Design and Validation of an Avionics System for a Miniature Acrobatic Helicopter

by

Kara Lynn Sprague

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2002

Author .....
.....
Department of Electrical Engineering and Computer Science
February 1, 2002

Certified by.....
.....
Eric Feron
Associate Professor
Thesis Supervisor

Accepted by .....
.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Design and Validation of an Avionics System for a Miniature Acrobatic Helicopter

by

Kara Lynn Sprague

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The avionics system for MIT's miniature autonomous helicopter was designed to safely achieve a robust, high-bandwidth feedback control system while meeting physical specifications. The system has provided an efficient platform for testing new closed-loop controllers and has yielded critical flight data used for developing simple models of small-scale helicopter dynamics. The helicopter has demonstrated successful flight under augmented controllers and has also performed three fully autonomous aileron rolls. The avionics system was used to examine software validation and certification techniques. Furthermore, experience in developing the system illuminated a number of design and implementation details that could be modified in future systems to improve robustness and safety.

Thesis Supervisor: Eric Feron
Title: Associate Professor

Acknowledgments

I would like to thank Vlad Gavrilets, Ioannis Martinos, Bernard Mettler, Eric Feron, Raja Bortcosh, David Dugail, Jan De Mot, Emilio Frazzoli, Masha Ishkutina, Ben Morris, Nancy Lynch, and Michael Earnst for their participation in and support of this project. The original flight system architecture was developed by Vlad Gavrilets and Alex Schterenberg. OpenGL open source software, developed by Jan Kansky of Lincoln Labs, was used for the HILSim graphics environment. This project was supported by NASA grants to MIT NAG2-1482, and CMU NAG2-1441, Draper Laboratory IR&D DL-H-505334 and the Office of Naval Research, under a Young Investigator Award.

Contents

List of Figures

Chapter 1

Introduction

With recent innovations in embedded and sensor technology, the use of automation in mission-critical and life-sustaining systems has become prolific. The complexity of these systems continues to grow as advances in both hardware and software technologies make possible new applications. Unfortunately, what can be modeled as a linear growth in system complexity translates into an exponential growth in safety assurance and verification analysis to provide a commiserate level of flight system certification [5]. Unless such systems are implemented cleverly, taking advantage of modularity and clear specifications, the cost of verification could very well be the limiting factor of the technical development in such safety-critical fields. Given the highly volatile market for software and hardware engineering and the speed at which technologies change, it is becoming increasingly difficult for safety-critical systems to both maintain safety standards and exploit opportunities to streamline and re-engineer systems to improve performance while reducing cost [6].

1.1 The Cost of Validation

The avionics industry provides a stunning example of how safety measures and concerns may be stifling the development of potential applications and uses for advanced technologies. The software validation specifications of the Federal Avionics Administration require assembly code-level checking to ensure that every branch of operating code is executed and runs correctly. Such low-level code verification is necessary since compiler certification is currently outside the capabilities of modern technology. Compilers are difficult to certify because they generally have very complex, monolithic implementations and frequently take

advantage of dynamic memory allocation, which introduces a level of non-determinism that renders code validation extremely difficult. Since compilers cannot currently be certified, every piece of mission-critical software that is generated by the compilers must be validated, line by line. One consequence of these rigorous application-level requirements is that a software engineer working on a safety-critical flight system will typically produce only three lines of onboard code per day. Consequently, systems develop very slowly and changes are usually limited in scope. The high certification cost thus fosters an atmosphere of evolutionary development and a tendency to build upon existing monolithic systems rather than implement a more maintainable and modular architecture.

These tendencies away from more modular designs are problematic for a number of reasons. In addition to keeping component complexity at a minimum, the modularization of flight system architectures would also serve to enable the incorporation of commercial off-the-shelf components into new designs. Such systems would also be able to better keep up with the pace of modern technology, since upgrades would not require an overhaul of the entire system. Unfortunately, there is little motivation for the producers of commercial off-the-shelf components to validate their products. The safety-concerned market is only a fraction of the total market for many products with potential uses in the avionics industry. In many cases, this small market share is not enough to motivate commercial product manufacturers to undergo processes to validate and certify their components. Consequently, such readily-available parts can rarely be used in safety-critical systems.

1.2 Motivation

The field of unmanned aerial vehicles (UAV) offers a good test bed for examining flight safety certification procedures. Such systems are becoming more and more popular for their great maneuverability and numerous applications in both military and civilian domains. Possible military applications include surveillance, hazard detection, and even combat. UAV's may also prove useful for aerial photography, remote area exploration, crop dusting, and disaster recovery operations. These aircraft provide the opportunity to streamline and develop verification and certification processes for new technologies without directly endangering human life and at a much lower cost than their full-sized counterparts. UAV's thus offer much more room for experimentation in not only systems and control engineering but also

in validation and certification processes.

UAV's also offer tremendous advantages in terms of small-vehicle dynamics for technological innovations. As sensor, communication, and computer technology evolve and shrink in size and weight, the potential for developing highly maneuverable unmanned vehicles increases. The miniaturization of technology is particularly relevant to the realm of small-scale unmanned aerial vehicles, in which large payloads exact high costs in maneuverability. As these craft become smaller in size, their moments of inertia decrease at an even faster rate, making them more agile. Miniature helicopters are among the most agile and maneuverable of the small-scale UAV's. Experienced radio-control (R/C) pilots of small helicopters routinely execute acrobatic maneuvers such as steady inverted flight, pop-ups, hammerheads, and barrel rolls. Though some systems based on small helicopters have already demonstrated autonomous operation [9], they display fairly modest performance when compared to the acrobatic maneuvers accomplished by expert R/C pilots with similar platforms. The potential of such systems to serve in applications requiring varying levels of autonomy is tremendous, but has not been fully explored.

1.3 Objectives

MIT's Laboratory for Information and Decision Systems (LIDS) Helicopter Project began with an effort to learn pilots' strategies for executing maneuvers by studying pilot commands to the vehicle in flight [8]. Since vehicle states were not recorded, this groundwork supplied only limited insight into small-scale helicopter dynamics and thus motivated an examination of the full state output of the helicopter to pilot inputs. Such data provides more insight into the dynamics of the vehicle in flight, thereby facilitating the development of models and controllers. The first introduction to MIT's instrumented, small-size helicopter equipped with a complete data acquisition system and capable of aggressive maneuvers was provided in [3]. The development of this test platform enabled the first fully recorded acrobatic flight in July of 2000. During this flight, pilot inputs and the state information of the vehicle were recorded while the helicopter performed hammerhead maneuvers and loops. The data collected from such flights has enabled the development and validation of a full-envelope helicopter model, as well as the design of the feedback control logic used in autonomous acrobatic flight. The ultimate goal of the helicopter project, to perform a

fully autonomous aggressive maneuver, was achieved on November 18, 2001. This flight, in which the helicopter performed three aileron rolls autonomously, demonstrated that the inherent agility of such systems is not severely limited by autonomous operation. Methods for verifying and validating the mission-critical software were evaluated in parallel with the development of the flight system, with the hopes that they would illuminate more efficient means to certify such systems and provide guarantees of safety.

This document presents the MIT helicopter and avionics system that was used to achieve autonomous aggressive flight. Chapter 2 details the technical aspects of the mechanical features of the helicopter and hardware involved in the design of the avionics system as well as the software system architecture and implementation details for the control system and ground station. Chapter 3 explores various system validation techniques and offers insight into the role these methods may play in the certification of safety or mission-critical systems. Recommendations for future work on similar systems are offered in Chapter 4.

Chapter 2

Flight System

The flight system, shown in Figure 2-1, is composed of the helicopter, avionics system, flight software and ground station. The test vehicle is a derivative of an Xcell-60 helicopter manufactured by "Miniature Aircraft USA." The 60 cm3 engine and carbon fiber frame give the craft a total weight of about 10 lb. While the basic rotorcraft is known for its reliability as a stable acrobatics platform, the avionics system is entirely custom designed. Mounted in a single aluminum enclosure suspended below the landing gear of the helicopter, the avionics consist of a CPU, a power regulator, five sensors to provide closed-loop feedback, and batteries. All sensors, with the exception of a pilot command receiver and servo control unit, are commercial-off-the-shelf components and are rated as flight-ready. The onboard and ground station software is entirely implemented in C and runs on QNX 4.25, a popular real-time operating system. The high vibration levels of the rotorcraft, which can interfere with the operation of sensors and equipment are attenuated by a passive vibration isolation subsystem, which consists of an elastomeric isolator suspension system for the avionics box. Each of these subsystems is described in greater detail below.

2.1 Vehicle Hardware

The test vehicle, shown in Figure 1, is an Xcell-60 hobby helicopter powered by a mixture of alcohol and oil. This model is favored by many R/C pilots for its ability to perform acrobatic maneuvers. The vehicle includes a twin-blade rotor with a 5 ft diameter and frame weight of about 10 lb. The main rotor assembly features a fairly stiff hub joint, which, according to flight test data, provides a fast angular rate response to cyclic inputs.

Figure 2-1: The flight vehicle equipped with avionics box.

The assembly also includes a Bell-Hiller stabilizer bar, which augments servo torque with aerodynamic moment to change the cyclic pitch of the blades and adds lagged rate feedback to improve the handling qualities of the helicopter.

Fast servos are necessary for a high-bandwidth control system able to perform aggressive maneuvers. Thus, the helicopter is equipped with fast Futaba S9402 servos. These servos feature a high torque rating of 111 oz-in and a 7 Hz small-signal bandwidth under half rated load, as measured with frequency sweeps.

Hobby helicopters have a high tail rotor control authority and low yaw damping that results in poor handling qualities. Artificial yaw damping is required for manual control. While this damping is usually achieved using a hobby gyro, the test vehicle uses the yaw rate measurement from the instrumentation package. This modification allows the gain on the yaw damping to be controlled by the onboard computer. The computer simulates gyro gains while the helicopter is flying in manual mode, and commands different gains while the vehicle is flying in automatic mode. While this modification facilitates an easier implementation of the control system, a drawback to this approach is that the vehicle becomes very difficult to fly in the case of an avionics or software failure.

An electronic governor with magnetic RPM pick-up sensor adjusts the throttle to main-

tain the commanded engine RPM. All flight operations are performed at 1600 RPM. The governor maintains tracking error to within 40 RPM, as measured by a ground-based stroboscope.

2.2 Avionics Package

The purpose of the avionics package for the test vehicle is three-fold. The first mission of the system is to log the sensor data from flight. High frequency data updates are recorded onboard in volatile memory until the end of a flight at which point they are transferred to the ground station. This flight data is critical to the development of models for small-scale rotorcraft and also provides valuable insight into the flight strategies of the pilot [2]. The second goal of the avionics system is to act as the control system for the helicopter, meaning that it must be capable of achieving high bandwidth feedback control. Accurate estimates of the vehicle states are achieved through algorithms described in [10]. The control system provides for control augmentation and fully autonomous acrobatic flight, allowing the vehicle to perform aggressive maneuvers such as loops and rolls completely autonomously. The third and final goal of the avionics system is to provide the ground operators a view into what is going on inside of the onboard computer. The ground station is the only portal through which the operator can obtain any information about the state of the control system. The helicopter communicates with the ground station through wireless LAN, providing low-frequency updates of logged flight data and a data channel for operator input into the control system.

2.2.1 Design Considerations

In addition to the operational requirements of the system, there are also a number of physical constraints imposed by the environment in which the avionics system is operating. The most important physical limitation on the design for a project of this scope is the weight of the system. In cooperation with Draper Laboratory, flight tests were performed with a dumb weight mounted on the vehicle's landing gear. The tests demonstrated that the pilot is able to perform acrobatic maneuvers with a payload of up to 7 lb.

Another vital characteristic of the avionics system is that it must be immune to the numerous sources of vibration inherent in a small helicopter. The primary source of vibration

on such a vehicle is the main rotor, spinning at roughly 27 Hz. Other sources of vibration include the engine, the tail rotor, and the tailboom bending resonance. These vibrations must be attenuated for an onboard sensor package to report reasonably accurate measurements, particularly in regards to gyroscope and accelerometer data. The control system must then be designed to fit into a single unit that is heavy enough for a passive vibration mount to be effective, but is light enough that the vehicle can still perform aggressive maneuvers. Simple calculations, described in [3], indicate that commercially manufactured elastomeric isolators can effectively attenuate the vibrations described above for a payload weighing more than 5 lb.

Although a compact system design is very effective for vibration isolation, it also compounds the issue of interference. Another physical constraint on the avionics system is that it must protect sensitive devices such as sensors and receivers from internal sources of electromagnetic (EM) and RF interference. Interference is a known cause of failure in such systems. The primary sources of interference in this avionics design are the wireless LAN transceiver and GPS antennae. The power supply circuitry also seems to contribute a fair amount of interferance. These system elements emit radio transmissions that can interfere with the R/C receiver, resulting in the pilot losing control of the vehicle. Common shielding precautions were used to alleviate electro-magnetic interference induced by the wireless LAN transceiver antenna and the GPS antenna is placed on the nose of the vehicle to isolate it from the rest of the avionics.

2.2.2 Avionics Description

The avionics package is a 7 lb data-acquisition and control system mounted on elastomeric isolators on the customized landing gear of the helicopter. The suspension system effectively attenuates high-frequency disturbance inputs from the main rotor and the engine, and can withstand a sustained positive acceleration of 3 g and a negative acceleration of 1 g. The aluminum box containing the onboard computer, wireless Ethernet transmitter, R/C receiver, and sensors helps to isolate the R/C receiver antenna from the radio interference emitted by the other onboard devices. Substantial range testing indicates that the best configuration for the receiver antenna is hanging directly below the helicopter. The antenna was encapsulated in a flexible plastic tubing and mounted to the vehicle frame such that the antenna hangs below the craft during flight but can bend to the side when the helicopter is

on the ground. Flight tests using this antenna configuration demonstrate that the antenna remains in place even during inverted flight.

The onboard computer runs at 266 MHz and is equipped with 32 Mb of RAM and 16 Mb of FLASH memory for data and program storage. The computer also includes four serial ports and four analog channels, all of which are used in communication with the onboard sensors. The single PC104 board is also equipped with a network entry port for connection to a wireless Ethernet transceiver.

The avionics system includes a separate pilot command receiver and servo controller board, which serves as the routing system for pilot and computer commands to the onboard servomechanisms. This unit is responsible for reading the pilot commands from the receiver, passing received commands through to the main onboard computer, and writing the commands from the main computer to the servos. The unit consists of two SX28 chips, which are used for measuring pilot inputs and driving the servos, and a PIC16F877 chip, which interfaces to both SX28 chips and the main computer. Optoisolators are used to provide different power sources to the logic bus and servo bus. This source isolation is needed to prevent inductive noise from interfering with the digital logic. The servo controller board is described in detail in [7].

In manual mode, the computer passes the original pilot commands through to the servos. The one exception is the tail rotor pitch command, which is computed by the main computer, and returned to the servo controller board. This command contains proportional yaw rate feedback. As noted above, this exception generates an unwanted dependence on the onboard computer; the vehicle is extremely difficult to fly without scheduled yaw gains and thus any problem with the flight computer or software could be catastrophic for the entire vehicle. In automatic mode, the computer either uses the pilot commands in conjunction with the estimated states to generate new commands to the servos, or the pilot commands are completely ignored and the computer generates its own commands based on the updated state estimate and the maneuver that the helicopter is attempting to perform.

The sensor package consists of an inertial measurement unit (IMU) with three gyros and three accelerometers, a single GPS receiver with pseudorandom code measurements, a barometric altimeter with resolution of up to 2 ft, and a magnetic compass. The IMU gyroscopes feature drift rates of less than 0.02 deg/sec during 10 minutes. The IMU drift rates are sensitive to vibrations and temperature. The Superstar GPS receiver from Canadian Mar-

coni provides 1 Hz position and velocity updates, with a latency of 1 sec. Due to the short flight time, relative navigation in the absence of Selective Availability gives approximately a 2 m circular error, and horizontal velocity errors are limited to within 0.15 m/sec. The Honeywell HPB200A barometric altimeter provides a very stable altitude reference except when the helicopter is affected by ground effect, in which downwash from the main rotor increases the total pressure measurement leading to faulty altimeter readings. The Honeywell HMC2003 triaxial magnetoresistive compass provides heading measurements, which are currently not used. This sensor requires an additional reset circuit to compensate for the magnetic field induced by the helicopter avionics.

2.2.3 Safety Analysis

The avionics hardware of the test vehicle is designed for fault detection and error recovery. Power loss to the computer, servo control board, or servos is perhaps the most damaging threat to the system. Each of the onboard batteries is wired such that voltages can easily be observed through battery meters on the ground station display. The main batteries, which power the flight computer, IMU, BAR, GPS, and CMPS, have the shortest lifetime. The voltage of these batteries is measured through one of the four analog inputs of the onboard computer. A hardware-implemented hot-swap system helps to eliminate in-flight power loss. For warm-up and ground routines, a ground battery is plugged in and the main flight batteries disconnected without interrupting the power to the rest of the system. The hot-swap thus provides more time in-flight without the fear of the power running low in the onboard batteries. The servo controller board, which is powered separately from the computer and other sensors in order to avoid interference, monitors it's own battery as well as the battery that powers the servos. In addition to monitoring battery voltages, the servo controller board also includes a fail-safe emergency mode. By the flip of a switch on the R/C transmitter, the entire flight control system, including all avionics hardware with the exception of the servo controller board and yaw rate feedback, can be bypassed and pilot commands sent directly to the servos. This feature is critical for error recovery in case the control system becomes deadlocked or one of the sensors fails completely in the middle of a flight.

Unfortunately, as is evident from a crash, the aforementioned safety mechanisms cease to be effective when the R/C receiver locks due to interference or some other unidentifiable,

non-control system-related problem occurs. Fault detection in such cases is difficult since the helicopter appears to be functioning correctly from the point of view of the control system. At the same time, the control and data-logging system may be used to detect unusual changes in the helicopter trajectory or to identify strange behavior, such as the craft speedily plummeting to the ground. The R/C receiver is able to detect when it has stopped receiving commands from the transmitter and responds by commanding all channels to zero. The control system interprets this set of commands as a command to hover. Thus, when reception from the transmitter is lost, the vehicle will automatically begin to hover. While this safety feature has been demonstrated in simulation, it has never been tested on the field. More investigation into how such fault detection systems could be used to initiate recovery processes is needed. Additionally, the current design is plagued by the problem that when the system does go down before logged data files can be transferred to the ground station, all flight data is lost. In the event of a crash or bizarre system failure, this data would be of great use in determining the source of the error and could provide insight into how similar problems could be avoided in the future. The obvious remedy to the loss of flight data is to write sensor data to the onboard non-volatile flash memory in flight. Unfortunately, this solution is problematic because such logging is a drain on the computer system resources. While technological innovations such as non-volatile RAM or enhanced high-speed I/O systems may provide a solution to this problem, they are on the horizon and currently not available.

2.3 Onboard Software

The software flight system runs on embedded QNX 4.25, a popular real-time operating system that is widely used in the aerospace industry. The onboard flight software consists of nine processes, including a bootstrap process that is started when the computer is first turned on, the main control process, an interrupt service routine for the IMU, device drivers for the servo controller board, GPS receiver and barometric altimeter, analog port monitors for the compass and battery, and a telemetry data server. At the highest level, the bootstrap process waits until it receives a message from the ground station telling it to start the main control process. After initialization and a handshake with the ground display application, the control process allocates seven shared memory buffers to hold the data from the IMU,

state estimate, servo commands, pilot uplink commands, GPS, altimeter, and compass. The former three buffers are filled in by the main loop of the control process. The later four buffers are filled in by driver processes forked by the control process before it enters its main loop. The data that is stored in the shared memory buffers is used in the main control loop to generate a state estimate of the helicopter in flight and to provide control augmentation and autonomous flight. In addition to an interrupt service routine for the IMU and drivers for the GPS, servo controller board, altimeter and compass, the control process also forks a process that provides battery readings from the analog port upon request and a process that provides the freshest telemetry readings from all of the shared memory buffers upon request. These process interactions are illustrated in Figure 2-2. The separate processes of the flight control system are described in detail below.

2.3.1 Process: Bootstrap

The bootstrap runs as a background process on the helicopter waiting for input from the ground station. Started when the computer first boots up and never exiting or returning, the bootstrap's entire purpose is to act as an omnipresent receiver for "Connect" and "Stop" messages from the ground station, and to spawn or kill the main flight control process when requested. The process first attaches the name "/heli/parent" to itself through the QNX name server so that the ground display application can locate it and send it messages. The bootstrap then waits for a "Connect" request from the ground station. When such a message is received, the process checks whether the flight control process is already running. If the main process is already running, the bootstrap kills the current instance of the process and spawns a new instance. The bootstrap then waits for confirmation from the control process that the spawn and setup were successful. A reply is then sent to the ground station informing it of the status of the main control process. If the child process starts correctly, the bootstrap waits for a "Stop" request from the ground at which point it sends a kill signal to the control process. A flow diagram depicting the control flow of the bootstrap process is given in Figure 2-3.

2.3.2 Process: Control

The main flight control process is spawned when a "Connect" request is issued to the onboard bootstrap process. The control process registers the name "/heli/main" with the

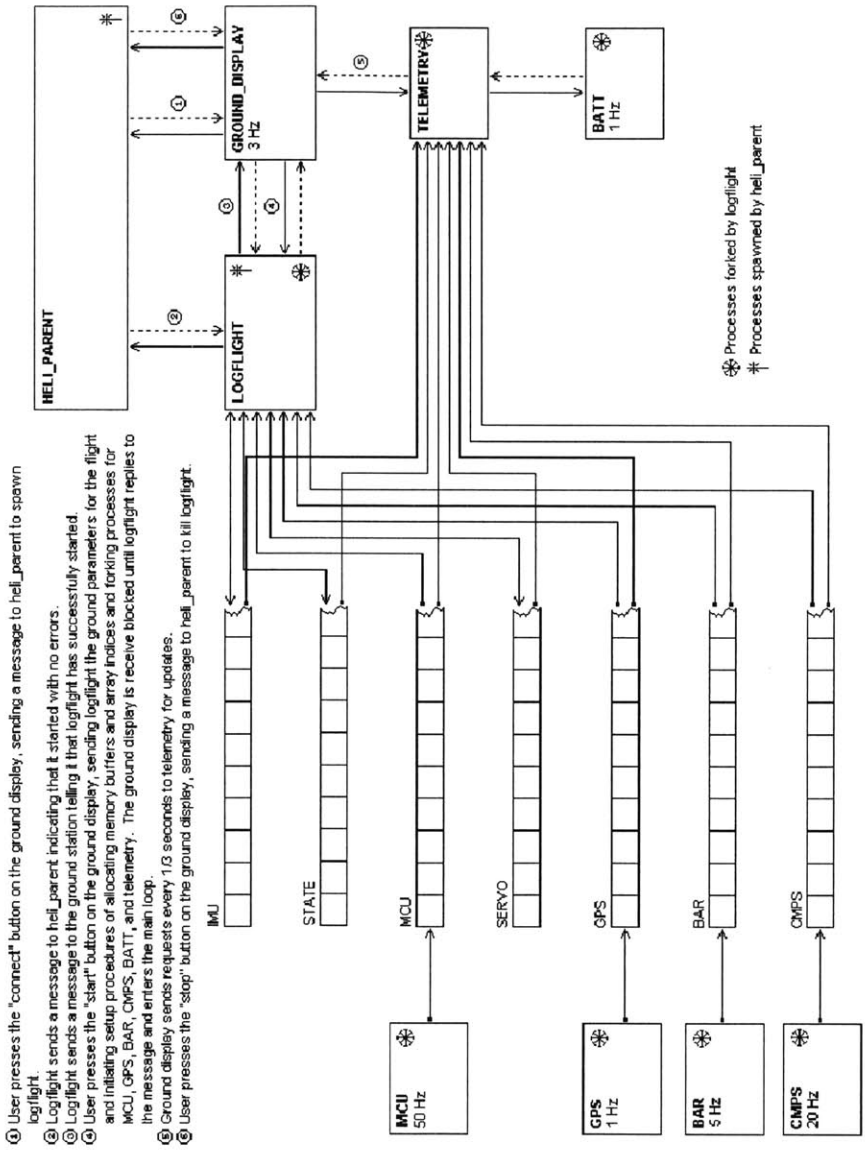

Figure 2-2: Block diagram of the flight software system interactions.

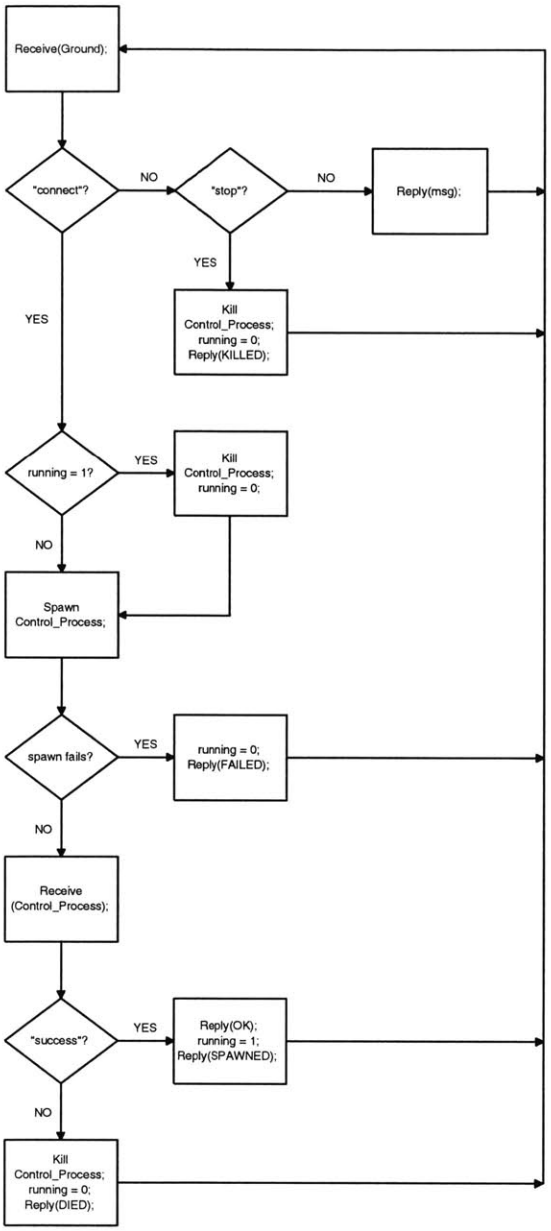

Figure 2-3: Bootstrap process block diagram.

QNX name server and locates the ground station by searching for the name "/ground/main", which is registered by the ground display application when it is first started. The control process then sends a message to its parent, confirming that the spawn succeeded. A message is then sent to the ground station to register the process with the ground display application. Once this message is sent, the process becomes receive-blocked until it receives a message from the ground station containing the parameters for the flight. This message is only sent when the ground station operator presses the "Start Logging" button on the ground station. The process uses these ground parameters to set up the names of the log files in which sensor data is stored. The control process then enters a series of initialization routines in which it allocates shared memory buffers to store packets for IMU data, state estimates, pilot uplink commands, servo commands, GPS data, altimeter data, and compass data, allocates global pointers to index into all of the shared memory buffers, attaches proxies for the IMU, servo controller board, GPS, altimeter, and compass drivers to kick when new messages arrive, forks the IMU interrupt service routine, drivers for the servo controller board, GPS, and altimeter, analog channel monitors for the compass and battery, and the telemetry data server, sets up a watchdog timer to ensure that the main loop runs at least every 12 msec, and finally registers a cleanup routine to be called upon reception of a kill signal. When all of this setup is complete, the control process replies to the ground station parameter message with an indication that it is ready to begin logging. The process then enters the main control loop.

The main loop of the control process integrates all of the sensor data into a state estimate that is used in control augmentation systems and autonomous flight. The loop waits to receive proxies from the various device drivers and interrupt service routines. In QNX 4.25, a proxy is simply a very short, canned message that does not require a reply. Timing is derived from the IMU, which ideally sends fresh data packets through a serial port to the computer once every 10 msec. Since the timing of the main loop is so critical to the success of the control software, a watchdog timer ensures that the loop runs at least every 12 msec. If the IMU skips a message or fails entirely, the flight control system will continue to run. Without the IMU data updates, however, the state estimate deteriorates quickly and manual intervention is required to save the vehicle.

The main loop receives proxies from six sources: the IMU interrupt service routine, the watchdog timer, the GPS device driver, the servo controller board device driver, the

altimeter driver, and the compass analog port monitor. Proxies from the IMU interrupt service routine and watchdog timer are handled in essentially the same manner. The only difference is in which packet is copied into the location of the freshest packet in the IMU buffer. In the case of a proxy from the IMU interrupt service routine, the new data packet from the IMU is copied into this location in the IMU buffer. If the watchdog timer triggers a proxy then the IMU is assumed to have dropped a packet and so the old freshest packet is copied into the location of the new freshest packet in the IMU buffer. After receiving a proxy from the IMU interrupt service routine or watchdog timer, the control process resets the watchdog timer and then copies the freshest IMU data it has into the new location in the IMU buffer. Every data packet in each buffer is time-stamped with the time it was received relative to the global time of the main control process. The global index into the IMU buffer is also updated to reflect the new location of the freshest IMU data. The IMU data must be calibrated for biases before it can be used in state estimation. Thus, there is a period of eighty seconds after the IMU has warmed up during which each IMU packet is used to accumulate an estimate of the biases. Once this calibration period is over, the accumulated biases are subtracted from the new IMU data. Every other IMU message, or at 50 Hz, the state is propagated using the two most recent readings from the IMU and the current state estimate. At this time, new commands to the servos are generated using the current state estimate, the commands from the pilot, and the control system specified by the ground station operator at the beginning of the flight. These commands are then passed through to the servos. Finally, the commands sent to the servos and the new state estimate are time-stamped and copied into their respective shared memory buffers. The respective global buffer indices for servo commands and state estimates are then updated.

Proxies from the remaining four sources, the GPS, servo controller board, altimeter, and compass drivers, indicate that the freshest data from these devices is available in their respective shared memory buffers at the index specified by their respective global indices into the arrays. New data from the GPS receiver, altimeter, and compass is used to update the state estimate in order to offset biases accumulated by the IMU in flight. Data from the servo controller board includes the commands from the pilot and is stored in the pilot uplink command buffer for use when the control process is generating new commands to the servos.

While the main control loop cannot run indefinitely due to the limited memory available

for storing the seven shared memory buffers, the loop is generally exited by sending a kill signal to the control process from the bootstrap process. The kill signal instructs the control process to go into its cleanup routine, which involves deleting the watchdog timer, killing all of the processes that were previously forked, and saving all of the shared memory buffers to files on the ground station. When the cleanup routine has completed, the control process exits and the onboard computer returns to its idle state, awaiting another command from the ground station.

2.3.3 Process: Inertial Measurement Unit

The IMU data is the most critical part of the feedback control system. The data not only provides the timing for state estimation updates and commands to the servos, it is also the primary source of data for estimating the state of the helicopter. Data from the GPS receiver, altimeter, and compass is only used to supplement the state estimate calculated with the IMU data. The IMU device driver is encapsulated within the main control process in order to provide the timing for the state estimation and controllers. The IMU is connected to the onboard computer through a serial port, sending new data packets at 100 Hz. The control process initializes the serial port using a custom serial port driver that is also used by the servo controller board and GPS device drivers. A built-in FIFO buffer on the serial port is used to ensure that bytes coming through the channel do not get dropped. The control process also attaches a proxy to itself. This proxy is used to notify the main control process that a new and complete data packet is available for use. The IMU driver uses an interrupt service routine, forked by the control process in its initialization sequence, to send notifications using this same proxy. Since the IMU is so important, the IMU interrupt service routine has the highest priority of all of the processes forked by the main control process. The interrupt service routine is essentially a state machine that tracks how much of the IMU data packet has arrived. The state machine synchronizes on a designated header byte for IMU packets and then fills in an IMU data packet as each new byte is received. Packet integrity is maintained through the use of a single-byte checksum, which is updated with the arrival of each new byte and then compared to the last byte of the incoming packet. If the checksums match, then the interrupt service routine returns a proxy, which is routed by QNX 4.25 to the process to which the proxy is attached. In this case, the proxy is sent to the main control process. If the checksums do not match, the state machine is reset back

to the start state to wait for another header byte.

A stand-alone process is used to test the IMU and the interrupt service routine. The stand-alone process is a dedicated driver for the IMU, meaning that it performs all of the IMU data reception steps of the main control process, but nothing else. The program has a single argument, which is the name of a log file to store the data received from the IMU while it is running. Just like the main control process, the stand alone driver initializes the serial port using a FIFO buffer. The program then sets up the interrupt service routine by attaching a proxy for the interrupt service routine to use to notify the driver of new packets and attaching the service routine, which is essentially the same as forking the interrupt service routine. The stand alone driver also tests the functionality of the watchdog timer for the main control loop. Another proxy is attached for the watchdog timer to trigger every 12 msec. The driver then sets up a kill signal to call a cleanup routine, just as in the main control process. Finally, the driver sets the timer and enters into its main loop, waiting for proxies from either the interrupt service routine or the watchdog timer. Proxies are handled by marking the current time, and resetting the timer. If the proxy was from the interrupt the new data packet is saved to the end of a buffer and the buffer index is updated. The new data is also printed to the screen. If the proxy was from the watchdog timer then the data pointed to by the index into the buffer is copied to the next location in the buffer and the index is incremented to point to this next location. Proxies from the watchdog timer imply that the IMU skipped a packet or failed in some way and so an error is indicated when this condition is encountered. The loop continues to receive proxies until the buffer is full or a kill signal is received. When the loop exits, the interrupt service routine is detached and the buffer contents are saved to the log file specified in the argument to the program.

2.3.4 Process: Servo Controller Board

As mentioned above, the servo controller board routes pilot commands from the R/C receiver to the onboard computer and drives the four servomechanisms of the helicopter with the commands generated by the computer. Although the main control loop handles the later job, outputting commands to the servo board at 50 Hz, a special process is responsible for the former task of acquiring new pilot commands. As in the case of IMU data acquisition, the servo controller board process is essentially a driver wrapped around an interrupt service routine. The interrupt handler for the servo controller board is simply a

state machine that tracks how much of the new data packet has arrived. The state machine synchronizes on a designated header byte for servo controller board packets and then fills in a pilot uplink data packet as each new byte is received. Packet integrity is maintained through the use of a single-byte checksum, which is updated with the arrival of each new byte and then compared to the last byte of the incoming packet. If the checksums match, then the interrupt service routine returns a proxy to the driver process. If the checksums do not match, the state machine is reset back to the start state to wait for another header byte.

The servo controller board process uses the same routine as the IMU driver to initialize the serial port and set the FIFO buffer level. A proxy for the interrupt service routine to trigger and the service routine itself are attached to the servo controller board process. The process ensures that the servo controller board is functioning properly by sending nine consecutive non-header bytes through the serial channel. The process then enters its main loop in which it waits for the arrival of a proxy from the interrupt handler before time-stamping and copying the complete packet composed by the interrupt service routine into the shared memory buffer for pilot uplink commands at the location specified by the local index into the array. This local index always points to the location one packet in front of the global index, which is used to update the local index of the main control process after a proxy is received from any of its child processes. The global array index is then incremented and a proxy is sent to the main control process to indicate that new pilot uplink data is available. Finally, the local array index is incremented so that it points to the next location to be filled in the shared memory buffer. The process then waits to receive another proxy from the interrupt handler.

The servo controller board driver is complemented by a stand-alone version which also generates a timed log of the servo controller board data. In addition to acquiring the data from the pilot uplink to the R/C receiver, the stand-alone driver also passes these commands through to its servo-driving outputs, emulating the behavior of the servo controller board in emergency mode. Since the stand-alone driver incorporates data logging and takes on the task of passing commands through to the servos, a couple more steps are required before it can enter into its main loop. First, the stand-alone process must set up a kill signal to call a cleanup routine so that logged data can be output to a file. A timer must also be initialized and set to trigger a proxy once every 20 msec, thus prompting the process to

drive commands to the outputs to the servos. The main loop then awaits the reception of proxies from two sources: the interrupt service routine and the timer. If a proxy from the interrupt handler is received, the stand-alone driver behaves just as described above. If a proxy is received from the timer, however, the most recently received pilot commands are driven onto the outputs to the servos and printed to standard out. The main loop continues receiving proxies until a kill signal is received, at which point the contents of the data buffer are saved to a file specified by the input argument to the stand-alone process.

2.3.5 Process: Global Positioning Satellite Receiver

The driver process for the GPS receiver follows the same mold as that for the IMU and servo controller board; the process consists of a driver wrapped around an interrupt service routine. The interrupt handler for the GPS receiver is just a state machine that tracks how much of the new data packet has arrived. The state machine synchronizes on a designated header byte for GPS device packets and then fills in a GPS data packet as each new byte is received. Packet integrity is maintained through the use of a two-byte checksum, which is updated with the arrival of each new byte and then compared to the last two bytes of the incoming packet. If the checksums match, then the interrupt service routine returns a proxy to the driver process. If the checksums do not match, the state machine is reset back to the start state to wait for another header byte. The GPS process uses the same routine as the IMU and servo controller board drivers to initialize the serial port and set the FIFO buffer level. A proxy for the interrupt service routine to trigger and the service routine are attached to the GPS process and a signal-catcher is set up to call a cleanup routine when the process is killed. The process then enters its main loop in which it waits for the arrival of a proxy from the interrupt handler before time-stamping and copying the complete packet composed by the interrupt service routine into the shared memory buffer for GPS data packets at the location specified by the local index into the array. This local index always points to the location one packet in front of the global index which is used to update the local index of the main control process after a proxy is received from any of its child processes. The global array index is then incremented and a proxy is sent to the main control process to indicate that a new GPS packet is available. Finally, the local array index is incremented so that it points to the next location to be filled in the shared memory buffer. The process then waits to receive another proxy from the interrupt service

routine.

The stand-alone version of the driver for the GPS receiver is exactly the same as the actual driver except that it prints each new packet to standard out immediately after it is received. Unlike the stand-alone versions of the IMU and servo controller board drivers, the stand-alone GPS driver does not currently generate a log file of its data.

2.3.6 Process: Barometric Altimeter

Whereas the drivers for the IMU, servo controller board, and GPS are all very similar in form and function, the device driver for the barometric altimeter deviates from these previous implementations. There are two key characteristics of the altimeter that necessitate a different approach to the driver than that used for the other devices: the altimeter outputs its data packets in ascii format and the device has an active handshake to initialize data readings. Due to these two conditions, the same custom serial port initialization routine employed by the drivers for the IMU, servo controller board, and GPS cannot be used by the altimeter. Instead, the altimeter driver uses another custom serial driver, which opens the serial port as a file descriptor and uses standard POSIX commands to interface to the serial device.

The driver process for the altimeter opens the serial port for writing and reading using a custom driver that accesses the port using file descriptors. As part of initialization, the altimeter is reset by writing a reset command to the file descriptor. When the device is reset, it should respond by writing to its output. These bytes are read and verified by the driver process from the serial port as a precautionary step. If the device fails to respond to the reset request, then there is something wrong and the device may have failed. Given that the device comes back online after being reset, the sampling frequency for the pressure readings is written to the serial port and the device is set for continuous sampling. The altimeter is configurable to output pressure readings within a frequency range from 1 Hz to 10 Hz. For the purposes of the helicopter, altitude updates at 5 Hz are sufficient.

Instead of using a separate interrupt service routine that is called each time a byte is received through the serial port, the altimeter driver attaches a proxy to the serial port file descriptor to trigger following the reception of any specified number of bytes. Since the altimeter outputs its data in ascii, which would necessitate a fairly large number of states, this implementation is less cumbersome than the state machine format used by the drivers

of the IMU, servo controller board, and GPS. Additionally, since the altimeter packets do not provide any kind of checksum error-checking, a byte-by-byte interrupt service would only be a drain on system resources. The proxy on the serial port file descriptor is initially armed to trigger after receiving a complete message from the altimeter. Within the main loop of the altimeter driver, the process waits to receive a proxy signaling a complete read. Once a proxy is received, the process compares the packet header to the known altimeter data packet header to verify that the packet reception loop is synchronized to the device. If the received header does not match, bytes are read from the serial port one by one until the first byte in the known header is encountered. A packet starting with this byte is filled in and the verification process begins again. Once a complete and valid packet is received, it is time-stamped and copied into the shared memory buffer for altimeter data packets at the location specified by the local index into the array. This local index always points to the location one packet in front of the global index which is used to update the local index of the main control process after a proxy is received from any of its child processes. The global array index is then incremented and a proxy is sent to the main control process to indicate that a new altimeter packet is available. Finally, the local array index is incremented so that it points to the next location to be filled in the shared memory buffer. The process then re-arms the serial port file descripter proxy on the reception of another complete packet and becomes receive-blocked until the proxy is triggered again.

The stand-alone altimeter driver is basically the same as the actual driver, except that valid readings are printed to standard out. The sampling frequency for the device is input as an argument to the program, though it defaults to 5 Hz if this parameter is not included. The stand-alone driver for the altimeter currently does not generate any log file.

2.3.7 Process: Magnetic Compass

The compass is used to provide low-frequency updates to the heading measurement of the state estimate. The magnetic compass is a completely passive device; magnetic field readings from the x, y, and z axes are used as inputs to three of the four analog ports of the onboard computer and there is no data transmission protocol. The real-time compass readings are combined with biases measured in a calibration routine to calculate heading updates for the helicopter during flight. The compass driver consists of a timer which fires at 20 Hz. Each time the timer expires, the three analog channels connected to the compass are examined

and their values are used to generate an updated heading measurement. Thus, the compass process is simply a driver for the analog port combined with a timer.

The process initializes the analog port by setting the control flags of the analog channels for unipolar, single ended operation with an external clock. The process then sets up the GPIO address and reads in the bias parameters that are stored in the onboard flash memory following a simple calibration routine. A proxy is attached to a timer and the timer is set to fire once every 50-msec. The process then enters its main loop, in which it waits for the arrival of a proxy from the timer. Each time the timer fires and a proxy arrives, the compass process resets the timer and then serially reads the three analog channels by setting the control and channel selection bytes for each channel and saving the value into a time-stamped packet pointed to by the local index into the shared memory buffer for the compass data. This local index always points to the location one packet in front of the global index which is used to update the local index of the main control process after a proxy is received from any of its child processes. The raw compass readings are converted to uncalibrated magnetic field readings in Gauss, which are then corrected for biases through application of the calibration parameters. Once this manipulation of the data is complete, the global array index is incremented and a proxy is sent to the main control process to indicate that new compass data is available. Finally, the local array index is incremented so that it points to the next location to be filled in the shared memory buffer. The process then waits to receive another proxy from the timer.

As with the drivers for the IMU, servo controller board, GPS, and barometric altimeter, the compass driver is also complemented by a stand-alone version which enables easy debugging. The stand-alone compass process matches the actual driver exactly except data is printed to standard out as each new packet is received.

2.3.8 Process: Battery

The battery process acts as a data server. When the telemetry process makes a request for information to the battery process by sending it a message with a particular identifier, the battery process reports the current reading on the analog channel to which the main flight batteries are connected. At the most basic level, the battery process waits for a message from telemetry and then packages a time stamp and a reading of the analog port into a battery packet and replies to the telemetry message with the battery packet. Battery data

is not logged, nor is it visible to the main control process.

2.3.9 Process: Telemetry

The telemetry process is a completely passive observer of the data stored by the main control process. The purpose of the telemetry process is to service requests from the ground station for the most updated state information of the helicopter without ever interfering with any of the flight-critical onboard processes. The telemetry process obtains its state information from the data that is collected by the various drivers forked by the main control process and placed into the shared memory buffers. When a device driver reads a new value from a sensor, as in the case of the IMU, servo controller board, GPS, altimeter, and compass, or the freshest value is updated in the main control loop, as in the case of the state estimate and servo commands, the new value is stored in the respective shared memory buffer and the global index into the array is incremented. After receiving a request from the ground station, telemetry updates its own local copies of the array indices with the current values of the global indices. Telemetry then generates a message containing copies of the freshest packets from all of the shared memory buffers as well as a reading from the battery meter. As described above, battery measurements are not stored in a buffer; readings are only made when the telemetry process makes a request. Each section of the message is separated from the rest by identifiers, which also provide indications for which type of data is located where in the message. The telemetry message is sent to the ground display application in the form of a reply to the original request message automatically sent three times a second from the ground station. The ground station then parses this message and outputs it in a format that is easily interpreted by the ground station operator.

2.4 Ground Station

The ground station is the only available interface to the onboard flight software. The ground station is simply a laptop that runs QNX 4.25 and is equipped with a wireless LAN transceiver so that it can transmit to and receive messages from the onboard computer during flights. The primary ground station application is a display program that provides the operator limited access to the flight computer. The ground display has three functions: it allows the ground operator to enter basic flight parameters such as file names for the

storage of flight data and the control augmentation system to be used in automatic flight, it initiates the onboard flight logging and control system software, and it provides feedback to the ground about the state of the software running on the helicopter and the state of the helicopter.

Figure 2-4 shows a screen shot of the ground display application. The interface is designed to give the ground operator as much state information about the helicopter and flight code as possible. The onboard software is initiated by pressing the "Connect" button, which sends a message to the omnipresent onboard bootstrapping process to spawn the main flight logging and control system process. If this control process is successfully started - meaning all variables are initialized, the process has registered with the QNX name server, and the ground station display process has been located - then it sends a message back to the bootstrap indicating success. Another message is sent from the main control process to the ground station to let the ground station know that it was started successfully and as a means of registering the main control process with the ground display application. This registration step is necessary for the ground station to later send messages directly to the onboard control process. If the control process manages to send a "success" message to the ground display, then the large red indicator between the "Connect" button and the "Start Logging" button turns green, the "Connect" button becomes inactive, and the "Start Logging" and "Stop" buttons are activated. If the bootstrap fails to spawn the main control process or the process dies during initialization, the indicator remains red and the "Connect" button is the only button that is activated.

Once the main control process is successfully started and has registered with the ground station, the ground operator can press the "Start Logging" button, which sends a message to the receive-blocked control process. This message contains the flight parameters that the ground operator can enter into the ground display such as the suffix to append to the file names for storing flight data and the choice for the control augmentation system to be used in flight. The choice of control augmentation systems is also reflected on the ground display, where the data labels for the pilot uplink commands are changed to reflect that which the pilot is commanding under the control augmentation system. Once these parameters are sent to the helicopter, the control process begins to allocate and initialize shared memory buffers and variables and fork processes for the sensors and telemetry. If all initialization processes are successful, the control process replies to the original message from the ground

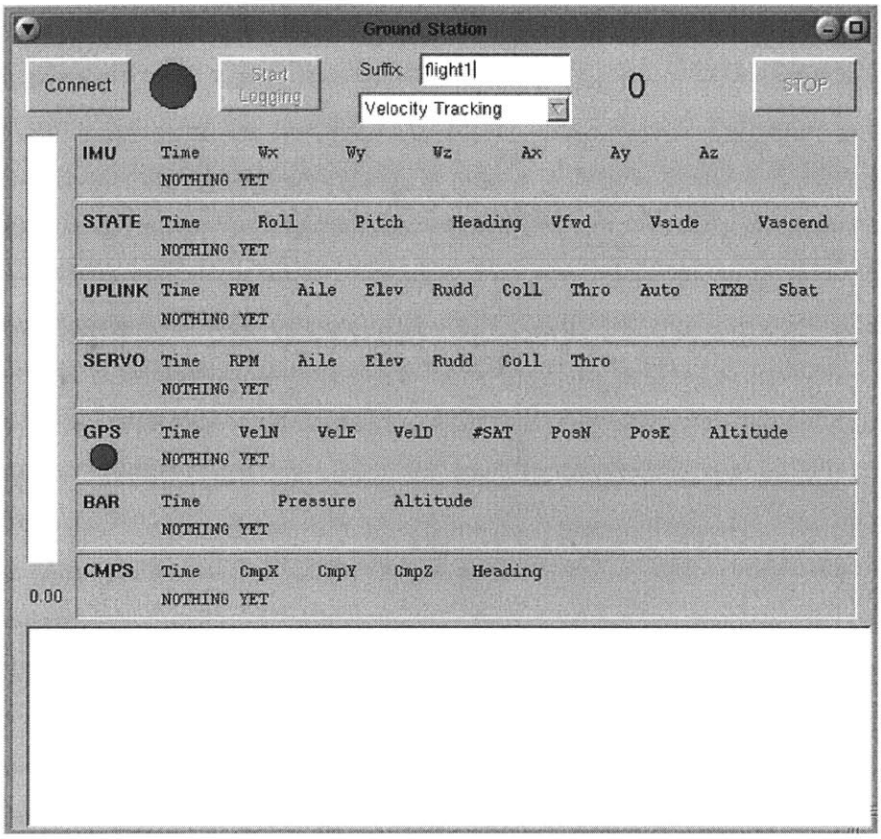

Figure 2-4: A screen shot from the ground display application.

station and then enters into its main loop, as described above. The "Start Logging" button becomes inactive and the "Stop" button remains the only activated button. If, however, the control process fails to allocate memory for the buffers or a process cannot be forked, the control process exits. In this later case, the ground display application indicates that the logging failed and the operator must reset the ground display application by pressing the "Stop" button.

While the onboard control process is in its main loop, the ground station automatically sends requests for new telemetry data at 3 Hz. The large indicator on the ground display flashes as new messages are received from the telemetry process. The counter located between the parameter input fields and the "Stop" button indicates the number of telemetry messages received. As each new message is received from telemetry, it is parsed and the data is output to the ground display in each of the appropriate fields. As noted above, telemetry messages consist of the freshest data readings from the IMU, state estimation, pilot commands, servo commands, GPS, barometric altimeter, compass, and battery monitor. The battery voltage is converted to a percentage of a fully-charged battery and output on the meter on the left side of the ground display. The actual voltage reading is displayed below the meter. Extensive battery tests comparing voltage to battery life yield a measure for how much time is left on the batteries when they are at a certain voltage. The battery meter is displayed in green until the batteries are believed to have only a third of their lifetime left, at which point the meter turns red. Unfortunately, the battery voltage has only a weak correlation to the time left on the battery. Field tests have shown that temperature is also a huge factor in battery performance. The battery meter, though useful, provides only an estimate of the battery lifetime. A final indicator on the ground display indicates if GPS is picking up enough satellites for the position and velocity information to be accurate. If GPS is receiving from fewer than five satellites, the indicator is red and any kind of automatic or augmented control flight is dangerous. The indicator is green when there are enough satellites to adequately supplement the control system with position and velocity updates.

At the end of the flight, the ground station operator can press the "Stop" button to send a message to the onboard bootstrapping process telling it to kill the main control process. This kill signal initiates the cleanup routine in the control process, which stops all of the previously forked processes on the helicopter and saves all of the logged flight data to files

on the ground station. At this point, the "Connect" button on the ground display is again activated and all other buttons are made inactive. The large connection indicator on the ground display is also again made red to show that there is no connection present.

Additional information about the state of the onboard processes is provided to the ground operator through the large text field at the bottom of the ground display. All responses from the bootstrap about the running state of the main control process are output to this area in addition to all text status and error messages from the control process after it has registered with the ground station. This area is particularly useful in displaying the progress of the files being copied from the onboard computer to the ground station at the end of a flight, since no other region on the ground display shows this information. The text field is also extremely useful for debugging onboard processes and inter-process communication.

Chapter 3

Software Validation

Despite the relative ease of testing hardware, such as sensors and actuators, it is extremely difficult to integrate and verify fault-tolerance in software systems. The safety aspects of software can be divided into two classes. The first concerns whether or not inputs or commands to the system achieve the desired, expected, and safe results. The second category addresses the issue of unanticipated inputs or commands and whether they still yield the desired, expected, and safe results [6]. Currently, verification and certification of flight-critical software involves a long, labor-intensive, manual endeavor consisting of extensive testing, process documentation, and inspections. This process not only consumes both time and money, it also does not scale well to the highly complex systems enabled by today's technology [5]. Both military and commercial avionics systems designers would benefit greatly from a streamlined software certification process that allows them to quickly evaluate the flight-readiness of new and commercial off-the-shelf software modules and systems.

Unfortunately, efforts aimed at mitigating the burden of flight system certification have been largely unsuccessful. Systems designed for robustness and modularity must still undergo extensive validation. Measures that test the performance of the system more directly by simulating running conditions are also not guaranteed to examine every branch in the flight software. Though extremely useful during development, software testing through simulation is not sufficient to guarantee correctness. Another venue for showing correctness is through theorem proving and model checking. Such techniques for formal verification show promise as alternative means for certification, but the environments in which these techniques are being developed and tested do not lend to their successful integration into

flight-system verification efforts. These various techniques for guaranteeing correctness and safety are evaluated in further detail below.

3.1 Robust and Fault Tolerant System Properties

Though all mission-critical software must be extensively tested and evaluated after specification and implementation, practices employed at the time of implementation also help streamline the process of verification and certification. One such practice is the use of hard modularity; systems designed as a collection of numerous self-sufficient sub-systems with clearly defined rules of interaction are easier to verify than large and complex monolithic designs. Modularity allows each component of the system to be treated as a black box with a well-defined input/output relationship. Since the complexity of a single component must be less than or, in the worst case, equal to the complexity of the system as a whole, verifying individual components is much more simple than a holistic approach to system verification. Modularity also helps to limit the propagation of errors. Once system integration takes place, proper functioning of the individual components can be assumed and verification is then limited to the component interactions. In addition to improving the overall testability of a system, modularity also provides for code and hardware reuse. Validated software drivers for the IMU, for example, can be duplicated in other systems with a commensurate level of assurance that they will function correctly.

The mission-critical flight software for the autonomous helicopter follows a practice of hard modularity. Specifically, each sensor has its own suite of tools including interrupt service routines, drivers, and binary-to-ascii conversion programs. The main control process is the glue that integrates each of the separate components into a single, cohesive flight system. Telemetry and real-time non-volatile data-logging processes are designed as pure observers; they do not alter the state of the control system or program data in any way. This approach to system design has a number of advantages. First, despite the fact that the IMU provides the timing for the main loop, the system functions entirely asynchronously. While synchronicity is sometimes beneficial is providing a deterministic order of events, it can also lead to race-conditions that are an added source of faults. Furthermore, synchronous designs are generally not accurate models of real-time systems. A second advantage of the modularity of the flight software architecture is that the components can be tested and

evaluated apart from the system, thus facilitating the search for and removal of detected errors and faults. Finally, the modularity provides an easy mechanism by which components can be easily integrated into or removed from the system.

In addition to a modular design, mission-critical software must also be designed to be robust to hardware glitches and failures, as well as software issues such as memory allocation problems and file handling errors. As mentioned above, the control loop relies on the IMU for timing. Specifically, if the state estimate is not updated at 100 Hz, as dictated by the arrival of a new data packet from the IMU, the control system will become unstable and the vehicle may crash. Though the IMU is rated to drop only about one packet out of every million, a watchdog timer is integrated into the control loop to ensure that the state estimate is updated at least every 12 msec, with or without a new IMU packet. The watchdog timer thus provides the pilot with a larger window of time to respond to an IMU failure, since it guarantees that the control loop will continue despite the loss of its critical input.

Another feature of the software design that makes the system more robust is the use of serial port FIFO queues in the interrupt service routines for the IMU, servo controller board, and GPS receiver. The interrupt service routine for each sensor is designed to trigger on the arrival of each single byte through the serial port. If, however, the computer is busy and an interrupt on the serial ports cannot be serviced immediately, the FIFO buffer ensures that the system will continue to receive bytes from the sensors without dropping or losing any. When the resources of the system are freed and it is able to service the interrupt, all of the bytes in the queue are handled, in the order they were received, just as if they had just arrived.

3.2 Hardware-In-The-Loop Simulator

The Hardware-In-the-Loop Simulator (HILSim) is a mechanism for testing the whole avionics system before it is out on the field plummeting at 50 m/s toward the ground. The HILSim enables the testing of new software releases and hardware configurations in the safety of the lab environment and without endangering the vehicle. The primary goal in the design of the HILSim is to include as much of the flight hardware in the testing loop as is possible. The HILSim configuration includes an exact duplicate of the flight computer

and onboard software as well as another servo controller board. The servo controller board drives a rack-mounted set of servos to simulate the servos installed on the actual helicopter. The servo positions on the rack are measured by coupling the actuators to potentiometers which output values to an analog to digital converter attached to the simulation computer at 100 Hz. In addition to reading the values of the servo positions, this computer is also responsible for outputting simulated sensor data to the sensor inputs of the clone computer. The helicopter equations of motion are integrated at the same rate, and simulated sensor outputs are generated and returned to the flight computer duplicate through inverse interfaces at the appropriate rates. The simulation computer is a 733 MHz Pentium III with several standard PCI and ISA expansion I/O cards. The simulation computer is connected to an OpenGL visualization server that renders the helicopter as it maneuvers around an artificial flight field. A diagram summarizing the HILSim setup is provided in Figure 3-1.

In addition to its primary role as a testing mechanism for the avionics system, the HILSim is also useful in training the pilot to fly using control augmentation systems. Failure conditions such as power outages and loss of sensor inputs can be effectively simulated such that the pilot is trained to respond appropriately. Finally, the visualization and data outputting capabilities of the HILSim can be harnessed to replay actual flights during which the sensor information was recorded and saved. This feature is useful for debugging the real-time software system and device drivers using real inputs.

The HILSim has proven to be an excellent and essential tool for analyzing the operational usability of the control system, including the ground station display and inter-process communication. However, though the HILSim offers a good indication that the system performs correctly in general, it in no way exhausts the possible conditions that actual test flights may produce. For instance, the HILSim does not incorporate any power concerns or possible loose connections to avionics hardware that would test certain branches of the flight-software. It could even be argued that the added value of a HILSim that reuses data from past successful missions is minimal; since the sensor outputs that are tested in the HILSim are known to have already run through the system without failure, they do not increase the overall test coverage. The only true certification that the HILSim can provide is that the system still accepts these already-tested inputs despite changes to the flight software.

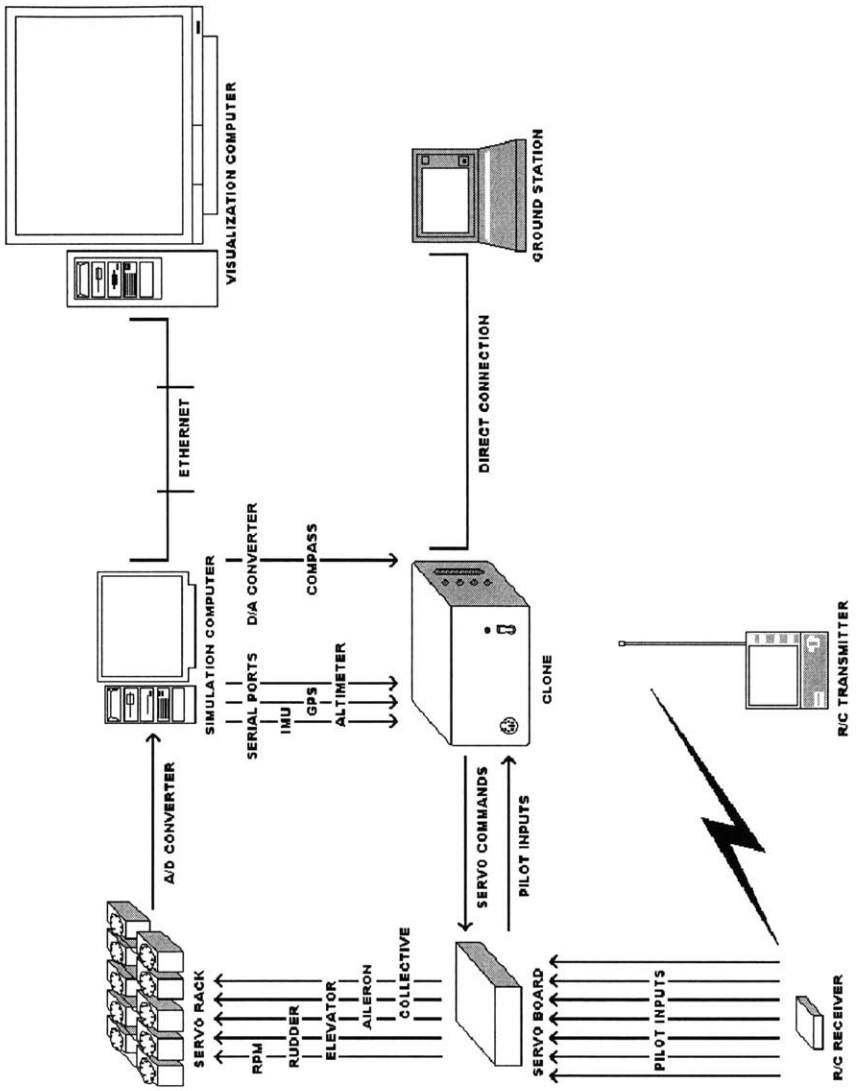

Figure 3-1: HILSim system interaction diagram.

3.3 Formal Verification

In order to completely test the mission-critical software and thereby certify the control system, formal verification of the flight systems in terms of timing analysis and model checking is required. Model checking is an automatic procedure to verify if a given system satisfies a given temporal property [4]. The application of such formal methods has been shown to have significant benefits in terms of safety and correctness of software. For example, formal methods, in combination with static analysis and dynamic testing, were employed with great success in developing formal requirements and the equivalent pseudo-code that manufacturers could use to develop the operational systems for the Traffic Alert and Collision Avoidance System that is now part of the standard cockpit equipment for major airlines [1]. The expectation was that the use of a formal specification and the ability to rigorously analyze and test the design logic and program structure would reveal errors and deficiencies that would have otherwise contributed to erroneous system behavior or deteriorated system performance.

While there is evidence that the application of formal methods can be used in a process to validate and certify mission-critical software, there also seems to be a significant gap between theory and application. Much of the effort surrounding formal verification methods has focused on the mathematical proofs needed to extrapolate useful trends and information from a formally specified problem. Unfortunately, the question of how applicable and realistic these methods are for use in a standard verification and certification process remains unanswered. Attempts were made to apply two different formal verification tools to the flight software of the autonomous helicopter, with only limited success. These experiences illuminated the huge obstacles that must be overcome for formal verification methods to become a feasible option in the world of safety-critical software certification. The tools were evaluated in terms of their real-world applicability and value added.

The Daikon Invariant Checker and the Polyspace Verifier represent two very different examples of the formal verification tools that are currently under development. Where Daikon is a university-grown tool designed to verify and supplement program specifications, Polyspace is gradually being accepted in the aerospace industry for its tremendous applications in software testing and verification. Despite their differences, the two tools share in common a very practical approach to applying formal verification methods to real

software systems.

3.3.1 Daikon Invariant Checker

The Daikon Invariant Detector, developed by Michael Ernst and colleagues at MIT, reports properties that are likely to hold at certain points in a program. These properties are mathematical expressions similar to what might appear in a formal specification or algorithm. These invariants are used to verify that the program meets the desired specifications and to better understand and more quickly debug programs. Invariance detectors such as Daikon may also have possible applications in performing timing analyses and in other areas of functional verification. One goal of applying Daikon to the flight software was to illuminate some new methods for verifying software without the required formalism of a logic proof or model checker.

Daikon works by instrumenting the code to be analyzed with observer processes that output variable values to a log file before and after each function call within the application. The log of variable values is generated while the software application executes. Post-processing of this log file yields insight into the invariants of the software application. The fact that the code to be analyzed must actually execute for Daikon to generate the data it needs for post-processing introduces the rather significant problem of system incompatibilities and general usability. QNX 4.25 is so substantially different from the native Linux environment of Daikon that the tool was extremely difficult to apply to the flight software. Attempts to copy the source files over to Linux for instrumentation and then copy the instrumented files back over to QNX to execute failed due to internal checks within Daikon to ensure that the code it is instrumenting will compile. Special system calls used in QNX such as Send() and Receive() are not accepted by Daikon compilation checker. Daikon was also compiled to run on QNX 4.25, but the executable consistently core dumped. Since QNX 4.25 does not offer any standard compilers or debuggers, the option of debugging the Daikon application on QNX fell outside of the scope of the project. A final effort to use Daikon on the flight code involved porting the system software to QNX 6, which is to QNX 4.25 what Windows 95 was to Windows 3.1. QNX 6 not only features GCC and GDB as its standard compiler and debugger, it also offers full support of POSIX. QNX 6 is, in effect, far more compatible with other contemporary operating systems than QNX 4.25. Unfortunately, this porting project involves significant changes to the flight software,

thereby rendering any validation of the ported software irrelevant to the certification of the software running on the actual helicopter.

3.3.2 Polyspace Verifier

While Daikon evaluates preexisting code and determines the rules by which the software functions, more formal techniques of software verification involve applying logic definitions and theorems to models based on the software. This type of certification is called logic or model checking and is much more difficult to apply directly to software. The Polyspace Verifier, developed by Polyspace Technologies, employs methods of abstract interpretation to perform static analysis of source code at compilation time in an attempt to detect possible run-time errors. Static verification is the process of checking the dynamic properties of a software application without ever actually executing the program. Compared to verification techniques that repeatedly execute a program with different inputs in order to test every possible computation path, static verification yields a dramatic improvement in running time. However, the difficulty of statically verifying a program still increases exponentially with the size of the application; static verification alone is not efficient enough to provide an effective means of certifying complex software applications. The Polyspace Verifier avoids this pitfall through the use of abstract interpretation, which is just another way of saying that the program evaluates an abstraction of the input software application, thereby decreasing the amount of work for the static verifier. Given source code, the tool enumerates places where there are attempts to read non-initialized variables, access conflicts for unprotected shared data in multi-threaded applications, referencing through null pointers, buffer overflows, illegal type conversions, invalid arithmetic operations, overflows or underflows of arithmetic operations for integers and floating point numbers and unreachable code.

As with Daikon, there is no QNX 4.25 executable offered for the Polyspace Verifier. However, since the tool only works on source code, it can still be applied to the system software by copying the core of the flight code over to a Linux machine on which Polyspace is installed. With only minimal tweaking, Polyspace was able to generate an analysis of the onboard software. The tool reported no critical errors in the flight code and only a few minor faults. While most of the indicated faults were generally harmless, their eradication does make the software more robust. The ability of the Polyspace Verifier to indicate unreachable code could also prove extremely useful in software certification processes.

3.3.3 Formal Verification in Safety-Critical Systems Certification

The effort to apply the tools to the flight software revealed a number of deficiencies in the current safety-critical system development and validation environment. The objective of efforts to apply formal verification techniques to analyze such software applications should be to provide a more efficient means to certify software systems. Unfortunately, these intentions are often thwarted by fundamental issues in system applicability. At the most basic level, there appears to be little or no interface between those developing mission-critical systems that require validation and those developing the applications that could be used to certify such systems. As evinced by experiences with Daikon, some verification applications simply do not execute on the very platforms that run the software most in need of verification. Though Daikon is currently only a university-sponsored project with limited scope, the capabilities of the tool could be much better explored and exploited if it could analyze safety-critical software on its native platform. Polyspace demonstrates a much better understanding of the requirements of the safety-critical software development industry, as it is a fully packaged tool that can be applied to any software system independent of the development platform of the software to be analyzed. As software becomes more and more prevalent in safety-critical systems, such easily applied and practical tools may prove useful in developing streamlined processes for software verification and certification.

3.4 Validation by Example: Flight Tests

The ultimate validation for the avionics system of the autonomous helicopter is that it works. The system has been used in a number of flight operations to log sensor and state data. Data has been collected from the helicopter while responding to step inputs and frequency sweeps commanded by the pilot. This data is used to generate and validate models of small-scale helicopter dynamics [10]. The recorded flight data also provides insight into the strategies of experienced R/C pilots in performing aggressive maneuvers [2]. Finally, the data offers a means of debugging more generally the helicopter and avionics system. Flight records show where sensors failed to deliver a packet on time or gave unusual readings. This information is critical to ensuring that the timing of the control system architecture is sufficient for closed-loop feedback control. This data would prove extremely useful in identifying the reasons behind many system failures, including the single crash of the Spring of 2001. The

low-frequency data collected by the ground station while the vehicle is in flight shows that the helicopter ceased responding to pilot commands shortly after take-off. The vehicle began to roll and accelerate downwards, plummeting to the ground and crashing. Though the ground station data offered some insight into the details of the crash, the high-frequency flight data, lost in the crash, contained many specifics that simply could not be captured by the ground station data. Though the data recording feature of the system failed in this particular instance, the flight still demonstrated that the receiver antenna configuration was inadequate and resulted in the design of a flexible antenna mount that braces the antenna in its optimal position even during inverted flight.

In addition to its achievements as a flight data recording system, the avionics system has also been used to demonstrate the full range of vehicle autonomy from manual control to operation under augmented control systems to fully autonomous flight. High bandwidth rate and velocity-tracking feedback controllers have been implemented on the system and successfully demonstrated on the field [10]. An iterative process involving flight tests, controller parameter tweaking, and simulations yielded augmented controllers that could be used by an expert R/C pilot to perform aggressive maneuvers. These augmented controllers were used as an intermediate step in the development of a fully autonomous control system. The avionics system fulfilled its ultimate objective on November 18, 2001, with a flight in which the MIT helicopter performed three fully autonomous aileron rolls.

Chapter 4

Design Recommendations

The experience of developing the avionics system for a miniature acrobatic helicopter has provided tremendous insight into how similar systems should be designed and implemented in the future. While the MIT helicopter and avionics system are capable of performing autonomous aggressive flights successfully, many aspects of the design and implementation of the system could be more streamlined or improved.

4.1 Flight Software Specifications

For the most part, the original flight software for the autonomous helicopter was not the product of rigorous specification processes followed by careful implementation. Instead, the initial system, incorporating only the ground station and flight data logging capabilities, was developed in the span of a few months and was only minimally documented. In hindsight, the original design architecture may not have been the best possible solution. The following description of a new system architecture and specifications for the processes involved offers a more robust and modular approach to the design of the flight system.

4.1.1 Development Environment

The proposed development platform for the next generation of flight system software is QNX 6, the newest release of the popular real-time operation system. Although QNX promises support for version 4.25 until at least 2004, Neutrino offers a number of advantages over the previous kernel, including threading support, full POSIX compliance, support for open-source GCC and GDB, and the CVS revision control system. These features allow QNX to

finally take advantage of the modern innovations in technology that are already a part of most contemporary operating systems. The most compelling reason to use Neutrino is that it is fully supported by QNX, whereas the company has ceased supporting QNX 4.25.

4.1.2 System Architecture Overview

The proposed system architecture for future systems is very similar to that of the current implementation except that it extends the modular design to include all components. While the most basic functions of the sensor operation and data-logging software adheres to the practice of hard modularity, the main body of the actual control software requires re-engineering to make it conform to the standards of the rest of the code. Taking advantage of the modular design of the control system, the flight software can be more easily evaluated for timing constraints and abstraction violations.

The recommended system is a distributed system with separate driver processes for each sensor, including the IMU, servo controller board, GPS, compass, and altimeter. While timing is still derived from the incoming IMU data packets, the process itself is not an integral part of the main control process, which is responsible only for periodically receiving updates from each of the sensors and incorporating the data into the state estimate and controllers. The sensor drivers offer a complete input output package much like the current implementation of the driver for the barometric altimeter; all interfaces are standard POSIX built on top of a single reusable serial driver. Additionally, all drivers include a watchdog timer to ensure packet updates and every stand-alone driver has data logging capabilities.

4.1.3 Process: Bootstrap

The purpose of the bootstrap process is to receive signals from the ground station, control the state of the onboard processes, and report the state of the onboard processes back to the ground station. The process responds to two commands from the ground station: start and stop. The start command begins the onboard processes required for data logging and control augmentation. The stop command ends any running processes by sending them kill signals and returns the entire flight software system to its idle state. The process reports success or failure in any of its actions to the ground station.

4.1.4 Process: Control

The main control process essentially serves as the receiver for proxies sent by the sensor drivers. Upon invocation, the process notifies its parent of success and locates the ground station, later sending a message to the main ground application in order to register itself with the ground station. The process then waits to receive the starting parameters, which at this point consist only of the controller that is to be used in the upcoming flight. After initializing shared memory buffers and forking driver processes for the IMU, servo controller board, GPS, altimeter, and compass, initializing shared memory buffers for the state estimate and servo commands, and forking telemetry and battery monitoring processes, the main control process finally replies to the original message of the ground station and then enters into its main loop. Functioning completely asynchronously, the main loop simply waits for messages from the various sensor drivers and then incorporates the respective data into the state estimate and controllers when it arrives.

The process exits its main control loop and enters a cleanup routine as soon as it receives a kill signal from its parent. The routine sends one more message to the ground station requesting the final flight parameters, which consist of a base directory and file name suffix to append to the names of the log files for the sensory data recorded during the flight. The cleanup routine then kills all of the children of the control process and dumps the contents of the seven shared memory buffers used to store individual data packets from the IMU, servo controller board, GPS, altimeter, compass, state estimate, and servo commands to these separate files.

4.1.5 Process: Sensor Drivers

The drivers for the IMU, servo controller board, GPS, and barometric altimeter all have the same basic structure with only minor changes to address the issues of different data packet sizes and initialization routines. These sensor drivers consist of a general driver process that is forked by the main control process packaged around a more basic serial port interrupt service routine. The serial port interrupt handler provides the transition timing for an internal state machine, which constructs a data packet, byte-by-byte. Once an entire data packet has been received and verified via a checksum, the packet is copied into the respective shared memory buffer and the array indices are updated. Each driver includes a

separate watchdog timer, which ensures that data packets are updated at a consistent rate. An important note is that this redundancy in driver implementation is only possible if the binary data transmission option is selected on the altimeter during initialization.

The compass driver operates very similarly to the other drivers, except that the watchdog timer is the only mechanism for updating the data packets in the shared memory buffers. This deviation is a result of the fact that the compass is attached through the analog channels of the onboard computer and have no internal timing loops to facilitate data transmission; the raw data is on the analog channels themselves. The battery meter has properities similar to those of the compass driver in terms of initialization and setting up the analog channels for reading, but it is designed to respond only to requests from the telemetry process for updated battery data.

Each sensor driver, including those for the compass and battery, has a corresponding stand alone version. The stand-alone drivers share the same functional code as the regular drivers in addition to code that prints the information of each packet received to standard out. Each stand-alone driver takes in a filename as a parameter which is used to generate a timed log file of all of the sensor data collected while the driver was being run. This log file follows the same respective format as does the main control process when it finally outputs logged sensor data to files.

4.1.6 Process: Telemetry

The telemetry process is designed to be a completely passive observer of the data that is being actively logged in the seven shared memory buffers as well as the data being generated in the battery process. Telemetry simply waits for messages from the ground station, which are in effect, requests for updated sensor data. The process then constructs a long reply message composed of copies of the freshest data packets by looking at the shared memory buffers at the location specified by the global index into the array and polling the battery process for new readings from the analog channel. Once it has constructed this message, telemetry sends the data in the body of a reply and then waits for another request from the ground station.

4.1.7 Ground Station

The primary jobs of the ground station are to communicate with the onboard bootstrap process to spawn and kill the main control process, send the starting parameters to the main control process, effecting initialization routines and eventual data logging, send the final parameters to the main control process cleanup routine to initiate saving the logged information to non-volitile data files, request and display updated telemetry data, and display as much information as possible about the state of the onboard processes. The primary change to the ground display application mandated by the modifications to the flight system software is simply that the ground display must be ready to receive a message from the control process cleanup routine after it sends a stop message to the onboard bootstrap process. After prompting the ground station operator for file saving preferences, including a root directory and filename suffix, it then replies with this information to the cleanup routine. All other behaviors of the ground display application remain consistent with the current implementation described in Chapter 2.

4.2 Hardware Considerations and Upgrades

In addition to basic modifications to the flight system software and ground station, the current avionics implementation could also be improved by upgrading the hardware components to their modern-day standards and employing a few system integration solutions.

4.2.1 Modern-Day Avionics Components

Many of the design and implementation decisions regarding the architecture of the flight software system were made to optimize the speed of the control loop, thus providing for higher-bandwidth controllers. In some cases, proper system design techniques were abandoned for the sake of meeting these control loop specifications. Such design trade-offs could be avoided by using more modern equipment. With the rate at which electronic components improve in terms of physical form and functional capability, new system designs are well out of date before the implementation phase is even completed. A majority of the hardware used in the helicopter avionics system is more than three years old and far behind the capabilities of modern-day components.

The new software architecture described above, which includes a completely modular

IMU process that is separate from the main flight control process and a new data logging scheme that stores flight data to non-volatile memory, could only be possible if a faster computer were used in the onboard avionics. These software modifications result in decreased system perfomance and fail to meet the timing constraints of the control laws using the computational power of the current avionics implementation. Replacement of the 266 MHz Cyrix onboard computer with a faster model would provide the speed required to make up for penalties in time and computation power demanded by the more modular system architecture and secure data logging functionalities described above.

Similarly the flash memory capacity of the onboard computer should be expanded so as to allow for the storage of an entire day of flight operations on a single card. While this upgrade is, for the most part, only relevant if the system is modified to save flight data to the non-volatile flash media during flight, it is still a worthwhile investment simply for the sake of making the system more easily expandable.

4.2.2 System Integration Solutions

In systems as complex as the helicopter and avionics package, the only workable solution to systems integration is to decouple systems whereever possible. For example, the control augmentation systems of the the helicopter work by decoupling the equations of motion for the helicopter, thus allowing individual state components to be controlled without strong interactions with the other elements of the state. Modularity is another means by which system components are effectively decoupled. The current implementation of the the flight system follows this idea of hard modularity for the most part. The deviations from this design principle in the software systems can be accounted for with the modifications to the flight system architecture described above. However, violations of the modularity principle in the hardware systems should also be addressed. The most glaring example of such an infraction is the coupling of the yaw gain scheduling with the onboard computer, which renders the vehicle nearly uncontrollable in manual mode without the help of the computer. Such dangerous system interactions and dependencies should be removed and replaced with more modular solutions that decouple different elements and limit fault propagation.

Another modification to the hardware system that would greatly improve the functionality and use is to connect the battery hotswap to a digital input so that amount of time spent on flight and ground batteries can be logged by the computer and displayed on the

ground display. This task is currently performed manually by the ground station operator, who keeps a log of the battery times through each flight operation. Not only would this hardware modification lessen the responsibility and task load on the ground operator, but it would also help to automate battery monitoring with respect to voltage as well as time. This data would be of enormous use in evaluating the safety of using the batteries at different voltage levels in terms of how long they are capable of supplying sufficient power to the electronics systems.

Finally, there is much room for improvement in terms of the mechanical design of the avionics system. A hatch should be added for the flight batteries so that they can easily be removed and replaced on the field without disturbing the rest of the system components. The container should feature a more adequate ventilation system, including a small fan to cool the power supply, so that parts do not overheat.

4.3 Best Practices

The recommendations for modifications to the flight system software and avionics hardware are complimented by some general process and design principles. The first such best practice is that procedures and processes should be automated whenever possible. This automation principle not only results in a more efficient development process, it also guarantees reproducibility. One area of the helicopter flight system that could be greatly improved with automation is the ground display. The purpose of the ground display application is to provide the ground operator an idea of the state of the helicopter by displaying all of the most recent data from telemetry. There are certain conditions that the ground station operator must verify before the vehicle can be operated with any control augmentation system. These conditions can be considered as a set of truth values, which, when combined through simple logic of ANDs, ORs, and NOTs, indicate whether the vehicle is safe or not. For example, the vehicle must be in the line of sight of at least four satellites for the positioning and velocity updates to be considered valid. The GPS data panel then includes an indicator that shows whether this condition is valid or not. If such methods of automatically checking the state of the helicopter, including more complicated means of verifying that sensors were updating properly, were used for all of the telemetry data of the helicopter, the flight system would be significantly more robust and safe. Although such state checks are used to some

degree onboard the vehicle, there is room for expanding these measures to take into account more failure conditions.

Automating safety checks for the flight code is only one of the ways that automation can be used to make the helicopter project more efficient and safe. Off-line processes involving code design, implementation, and testing can also be automated as a means of lowering the length of the development cycle. Currently, the helicopter software suite includes scripts for automated version control and system backups. Additional scripts to automate code verification processes would significantly reduce the overhead of using such methods in system development and would therefore encourage safer development practices.

While automation is helpful in making processes more efficient, it still does not replace the need for rigorous planning and discipline. Flight tests with the vehicle require careful preparation and thus cannot be changed on the fly without threatening the integrity of the mission. As a general rule of thumb, flight and design plans should never be changed once they have been agreed to and finalized. Similarly, the flight system should remain a black box while on the field; the system should undergo thorough laboratory testing after any modification. For example, if the vehicle's flight battery happens to run low for no apparent reason on the field before the scheduled flights are completed, the rest of the flight test should be cancelled. These seemingly over-conservative practices are necessary to ensure that the system is never unnecessarily placed in potentially dangerous situations.

A final best practice that helps to ensure system safety is to avoid shortcuts or hacks in the design and implementation. Specifically, when two components should logically be separate but can be implemented faster in a monolithic design, adhere to the principles of modularity. Furthermore, when the easy way to engineer a component is not the same as the right way to engineer the component, development should also follow the path of the right way.

4.4 Conclusion

Though the avionics system for the MIT miniature autonomous acrobatic helicopter demonstrated great success in achieving its goals on the field, from a system design perspective there is tremendous room for improvement. Ultimately, similar commercial systems should be easily expanded and modified to provide further or improved functionality. Keeping in

mind the high cost of verifying and validating such mission-critical systems, the design must then be kept modular. Ideally, a complete avionics system is but a unique configuration of individually documented and certified, re-usable components whose interfaces are both simple and robust. New or commercial technology can thus be incorporated as additional or updated system components. The solution to the growing problem of certifying and verifying safety-critical systems seems to largely involve designing more robust and adaptable systems that do not require extensive modifications to exchange or add components. Efforts aimed at methodologically providing guarantees about correctness and safety, such as the formal verification methods examined in Chapter 4, has also produced compelling results. However, this work assumes that systems will continue to grow in complexity, rather than focusing on techniques to design systems that scale and grow without necessarily becoming more complex.

Bibliography

[1] H. Abdul-Baki, J. Baldwin, and M. Rudel. Independent Validation and Verification of the TCAS II Collision Avoidance Subsystem. *IEEE Aerospace and Electronics Systems Magazine*, 15(8), 2000.

[2] V. Gavrilets, E. Frazzoli, B. Mettler, M. Piedmonte, and E. Feron. Aggressive maneuvering of small autonomous helicopters: A human-centered approach. *International Journal of Robotics Research*, 2001.

[3] V. Gavrilets, A. Shterenberg, M. Dahleh, and E. Feron. Avionics System For a Small Unmanned Helicopter Performing Aggressive Maneuvers. Philadelphia, PA, October 2000. 19th AIAA Digital Avionics Systems Conference.

[4] P. Hsiung. Concurrent Embedded Real-Time Software Verification. Computers and Digital Techniques, IEEE Proceedings, 2000.

[5] M. Lane. Predicting the Reliability and Safety of Commercial Software in Advanced Avionics Systems. Philadelphia, PA, October 2000. 19th AIAA Digital Avionics Systems Conference.

[6] R. Loesh, A. Gosnell, T. Benoist, R. Wyskida, and J. Johannes. An Engineering Approach to Critical Software Certification. 32nd Annual Hawaii International Conference on Systems Sciences, 1999.

[7] I. Martinos. *System Integration and Part Design For a Small Autonomous Helicopter.* PhD thesis, Tufts University, Medford, Massachusetts, 2001.

[8] M. Piedmonte and E. Feron. Aggressive Maneuvering of Autonoumous Aerial Vehicles: A Human-Centered Approach. Snowbird, UT, October 1999. International Symposium on Robotics Research.

[9] C. Sanders, P. Debitetto, E. Feron, H. Vuong, and N. Levenson. Hierarchical Control of Small Autonomous Helicopters. Daytona Beach, FL, December 1998. IEEE Conference on Decision and Control.

[10] K. Sprague, V. Gavrilets, D. Dugail, B. Mettler, and E. Feron. Design and Applications of an Avionics System for a Miniature Acrobatic Helicopter. Daytona Beach, FL, 2001. 20th AIAA Digital Avionics Systems Conference.