by the FCS processor using its general-purpose digital I/O lines, doing so is not the most effective way to perform this task for several reasons. First, because the pulse widths and periods are on the scale of milliseconds, their time scale is extremely long in terms of the instruction execution rate of the FCS processor, which will be running at a frequency of tens or hundreds of MHz. This means that the FCS processor must waste enormous amounts of time in counting loops to measure or generate the pulses, or internal timers and interrupts must be used. Using timers and interrupts to generate the pulses makes the FCS programming much more complex, and can lead to problems achieving the computational throughput required by the flight control algorithms themselves. Second, using the FCS processor's general purpose I/O lines to interface directly to the receiver and servos directly limits the number of channels of control information and servos that can be controlled and limits the flexibility of the system because any time this numbers are changed, the FCS code itself will have to be modified.

A better solution that is shown in Figure 1 is to use the FPGA to implement a custom hardware interface to the RC receiver and servos. In this way, the custom hardware can deal with the long time frame of the standard RC pulses and convert their length directly into digital information that can simply be read or written by the FCS processor as needed.

In addition to the interface to the receiver and servos described above, the FCS must interface to sensors and other hardware aboard the aircraft. This includes interfacing to a COTS Global Positioning System (GPS) receiver, which provides both position, and velocity information to the FCS. This interface is most often performed via a standard communications ports such as RS-232. The FCS must also interface to an Inertial Measurement Unit (IMU). The IMU contains solid-state gyros and accelerometers and uses these to determine the attitude of the UAV within the three dimensions of pitch, roll, and yaw. This attitude information is send to the FCS for use in the aircraft control algorithms. Most IMU's also communicate with the FCS processor using an RS-232 interface. Finally, the FCS must also interface to various sensors in the aircraft. These may include a magnetic compass, and airspeed, altitude, or surface pressure sensors. These sensors will most likely be analog in nature, so an Analog-to-Digital Converter (ADC) will be required for the FCS to read their outputs. Here custom hardware on the FPGA within the FCS can be utilized to select the proper sensor for reading, start and monitor the A-to-D conversion process, retrieve the digital result from the ADC, and send it to the FCS processor at the requested time.

The standard peripherals required in the FCS may be included in the microcontroller selected for the FCS itself, or integrated with the microcontroller on the FCS board. The COTS board selected for the FCS will also have a development environment including a C or C++ compiler, and may include a simple operating system to ease software development and porting.

The mission processor system will be responsible for executing the application software necessary to complete the required UAV mission. This may include such things as complex search algorithms to find the shortest flight path to visit a specific set of waypoints for surveillance applications, developing the most efficient search pattern given current wind and weather conditions for search and rescue applications, or adaptive, intelligent control algorithms for autonomous, collaborative, multiple-UAV swarming for combat support operations.

The mission processor board will contain the processor selected to execute the mission application software as well as its associated memory and I/O systems. The mission processor will be a high performance processor that is capable of executing the complex algorithms inherent in the application software. As such, it is likely to have a much higher clock speed and processing throughput and a larger memory system than the FCS processor. The mission processor will also run a full blown operating system such as Linux or Windows CE and have a powerful and robust C or C++ based development system. The mission processor board will also include a high-speed UART or perhaps an Ethernet system that can be used to interface, via a radio modem, to the ground control station (GCS). The GCS will be used to monitor the performance of the autopilot and the aircraft both during the development stage, and later during the operational phase. However, the system will be capable of operating autonomously without an interface to the GCS. Finally, the mission processor board may also have interfaces to mission-related sensors, which may include high-resolution pictures or video. The data from these sensors may actually be processed on the mission processor board (e.g., image recognition for targeting), or compressed and/or stored on-board or transmitted directly to the ground station.

There will be in interface between the FCS processor board and the mission processor board. This interface will be implemented using a bus system for data transmission and reception. A standard bus, such as the PC-104 or compact PCI bus may be utilized to implement this bus, or a custom bus standard may be implemented using the general-purpose digital I/O's of the two processors.

To facilitate development or porting of flight-control or mission software to this architecture, standard API routines to communicate with the peripherals on each board and to communicate between the two processors will be developed. In the case of the FCS processor, the API routines will facilitate communications between the processor and the GPS system, the IMU, the RC receiver and servos, and the other installed sensors. Examples might include a routine called *receive_LLA_report()* that receives the latitude, longitude and altitude (LLA) report from the GPS system, or *get_pressure_altitude()* that uses the A/D converter to read the analog pressure altitude sensor and converts it to feet or meters of altitude above sea level. In addition, the API for the FCS will include routines that communicate with the mission application processor. Examples of this API might include a *report_position()* routine that can be used to transmit the current position (and altitude, heading, speed, etc.) to the mission processor, or a *go_to_waypoint()* routine that can be used to direct the FCS to fly the aircraft to a specific set of GPS coordinates.

In the case of the mission processor, the API will include routines that interface to the mission sensors, process mission data, communicate with the GCS, and communicate with the FCS. An example of the first type of routine might be one called *establish_video_link()* that sets up a continuous link between an on-board video camera and the GCS. A routine called *compress_and_store_image()*, which compresses a still image from an on-board camera, or video capture board, and stores it in an on-board flash memory system might be an example of the second type of routine. A typical GCS communication routine might be called *send_aircraft_state()*, which would transmit the current speed, altitude, heading, position, and roll, pitch, and yaw angles across the radio modem to the GCS. Finally, routines such as *maintain_specified_altitude()*, which transmits a new altitude to be held, or *orbit_waypoint()* might be examples of routines used to communicate with the FCS.

III. Task and Deliverables Description

The goal of the complete project is to develop a new architecture and implementation of an autopilot system for small UAV's as described above. In order to develop this system, the following tasks would be required to be performed:

Task 1 – Survey the available off-the-shelf CPU and FPGA systems to determine potential candidates for the FCS processor board. Select the most appropriate candidate or candidates.

This task will use the requirements developed in previous autopilot development work by the proposers to select the best candidate board for implementation of the FCS processor board. The requirements for the mission processor board as they pertain to the interface between the two boards will also be considered in the selection process. Although the goal is to select a readily available COTS board to satisfy the requirements of the FCS, it may be necessary to develop some custom hardware or printed circuit boards to interface more than one commercial component together to implement the FCS board. The deliverable from this task will be the selection, purchase, and setup of the FCS processor board and associated development environment to begin implementation of the autopilot system.

Task 2 – Implement basic FCS system on selected FCS processor board and flight test.

This task will utilize the selected FCS from Task 2 to implement a basic autopilot system and flight test it. This is the most effective way to evaluate the FCS for its ability to perform the required functions. In this task, the FCS architecture developed by the proposer and the VCU students for the AUVSI competition UAV will be utilized and the mission processor function will be performed by the ground control station. The deliverable from this task will be a flight test demonstration of the FCS controlling a small UAV.

Task 3 – Develop requirements for the autopilot for the targeted class of applications.

This task will develop and document the requirements for the autopilot system. This includes both the FCS and mission processor boards. Typical algorithms for both systems will be examined to help determine the processing throughput and memory systems required for each processor. Characteristics for the targeted types of UAV's will be used to help determine constraints such as maximum weight and power consumption. Typical sensor systems will be examined to determine interface requirements for both processors. The deliverable from this task will be a requirements document that will describe the requirements for each part of the autopilot system.

Task 4 – Survey the available off-the-shelf CPU systems to determine potential candidates for the mission processor board. Select the most appropriate candidate or candidates.

This task will utilize the requirements developed in Task 3 to select a candidate for the mission processor board. The processing power, form factor, development environment and operating system, availability, and cost will be major factors in selecting this system. The deliverable from this task will be the selection, purchase, and setup of the mission processor board and associated development environment to begin implementation of the complete autopilot system.

Task 5 – Implement mission processor board and integrate with FCS processor board and flight test.

This task will develop the processor board selected in Task 4 into the mission processor. This involves interfacing the mission processor board to the FCS, the communications link, and the mission sensors. Included in this task will also be the physical packaging of the entire system into the final form factor. Once the final system is developed, it will be flight tested using the FCS algorithms developed in Task 3, and a test mission algorithm. The mission algorithm will be something similar to a "shortest path" algorithm that, given a set of waypoints up-loaded from the GCS for the UAV to visit, will determine the order of the waypoints that will minimize the distance traveled by the UAV and direct the aircraft along that path. The deliverable from this task will be a flight demonstration of the completed autopilot system controlling a small UAV.

Task 6 – Develop standard API for mission processor and FCS boards.

This task will develop the complete API that will be used by the software on the FCS and mission processor board to communicate with the associated hardware on those boards, and to communicate between the FCS and mission processor software. The first part of this task will be the process of gathering the requirements for the API itself and designing the level of interaction between the software components and between the software and hardware. Input from potential users of the autopilot system will be gathered to assist in this process. All of the routines will be tested in operation on the autopilot system. The deliverables from this task will include an API reference guide that will describe each of the API routines, how they function, what their arguments and return values are, and any known issues with them.

Task 7 – Port example FCS and mission algorithms provided by ARL onto autopilot system and demonstrate

This task will demonstrate the autopilot system using more sophisticated flight control algorithms provided by ARL. A potential candidate for the flight control algorithm is the learning-based technique being developed by Oregon State under ARL direction. For the mission algorithm, the collaborative software for UAV swarming demonstrated by ARL and APL might be used. The deliverable from this task will be a flight demonstration of one or more UAVs being controlled by the autopilot executing the selected algorithms. The number of UAVs in the demonstration will of course, be limited by the availability of UAV platforms and the number of the developed autopilot systems that have been produced.

Task 8 – Develop and deliver documentation of autopilot system.

> This task will entail the development of complete documentation of the developed autopilot system, the API, and the demonstration software. The deliverables from this task will include an autopilot user's guide that explains the use of the autopilot system, how to program it, and how to add additional hardware components onto it. Also included will be an autopilot reference guide that will describe in detail how the system hardware and software are implemented. This document will be a reference that will be useful in the future in porting the system to newer state-of-the-art processor boards. Finally, a user's guide for the demonstration algorithms described above will be developed. This document will be useful for users who wish to port their mission or flight control software to the autopilot in place of the demonstration algorithms.

IV. Completed Work and Included Documents

The initial funding and completion date specified under the subject research agreement were sufficient only to complete Tasks 1 and 2. The flight control system (FCS) developed under these tasks is detailed in the FCS final report. The Ground Control Station (GCS) is detailed in the GCS final report. Finally, a brief Users Guide which details how to start up the GCS software and configure it for flight is included. It is assumed that the operator is completely familiar with the operation of RC aircraft such as the FQM-117B RCMAT, therefore, details on its operation are not included.

VCU Autopilot System
Flight Control System
Final Report

July 5, 2006

I. FCS Processor

This project had resulted in the development of an operational FCS for small UAVs based on an embedded processor. This embedded processor is coupled with Field Programmable Gate Array (FPGA) resources to implement custom hardware and sensor interfaces. The FCS is based on the Suzaku processor board, which uses a Xilinx Spartan-3 FPGA. The Spartan-3-based Suzaku board implements a 50 MHz, 32-bit processor called the Microblaze. The Microblaze is a *soft-core* processor. That is, the processor is not actually built in dedicated silicon on the device like the PowerPC processor in the Virtex-II, but is implemented using standard FPGA logic resources just like user-defined custom logic. This approach makes the processor architecture more flexible in that the application designer can actually change it, but results in lower overall processor performance.

In addition to the Spartan-3 device upon which the Microblaze processor can be implemented, the Suzaku board contains 16 M Bytes of SDRAM and 8 M Bytes of Flash memory. The Spartan-3 device on the Suzaku comes pre-configured with Microblaze processor on it with a micro version of Linux called uCLinux. The GNU tool chain can also be used to develop applications for the Microblaze processor on the Suzaku. The Suzaku board is shown in Figure 1.

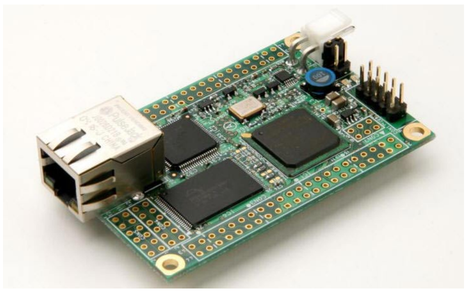


Figure 1. The Suzaku processor board

The salient characteristics of the Microblaze processor on the Suzaku board are listed in Table 1. For comparison, the characteristics of the Suzaku board used in the VCU autopilot are also included. In order to help assess the performance of the processors on each board, the Drystone integer and Linpack floating point benchmarks were compiled and executed on each board. Note that none of the processors on these boards have a built-in hardware floating point unit, so all floating-point operations are performed in software using numerical library routines.

Table 1. Suzaku board statistics

Processor Board	**Processor**	**Memory**	**Operating System**	**Drystone 2.1 MIPS**	**Linpack MFLOPS**
Suzaku	50 MHz 32-bit Microblaze RISC	16 M Bytes SDRAM 8 M Bytes Flash	uCLinux	NA	0.085

II. FCS System Architecture

The architecture of the FCS is shown in Figure 2. The processor, it's associated memory, and the uCLinux operating system is shown as a separate subsystem in the diagram. With the exception of device drivers needed to interface to peripherals, no modifications to the processor/memory/OS subsystem were made for this application. All of the custom hardware added for this application was interfaced to the processor via an external I/O bus, in the case of the Suzaku, the OPB bus.

A total of two UART modules were needed for this application to provide RS-232 interfaces to the Ublox Antaris GPS receiver and the 900 MHz radio modem to communicate with the ground control station (GCS). Depending on the specific processor board, some of these UARTs were available within the processor itself. Additional UARTs were added to the system by implementing them in the FPGA as required.

In addition to the UARTs, an interface was required to the Analog to Digital Converters (ADCs) used to digitize the outputs of the analog sensors. These analog sensors included the pressure sensors for airspeed and altitude, the IR sensors, where applicable, and the measurement of the state of the onboard batteries. The ADCs are designed interface to other devices via a Serial Peripheral Interface (SPI) bus. An SPI interface core was implemented in the FPGA resources to manage the SPI communications between the processor and the ADCs. This core allowed the processor to control and received data from the ADCs by performing simple I/O register read and write operations.

In addition to the sensors, the FCS needs to monitor the values of the input commands from the safety pilot. An external UAV switch handles the switching of control from the safety pilot to the FCS. This device, developed by Electrodynamics, Inc. (http://electrodynam.com) allows completely independent operation of the UAV by the safety pilot regardless of the state of the FCS. Switching of the inputs to the servos from the safety pilot receiver to the FCS is controlled by one of the channels from the safety pilot receiver. Thus, the safety pilot can turn over control of the UAV to the FCS and take it back at any time. The requirement for the capability to do this is in a system intended to test unproven hardware, software, and flight control algorithms is obvious.

Even with the UAV switch, however, the FCS still needs to read the inputs from the safety pilot receiver. The state of the channel which controls the safety pilot switch (i.e., the manual/autopilot mode switch) is used to tell the FCS when it should begin execution of the algorithms for control of the aircraft. The values on the channels which control the UAVs control surfaces, such as elevator, aileron, rudder, and throttle, are used by the FCS to determine neutral trim positions, and are telemetered to the ground control station by the FCS for debugging purposes.

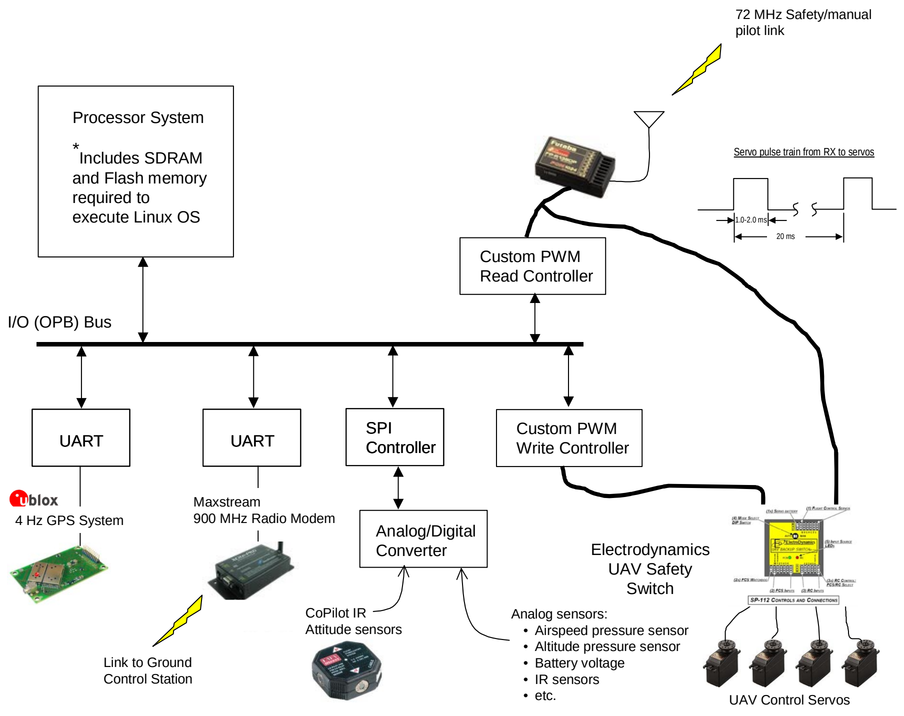

72 MHz Safety/manual pilot link
Processor System
*Includes SDRAM and Flash memory required to execute Linux OS
Servo pulse train from RX to servos
1.0-2.0 ms
20 ms
Custom PWM Read Controller
I/O (OPB) Bus
UART
UART
SPI Controller
Custom PWM Write Controller
ublox
4 Hz GPS System
Maxstream 900 MHz Radio Modem
Analog/Digital Converter
Electrodynamics UAV Safety Switch
CoPilot IR Attitude sensors
Analog sensors:
• Airspeed pressure sensor
• Altitude pressure sensor
• Battery voltage
• IR sensors
• etc.
Link to Ground Control Station
UAV Control Servos

Figure 2. Architecture of the Flight Control System

The pulse train output by the safety pilot receiver, which normally goes directly to the servos, is shown in Figure 2 as well. This pulse train has a period of 20 ms and the individual pulses are 1.0 ms to 2.0 ms wide. The servo output position is proportional to the width of the individual pulses. Because the period, and even the width of the pulses, is very long with respect to the processor's clock period, it is not efficient for the processor to be directly involved in reading the width of the individual pulses. Therefore, a custom core was developed using VHDL and the FPGA development tools, to read the incoming pulses from the safety pilot receiver. This core contains a 16-bit counter for each incoming channel. When the start of a control pulse is detected, this counter is started, and when the end of the pulse occurs, the counter is stopped. The input clock to the counter was setup with a period such that the difference between the resulting count for a 1.0 ms pulse and a 2.0 ms pulse resulted in 10 bits of resolution. Once the counter has stopped, the contents of the counter is placed in a register corresponding to the specific channel. To obtain the value of the current command pulse width for any incoming channel from the receiver, all the processor has to do is perform a simple I/O read from a specific I/O address.

In a similar manner, a custom core was developed to generate the required output pulses to operate the UAVs servos when the FCS is operating the aircraft. In this case, the processor performs a simple I/O write operation to the specific address for the servo in question. The core then uses this value to load a counter every 20 ms. This counter is then started and outputs a pulse that begins when the count begins and ends when the count reaches zero. The core repeats this process every 20 ms using the current values stored in each register. Therefore, the processor only needs to write a value out to a specific servo's control location when it desires to change the value of the current pulse width (i.e., when it wants to move that servo).

In addition to the components described above, additional hardware resources can be added to the user-programmable logic inside the FPGA as required to assist the processor in performing the flight control function. Examples might include custom cores to operate additional types of actuators such as pneumatic or electromagnetic components used in different types of "morphing" aerodynamic structures, custom cores to generate pulse width modulated signals for AC motor control for electric aircraft, or interfaces to complex sensors that use some other communications protocol besides RS-232 or SPI. These cores can be designed and simulated using VHDL and added to the FPGA as long as additional resources are available within it.

III. Flight Testing

During flight testing, the ground control station (GCS) developed as part of the VCU autopilot system is utilized to monitor the flight of the aircraft, the performance of the FCS, tune the FCS parameters, and record flight data. Figure 3 shows typical FSC displays from a test flight.

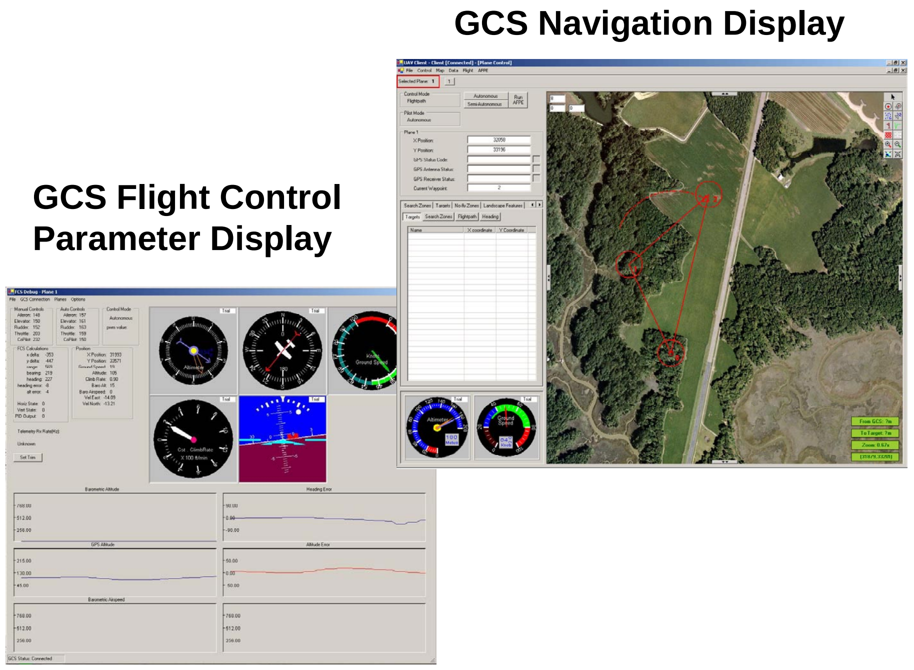
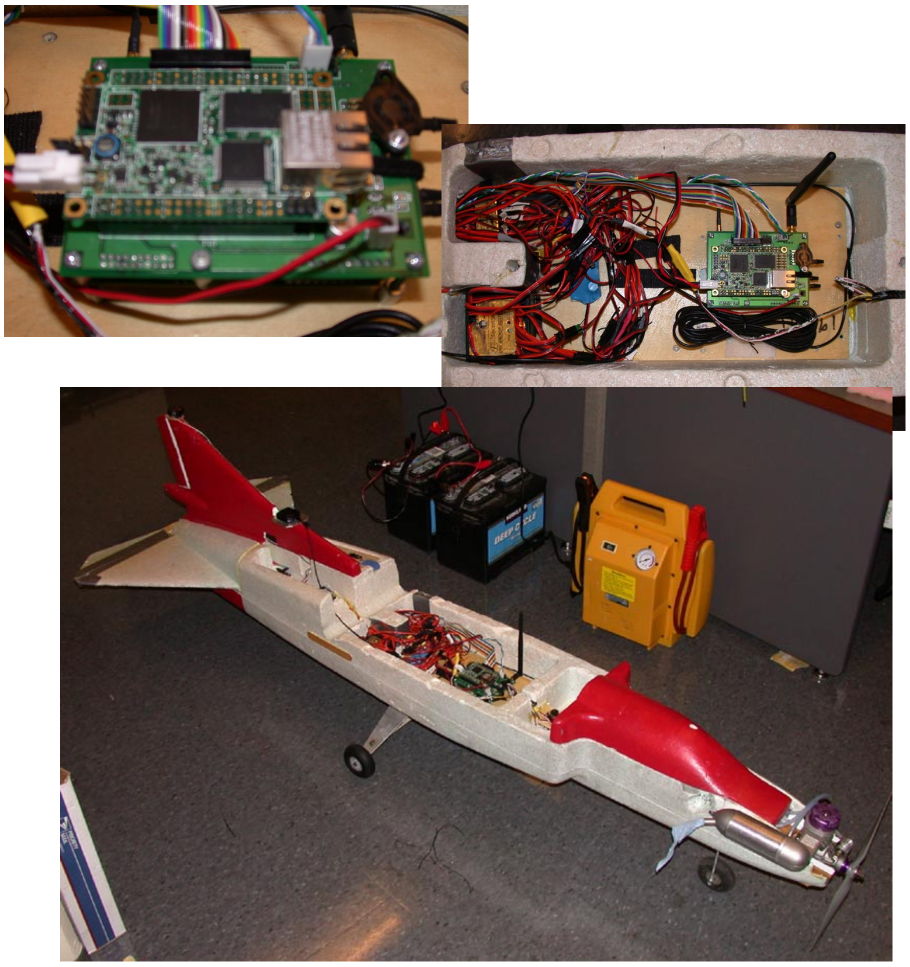


Figure 3. GCS Display

The FCS system was installed in an FQM-117B “Mig” aircraft provided by ARL for testing. Several test flights were conducted on the aircraft to insure proper operation. Figure 4 shows the system installed in the ARL Mig test aircraft.



4. FCS installed in the ARL Mig aircraft

VCU Autopilot System Ground Control Station Final Report

July 5, 2006

I. Summary

This report presents the details the design of the ground control system for VCU’s autonomous unmanned aerial vehicle (UAV) flight control system. The ground control system is used to control the operation of the aircraft in concert with the on-board flight control system.

The system has been developed to allow multiple UAVs to be flown and controlled simultaneously by one ground station. To facilitate this, the ground station software is based on a client-server model that allows multiple human operators to control the plane and/or analyze the real-time data being collected.

In addition to the multiple UAV capability, the system provides a user interface for photograph storage and display, automatic flight path assignment to visit all targets in the least amount of time, and the ability to alter mission data such as targets in mid-mission. The system has the ability to overlay an aerial photograph of the operational area onto a real-time map so that the location of the UAVs and targets can be related to the land features below.

The following sections describe the operation and implementation of the various components of the ground control system.

I. System Description

A. Functionality Required

The GCS system serves as the human interface to one ore more UAVs in the air. As such, it must provide the user with two basic functions: control over the flight of the UAVs themselves and easy, effective access to the data that the UAVs collect. Additionally, because multiple UAVs are under the control of one ground station, it is important that the GCS system be designed so that multiple planes can be controlled with the minimal amount of confusion. In order to accomplish this goal, the GCS was designed to allow for multiple operator terminals so that more than one person can be controlling the planes and/or analyzing the data simultaneously. The system was designed to support an essentially unlimited number of aircraft although it has only been tested with two UAVs in the air and practically, the number is limited by the number of USB ports on the GCS computer.

1. Control

To control the UAVs, first the user must be able to define the mission. The user can define three types of markers:

- **Targets** mark locations that need to be visited and photographed.
- **Search zones** define a region that is covered by a search pattern. The UAV(s) will take a series of pictures that will cover this entire area.
- **No-fly zones** define areas that the UAV(s) must not fly over. If the plane is in an autonomous control mode (see below) the GCS will define flight paths to lead the UAVs around these no-fly zones.

All of these markers may be defined before or during the flight. This enables the user to alter the mission in progress. The user is also able to define the markers visually with the mouse on the real-time map, or the user may define them in a separate form using precise coordinates.

Once one or more UAVs are in the air, the user is able to choose from four control modes for each plane:

- **Fully autonomous** mode means full control of the UAV’s flight path is given to the GCS software. The GCS automatically allocates targets and assigns flight paths to each UAV in this mode so as to complete the mission in the shortest amount of time.
- **Semi autonomous** mode means that a UAV is assigned an ordered or unordered set of one or more targets and search zones by the operator, and the GCS automatically defines flight paths that guide the UAV around any no-fly zones and to search the required search areas. If the list is unordered, the targets is covered in the most efficient pattern.
- **Flight path** mode means that the user defines an actual flight path for the UAV to take. In this mode, the GCS will permit the user to guide a UAV through a no-fly zone, though an alert is displayed to make sure the user truly wishes to do this. This mode gives the user complete control over where the UAV flies.
- In **heading** mode, the user supplies a direction of flight and the UAV will simply fly along this heading. Once again, no-fly zones are disregarded.

Beyond the control modes, the user is able to fine-tune the UAVs' flight parameters. The user is able to adjust the minimum and maximum values for control signals (throttle, ailerons, rudder, and elevator), tweak the PID control loop constants to adjust flight performance, and define an arrival range for targets. The arrival range defines the maximum distance from a target that the UAV must reach before it is considered to have reached the target.

2. Data Management

In order for the system to be useful, the data it collects must be presented in a clear manner. To this end, several features are implemented to make data easily accessible to the user. First, the user is able to completely customize their real-time map. They may select custom color schemes, choose which objects are displayed and which are not, and adjust the viewing area of the map (translation and zoom). This enables them to view exactly what real-time data they want to see and to hide what they don't.

Second, the user is able to access pictures taken by the UAVs in two ways. On the real-time map, a small icon is displayed on the map at each location where a picture has been taken. The user may click this icon to view that picture. Alternatively, the user may call up a complete list of pictures by filename and select the picture they wish to view that way.

Third, a playback mode is provided. In playback mode, the users will see a real-time replay of any saved mission. The playback speed will also be adjustable, if the user wishes to view the playback slower or faster than real time.

Finally, a complete human-readable log file is generated of all system activities. Should the system fail completely, this file can be reviewed outside the system to analyze the error.

B. System Architecture

To achieve the described functionality, a client/server model is used. The client will provide the human-machine interface where the above features is made available. Any number of clients may connect to the server via a wireless or wired LAN. The clients will come in two varieties: control and recon. Control clients can alter the mission, change the control mode of the UAVs, and view all incoming data (pictures, video feeds, etc.). Recon clients, on the other hand, are optimized for viewing the incoming data. Before it can send any control commands, a client must authorize itself by providing login information; recon data is available to any client that connects.

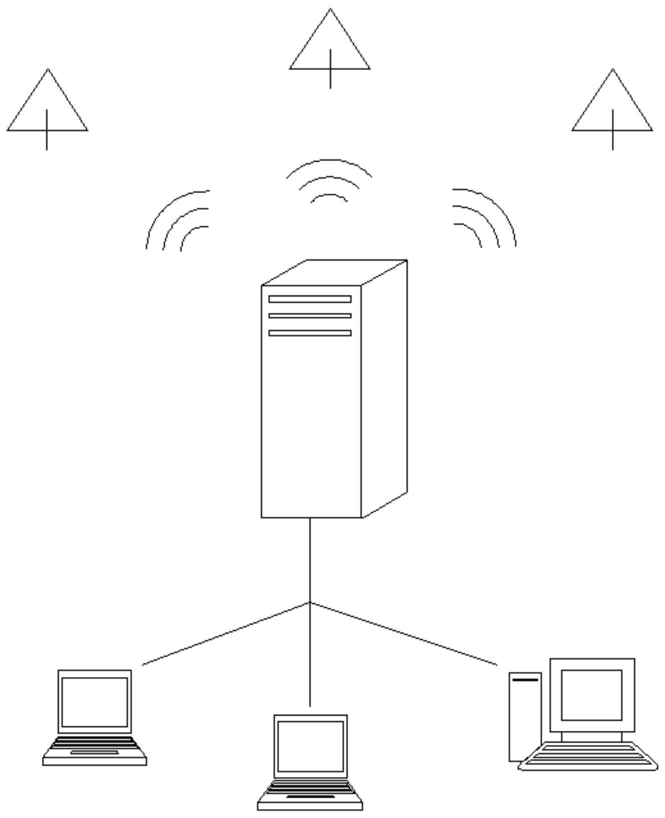


Figure 1: Diagram showing design of system where server acts as mediator between multiple operators and planes.

The server handles the actual communication with the UAVs and the synchronization of the clients. Radio modems using a custom-designed protocol called VACS (VCU Aerial Communications Standard) connect the server to the UAVs. Through this link the server will upload flight paths and download status information such as the UAVs' telemetry. The video feed is downloaded through a separate antenna, but it is also captured by server and streamed to all of the clients. The server will capture still images from the video stream whenever the plane flies over a target or a client requests a picture.

The server and clients use XML messaging to communicate over a star model LAN. When a control client changes a UAV or mission value (adds a target, changes a PID constant, changes control mode, etc.), the client only messages the server. This change does not appear on the GUI until the server echoes the change out to all the clients, keeping them synchronized.

II. Implementation

This section describes the details of how the system was built, including details of all four subsystems, the hardware that was used to implement the operator terminals and the communication channels, how the systems are interfaced, and a schedule of development. The schedule of development also includes a discussion of how the design has been tested continuously throughout the project.

A. Subsystem Breakdown

In order to break the design down into manageable parts that could be completed somewhat independently by individual team members, the whole system was broken down into several subsystems. At the top-most level, the system was broken into two main parts: The flight control system that actually manipulates the plane's controls to fly from waypoint to waypoint and the ground control system that allows a human operator on the ground to provide the plane with its waypoints, monitor its progress, and analyze the reconnaissance data gathered.

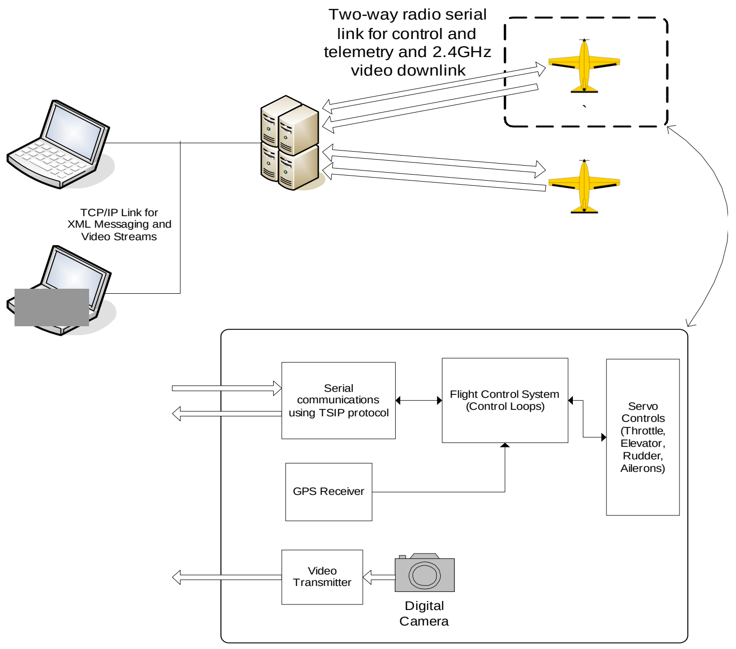

Two-way radio serial link for control and telemetry and 2.4GHz video downlink
TCP/IP Link for XML Messaging and Video Streams
Serial communications using TSIP protocol
Flight Control System (Control Loops)
Servo Controls (Throttle, Elevator, Rudder, Ailerons)
GPS Receiver
Video Transmitter
Digital Camera

Figure 2: Overview of Skyline UAV system.

The ground control system is broken down into the following major subsystems:

1. GCS client software
2. GCS server software
3. Autonomous flight path control software
4. Communications handling on-board the UAV

1. Client Software

All of the ground control station software is implemented in C# (a flavor of C++) using Microsoft's .Net architecture. The client side of the ground station is broken down by C# class in Figure 3. With the exception of *XmlNetworkComm*, *PlaneGraphic*, and the Auto Flight Path Engine, all classes represent forms. The *Client* form is the parent form for the client application. All other forms that make up the client are contained within it.

First, the user must log in to the server. This requirement is represented by the dashed lines labeled "locks" on the figure. The user chooses a name for their client (a human-recognizable name that displays in the server window) and then the IP address of the server. The *Choose Server* form maintains a list of all previously used server addresses from which the user may select a connection. Once connected, the client user is allowed to open the control and recon clients.

i. Control Features

The *Control Client* form is the main form in the client system. From here, the user has complete control over any UAV in the system and can view any data relayed from the UAVs. There are a number of sub-forms to support these two basic functionalities. On the control side, the most important requirement is that the user be able to define the mission. A mission is defined by:

- **Targets**: single locations the UAV needs to photograph
- **Search zones**: areas of land the plane must photograph
- **No-fly zones**: areas of land the UAV must not fly over

Any of these mission features may be added directly within the control client by drawing them on the real-time map with the mouse, or they may be added more precisely via forms that take GPS coordinates as inputs. Additionally, any of these features may be added to or removed from the mission at any time, allowing the user to redefine the mission even as the UAVs are completing it.

When the mission begins (defined as the time when the first plane is moved from manual to autonomous piloting), the Auto Flight Path Engine (discussed in detail on page 12) is called to set the UAV's flight path. After that time, this engine is called whenever the mission changes or another UAV goes up. Note that the Engine will only define flight paths for UAVs in the fully autonomous or semi autonomous modes (see the section on control modes below). However, it will still be used to determine if the user-specified flight path or heading will carry the UAV over a no-fly zone, in which case a warning is displayed, and the user will have to confirm the orders.

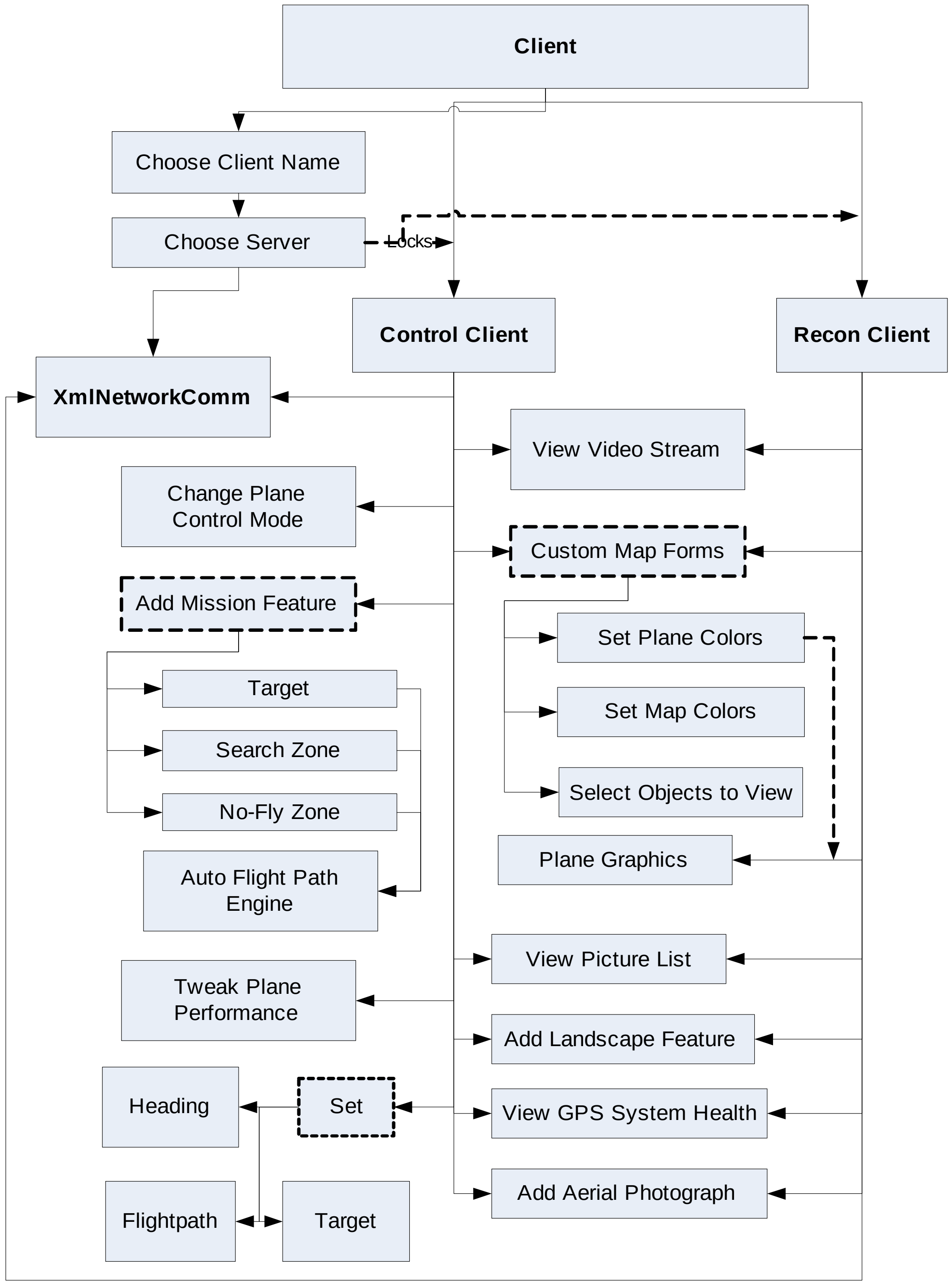


Figure 3: Client application class diagram

The next feature allows the user to change each UAV’s control mode. The four modes are:

- **Fully autonomous**: the system has complete control over all UAVs in this mode and will complete the mission the user has defined without further input from the user. It will optimize each UAV’s flight path while avoiding all no-fly zones, legally completing the mission in the shortest possible time. The user has full ability to redefine this mission at any time, however, and the system will immediately recalculate the necessary flight paths.

- **Semi Autonomous:** the user specifies an ordered list of targets for the selected UAV, and the system will send the UAV through that order immediately, guiding it around any no-fly zones
- **Flight path**: the selected UAV flies the defined flight path, ignoring no-fly zones. The system, however, will generate a warning, verifying that the user wishes to send the UAV over the no-fly zone before sending the UAV off along the flight path.
- **Heading**: the UAV flies along the set heading, once again ignoring no-fly zones

For each of the latter three modes, there is an associated Set form that allows the user to set the goal. When flying to a target, the user may create a new target or select a previously defined target (in this case, the user will have the option of removing this target from the list that the fully autonomous UAVs are covering). When following a flight path, the user defines the flight path by drawing it on the map with the mouse, or by entering coordinates into a form. Finally, when flying along a heading, the user may select a head between zero and three hundred sixty degrees. A small graphic will show the user the selected heading relative to the UAV's current position, so they can verify the direction.

The final control feature allows the user to tweak each UAV's flight performance. This feature is divided into two parts. First, the user is able to set the minimum and maximum positions for the plane control servos (the servos that control the throttle, ailerons, rudder, and elevator). This will allow the user to restrict the maximum turn radius or the maximum rate of climb, for example. Second, the user is able to tweak the PID control loop constants. This will allow the user to adjust the response of the UAVs (making them turn harder or softer for a given error in heading, for example).

ii. Reconnaissance Features

The other half of the control client enables the user to view all data gathered by the UAVs. This mirrors exactly the functionality of the recon client. The only difference is that the recon client is optimized for viewing data, since all the control features is removed. This optimization will occur mainly in the form of data access being moved out of menus and onto the recon client where it is more readily available.

There are three main types of data that is collected: pictures, streaming video, and plane status information. Pictures is automatically captured from the video stream by the system whenever a UAV reaches a target or passing over a search zone. Additionally, the user may have the system snap a photo at the UAV's current location by clicking a button. These pictures is available to the user both as clickable icons on the real-time map (displayed at the point where they were taken) and as a file list within a sub-form. The streaming videos from each UAV can be opened in a sub-form, with an "always on top" option available. Plane status information includes telemetry data, control servo information, and GPS health information. The current design calls for the telemetry and control servo information to be available on the main form and for the GPS health data to be available on a sub-form, though this design may be subject to change.

Given all this data, it is anticipated that the user could be easily overwhelmed, so display customization forms is implemented. First, the real-time map is highly customizable, allowing the user to pan and zoom the map view, change the color of the map items, or even hide items all together. Second, the user is able to set the color for each plane and the mission objectives assigned to it (which target it is flying to next, for example). Last, the streaming video and still pictures, as mentioned above, is available in sub-forms so they can be hidden when not necessary.

For additional ease-of-use, there is two additional map features: an aerial photograph and a landscape marker. Via a sub-form, the user may specify an aerial photograph of the mission area, along with GPS coordinates of two diagonal corners, and this aerial photograph will then be drawn behind all the mission features on the real-time map. This will give the user perspective on where the UAVs relative to landmarks. If an aerial photograph is not available or not up-to-date, the user may use the landscape feature to mark map locations. This tool is used in a manner identical to the no-fly and search zone tools, but it does not affect the mission. Thus it can be used to mark important locations on the map.

iii. Supporting Classes

The *Plane Graphics* class saves information necessary to draw each UAV on the map. This information includes such items as the color of that plane, the last ten locations of the UAV, and the UAV's current flight path. This class is used by both the control and the recon client to save this information. Additionally, the class that sets the UAV colors must associate with this class to update the UAV's color.

Finally, the *XmlNetworkComm* class is used to connect to the server. All three main forms (client, control client, and recon client) use this class. The connection is first established by the client. Once established, this connection is passed to each control and recon client that is opened within the client application, allowing them to talk directly to the server. Further discussion of this class is available in the interface section.

2. Server Software

The server application has the responsibility of providing a communications hub between all of the user clients and all of the planes in the air. The software is organized into three main functions: communication with the clients (via an *XMLNetworkCommServer* object), communication with the planes (via *UAVComm* objects), and a main class (*MainForm*) that ties the two ends of the server together and provides a user interface for configuration and feedback on the system status.

In addition to translating and forwarding messages between multiple clients and multiple planes, the server maintains mission data among all of the clients. Target lists, flight zones, and plane status are all stored centrally on the server so that all clients are working with the same mission data.

When the server is launched, it automatically begins listening for TCP connections from a client on port 5510. At this point clients can connect and set up mission data, but in order to do anything useful one or more planes must be configured. To set up a new plane, once the radio modem for that plane has been connected to one of the serial ports, the user must select the appropriate serial port and assign a plane ID number to it. At that point, a *UAVComm* object is created for the plane and isgin accepting TSIP packets from the serial port and tracking the plane's telemetry. The plane is now also available for control commands (i.e. upload new flight path) from a client or from the auto flight path engine.

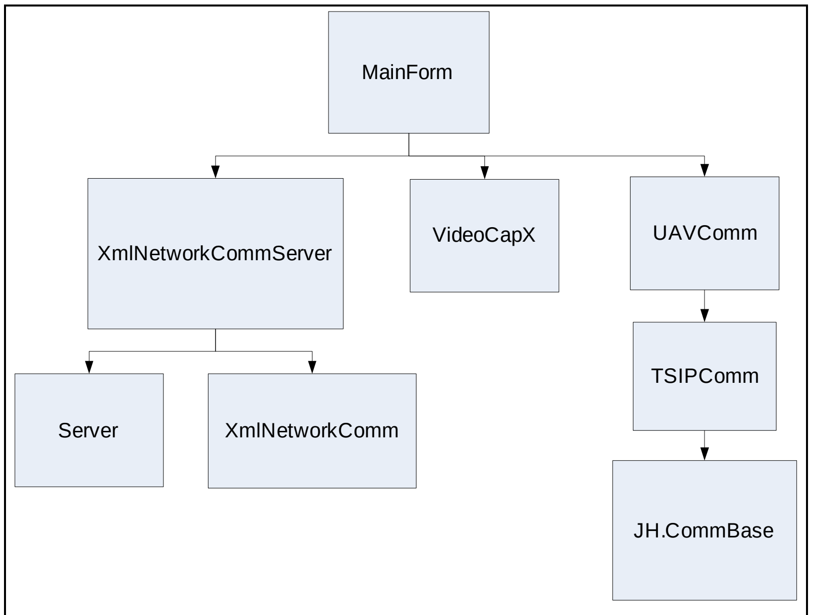


Figure 4: Class diagram for server application.

The *UAVComm* class represents a plane in the system. It handles everything from the low-level serial port interface to maintaining information on the plane it represents and making it available to the MainForm object. A set of events are provided by *UAVComm* and are signal when certain events happen, such as reception of new telemetry data, or when it receives acknowledgement from the plane of an update flight path. The *MainForm* object may then take whatever action is required to respond to the event.

Low level serial port interactions are handled by the *CommBase* class, created by John Hind[1]. The *VACSComm* class inherits from *CommBase* and parses the incoming data bytes into VACS packets which are then represented by *VACSPacket* objects. Finally, the *UAVComm* object is inherited form *TSIPComm* and actually parses these *VACSPacket*'s to extract the data contained in them.

More details of the VACS protocol used for ground to plane serial communications can be found in the interface section below.

Incoming XML messages from the clients are delivered via the *MessageReceivedEvent* of the *XmlNetworkComm* class to the *MainForm* object where they are handled.

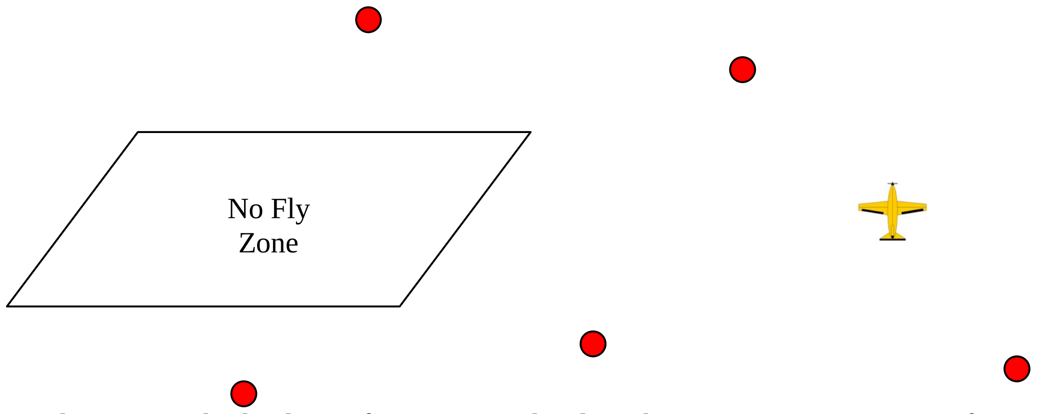
Another feature that must be implemented by the server is video capture. Each plane is transmitting a video signal that must be captured digitally via the Dazzle DVC 80. Also, for this video signal to be used by multiple clients, it must be made available over the network. We have purchased a license for an Active X component called VideoCapX[2] that will provide us with both video capture and video streaming functionality. Once one or more planes are configured, the server GUI will display a list active planes and available video capture devices. The user will then be able to link a particular plane to a capture device. Once a video signal is associated with a plane, the *VideoCapX* plugin is used to set up a video streaming server on the appropriate port.

[1] Article and software available at: http://msdn.microsoft.com/msdnmag/issues/02/10/netserialcomm/
[2] Made by Fath Software: http://www.fathsoft.com/videocapx.html

The port will either be given to the clients in an XML message, or a standard port assignment scheme based on the plane's ID number is used. For example, the video port could be the XML port (5510) plus the plane ID, so that video for plane number one would be available on port 5511.

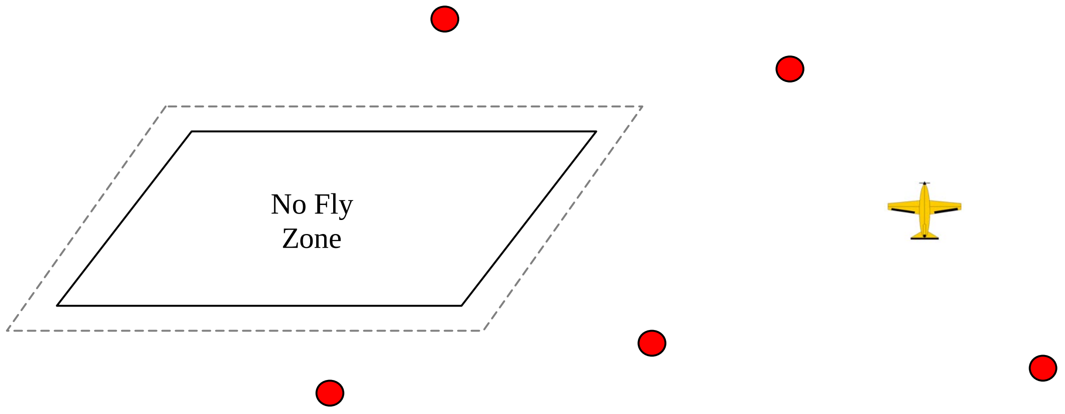
3. Autonomous Flight Path Engine

The autonomous flight path engine is designed to take a set of user-entered targets and calculate the shortest flight path to cover all of those waypoints. It is capable of dividing these

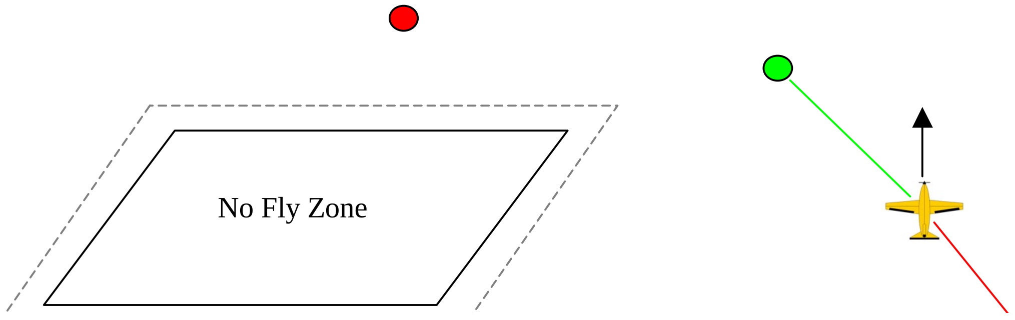


waypoints between multiple planes if necessary. The algorithm outputs its own series of waypoints, constituting the flight path, back to the plane(s) via the control clients and server.

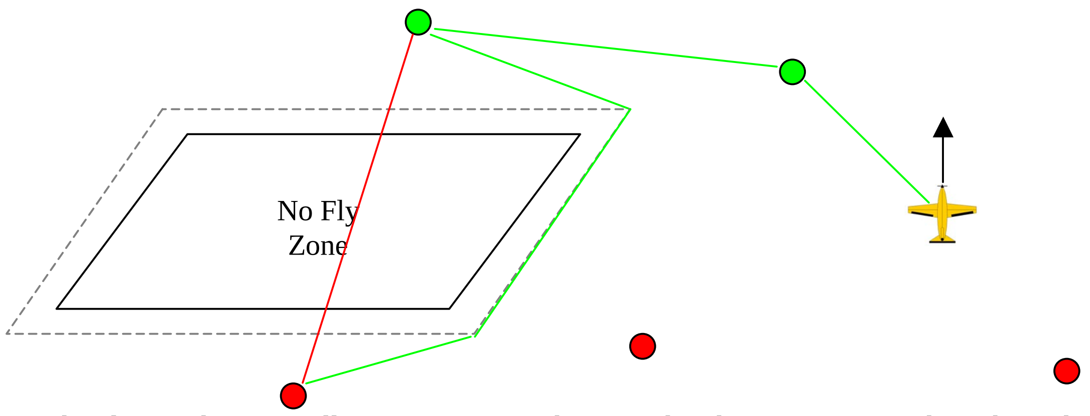
The first step of the algorithm is to offset the no-fly zones to give any plane flying near one a safe buffer distance.



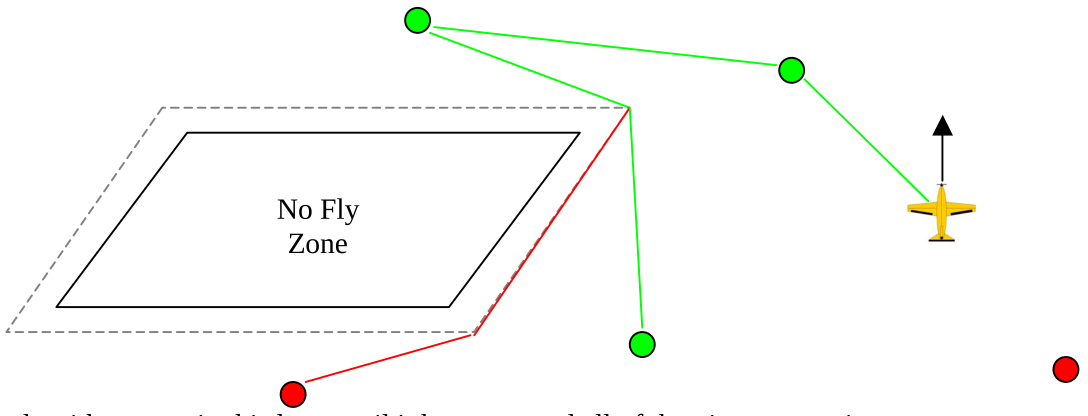
The next stage checks to see if the waypoints passed to it are already ordered or need to be ordered. If ordering of the waypoints is necessary, the engine takes a weighted average of the distance to the next waypoint and the bank angle required to reach that waypoint to choose the next waypoint.



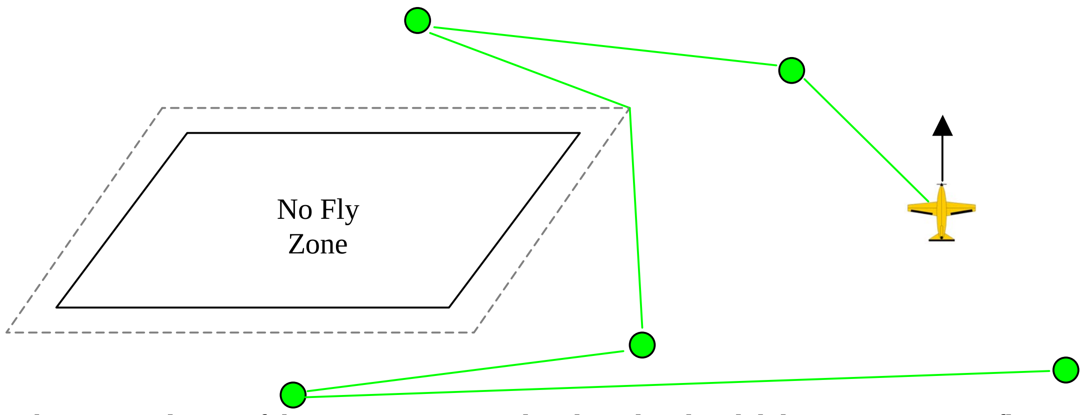
It also checks to see if the path to that waypoint requires crossing a no fly zone and if so, recalculates the shortest path around the no fly zone.



During this detour, the target allocation engine makes sure that there is no point where the path would be shorter to go to a different waypoint, and if it finds a shorter path, the course is altered to this path.



The algorithm stays in this loop until it has traversed all of the given waypoints.



Once this is complete or if the waypoints were already ordered and did not cross any no-fly zones, the target allocation engine makes adjustments to the flight path in order to take the turn radius of the plane into account. By doing so, the flight path has been adjusted using the planes' actual physical capabilities and leads to a smoother flight and less stress on the PID algorithms.

4. UAV Serial Communications

The communications software running on the FPSLIC on-board the UAV is largely based on last year's design; with a few significant changes. The biggest change is that serial port transmission and reception, for both the radio link to the ground and the link to the GPS, has been improved to be completely interrupt-driven. We believe that this will allow for more efficient use of processor time, and will fix problems that would often cause the UAV software to hang and stop transmitting telemetry packets.

The overall design of the UAV software has also be made more modular in order to facilitate the integration of the flight control software being developed by another team, and the communications software we are developing. We have adjusted the interface so that we can update the waypoints during a mission, control flight parameters, etc., using our new communication protocol.

B. Hardware

The communications and control system for the ground control system consists of standard computers and networking hardware for the ground control station, radio communications equipment for the transmission of telemetry, control commands, and live video feeds, and finally the Atmel FPSLIC development board that controls the plane itself. Details of these hardware systems are discussed in the following sections.

1. Ground Control Station

The hardware for the ground control station is, essentially, one or more networked PCs running the GCS software, and a radio modem and video receiver for each plane. One computer must run the server software, although the same computer may also run the client software, so the system can be used with only one PC. The server computer handles all communications with the planes, and must have one radio modem and connected for each plane.

Because the radio modem connects to a 9-pin RS232 serial port, and most modern computers have zero to one of these ports, a USB to RS232 adapter is used to provide serial connections for more than one modem. Video capture is done via a Pinnacle DVC 80 A/V capture device; again, one is required for each plane for which video is to be captured. The captured video is recorded into MPEG video format for later review, and a live feed will also be broadcast so that all of the clients can simultaneously view the video as it comes in.

Additionally, the video signal are split and displayed on stand-alone monitors. This is to ensure that the video will always be viewable, despite any problems with the capturing software or hardware. We believe it will also provide a useful debugging tool in case of video reception problems during missions.

In order to allow for multiple mission operators, multiple computers must be utilized and connected via a wired or wireless TCP/IP network. The network provides the mechanism for the two types of communications required between the client and server software: an XML based message passing system, and the streaming video.

Both the client and server are written in C# using the Microsoft .NET architecture. The C# programming language and extensive libraries available with .NET allows us to develop the

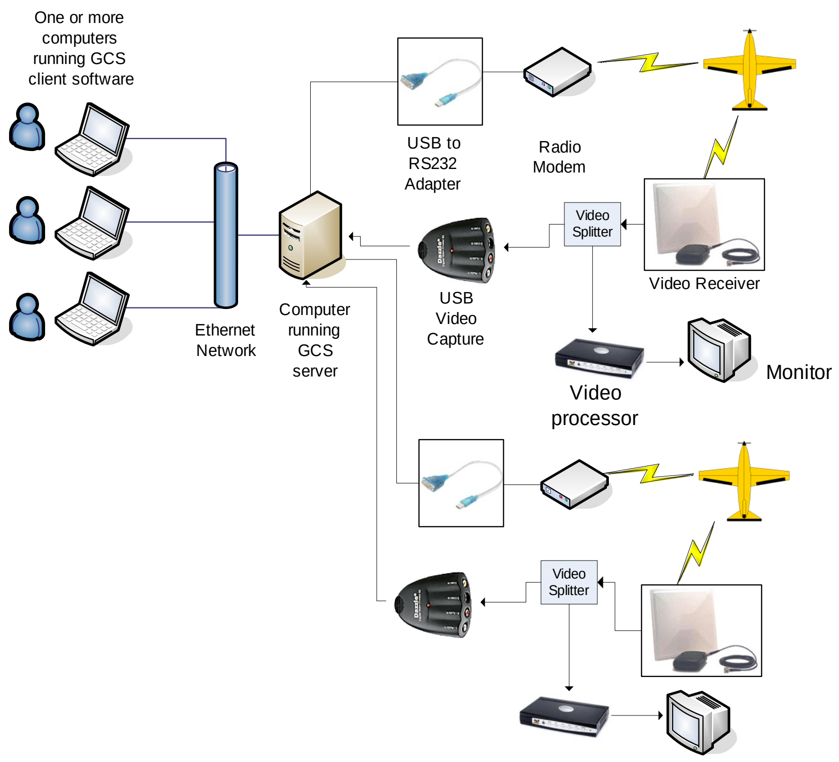


Figure 5: Connection diagram showing each device used to connect the planes in the air to the PC ground station.

complex features that is required by our application and still provide quick and easy GUI development.

2. Communications Equipment

For communications between the GCS and the plane, a MaxStream 9XStream 900MHz radio modem is used to provide a bi-directional 9600 baud serial connection. This connection is used to send commands to the airplane, and to download telemetry data from the airplane. In order to provide enough serial ports on the laptop computer running the server software, a USB to serial converter[3] is used to provide additional serial ports.

For video, the same wireless video system from www.wirelessvideocamera.com that was used last year is being used with a new camera. The system, part number AAR05, includes a directional antenna and provides a range of 5 miles.

3. UAV

The main controller on-board the UAV is an off-the-shelf Field Programmable Gate Array (FPGA)-based computing board called the Suzaku. The Suzaku contains a Xilinx FPGA which, in turn, includes a 32-bit RISC microprocessor called a Microblaze. The Microblaze runs a

[3]Converter made by Startech.com, and Mfg. part number is ICUSB232. Purchased from CDWG (http://www.cdwg.com/shop/products/default.aspx?EDC=320583).

version of the Linux operating system and the flight control application executes on top of the Linux OS. More details on the flight control system is contained in the FCS report.

C. Subsystem Interfaces

This section describes the interfaces between the four main subsystems of the Skyline GCS system: Client to Server, Server to UAV, and Auto Flight Path Engine to Client/Server.

1. Client/Server Interface

The client and server communicate via an XML messaging system designed specifically for this project. This system is implemented in the *XMLNetworkCommClient* class and the *XMLNetworkCommServer* class, both of which inherit general features, such as message type definitions, from their shared parent class *XMLNetworkComm*. The *XMLNetworkCommServer* makes use of a slightly modified version of a generic C# TCIP/IP server module written by Asad Aziz[4]. The TCP server software is implemented in the *Server* class. The TCP socket handling for the client is considerably simpler than the server, was written specifically for the *XMLNetworkCommClient* class, and is contained in the *Client* class.

A set of predefined message objects, serialized into XML messages, is passed between the client and the server via the TCP socket. A sample of a message class definition and the resulting XML message that is sent over the connection when the object is serialized and transmitted is shown in Figure 6.

Telemetry	Transmits telemetry information to clients
GPSHealth	Contains GPS Health info for a plane
Targets	From Server: Updated Target List From Client: Add or remove target(s)
RequestTargets	Used by client to request the current target list
FlightPath	From Server: Updated FlightPath From Client: Change FlightPath
RequestFlightPath	Request current flightpath for a plane
FlightZone	From Server: Current FZ List From Client: Add/Remove FZ
RequestFlightZone	Request current FZ list from server
PicTaken	Sent when new picture is captured
RequestPic	Request a picture be captured
Hello	Sent by server upon connection. Assigns Client ID.
Identify	Should be sent by client in response to Hello. Gives identifying string for the client connecting

[4] Article and code available at: http://www.csharphelp.com/archives3/archive486.html

NewPlaneActive	Notifies clients when a new plane becomes available
ControlMode	How the plane is being controlled (autonomous, semi-autonomous, etc)
PIDParam	Server: Sends current PID Parameters Client: Update PID Parameters
PIDParamRequest	Client requests PIDParam message for a plane
PIDParamUpdate	Causes server to update its PID Parameters by downloading them from the plane

Table 1.

In addition to the XML messages, the server application makes available streaming video for each plane to all of the clients. The video is implemented on both the client and server ends by the VideoCapX ActiveX plugin from Fath Software, Inc.

Telemetry	Transmits telemetry information to clients
GPSHealth	Contains GPS Health info for a plane
Targets	From Server: Updated Target List From Client: Add or remove target(s)
RequestTargets	Used by client to request the current target list
FlightPath	From Server: Updated FlightPath From Client: Change FlightPath
RequestFlightPath	Request current flightpath for a plane
FlightZone	From Server: Current FZ List From Client: Add/Remove FZ
RequestFlightZone	Request current FZ list from server
PicTaken	Sent when new picture is captured
RequestPic	Request a picture be captured
Hello	Sent by server upon connection. Assigns Client ID.
Identify	Should be sent by client in response to Hello. Gives identifying string for the client connecting
NewPlaneActive	Notifies clients when a new plane becomes available
ControlMode	How the plane is being controlled (autonomous, semi-autonomous, etc)
PIDParam	Server: Sends current PID Parameters Client: Update PID Parameters
PIDParamRequest	Client requests PIDParam message for a plane
PIDParamUpdate	Causes server to update its PID Parameters by downloading them from the plane

Table 1: List of currently proposed XML messages.

```
public class HelloMsg : Message
{
      public HelloMsg()
      {
            messageType = MessageType.HelloMsg;
      }
      public HelloMsg(string wdpn, int cid, int numop)
      {
            messageType = MessageType.HelloMsg;
            WorkingDirPathName = wdpn;
            clientID = cid;
            numOfPlanes = numop;
      }     public string WorkingDirPathName;

      public int clientID;
      public int numOfPlanes;
}

<?xml version="1.0" encoding="utf-16"?>
<Message xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="HelloMsg">
  <srcID>0</srcID>
  <messageType>HelloMsg</messageType>
  <WorkingDirPathName>/</WorkingDirPathName>
  <clientID>1</clientID>
  <numOfPlanes>0</numOfPlanes>
</Message>
```

Figure 6: Example of XML Message class and resulting XML after serialization.

2. Server/UAV Interface

As mentioned previously, the planes communicate with the ground station server software via a 9600 baud radio serial link. The protocol used over the serial link is called the VCU Aerial Communications Standard (VACS).

The lowest level of the VACS communications standard is the serial packet format. Each VACS packet includes two sync bytes, a set of header fields, a variable length data payload, and a two-byte checksum.

Byte	Field
0	Sync1 Byte (0x76, ‘v’)
1	Sync2 Byte (0x63, ‘c’)
2	Destination Address
3	Source Address (address of the sending host)
4	Message ID (Low Byte) (Defines type of message)
5	Message ID (High Byte)
6	Length N (low byte) (Gives length, in bytes, of the data payload)
7	Length N (high byte)
…	Data payload (N bytes of data)

N+8	Checksum A
N+9	Checksum B

Table 1: VACS serial packet format

Checksum Calculation

The checksum is calculated using a 16-bit Fletcher algorithm, similar to the TCP protocol error detection scheme. The checksum calculation includes the data bytes as well as the destination address, source address, message ID, and length. It does not include the Sync bytes. The two bytes of the checksum, ChkA and ChkB, and calculated as follows:

```
ChkA = 0;
ChkB = 0;
// Array bytes contains the data to which the
// check sum is being applied
for(int i=0; i<num_bytes; i++)
{
      ChkA = ChkA + bytes[i];
      ChkB = ChkB + ChkA;
}
```

It is important that this operation is done using 8-bit operations. If the operation is being implemented with larger numbers, the resulting checksum bytes should be logically AND'ed with 0xFF to clear all but the lower 8 bits.

Addresses

The current VACS protocol defines only an 8-bit address space, giving a total of 255 possible addresses. An implementation is free to assign addresses as desired, however the address 255 is recommended to be reserved for a broadcast address and address 0 to be reserved for a Ground Control Station, in such situations where there is a single controlling GCS.

Reserved Addresses

GCS – 0x0
Broadcast – 0xFF

Message Types

VACS defines the serial packet structure used to communicate, however the actual data that is communicated over the link depends on the implementation of the vehicle and ground station using it. A message vocabulary is defined as a set of messages (each with a unique message ID) with defined data payload formats. Each message type can be either a **report** type or a **command** type.

Reports

Report messages are sent by the UAV to give information about its status; they can be sent periodically or only when requested. A report message can be requested from a UAV by sending an empty (no data payload) message with the message ID of the report desired, addressed to the UAV.

Commands

Commands are sent by the ground station to a specific UAV or broadcasted to all UAVs on the channel (by addressing the message to the broadcast address). When a command is received, the receiving UAV should send back an acknowledgement. The acknowledgement message is an empty message with the message ID of the command being acknowledged.

Inter-vehicle Communications

The VACS system should be built to support communication among vehicles in the air without the involvement of a GCS. This can be used in the test/implementation of cooperative mission algorithms, and may also prove useful for long-range communications using multi-hop networking. This issue needs to be looked at, and may require redesigning the command/report method. Or, it may not. It is possible for a plane to send a command addressed to another plane, or to request a report from another plane.

Special Messages

All vehicles using the VACS protocol should implement certain messages. For now, one message is required to identify what type of plane is transmitting. More special messages may be defined in the future.

Identify Message (0x00)

If a single ground station is going to support different vehicles (or two similar vehicles with different software versions), with potentially different message vocabularies, there must be a common way to identify what type of vehicle a message comes from. The VACS system provides two values to identify a vehicle and the types of messages it supports: The **vehicle type** and a **version**. Vehicle type is a 16-bit integer, version is an 8-bit integer (Vehicle type could potentially be represented as a string, and this is being considered). All vehicles should implement a common “identify” message, with message ID 0. This message should be broadcast when the vehicle is turned on, and sent in response to a request.

Byte	Description
0	Vehicle type high byte
1	Vehicle type low byte
2	Version
…	Optional: Additional data bytes specific to an implementation

Table 2: Data payload format for identify message (ID: 0x0)

Ground Station Implementation

The communications system discussed here is intended for a system with one GCS controlling multiple UAVs. Some redesigning of the basic communication and data storage system in the

GCS is required in order to support multiple UAVs of different types and to allow quick run-time configuration of the message protocols used by the planes. Two primary new features are planned for the GCS: The use of a “property tree” for storing all the system state information, and the definition of a generic interface for plane communications.

Property Tree

The property tree is a concept borrowed from the open source flight simulator FlightGear. A property tree is a collection of nodes arranged in tree form, with each node storing a value of one of several types, such as double, int, or string. A property location is represented by a string such as “/uav[0]/gps/longitude”, where longitude is a child node of gps, which is a child node of uav[0]. The property tree allows multiple nodes of the same name to be referenced by index, as done with the node named ‘uav’. Nodes do not have to be pre-defined, one code module can add a new node and it will become available immediately to others, however in practice some nodes (such as for longitude, latitude) will be standard for all planes.

The property tree (or branches of the tree) can be easily represented as XML documents for communication or storage. Logging of a mission can be achieved by writing out the entire tree, selected properties, or perhaps changes in properties to a file. Additionally, logging can be easily configured by a file that specifies what properties are to be logged and how frequently. As an example, suppose a new sensor is added to a plane: once the sensor value is assigned to a property (details on how this assignment can be configured will be presented further on) it can automatically be logged to a file or printed to a windows form just by referencing it’s property value; no GCS re-compile is necessary.

Last two bytes of data are checksum. This 16-bit number is the sum of type and data bytes.

Hex Code	ASCII	Length	Description
			GCS to UAV
0x47	G	2	Request GPS Health status from plane
0x41	A	2	Request PID Parameters
0x50	P	TBD	Upload new PID Parameters including radius of target boundary
0x55	U	2	Set camera mode to Auto
0x4D	M	2	Set camera mode to Manual
0x4C	L	>=3	Upload new flight path
0x4F	O	10	Upload new origin
0x44	D	2	Request Current Origin
0x46	F	2	Request Current Flight Path ('f' packet)
			UAV to GCS
0x67	g	>= 1	GPS Health packet: Contains number of locked on satellites and error code
0x73	s	25	Telemetry Information
0x61	a	TBD	PID parameters
0x70	p	2	Acknowledge PID parameters (sent when P packet with matching checksum is received)
0x75	u	2	Acknowledge Camera mode set to Auto
0x6D	m	2	Acknowledge camera mode set to Manual
0x66	f	>3	Current flight path
0x6C	l	2	Acknowledge new flight path
0x6F	o	2	Acknowledge new origin
0x64	d	10	Current Origin

Table 2: Packet codes for UAV packets. First byte of the data packet contains the code indicating

3. Auto Flight Path Engine Interface

The Auto Flight Path Engine is a class that is used by both the client application and the server application. It is interfaced via a set of methods.

The Auto Flight Path Engine interfaces with both the server and the client depending on the control mode of the plane. When the plane is in fully autonomous mode, the server calls the Auto Flight Path Engine to create a flight path for each of the active, autonomous planes. The Auto Flight Path Engine is given each plane's telemetry, a set of targets and search zones, and any no fly zones. It then calculates and returns the shortest flight path for each plane that together completes all objectives in the most time efficient way, and these flight paths are returned to the server to be passed on to the planes.

When the plane is in semi-autonomous mode, the client calls the Auto Flight Path Engine. This time the client passes the targets, search areas, plane telemetry, and no fly zones. The Auto Flight Path Engine then calculates a flight path to complete the user-inputted objectives in the sequence given, without flying through any no-fly zones, and returns this flight path to the calling client who passes it on to the plane via the server.

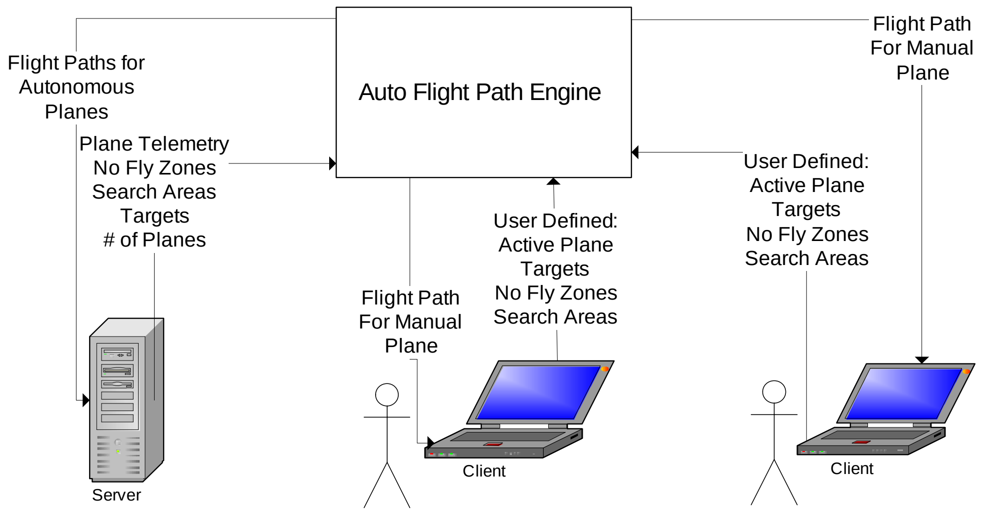


Figure 7: Input/Output diagram for Auto Flight Path Engine.

VCU-UAV Skyline GCS
User's Guide

I. Ground Station Overview

The ground station is based on a client-server architecture. A server application communicates with the plane(s), and acts as a gateway to one or more client applications which provide the primary user interface. The server must run on the computer that is connected (via serial port) to the radio modem, however the clients can run on the same computer or different computers on the same network.

Starting the System

The server application must be started first. The mission and plane must be set-up on the server, and then any number of clients can be connected to view/control the uav(s). The following sections describe the operation of the server and clients separately.

II. Using the Server

Data Directory

After launching the server application, the first dialog you will see is prompting you to choose a data directory. This is where any captured data is stored, such as telemetry logs, image captures, and video. Use the "Change" button to navigate to the directory you would like to use, and then click "OK".

Mission Name

The next dialog asks you to specify the mission name. The mission name is used to identify specific flights. A default mission name is generated for you based on the date and time, however additional information may be helpful if you intend to review the data later. A directory with the same name as the mission name will be created in the data directory you already specified, and this is where saved data for this flight will be saved.

Mission Location

Before you can add any planes to the system, you must set the mission location. The FCS system uses a flat-earth approximated Cartesian coordinate system for position. Longitude and Latitude are converted to two 16-bit numbers for x and y position, relative to a location specific origin. This origin is specified by setting the mission location, and is then uploaded to the planes to be used for FCS calculations.

Because the x and y position have a range of 0-65535 and are in units of meters, the maximum operating area is around 65 sq. km. The origin should be set to a point south-west of the operating area. Ideally, the origin should be around 32km south, and 32 km west. The following formula can be used to find that location:

$$OriginLat = CurrentLat - 32000 / LAT\_MPERDEG$$
$$OriginLon = CurrentLon - 32000 / LON\_MPERDEG$$

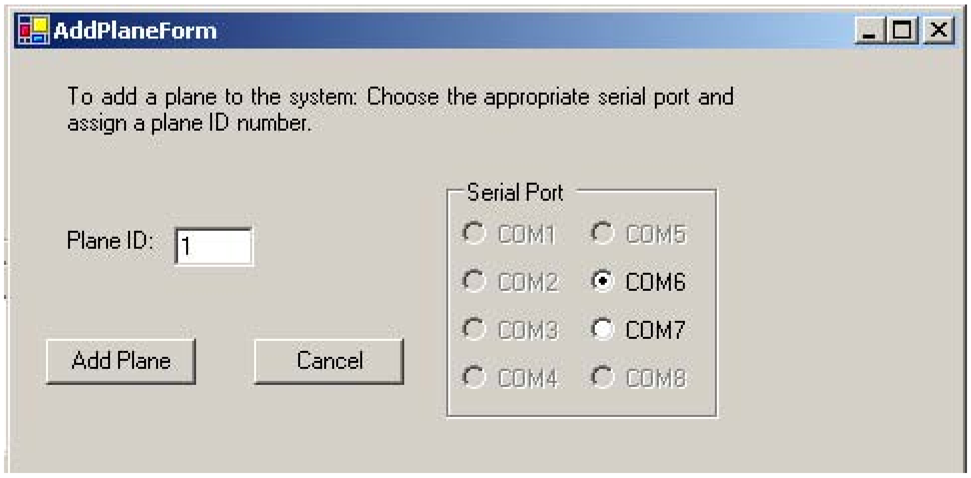
Where LAT_MPERDEG (meters per degree of latitude) is always 110898, and LON_MPERDEG (meters per degree longitude) varies depended on latitude, but can be calculated by:
$LON\_MPERDEG = 110898 * \cos(latitude)$

To set the mission origin:
1. Click “Configuration -> Set Mission Location”
2. Create a new mission location or select a pre-existing one

Alternatively, the list of locations can be modified by editing the XML file locations.xml located in the GCS application directory.

Adding a plane

The next step is to add a plane and connect it to a com port.


AddPlaneForm

To add a plane to the system: Choose the appropriate serial port and assign a plane ID number.

Plane ID: 1

Serial Port: COM6 selected

1. Select “Planes->Add a Plane” from the main menu
2. Select an ID number for the plane (the default of 1 is should always be acceptable)
3. Select the COM port the plane’s modem is connected to. If a com port is grayed out, it means that the port either does not exist or that it is already opened by another program.
4. Click “Add Plane”

The plane should now be displayed in the Active planes list on the server, along with a running count of all packets received from and transmitted to the plane.

Monitoring clients

Information on currently connected clients is displayed in the “Current Connections” panel

Assigning video capture

If a video capture device is connected to the server computer, it can be associated with a plane. To do so,

1. Right click on the plane in the active planes list
2. Select “Video Capture ->”
3. Select the capture device you want to associate with the plane.

Now, video streams will be available to the payload client, still image captures can be triggered from the payload client, and when you start recording flight data, the video stream will be saved as well.

Mission Recording

To start recording mission data, press the start recording button on the server. All telemetry data will be saved in an XML format to a file in the mission data directory. This data can then be extracted to a tab-delimited file using the *logextractor* program.

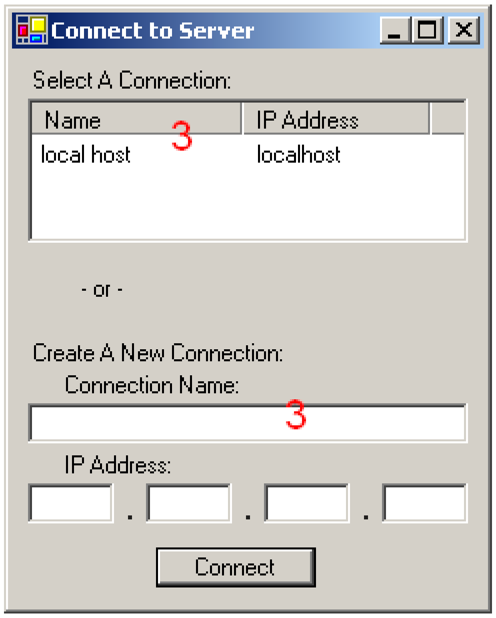
III. Using the Control/Payload Client

The control client provides two different functionalities in one application: control of the plane and a special purpose client for viewing video captured from the plane. The payload client is not necessary for flight control, however.

Connecting to Server

Launch the gcsClient application.

1. Select "File->connect" from the main menu.
2. Enter a client name or select OK to use the default. The



Name	IP Address
local host	localhost

3. Select a server IP address from the saved list or enter a new address
4. Click "Connect"
    1. If you connect successfully, you may now use the control and payload clients
    2. If you do not, verify the server address and repeat 1-3

Using the Control Client

To start the control version of the client, select "File->Control" from the main menu. You will be presented with the main control client display, including moving map.

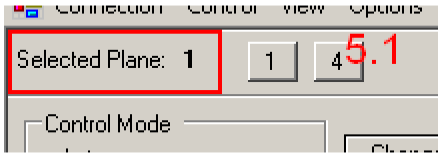
Adding a background image

In order to greatly increase the usefulness of the moving map display, an aerial image can be displayed as background rather than the default green background. To do this an aerial image with known coordinates is required. The image must be oriented so that up is north. The image location can be specified as either the longitude and latitude of the north-west and south-east corners, or as the north-west corner and a width and height of the image expressed as degrees of longitude (width) and latitude (height). To add a background image:

1. Select “Map->Background Image”
2. Select an already saved background image and press “OK”, or
3. Select an image by clicking the “Choose Image” button
4. Specify the image location coordinates
5. Give the picture a name to save it as
6. Press “OK”

Alternatively, the saved images can be modified manually by editing the XML file “BackgroundImageFile.uav” in the GCS application directory.

Selected Plane



A button is displayed for each active plane along the top of the client display (just below the main menu). To select a plane, simply click the button labeled with its ID number. Once selected, all data displayed will be relevant to the selected plane, and all data input (such as flighpath, PID parameters, etc.) will be applied to the selected plane.

Map Display

The largest component of the control client is the moving map display. This shows the location of any planes in the system and their flight paths, as well as any targets or zones added by the user.

There are several ways to navigate the view on the map. The map position can be moved by “dragging” with the right mouse button. Alternatively, there are four buttons (labeled with arrows) on each side of the map that can be used for scrolling the map location.

Zooming can be performed by clicking the zoom-in/zoom-out buttons on the toolbar in the top-right corner of the map, or by using the scroll-wheel on a suitably equipped mouse (You may have to make the map control “active” by clicking on it before it will receive scroll-wheel inputs).

The map can be centered on the selected plane by clicking the center on plane button

The map can be center on the current background image by clicking the center on image button.

Setting FCS PID parameters

PID parameters must be uploaded to the plane before engaging the autopilot. PID parameters can be set from the control client by selecting “Flight->Set PID Parameters” from the main menu. Be sure that the plane you want to control is selected first.

Load from File

To load previously saved PID parameters from a file:

1. Click “Browse”, and select the saved file you want to load
2. Click “Load parameters”.
3. The saved parameters should be displayed in the parameter text boxes.

Save to File

To save current parameters to a file for later use, simply click “save parameters as”, and enter the filename you would like to save.

Upload Parameters to Plane

To transmit the parameters to the plane, you can click the OK button, or the Apply button. The only difference being that OK will close the form, and Apply will leave it open.

Requesting Parameters from Plane

The plane’s current parameters can be requested by clicking the “Refresh” button. This causes a PID request message to be sent to the plane, and when the response is received, the PID Parameter form will be updated with the values.

Assigning Flightpath

Perhaps the most important functionality of the control client is assigning a flightpath for the FCS to fly. This is done graphically using the map-display.
To assign a flightpath to a plane:

1. Make sure that the desired plane is selected
2. Click the center on plane button, if necessary, to put the plane on the map
3. Select flight path mode by clicking the flight path button on the map toolbar
4. Draw the flight path on the map by single clicking for each waypoint, and double-clicking the last waypoint to complete the flight path.

Once entered by double clicking, the new flight path will not be displayed until acknowledgement of its reception is received from the plane.

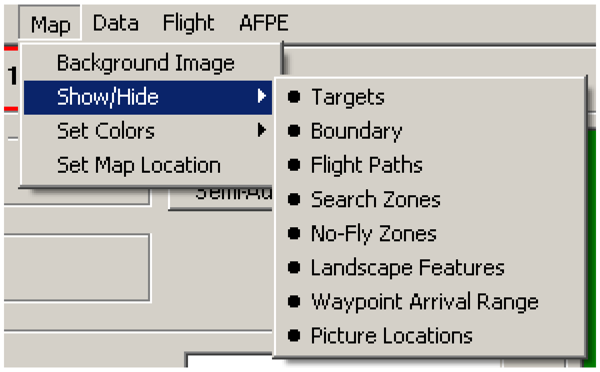
GCS Location

The ground station location can be set on the map using the button on the map toolbar. Simply click the button to select the tool, and then click on the GCS location on the map display. One the GCS location is set, the distance from plane to GCS location is display in the bottom right corner of the map.

Map Options

Several other options can be adjusted for the map, such as what to display, and different colors to use for display.

1. The "Map->Show/Hide" menu allows certain things to be included or excluded from the map. \

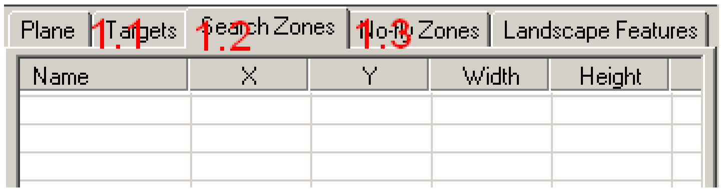


2. Similarly, the "Map->Set Colors" menu allows the colors used for most items to be customized.

3. "Options -> Set Map Location" allows user to change the area displayed in the map. Simply enter the mission coordinates of the desired upper-left corner of display area and click "OK"

Mission Definition Objects

Although the flight path uploaded to the plane is the only thing that controls the vehicles flight, other items can be added to the map for reference. These include Targets, Search Zones, and No-Fly zones.

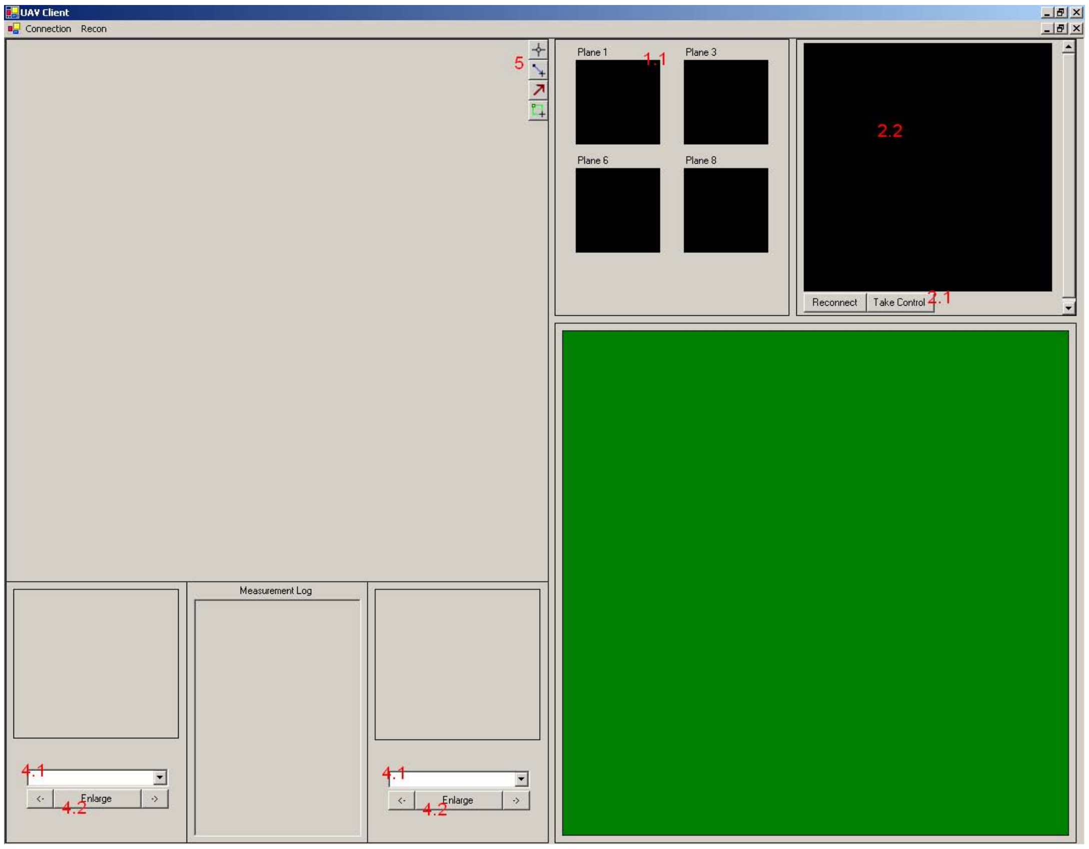


Name	X	Y	Width	Height

1. Targets
   1. Use the ⦿ tool to add targets directly to the map
   2. Use the add button on the Targets tab page to add targets via coordinates
2. Search Zones
   1. Use the ⁙ tool to add search zones directly to the map
   2. Use the add button on the Search Zones tab page to add search zones via coordinates
3. No-fly Zones
   1. Use the ▩ tool to add no-fly zones directly to the map

    2. Use the add button on the No-fly Zones tab page to add no-fly zones via coordinates
4. Deleting
    1. To delete mission definition object(s), select the object(s) to be deleted in the list and press the delete key on the keyboard

IV. Using the Payload Client



1. Selecting a video source
   1. Double-click on the thumbnail video feed to view it in the large feed window

2. Controlling camera viewing angle
   1. Click “Take Control” to take over control of the camera positioning
      1. You will be asked to authenticate with a password
   2. Move the mouse around the window to move the camera
   3. Right click to center the camera (“center” is normal to the body of the plane)

3. Capturing an image
   1. Click on the video feed (main feed or thumbnail feed) to capture an image from that feed

4. Viewing images
   1. Select an image from one of the two dropdown menus to view its thumbnail
   2. Click “Enlarge” to show it enlarged in the main image window

5. Measuring images
   1. The ⌖ tool obtains a point’s coordinates
      1. Click the tool button

   2. Hover the mouse over the image to obtain a point’s coordinates
   3. Click to save the coordinates to the measurement log
2. The tool measures distance
   1. Click the tool button
   2. Draw a line on the map
   3. The length of the line is printed to the measurement log
3. The tool finds heading
   1. Click the tool
   2. Draw a line on the map
   3. The heading of the line is printed to the measurement log
4. The tool calculates area
   1. Click the tool
   2. Draw a rectangle on the map
   3. The area of the rectangle is printed to the measurement log