An Embedded Controller for Quad-Rotor Flying Robots Running Distributed Algorithms

by

Brian John Julian

B.S., Cornell University (2005)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author ....
Department of Electrical Engineering and Computer Science
August 20, 2009

Certified by ....
Daniela L. Rus
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

This work is sponsored by the Department of the Air Force under Air Force contract number FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

An Embedded Controller for Quad-Rotor Flying Robots Running Distributed Algorithms

by

Brian John Julian

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Multiple collaborating quad-rotor flying robots are useful in a broad range of applications, from surveillance with onboard cameras to reconfiguration of wireless networks. For these applications, it is often advantageous to have the robot team be a distributed system. In this thesis, an embedded controller capable of running distributed algorithms is presented for the quad-rotor flying robot.

The robot platform is first characterized to help guide the design of the embedded control module. These modules are fabricated and tested on the quad-rotor flying robots in both indoor and outdoor environments. To propagate state estimates throughout the robot team, a location-based multi-hop algorithm is proposed. Network limitations, such as sub-optimal bandwidth and finite communication range, are implemented in hardware-in-the-loop simulations to determine system performance. A novel coverage algorithm for multiple hovering robots with downward facing cameras is then demonstrated on the embedded controller. The results from numerous indoor and outdoor experiments are discussed.

Thesis Supervisor: Daniela L. Rus
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This research would not have been possible without the support of those around me. I would like to thank my advisor Daniela Rus, whose steadfast confidence in my work kept me excited and motivated through the toughest challenges. I am indebted to all the members of the Distributed Robotics Laboratory for their willingness to endure late nights even if not for their own projects. In particular, Mac Schwager and Michael Angermann have my highest respect for their guidance and good humor. Carrick Detweiler and Iuliu Vasilescu also deserve special recognition for the many valuable discussions and debugging sessions.

I would also like to thank MIT Lincoln Laboratory for graciously funding my graduate education. Special thanks goes to:

- My group leaders Craig Perini and Steve Forman, for providing critical feedback of my performance as a student and as an engineer.

- My division leaders Eli Niewood and Bob Davis, for supporting my educational and professional development over the years.

- My mentor Andrew Stimac, for offering thoughtful reviews of my publications and this thesis.

Most importantly, my work would be meaningless without the love from my family. I would like to thank my parents, Kathleen and John, and my sister, Dawn, for their optimism and enthusiasm. Lastly, to my beautiful wife Bina, whose unconditional love and support continuously inspires me in all that I do.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The payload for an Unmanned Aerial Vehicle (UAV)[1] is typically considered a separate subsystem from the vehicle itself. Engineers spend much of their resources designing the payload in an effort to maximize its performance. This type of prioritizing can cause the UAV to become an afterthought; the engineers may be tempted to select a preexisting vehicle based on convenience, cost, or familiarity. Ironically, the vehicle can have as much of an impact on system performance as the payload itself. Consider a surveillance application where the payload is a downward facing camera. Increasing the altitude of the camera results in a wider field of view, much like the effect of a zoom lens. In fact, one can argue this maneuver is more effective than "zooming" since it does not change the lens speed, allowing the optics to perform identically at all vehicle configurations. Here the integration of the UAV into the payload forms the basis of a high performance robot.

This thesis concentrates specifically on the quad-rotor flying robot, an element of the rotary wing subclass of UAVs. A collaborating quad-rotor robot team, like the one seen in Figure 1-1, is capable of outperforming the traditional single (usually fixed-wing) UAV by providing multiple payload configuration capabilities. In the surveillance example, the resulting image may need to fully cover a defined environment. With a single UAV, there are limited number of configurations to accomplish

[1] Currently referred to as Unmanned Aircraft Systems (UASs) by the U.S. Department of Defense, who has modified the previous definition of UAVs to provide a working definition of an "unmanned system" [14].

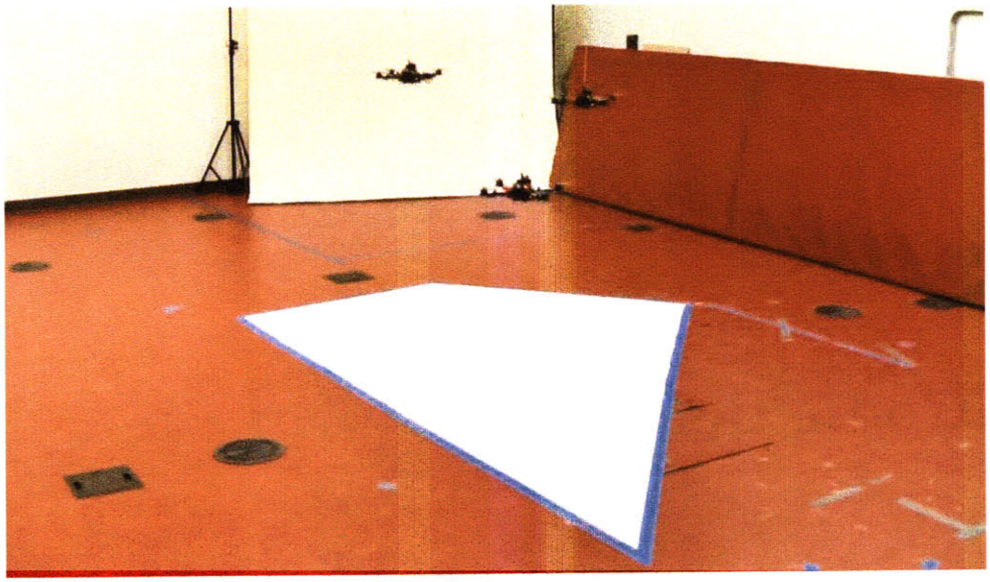


Figure 1-1: This three quad-rotor flying robot system successfully demonstrated our distributed algorithm for optimal coverage with downward facing cameras [52]. Such robot teams will augment current surveillance systems to provide currently unachievable performance.

this task, bounding the overall performance of the system. By having multiple quad-rotor flying robots, we are able to continuously place the cameras in an optimal configuration, even in a dynamic environment. This ability again allows for "zooming," except here we are decreasing the altitude to increase resolution over areas of importance.

The use of multiple payloads also reduces the required performance specifications of the payload itself. An expensive, high-performance payload can be replaced by affordable, commercially available components. In the surveillance example, scientific grade CCDs are often fabricated to produce a high resolution, wide angle camera. Equal or superior performance is achieved by deploying several consumer cameras that are able to be positioned closer to the environment. As a result, total system cost is significantly reduced. Figure 1-2 shows one frame of a video mosaic from an outdoor coverage experiment with three quad-rotor flying robots; we were able to deploy inexpensive video cameras to obtain a high resolution view of the environment.

Such multi-robot systems are inherently robust against robot failures. If failures

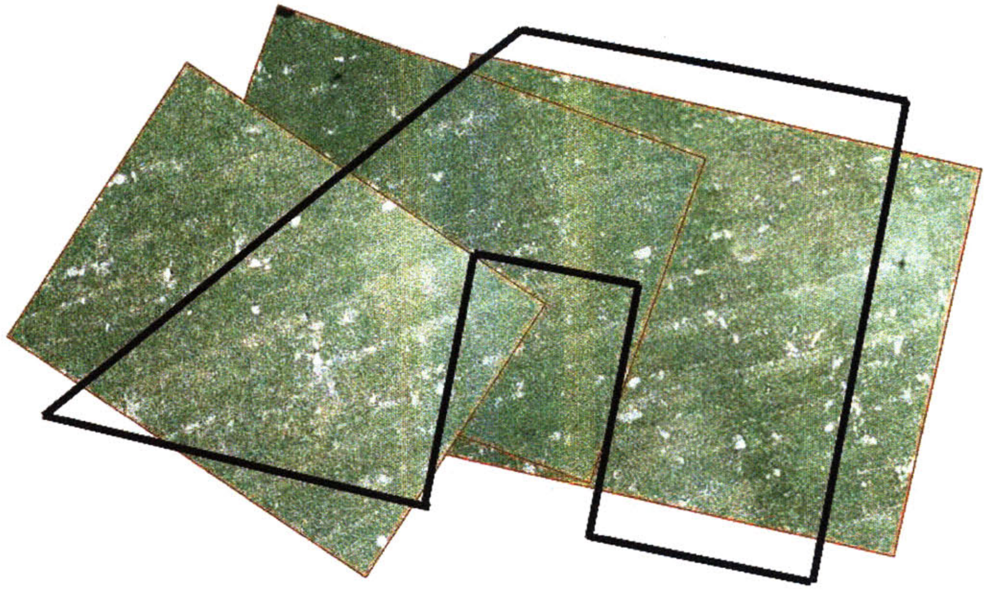


Figure 1-2: The same system from Figure 1-1 was deployed outdoors with off-the-shelf miniature camcorders. By creating a video mosaic, a higher resolution view is obtained of the environment compared to using a single camera.

occur during operation, the system can accommodate for the lost payloads by reconfiguring itself. This situation is catastrophic for a single UAV system; vehicle failure often results in mission failure. By making the system purely decentralized, we can completely avoid dependencies on a central controller, further increasing robustness. In addition, such autonomy allows the team to accomplish missions that are normally considered "out of range" of the ground stations [50]. Thus, we are not only motivated to design an onboard embedded controller for the quad-rotor flying robots, but have it be capable of running algorithms in distributed fashion.

Applications for a distributed quad-rotor robot team are not limited to surveillance. Figure 1-3 shows a geological survey mosaic produced from multiple images taken by one quad-rotor flying robot. Using multiple robots would provide wider tree canopy coverage over the same period of time. They could also be used in a similar fashion to provide valuable information to firefighters during a forest fire. In fact, collaborating quad-rotor flying robots can be used to assist rescuers in many natural disaster situations: deploying motion and acoustic sensors to locate earthquake

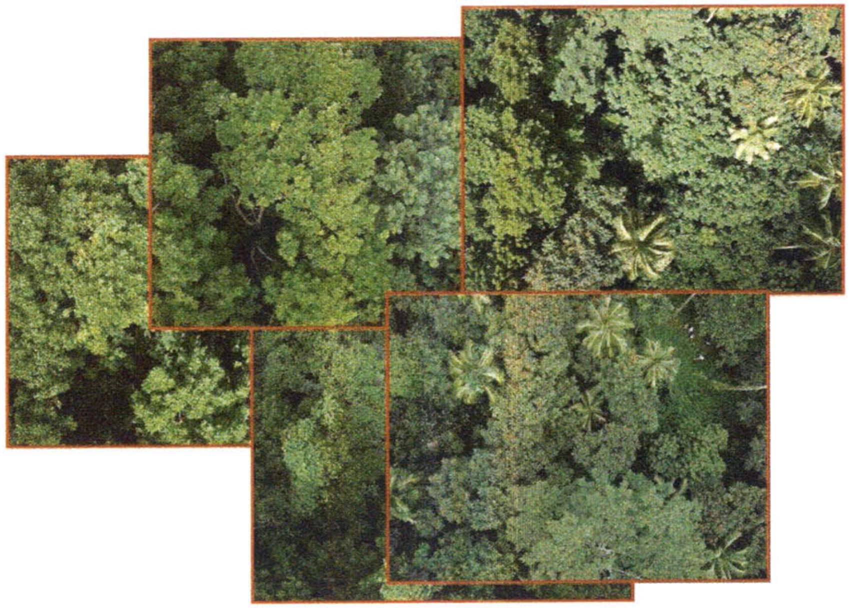


Figure 1-3: A geological survey using a single quad-rotor flying robot was conducted in March 2008 by Jan Stumpf and Daniel Gurdan of Ascending Technologies GmbH (AscTec). Here we see the tree canopy on the island of Moorea, French Polynesia.

victims; distributing food and first aid supplies to stranded flood victims; and monitoring lava flow to maximize time for evacuation during volcanic events. The defense applications are just as plentiful; these systems are expected to perform many of the dull, dirty, and dangerous roles currently performed by the warfighter [14].

We believe research in distributed quad-rotor robotics will have its greatest impact in telecommunications. The ability to act as wireless routers allows these robots to form a three dimensional mobile ad hoc network, like the one in Figure 1-4. When combined with advanced localization techniques [28], this network can extend into urban environments denied of a reliable GPS signal. The ultimate vision of the DoD is for such an aerial robot team to interact with unmanned ground and land systems to extend and augment the capabilities of the warfighter [14].

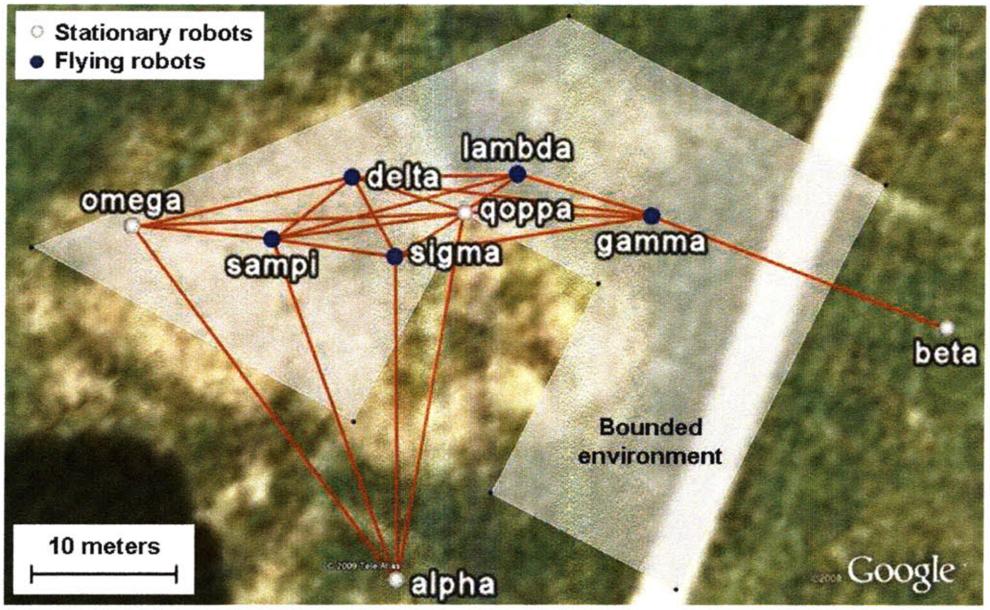


Figure 1-4: An example mobile ad hoc network graph from a quad-rotor flying robot experiment is plotted in Google Earth. This fully connected network consists of five flying and four stationary robots equipped with our embedded controller.

1.1 Contributions to Robotics

For our system of quad-rotor flying robots, we designed an onboard embedded controller capable of running algorithms in distributed fashion. For indoor operation, we use a motion capture system to wirelessly relay position information for low level control. For outdoor operation, a commercially available module allows for GPS, altitude, and compass waypoint control. In addition, we implement a mobile ad hoc network infrastructure to propagate state estimates necessary for the robots to self-organize. This system provides us with a high performance testbed for implementing distributed algorithms designed for a networked team of hovering robots.

In this thesis, we provide the following contributions to the field of robotics:

1. *Embedded Control Module Design* - Hardware and software designs are discussed for the embedded control of a quad-rotor flying robot. The control structure for both indoor and outdoor flight is presented with parameters specific to a commercially available robot platform.

2. *Information Management for Distributed Algorithms* - Embedded software routines are constructed to provide the distributed algorithm with real-time system information. Low level, fast executing functions supporting overall system stability are given priority over high level, slow executing processes.

3. *Location-Based Multi-Hop Strategy* - A deterministic algorithm is presented that efficiently propagates state estimates among a robot team without the need of a routing scheme. The algorithm is shown to be effective in solving distributed control problems while being efficient in terms of communication bandwidth and computational complexity.

4. *Optimal Coverage with Downward Facing Cameras* - The embedded control module is used to optimally position downward facing cameras installed on the quad-rotor flying robots over a bounded environment. Experiments are conducted using both the indoor and outdoor flight configurations.

1.2 Technological Challenges

There were numerous hardware, software, and control challenges we faced during development. The advanced dynamics of the quad-rotor robot inherently makes the control system more complicated than traditional fixed-wing UAVs; the fast time-response of the robot plant that allowed for indoor flight is the same characteristic that sometimes led to unstable behavior. The robots also experienced dynamic coupling between the high level distributed algorithm and low level control, which required careful tuning of feedback parameters and control loop rates. We understand that the final configuration may not be theoretically ideal, but instead results in acceptable performance in the heuristic sense.

There were also many challenges associated with acquiring state estimates. Our urban location at the Massachusetts Institute of Technology (MIT) often resulted in an inaccurate GPS and/or magnetic compass readings. In addition, the pressure sensors used for altitude control experience significant drifts with time, a characteristic that had to be accounted for during experiments. These difficulties with outdoor flight made using the indoor motion capture system appealing, although the limited sensing volume restricted us to a maximum of three simultaneous flying robots.

Once state estimates are acquired, they need to be communicated among the robot team. Here we experienced inconsistencies in connectivity and performance of the radio technology, especially in noisy environments. We do not yet understand how to construct an accurate communication model, which prevents us from considering all possible modes of failure. The distributed algorithms we develop are robust against communication failure, but the system supervisor functions (e.g. collision detection) are not.

1.3 Thesis Organization

This thesis is divided into seven chapters. Chapter 2 presents related work on quad-rotor flying robots, mobile ad hoc networks, and coverage algorithms. Chapter 3 discusses the robot platform and the embedded controller hardware for decentralized control. Chapter 4 introduces a novel location-based algorithm for multi-hopping state estimates with the robot team. Chapter 5 presents a novel distributed algorithm that moves the robot team with downward facing cameras to optimally cover a bounded environment. Chapter 6 discusses the experiments from the system and algorithms we describe in the previous chapters. We conclude in Chapter 7 with final thoughts and lessons learned.

Chapter 2

Related Work

Since Archibald Low's invention of the Aerial Target in 1916 [57], UAVs have been one of the most popular research topics in engineering. By removing the human from the aircraft, a vehicle of a smaller size (and thus smaller signature) can perform assignments with increased survivability, endurance, and maneuvering capabilities [14]. Rotary wing UAVs have distinct advantages over conventional fixed-wing aircrafts since they can takeoff and land in limited spaces and easily perform stationary hovers [6]. Compared with a conventional helicopter with a single large rotor, a quad-rotor flying robot uses much smaller rotors that allow for closer approaches to obstacles without fear of rotor strike [33, 47].

By extending the work of general UAV teams, we believe the distributed quad-rotor platform can significantly increase the performance of current mobile sensor networks in tasks such as surveillance [15], target tracking [10, 13, 39], and team navigating [49]. We are inspired by numerous research platforms designed for the autonomous operation of multiple UAVs. How et al. at MIT have developed a system capable of evaluating coordination and control algorithms for teams of eight fixed-wing autonomous UAVs [36]. Beard et al. at Brigham Young University have constructed an experimental platform for decentralized path following control of small air vehicles in high-wind conditions [9]. This list is far from comprehensive, as research labs at Stanford University [58], University of California Berkeley [54], Vanderbilt University [21], and University of Essix [34] have also developed such systems.

2.1 Quad-Rotor Flying Robots

To date, a significant amount of research on quad-rotor flying robots has focused on the dynamics of a single vehicle. Early work by Hauser et al. studied dynamic models for nonlinear control of a vertical takeoff and landing (VTOL) aircraft [27], from which Shim et al. compared several control designs for autonomous helicopters [55]. In 2002, Pounds et al. and Hamel et al., whom both included Robert Mahony, published dynamic models for the quad-rotor flying robot known as the X4-flyer [26, 48]. The X4-flyer was further modified to produce favorable stability properties in [47]. Recent work in characterizing the flight dynamics of a quad-rotor flying robot has led to several variations of inertial-based controllers [19, 33, 43]. In particular, the high frequency controller proposed by Gurdan et al. will be used for this thesis [24].

A popular extension of autonomous quad-rotor flight is through the use of vision-based control [23]. Altug et al. used a ground-based, two camera pose estimation method to control a quad-rotor flying robot [7]. Soon after Tournier et al. implemented a fixed moiré target to sustain a stable hover with a robot mounted camera [59], while Kemp employed a dynamic programming algorithm to develop a filter capable of vision-based control [38]. Flowers et al. furthered this work by developing an onboard FPGA platform to minimize drift via Harris feature detection and template matching [20].

Other researchers have concentrated on autonomous trajectory tracking and motion planning. Salazar-Cruz et al. implemented a nested saturation control algorithm to track a global trajectory [51]. Bouktir et al. proposed a minimum time trajectory method derived from a nonlinear optimization using algebraic spline functions [11], while Meister et al. characterized numerous collision avoidance strategies based on range sensing capabilities [42]. He et al. navigated a quad-rotor flying robot indoors using an Unscented Kalman Filter extension of the Belief Roadmap algorithm [28]. This work under Nicholas Roy at MIT was the basis of further research efforts to investigate simultaneous localization and mapping (SLAM) using stereo vision and laser odometry [4, 5, 8]

Concerning multi-agent systems, the literature has been somewhat limited for quad-rotor implementation. The Real-time indoor Autonomous Vehicle test ENvironment (RAVEN) developed in the Aerospace Controls Laboratory at MIT studies long-duration multivehicle missions in a controlled environment [35]. The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) developed in the Hybrid Systems Laboratory at Stanford University uses quad-rotor flying robots to autonomously track a given waypoint trajectory [30]. At the time of our work, both RAVEN and STARMAC relied on a centralized computer for multi-agent coordination. Recent work to STARMAC is pursuing onboard optimal control strategies [3], for which Hoffmann et al. have published advances in decentralized collision avoidance [31] and information-seeking guidance [32].

2.2 Mobile Ad Hoc Networks

Concerning our algorithm for mobile ad hoc networks, a substantial body of work exists on location-based routing. Haas proposed a zone-based routing protocol using a radius parameter to reduce the number of control messages [25]. Ni et al. developed a distance-based scheme to decide when a node should drop a rebroadcast [44], while Sun et al. adapted a similar scheme for setting defer times [56]. Ying et al. discussed how these ad hoc schemes influence flooding costs [17].

Our proposed algorithm is related to this body of work in that location is used to broadcast information through a mobile ad hoc network. However, instead of routing actual data packets to a predetermined receiver, we are deterministically transmitting state information to be used by the entire team of robots. This allows all transmissions to be treated as simple broadcasts, for which the sender uses the algorithm to select state estimates. This strategy is applicable for many distributed control problems, such as coverage control algorithms for mobile sensing networks.

2.3 Coverage Controllers

For our work on optimal coverage with downward facing cameras, we are inspired by recent literature concerning the optimal deployment of robots for providing sensor coverage of an environment. Cortes et al. introduced a stable distributed controller for sensor coverage [16] based on ideas from the optimal facility placement literature [18]. This approach involves a Voronoi partition of the environment, and has experienced several extensions [22, 46, 53]. One recent extension described in [41] proposed an algorithm for the placement of hovering sensors, similar to our scenario.

Our proposed method is related to this body of work in that the distributed control algorithm is derived by taking the gradient of a cost function. However, the cost function we propose does not involve a Voronoi partition. Instead, it relies on the fields of view of multiple cameras to overlap with one another. Another distinction from previous works is that the configuration space of the robots is different from the coverage space. This allows for optimal coverage of complex (non-convex, disconnected) environments [52].

Chapter 3

Robot System

The quad-rotor flying robot fleet at DRL[1] is composed of five Ascending Technologies (AscTec) Hummingbirds [2]. Each is equipped with an AscTec AutoPilot module to stabilize the robot in pitch, roll, and yaw. In addition, an onboard GPS module, pressure sensor, and magnetic compass allows for GPS, altitude, and compass waypoint control during outdoor operation. We developed an onboard embedded control module that acquires state estimates from the robot and wirelessly communicates them to other robots using commercially available radio modules. The same microcontroller runs the distributed control algorithm, which allows the robot team to self-organize in both indoor (motion capture system) and outdoor (GPS enabled) environments.

3.1 Quad-Rotor Platform

The AscTec Hummingbird, as shown in Figure 3-1, is a classic four-rotor helicopter with clockwise and counterclockwise rotating propellers located on the forward and side axes, respectively[2]. All propellers are equidistant from the center of the vehicle, providing an under-actuated platform ideal for hovering.

Consider steady-state operation when all four propellers are producing equal thrust totaling the force of gravity on the vehicle. Since the pair of counter rotating

[1]The Distributed Robotics Laboratory at MIT.

[2]The clockwise/counterclockwise pair ordering is irrelevant; the clockwise and counterclockwise rotating propellers could have been located on the side and forward axes, respectively.

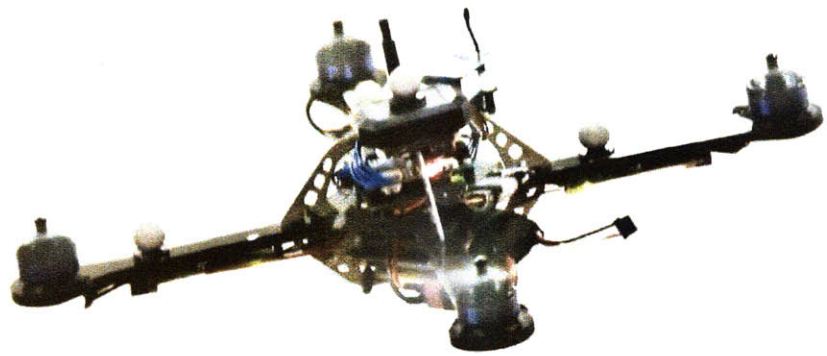


Figure 3-1: This photograph, courtesy of CSAIL photographer Jason Dorfman, shows one of our AscTec Hummingbird robots autonomously flying indoors using a motion capture system.

propellers are spinning at equal speeds, the moment of inertia about the downward-axis is balanced, resulting in a stable hover. Movement about the downward axis is accomplished by increasing or decreasing thrust equally on all propellers. To provide translational forward movement, a differential thrust is produced between the clockwise rotating propellers to tip the robot about the side-axis. Due to propeller symmetry, all gyroscopic effects are balanced, resulting in a stable pitch. Translational side movement is similarly produced using the counterclockwise rotating propellers to roll about the forward-axis. For rotational movement about the downward-axis, a differential thrust is produced between the pair of clockwise and counterclockwise rotating propellers. As long as propellers of like rotation remain at equal speeds and total thrust balances the gravitational force, a stable yaw results.

3.1.1 Dynamic Model

The dynamic model for a quad-rotor flying robot is shown in Figure 3-2. The body frame defines the robot's translational vectors of forward, right, and downward as positive x^b, y^b, and z^b, respectively. The Euler rotational angles of roll, pitch, and

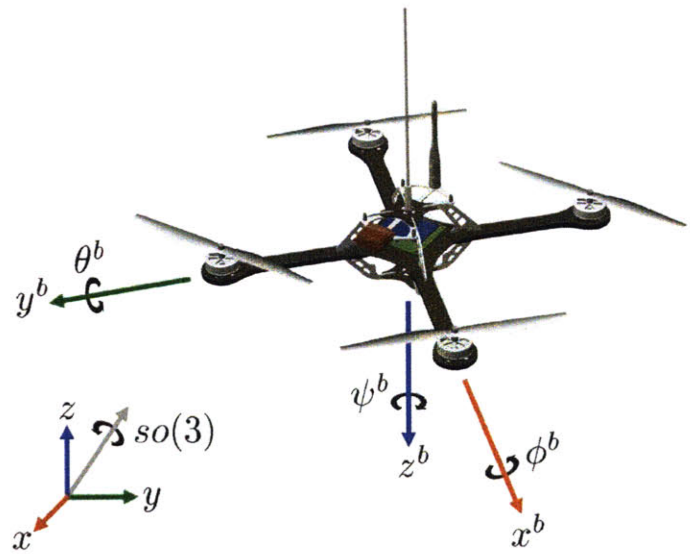


Figure 3-2: The dynamic model shown for a quad-rotor flying robot is used throughout this thesis. The local dynamics of the robot are described using the body frame $(x^b, y^b, z^b, \phi^b, \theta^b, \psi^b)$, while the global orientation is described using the global frame (x, y, z, r^x, r^y, r^z).

yaw are defined as positive ϕ^b, θ^b, and ψ^b, respectively. When describing the local dynamics of a given robot i, the body state is defined as $\vec{p}_i^{\,b} = [x_i^b, y_i^b, z_i^b, \phi_i^b, \theta_i^b, \psi_i^b]^T$ in $\mathbb{R}^3 \times \S^3$. However, the global state $\vec{p}_i = [x_i, y_i, z_i, r_i^x, r_i^y, r_i^z]^T$ in $\mathbb{R}^3 \times \S^3$ is used to locate the robot with respect to an earth-based environment, where r^x, r^y, and r^z are the so(3) angles of the global axes x, y, and z.

For transformations of translational vectors from the global frame to the body frame, we define the rotational matrix

$$\mathbf{R}_b(\vec{p}_i) = \begin{bmatrix} t_c r_i^y r_i^x + r_i^z \sin(\Psi_i) & t_c r_i^y r_i^y + \cos(\Psi_i) & t_c r_i^y r_i^z - r_i^x \sin(\Psi_i) \\ t_c r_i^x r_i^x + \cos(\Psi_i) & t_c r_i^x r_i^y - r_i^z \sin(\Psi_i) & t_c r_i^x r_i^z + r_i^y \sin(\Psi_i) \\ -t_c r_i^z r_i^x + r_i^y \sin(\Psi_i) & -t_c r_i^z r_i^y - r_i^x \sin(\Psi_i) & -t_c r_i^z r_i^z - \cos(\Psi_i) \end{bmatrix} \tag{3.1}$$

where $\Psi_i = \sqrt{r_i^x r_i^x + r_i^y r_i^y + r_i^z r_i^z}$ and $t_c = 1 - \cos(\Psi_i)$. Here, $\mathbf{R}_b(\vec{p}_i)\vec{q}$ transforms the vector $\vec{q} \in \mathbb{R}^3$ from the global frame to the body frame of robot i. In addition, we define the yaw of the robot to be

$$\psi_i = \arctan 2\left(-t_c r_i^y r_i^x - r_i^z \sin(\Psi_i), t_c r_i^x r_i^x + \cos(\Psi_i)\right) \tag{3.2}$$

which aligns with the z-axis of the global coordinate system.

3.1.2 Control for Indoor Operation

The onboard AutoPilot module uses a three-axis accelerometer and three-axis gyroscope to provide attitude stabilization for the quad-rotor flying robot. Let $\vec{\gamma}_i^f = [\ddot{\phi}_i^f, \ddot{\theta}_i^f, \ddot{\psi}_i^f]^T$ in $\mathbb{R}^3$ be the rotational acceleration estimates from the fused inertial data. Three independent proportional-derivative (PD) control loops use the inertial data to update the torque input to all rotors at 1 kHz. Let $\vec{u}_i^b = [u_{\phi_i}^b, u_{\theta_i}^b, u_{\psi_i}^b, u_{z_i}^b]^T$ in $\mathcal{U}_\angle^3 \times \mathcal{U}_z$ be the pitch, roll, yaw, and thrust command inputs to the AutoPilot, respectively, where $\mathcal{U}_\angle$ and $\mathcal{U}_z$ are defined in the AutoPilot technical manuals [1]. The pitch, roll, and yaw command inputs are scaled and summed with both the proportional and derivative terms to provide a "heading-lock" control structure. The thrust input is scaled and summed directly with the torque input to all rotors. This attitude controller, which is based on the work of Gurdan et. al [24], is shown in Figure 3-3.

We conducted autonomous indoor flights using a CSAIL[3] laboratory equipped with a Vicon motion capture system. This system uses 16 high resolution infrared (IR) cameras to triangulate the $\mathbb{R}^3$ global coordinates of IR reflective spherical markers in the sensing environment. As seen in Figure 3-1, these markers are installed on each robot in a predetermined, unique configuration. Careful consideration is taken to avoid Euclidean symmetry among all configurations. By constructing a digital model of each configuration, the motion capture system is able to measure the global state $\vec{p}_i$ of each robot at a rate of 120 Hz. The states are then broadcasted wirelessly at a rate of 50 Hz. The system configuration is shown in Figure 3-4.

For waypoint position control, we designed an embedded control module that runs two different proportional-integral-derivative (PID) control loop configurations. Let

[3]The Computer Science and Artificial Intelligence Laboratory at MIT.

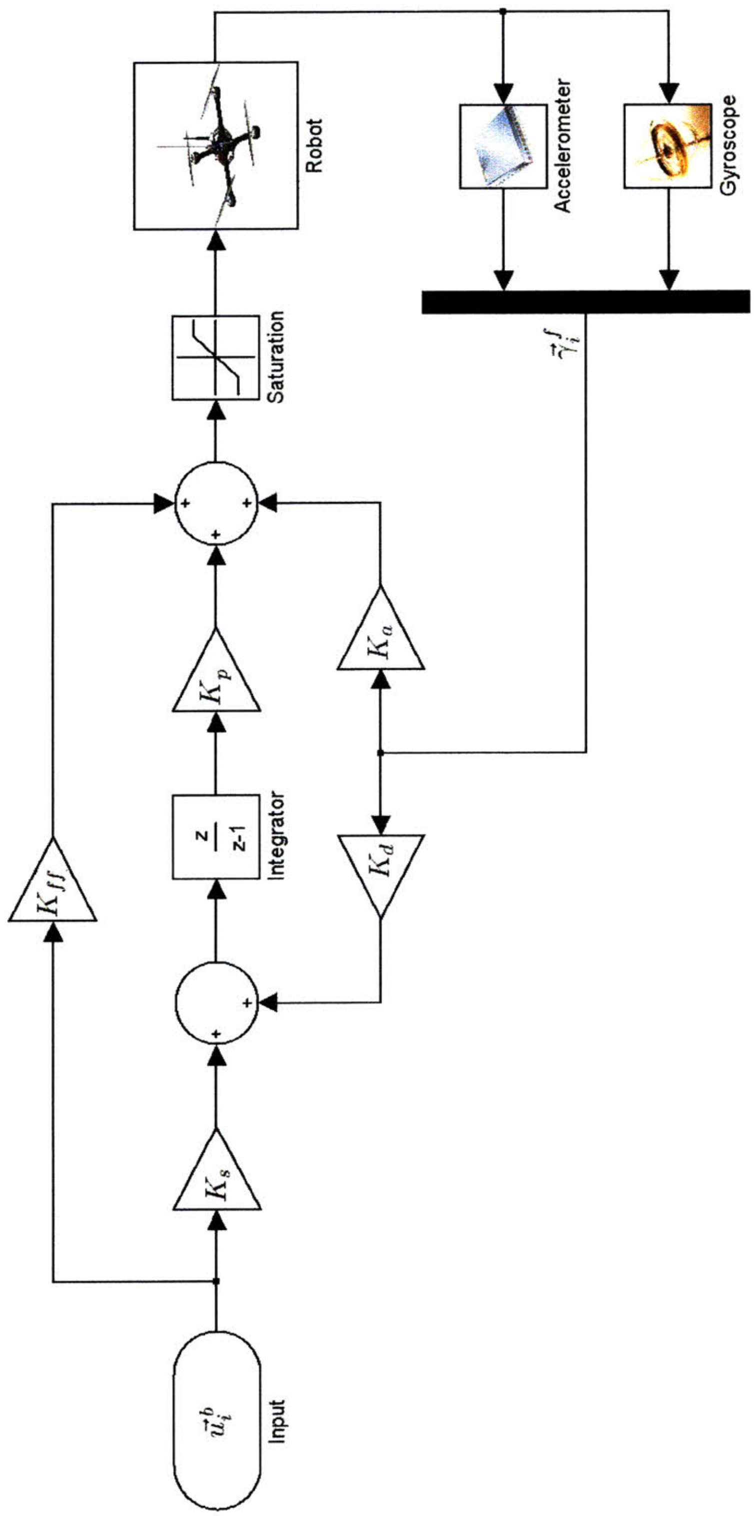


Figure 3-3: The basic PD control structure of the AutoPilot control loops for pitch, roll, and yaw stabilization [24].

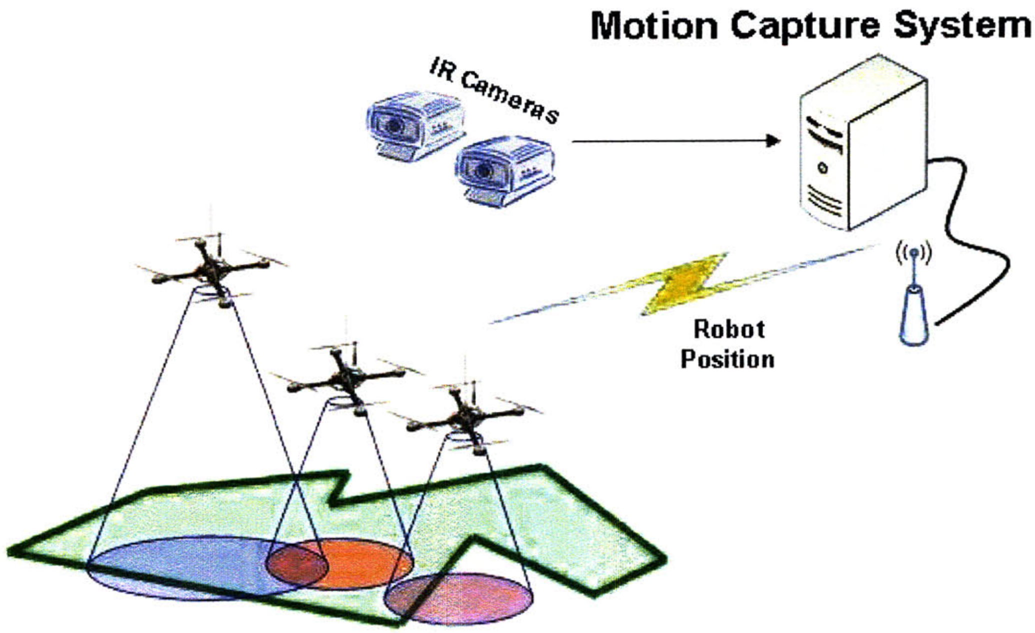


Figure 3-4: A motion capture system measures the position of the robots and broadcasts this information wirelessly to the robot team.

$\vec{w}_i = [x_w, y_w, z_w, \psi_w]^T$ in $\mathbb{R}^3 \times \S$ be the current waypoint of robot i in the global frame. The embedded control module first acquires the broadcasted position data via a radio module and subtracts it from the waypoint to give the global error vector

$$\vec{e}_i = \begin{bmatrix} x_w - x_i \\ y_w - y_i \\ z_w - z_i \\ \psi_w - \psi_i \end{bmatrix} \tag{3.3}$$

where ψ_i is defined in Equation (3.2). The global error vector is then rotated to align with the body frame of robot i to give

$$\vec{e}_i^{\,b} = \begin{bmatrix} \mathbf{R}_b(\vec{p}_i) & 0 \\ 0 & 1 \end{bmatrix} \vec{e}_i \tag{3.4}$$

which feeds into two different PID modules. For yaw and thrust, the standard PID

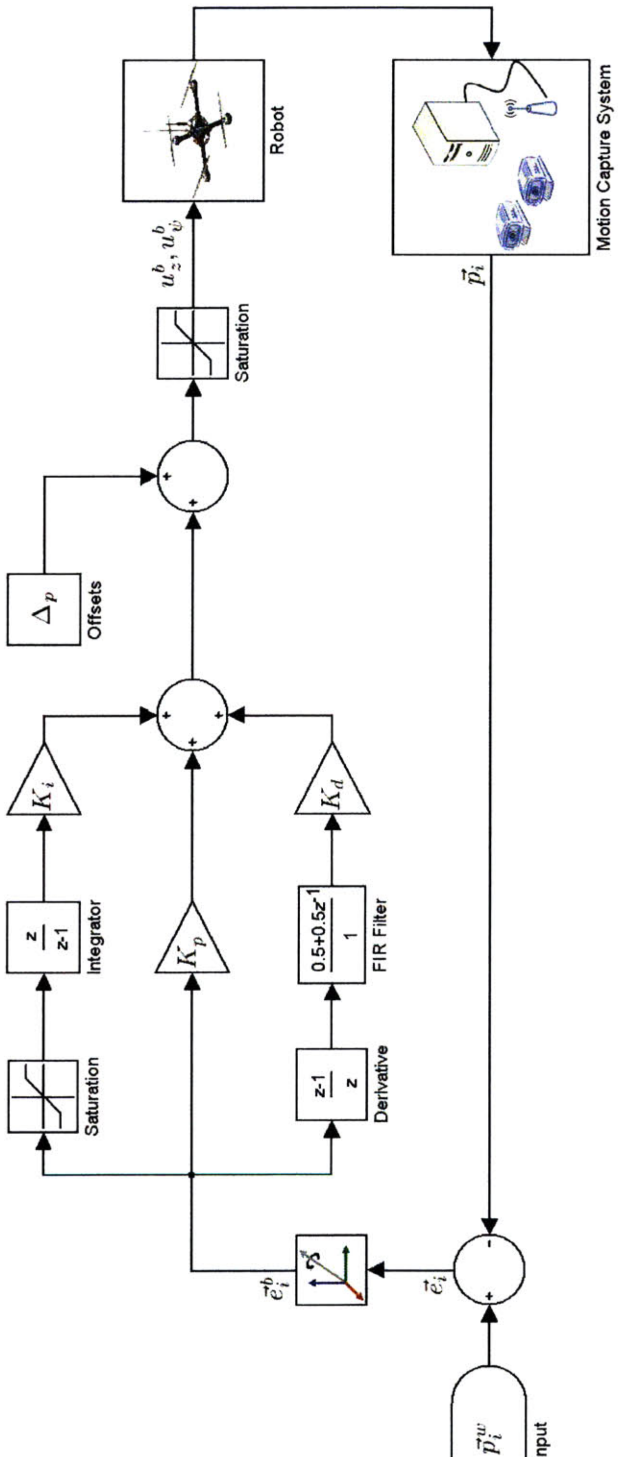


Figure 3-5: The classic PID control structure of the embedded waypoint position control for z_i and ψ_i. All computations outside the motion capture system are done onboard the robot.

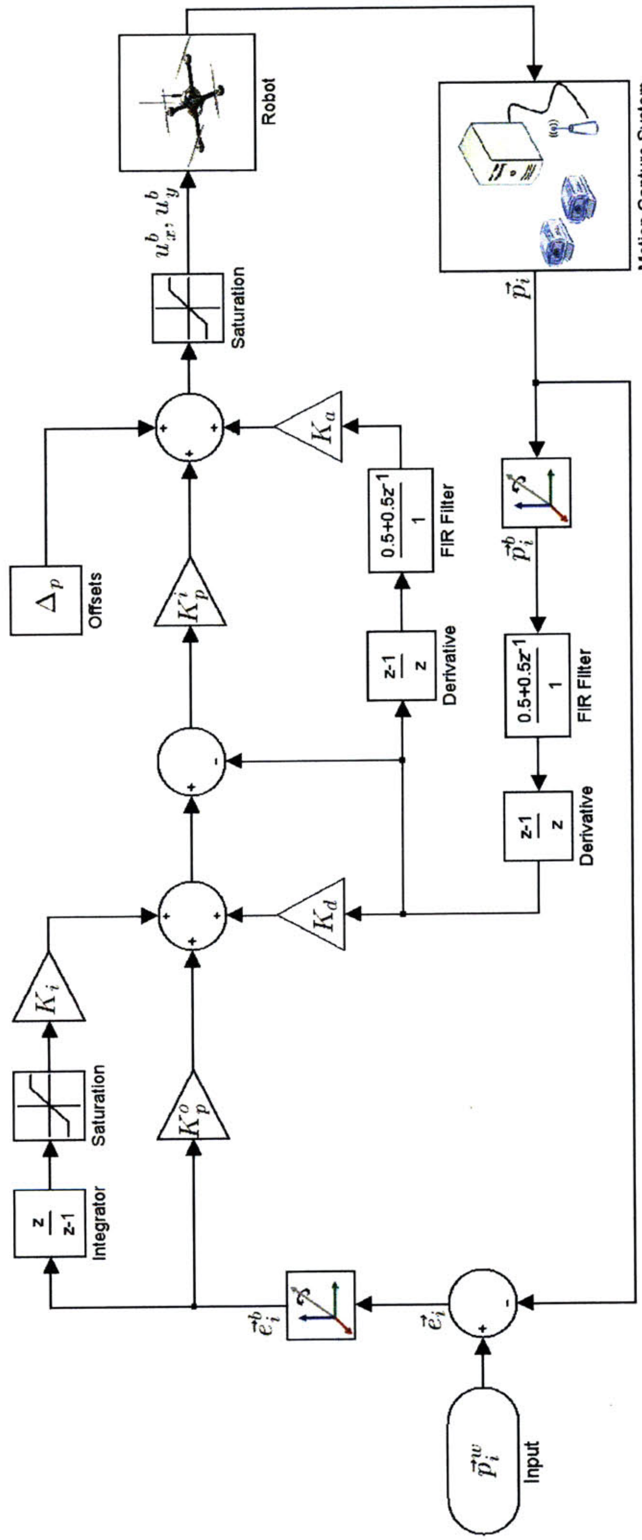


Figure 3-6: The cascaded PID control structure of the embedded waypoint position control for x_i and y_i. All computations outside the motion capture system are done onboard the robot.

control loop shown in Figure 3-5 is used since $[u^b_{\psi_i}, u^b_{z_i}]$ is proportional (although nonlinear) to $[\psi^b_i, z^b_i]$. However, this relationship does not exist between $[u^b_{\phi_i}, u^b_{\theta_i}]$ and $[x^b_i, y^b_i]$. Thus, the cascaded PID control loop shown in Figure 3-6 is used for pitch and roll. Both control loops incorporate low pass filters for the discrete derivative blocks, saturation limits for the discrete integrator blocks, and adjustable output offsets. This embedded waypoint position control system produces command inputs to the AutoPilot module at a rate of 33 Hz.

3.1.3 Control for Outdoor Operation

The AutoPilot module uses a GPS module, pressure sensor, and magnetic compass in combination with the inertial data to provide outdoor waypoint control for the quad-rotor flying robot. To use this off-the-shelf waypoint control, we programmed the embedded control module to provide global waypoint position commands, $\vec{p}_i^{\,w}$, to the AutoPilot module instead of lower level command inputs, $\vec{u}_i^{\,b}$. In addition, the state estimates, $\vec{p}_i$, are polled from the AutoPilot module to be used by our high level distributed algorithm. We designed the waypoint structure to be identical for both indoor and outdoor operation, which allows the high level algorithm to be agnostic to the system's low level control.

3.2 Embedded Control Module

The embedded control module is designed around a single NXP LPC2148 microcontroller [45]. The ARM7TDMI-S microcontroller possesses a 32-bit wide data path, 32 (+8) kB of static random access memory (SRAM), and 512 kB of flash program memory. Figure 3-8 shows a diagram of the microcontroller architecture relevant to the processes described in this thesis. We operate the microcontroller with an external crystal oscillating at 14.7456 MHz. Combined with a internal phase lock loop (PLL) multiplier of $4x$, we achieve a system clock frequency of 58.9824 MHz. The ARM7TDMI-S processor operates at this frequency on the local bus to control the SRAM, flash memory, and fast general purpose input/output (FIO). The vectored

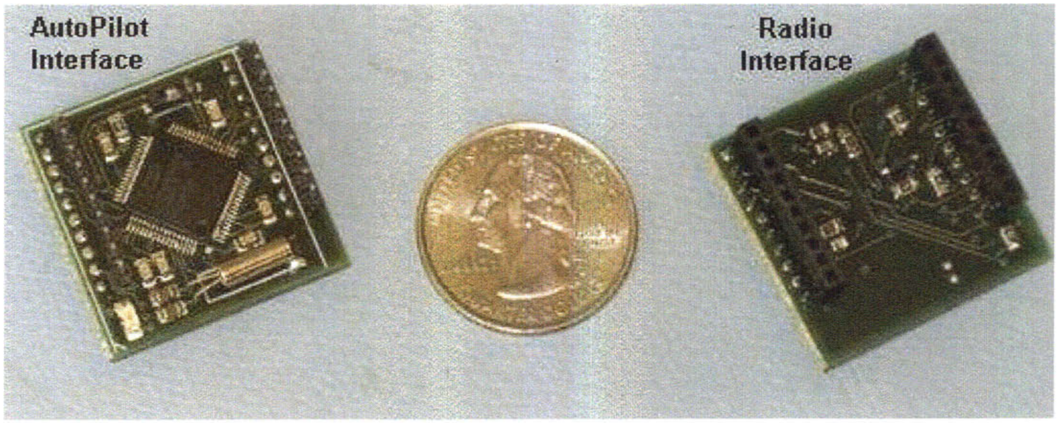


Figure 3-7: The embedded control module shown has adopted the Digi XBee footprint such that the original AutoPilot connector module can be used for power and serial communication.

interrupt controller (VIC) operates on the advanced high-performance bus (AHB) to process interrupt requests during code execution.

Bridging to the AHB is the VLSI peripheral bus (VPB), which connects all peripheral blocks at the system clock frequency implemented by a unity VPB divider. The system control block maintains several low level system features not related to specific peripheral devices (e.g. memory mapping). The two system timers (TMR0 and TMR1) are extensively used in the embedded software to trigger high priority interrupts and control external match pins for status LEDs. Two universal asynchronous receive/transmit (UART0 and UART1) are used to communicate to the AutoPilot and radio modules, respectively. Since UART0 does not support automatic flow control, an external interrupt (EINT) pin and a FIO pin are used for manual implementation. One additional EINT pin is connected to a mechanical switch to allow in-situ changes between indoor and outdoor operation.

3.2.1 Interface to AutoPilot Module

The embedded control module interfaces with the AutoPilot module via six electrical signals: ground (GND), 3.3V power (VD3.3), transmit output (TXD_AP), receive input (RXD_AP), request to send (RTS_AP), and clear to send (CTS_AP). As shown

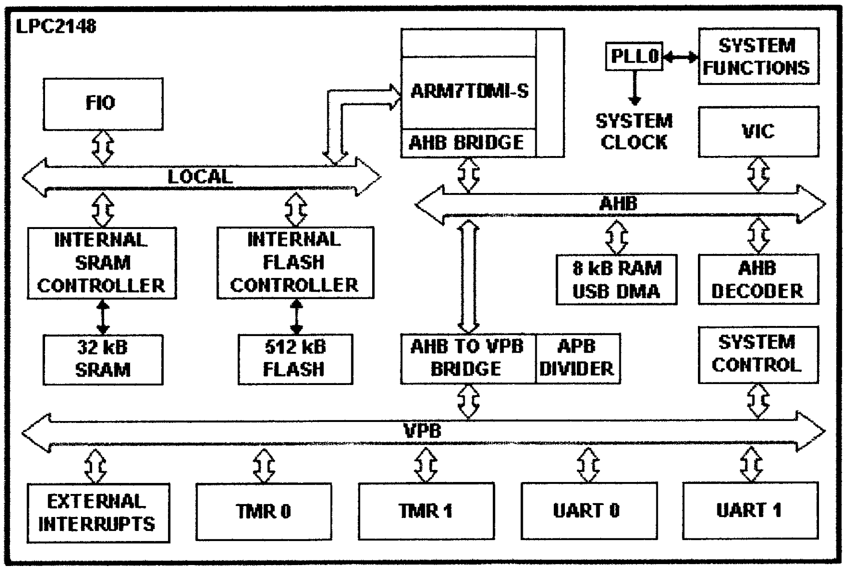


Figure 3-8: An abbreviated diagram of the NXP LPC2148 microcontroller [45]. System blocks not relevant to the operation of the embedded control module are hidden for clarity purposes.

in Figure 3-7, the socket provided by the AscTec connector board has the same footprint as the Digi XBee module in Figure 3-9. The GND signal is the floating level of the negative terminal of the robot's lithium-ion battery. The VD3.3 signal is a filtered voltage produced by a low noise linear regulator located on the AscTec connector board. We use this signal as our power source for the control and radio modules.

TXD_AP and RXD_AP are connected to the UART1 receive and transmit pins, respectively. This communication channel is configured for 8-bit words with no parity and 1 stop bit transmitted at 57.6 kbps. Outgoing information from UART1 is held in its 16-byte transmit buffer until the AutoPilot module pulls RTS_AP low. This signal is connected to the clear to send pin of UART1. UART1 automatically manages flow control, sending data from the transmit buffer over RXD_AP until all data is sent or the AutoPilot module pulls RTS_AP high. Outgoing information from the AutoPilot module is requested by pulling the CTS_AP low with the request to send pin of UART1. UART1 automatically pulls CTS_AP high when the 16-byte receive buffer contains at least 8 bytes. CTS_AP is reasserted as soon as the receive buffer contains 4 bytes or less.

3.2.2 Interface to Radio Module

The control module interfaces with the radio module via eight electrical signals: GND, VD3.3, reset (RST), bootloader (BST), transmit output (TXD_RF), receive input (RXD_RF), request to send (RTS_RF), and clear to send (CTS_RF). As previously stated, power over VD3.3 is drawn from the AscTec connector board. RXD_RF and TXD_RF are connected to UART0 in identical fashion with respect to the AutoPilot module and UART1. However, UART0 does not support automatic flow control. Instead, RTS_RF is connected to an EINT pin, which triggers an interrupt to manually enable or disable the data transfer from UART0. Likewise, CTS_RF is connected to a FIO pin, whose level is set during UART0 interrupts based upon the state of the receive buffer.

RST and BST are only used when programming the microcontroller. Each signal originates from a digital output pin on the radio module. By pulling RST low while BST is low, the microcontroller enters a bootloader state, which allows UART0 to write a binary program file into the flash memory. This configuration allows us to reprogram the microcontroller wirelessly while the control module is installed on the robot.

3.3 Radio Module

As shown in Figure 3-9, we use two different, yet interchangeable, IEEE 802.15.4 radio modules, depending on the environment. For indoor operation, we use a Digi XBee-PRO module that operates in the 2.4 GHz frequency band. Ideally, this module can transmit at 250 kbps with a maximum line-of-sight range of 1.6 km, although in testing we have experienced performance ceilings of typically one-fourth these values. For outdoor operation, we use a Digi XBee-XSC module that operates in the 900 MHz frequency band. Like the XBee-PRO, we underperform the reported specifications of 9.6 kbps and 9.6 km.

Figure 3-9: The Digi XBee-PRO and Digi XBee-XSC radio modules are used for indoor and outdoor operation, respectively.

Although these XBee modules are capable of more complex network topologies, they are used as simple broadcast beacons to be heard by all other modules in communication range. Prior to robot deployment, all robots are programmed to use the same channel. Integrated retries and acknowledgement capabilities are disabled since network flooding schemes do not use the concept of "destination" nodes. This configuration requires manual implementation of channel access methods, which we will discuss in Chapter 4. Our goal is to reduce total communication overhead such that the baud rate is maximized.

Chapter 4

Mobile Ad Hoc Network

Robots in a team need to communicate state estimates to self-organize. Since many applications require the team to spread over large-scale domains, resulting distances between any two robots can become larger than their capable peer-to-peer transmission ranges. These configurations require multi-hop networking to distribute state information over the entire system. To facilitate the transportation of data packets in a multi-hop fashion, many mobile ad hoc networks implement sophisticated routing schemes. Due to the mobile nature of such networks, these schemes consume a significant amount of communication capacity for maintaining knowledge about network topology. While some routing strategies take spatial configurations into account, the robots are agnostic to the relevance of the actual data being transferred. There is no concept of data importance from the robots' point of view, often resulting in the suboptimal allocation of communication resources (e.g. time, bandwidth, power) to transfer packets.

The strategy in this chapter[1] allows robots to better manage communication resources for relaying state estimates. Since the collaboration of robots takes place in the physical world, spatial relationships between robot states can give insight into the importance of transferring each estimate. This location-based approach gives a quantitative answer to the question: how important is it for one robot to broadcast state information about another robot? We represent the importance of transmitting

[1]The majority of this chapter was published in [37]

a state estimate as a function that is inversely proportional to the distance between robots.

From this importance function we develop a deterministic algorithm that ensures state estimates propagate throughout a robot network. The proposed location-based algorithm is efficient in terms of bandwidth and computational complexity; it does not require network topology information to be transmitted or computed. We used Monte Carlo simulations to show increased propagation rates of state estimates in local neighborhoods. Then with real control and wireless hardware, we simulated a nine robot team running a Voronoi coverage controller to show the algorithm's effectiveness in solving distributed control problems. Experimental results for the propagation of state estimates are presented in Chapter 6 for five AscTec Hummingbird quad-rotor flying robots and four stationary robots.

4.1 Importance of Broadcasting State Estimates

A common assumption for distributed control algorithms is that robots have access to state estimates of other nearby robots. This assumption is often translated into unrealistic requirements on communication range. The most common requirement is that estimates need to be directly shared between robots that are within a specified distance. Another common requirement is for information to be shared between robots of a defined spatial relationship (e.g. adjacent Voronoi regions [16] or overlapping fields of view [52]).

These communication requirements are too simplistic to be realized in practice. Actual network topologies depend on more than simple distance criteria, such as environment geometry, channel interference, or atmospheric conditions. Even if transmission ranges are ideal in the physical sense (e.g. the ideal disk model), spatial relationships for certain distributed controllers cannot guarantee peer-to-peer connectivity. Figure 4-1 shows a configuration where a direct communication link cannot be created between the Voronoi neighbors i and j. Moreover, robots that are spatially disconnected may decide not to route state estimates to one another. If they

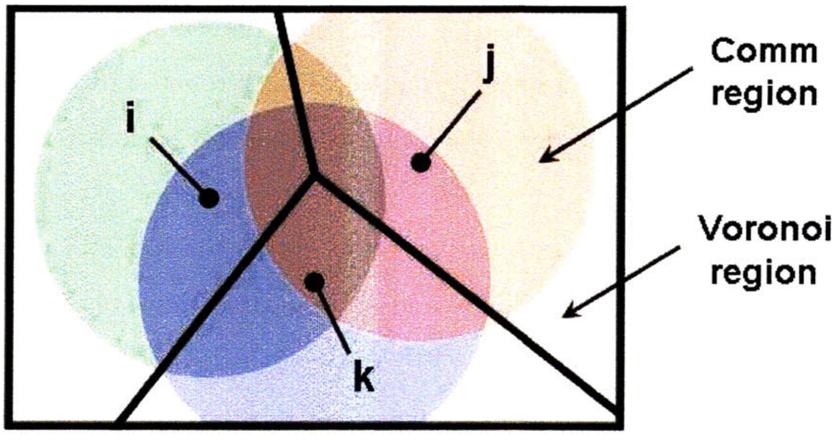


Figure 4-1: A simple example where robots i and j share a Voronoi boundary but cannot communicate their state estimates directly. This problem is easily resolved using a mobile ad-hoc network topology to route information through robot k.

move to become spatially connected, the lack of shared data will prevent the robots from learning about their new neighbors. Thus, no new communication links will be established. We are motivated by these serious and unavoidable complications to develop an algorithm that ensures state estimates flow throughout a team of robots.

4.1.1 Broadcast Scheme

Consider n robots moving in a global space[2], $\mathcal{P}$. Each robot, $i \in \{1, \ldots, n\}$, knows its current state, $\vec{p}_i(t) \in \mathcal{P}$, by some means of measurement (e.g. GPS or visual localization). We propose that each robot maintains a list of state estimates, $[\vec{p}_1(t_{i1}), \ldots, \vec{p}_n(t_{in})]$, where t_{ij} denotes a time stamp at which robot i's estimate of robot j's state was valid. We have that $t_{ij} \leq t$ and $t_{ii} = t$. Each robot's state estimate is initialized to infinity to indicate that a valid estimate is lacking, except for its own state which is always current.

We desire to communicate state estimates throughout the robot network. For simplicity, we use Time Division Multiple Access (TDMA)[3] to divide the data stream

[2] Although it is easiest to think of the space being $\mathbb{R}^2$ or $\mathbb{R}^3$, the strategy we describe is equally useful with more detailed state estimates (e.g. velocity, acceleration, joint positions, state machine information, etc.)

[3] In this chapter we primarily discuss implementing the proposed strategy using TDMA; however, many other channel access methods are appropriate (e.g. FDMA or CDMA).

into time slots of equal length, l. During a time slot, one assigned robot is allowed to broadcast over the shared frequency channel. Other robots broadcast one after the other in a predetermined order. One complete broadcast cycle is referred to as a frame.

To broadcast its own state estimate once per frame, the robot's time slot must be long enough to transmit the estimate and an associated time stamp. Such a time slot is considered to have length of $l = 1$. Clearly time slots of unit length are not sufficient to transmit information throughout the network; each robot would only be updated with the state estimate of its neighbors with direct communication on the network. For multi-hop networking, the robots need longer time slots to broadcast the estimates of other robots.

One naive strategy is to assign a time slot length equal to the number of robots, $l = n$, so that each robot can broadcast its entire list of state estimates, thus creating a simple flooding scheme. Robots that are adjacent on the network use this information to update their own list, retaining only the most current state estimates. The process is repeated for each time slot, naturally propagating state estimates throughout the network without the need of a complicated routing protocol.

Although simple to implement, this strategy is not scalable for a large number of robots. Consider the rate a system can cycle through all time slots to complete one frame. This frame rate, r_f, gives insight into how quickly state estimates are being forwarded, and therefore how confident distributed controllers can be in using the estimates. For a network of fixed baud rate, r_b, the maximum frame rate[4] is given by

$$\max(r_f) = \frac{r_b}{lns} \tag{4.1}$$

where s is the data size of a state estimate and its associated time stamp. For $l = n$, increasing the number of robots in the system will decrease the frame rate *quadratically*. This inherent trade-off provides motivation to reduce the length of the time slot. However, if a robot cannot broadcast *all* state estimates within one time

[4]We are ignoring overhead associated with TDMA (e.g. guard periods, checksums, etc.)

slot, which estimates are considered more important to broadcast?

4.1.2 Importance Function

Many distributed controllers are dependent on spatial relationships between robots. When selecting which state estimate to broadcast, the selection process should also depend on these same spatial relationships. This makes sense because a robot's state is more likely to be useful to controllers in proximity. However, it cannot be considered useless to controllers that are distant due to the mobile nature of the system. We propose that the importance of robot i broadcasting robot j's state estimate is inversely proportional to the distance between robot states.

Since the robots only have access to the state estimates they receive, the following importance function, $g : \mathcal{P} \times \mathcal{P} \mapsto (0, \infty]$, uses a distance estimate to give

$$g_{ij}(t) = d\left(\vec{p}_i(t), \vec{p}_j(t_{ij})\right)^{-\alpha} \tag{4.2}$$

where $d : \mathcal{P} \times \mathcal{P} \mapsto [0, \infty)$ is a distance function and $\alpha \in (0, \infty)$ is a free parameter, both of which are selected for the given distributed controller. For example, a Voronoi coverage controller dependent on linear spatial separation may use a Euclidean distance function with $\alpha = 1$. This same distance function is appropriate for a sensor-based controller dependent on light intensity, although $\alpha = 2$ may be used since light intensity decays quadratically with distance from the source. Conversely, the distance function does not need to be Euclidean or even of continuous topology, such as for truss climbing robots with a finite configuration space. In any case, a robot should consider its own state estimate to be the most important to broadcast. This is reflected in the model since g_{ii} is infinite for any valid $d(\cdot, \cdot)$ and α.

4.2 Location-Based Algorithm for Broadcasts

We use the importance function in Equation (4.2) to develop a deterministic algorithm. For a given time slot, this algorithm selects which state estimates a robot will broadcast. We first describe a probabilistic approach to help formulate the final algorithm.

4.2.1 Probabilistic Approach

Consider a robot that needs to select l state estimates to broadcast during its time slot. We provided motivation in Section 4.1.2 that some selections are more important than others. However, the robot should *not* systematically select the state estimates associated with the highest importance; doing so can prevent estimates from fully dispersing throughout the system. Instead, we propose that the probability of robot i selecting the state estimate of robot j is

$$P^{ij}_{\mathcal{M}_i}(t) = \frac{g_{ij}(t)}{\sum_{k \in \mathcal{M}_i} g_{ik}(t)}, \quad j \in \mathcal{M}_i \tag{4.3}$$

where $\mathcal{M}_i$ is the set of robot indices associated with selectable estimates.

Prior to the first selection for a given time slot, $\mathcal{M}_i$ is the set of all robot indices. From the full set the robot always selects its own state since it has infinite importance. The robot then removes its index from $\mathcal{M}_i$ to prevent wasting bandwidth. Since Equation (4.3) is a valid probability mass function, the robot can simply choose the next state estimate at random from the corresponding probability distribution, then remove the corresponding index from $\mathcal{M}_i$. This means estimates of closer robots are more likely to be chosen than ones that are farther away. By repeating this process, the entire time slot of length l can be filled in a straightforward, probabilistic manner.

4.2.2 Deterministically Selecting Estimates

It is not ideal in practice to probabilistically select which state estimates to broadcast. Consecutive selections of a particular robot index can be separated by an undesirably long period of time, especially concerning distant robots. By developing a location-based deterministic algorithm, we can increase the average rate at which all state estimates of a given time stamp will propagate throughout a team. In the deterministic case, propagation time is bounded above by the longest path taken among the

Algorithm 1 Deterministic Method for Selecting State Estimates

n is the number of robots in the system and l is the time slot length.

Require: Robot i knows its state $\vec{p}_i(t)$ and the state estimate of other robots $\vec{p}_j(t_{ij})$.

Require: Robot i knows its running counter $[c_{i1}, \ldots, c_{in}]$.

$\mathcal{M}_i \leftarrow \{1, \ldots, n\}$

for 1 to l **do**

$P^{ij}_{\mathcal{M}_i}(t) \leftarrow \frac{g_{ij}(t)}{\sum_{k \in \mathcal{M}_i} g_{ik}(t)}, \ \forall j \in \mathcal{M}_i$

$c_{ij} \leftarrow c_{ij}[1 - P^{ij}_{\mathcal{M}_i}(t)], \ \forall j \in \mathcal{M}_i$

$k \leftarrow \arg\max_{k \in \mathcal{M}_i}(c_{ik})$

$\mathcal{M}_i \leftarrow \mathcal{M}_i \backslash \{k\}$

$c_{ik} \leftarrow 1$

end for

return $\{1, \ldots, n\} \backslash \mathcal{M}_i$

estimates. No such bound exists in the probabilistic case, resulting in a positively skewed distribution of propagation times and a larger mean.

We propose that each robot maintains a list of counters, $[c_{i1}, \ldots, c_{in}]$, which are initially set to a value of one. Using the probability mass function in Equation (4.3), each counter represents the probability that the corresponding index has *not* been selected. Consider a robot's first selection, which will always be its own index. The probability, $P^{ii}_{\mathcal{M}_i}(t)$, of selecting index i is equal to one, while all other probabilities, $P^{ij}_{\mathcal{M}_i}(t)$ subject to $j \neq i$, are equal to zero. This implies that the counter c_{ii} is multiplied by $[1 - P^{ii}_{\mathcal{M}_i}(t)] = 0$, or a zero probability of not being selected, while all other counters, c_{ij}, are multiplied $[1 - P^{ij}_{\mathcal{M}_i}(t)] = 1$, or a probability of one. By selecting the index with the lowest counter value, we are deterministically guiding our method to behave according to the probability distribution described by Equation (4.3). The selected index (in this case i) is removed from the set $\mathcal{M}_i$, and its corresponding counter (c_{ii}) is reset to a value of one. This process is iteratively applied to completely fill a time slot with l state estimates, with counters maintaining their values between frames. The complete deterministic strategy is given in Algorithm 1.

Proposition 1 (Deterministic Strategy Time Complexity) *The time complexity of Algorithm 1 is*

$$T(n, l) = O(nl) \tag{4.4}$$

where n is the number of robots and l is the time slot length.

Proof 1 *Assuming $l < n$, calculating (4.3) for all $j \in \mathcal{M}_i$ takes $O(n)$ time. Since all other loop operations are less than or equal to $O(n)$ time, Algorithm 1 is of $O(nl)$ time.*

Remark 1 *The time complexity of Algorithm 1 is bounded above by $O(n \log(n))$ time. This case is for simple flooding when $l = n$, resulting in loop operations of less than or equal to $O(\log(n))$ time.*

4.3 Network Simulations

We provide insight into the performance of the location-based algorithm in two ways: we conducted Monte Carlo simulations for 100 stationary robots, and we used the embedded control module to simulate nine robots running a distributed coverage algorithm. We first describe the Monte Carlo simulations used to measure information propagation throughout the robot team. Propagation time is the main performance metric for the algorithm. This metric depends on the length of the time slot, or in other words, the number of state estimates communicated during one robot broadcast. We compare these results to the case when the time slot length equals the number of robots, since allowing robots to broadcast every state estimate is the simplest multi-hop scheme. This scheme is referred to as simple flooding.

4.3.1 Monte Carlo Simulations

In a MATLAB environment, we simulated a team of 100 stationary robots arranged in a 10×10 square grid. Each robot, initialized knowing only its own state estimate, was able to receive broadcasts from its adjacent neighbors along the vertical and horizontal directions. Each robot ran Algorithm 1 in distributed fashion. Over 1000 Monte Carlo simulations were executed for time slots of varying lengths, with each run having a random order for the time slot assignments. For the 2×2, 4×4, 6×6,

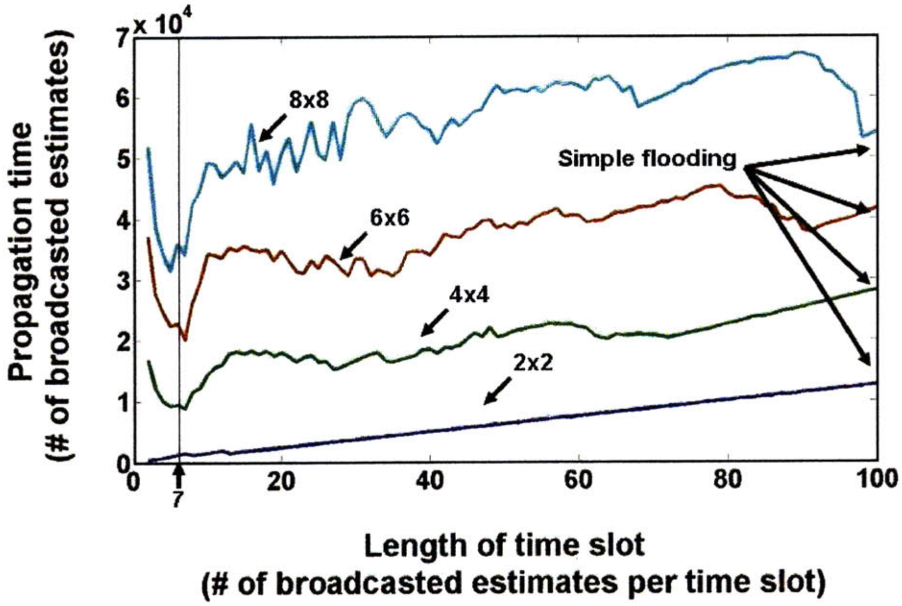


Figure 4-2: This figure shows the average propagation time for the location-based algorithm running on a 10×10 stationary robot grid. Averages were taken over 1000 Monte Carlo simulations. For small subgraphs (i.e. 2×2), update rates of state estimates increased with decreasing time slot lengths. For larger subgraphs, the optimal length was around $m = 7$.

and 8×8 subgraphs centered on the 10×10 graph, we measured the time it took for all subgraph members to exchange state estimates.

Figure 4-2 plots average propagation time for the Monte Carlo simulations. For the smallest subgraph (i.e. 2×2), state estimates propagated faster with smaller time slot lengths. This relationship makes sense since we are maximizing the frame rate, thus increasing update rates for the local state estimates of highest importance. As the subgraph size increases, very small time slot lengths become less effective at propagating estimates, especially between robots at opposite sides of the subgraph. By using a slightly larger time slot length, a significant improvement in performance over simple flooding is obtained; propagation times for all subgraphs decreased by more than 47% using a time slot length of $m = 7$. Analyzing such Monte Carlo plots provides a heuristic technique for selecting an acceptable time slot length for a given control problem.

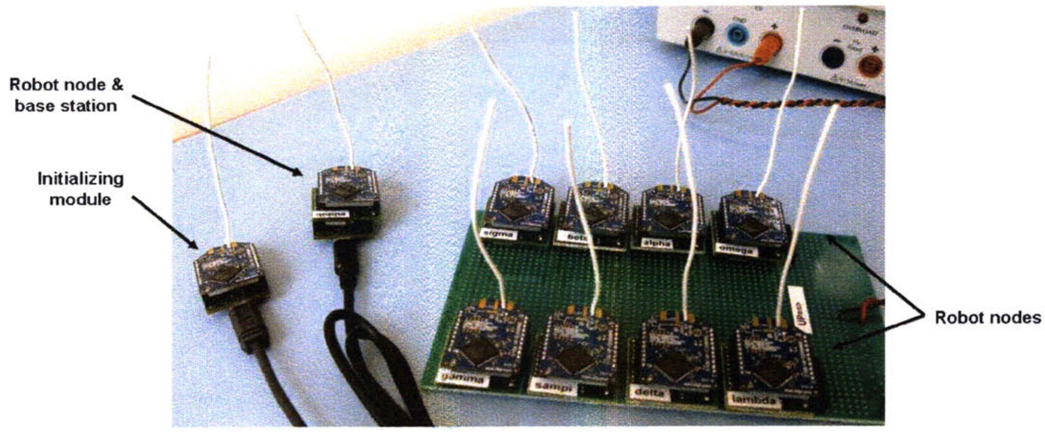


Figure 4-3: A testbed for simulating distributed algorithms using the embedded control module and XBee XSC. Each module locally ran a Voronoi coverage controller and wirelessly communicated state information to other modules in the virtual robot team. A communication range was implemented to add complexity to the wireless network.

4.3.2 Voronoi Simulations on the Embedded Control Module

We then tested the algorithm in a simulated robot scenario using the embedded control modules as shown in Figure 4-3. We implemented a Voronoi coverage controller [16] on nine modules, each using a 900 MHz XBee XSC module to wirelessly broadcast state estimates during its assigned time slot. Each control module simulated the dynamics of a flying robot, creating a virtual distributed robot team. In addition, a communication range was implemented such that packets from "out-of-range" robots were automatically dropped. We investigate the performance of the location-based algorithm in a simple scenario where nine virtual robots were tasked to cover a square area. For this scenario the optimal configuration is for the robots to be arranged in a 3×3 square grid, thus minimizing the total cost of the Voronoi cost function.

For the location-based algorithm, a time slot length of $l = 3$ was selected using the Monte Carlo technique previously discussed. We also selected the Euclidean distance function with $\alpha = 1$ given that the Voronoi coverage controller is linearly dependent on such distance. Each state estimate for the virtual flying robot is constructed of six 32-bit integers (robot identification, time stamp, latitude, longitude, altitude,

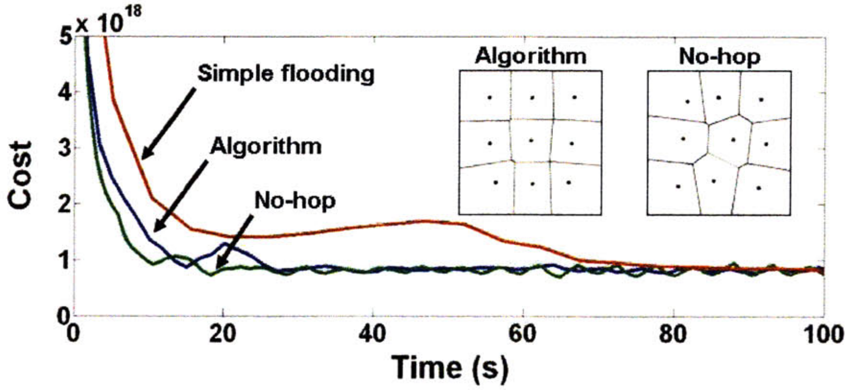


Figure 4-4: Coverage costs are shown for a nine robot system simulated on the embedded control module running a Voronoi coverage controller. The system has a frame rate of 1.7 Hz when using a no-hop scheme ($l = 1$). The system initially performs well, but its inability to multi-hop state estimates resulted in a suboptimal final configuration that does not minimize the Voronoi cost function. A simple flooding scheme ($l = 9$) improved steady state performance, however, the slow frame rate of 0.2 Hz caused the system to initially oscillate in a high cost configuration. The location-based algorithm with a time slot of length $l = 3$ performed the best overall by combining fast update rates with multi-hop capabilities. The final Voronoi configurations for the algorithm and no-hop simulations are also shown.

and yaw), resulting in a data size, s, of 192 bits. Given that the wireless hardware could reliably operate at 3000 baud, the resulting frame rate was about 0.6 Hz. For comparison, the simple flooding ($l = 9$) and no-hop ($l = 1$) schemes ran at about 0.2 Hz and 1.7 Hz, respectively. Figure 4-4 shows the resulting coverage cost profiles from these simulations. The location-based algorithm had better initial performance than the simple flooding scheme and better steady state performance than the no-hop scheme. The final Voronoi configurations for the algorithm and no-hop simulations are also shown.

Chapter 5

Optimal Coverage with Downward Facing Cameras

As discussed in Chapter 1, there are many applications for a team of collaborating quad-rotor robots. In this chapter[1], we introduce a novel algorithm that optimally covers a bounded environment using a team of quad-rotor flying robots with downward facing cameras.

We are motivated by a basic information content principle for digital cameras: minimal information per pixel. Using this metric allows for the incorporation of physical, geometric, and optical parameters to give a cost function that represents how well multiple cameras cover a given environment. For our application, we simplify the coverage problem in that the cameras are mounted to the robots in a downward facing orientation. The simplified cost function leads to a gradient-based distributed control algorithm that positions the robots into an optimal configuration.

5.1 Optimal Camera Placement

We desire to cover a bounded environment, $Q \subset \mathbb{R}^2$, with the fields of view from a number of cameras. We assume Q is planar, without topography, to avoid the complications of changing elevation or occlusions. Let $\vec{p}_i^c \in \mathcal{P}_c$ represent the state of

[1]The majority of this chapter was published in [52]

a camera, where the state-space, $\mathcal{P}_c$, will be characterized later in Section 5.1.1. Each camera is mounted to a robot moving in the global configuration space, $\mathcal{P}$. We want to control n robots in distributed fashion such that their camera placement minimizes the aggregate information per camera pixel over the environment,

$$\min_{(\vec{p}_1^c,\ldots,\vec{p}_n^c)\in\mathcal{P}_c^n} \int_Q \frac{\text{info}}{\text{pixel}}\, d\vec{q}. \tag{5.1}$$

This metric makes sense because the pixel is the fundamental information capturing unit of the camera. Consider the patch of image that is exposed to a given pixel. The information in that patch is reduced by the camera to a low-dimensional representation (i.e. mean color and brightness over the patch). Therefore, the less information content the image patch contains, the less information will be lost in its low-dimensional representation by the pixel. Furthermore, we want to minimize the accumulated information loss due to pixelation over the whole environment Q, hence the integral.

5.1.1 Single Camera

Information per pixel can be represented as the product of two functions, $f : \mathcal{P}_c \times Q \mapsto (0, \infty]$, which gives the area in the environment seen by one pixel, and $\Phi : Q \mapsto (0, \infty)$ which gives the information per area in the environment. The form of $f(\vec{p}_i^c, \vec{q})$ will be derived from the optics of the camera and geometry of the environment. As shown in Figure 5-1, the function $\Phi(\vec{q})$ is a positive weighting of importance over Q and should be specified beforehand (it can also be learned from sensor data, as in [53]). For instance, if all points in the environment are equally important, $\Phi(\vec{q})$ should be constant over Q. If some known area in Q requires more resolution, the value of $\Phi(\vec{q})$ should be larger in that area. This gives the cost function

$$\min_{\vec{p}_i^c\in\mathcal{P}_c} \int_Q f(\vec{p}_i^c, \vec{q})\Phi(\vec{q})\, d\vec{q}, \tag{5.2}$$

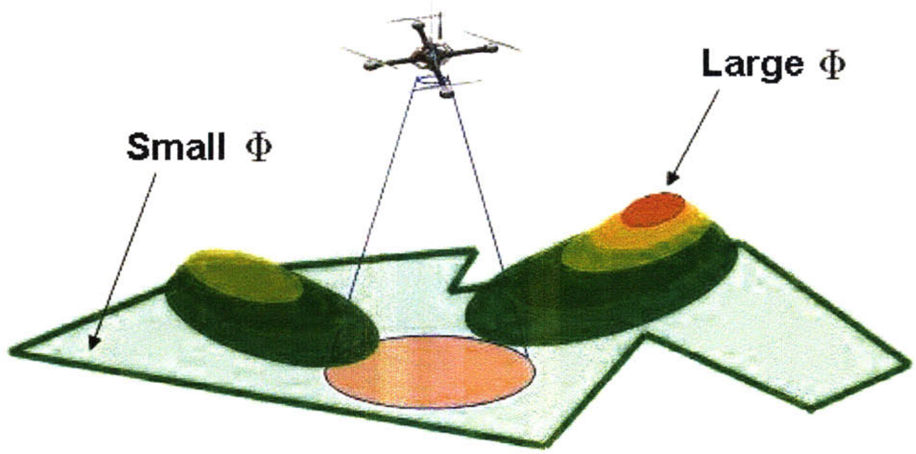


Figure 5-1: An example importance profile is shown for this environment. An larger Φ implies that a particular area is of more importance. This figure also illustrates how changes in surface elevations can be represented even though the environment is in $\mathbb{R}^2$.

which is of a general form common in the locational optimization and optimal sensor deployment literature [12, 18].

The state of a camera, $\vec{p}_i^c$, consists of all parameters associated with the camera that effect the area per pixel function, $f(\vec{p}_i^c, \vec{q})$. We consider the special case in which the camera is mounted to the robot in a downward facing orientation. The resulting relationship between the camera and robot state is $\vec{p}_i^c = [x_i, y_i, z_i]^T$, giving a camera state space $\mathcal{P}^c = \mathbb{R}^3$. Here, z_i is the height of the camera above the environment, Q, and $\vec{v}_i = [x_i, y_i]^T$ is the center point of the field of view, $\mathcal{B}_i$. For now, we define a circular field of view to be the intersection of the cone whose vertex is the focal point of the camera lens with the subspace that contains the environment, as shown in Figure 5-2. We have

$$\mathcal{B}_i = \left\{ \vec{q} : \frac{\|\vec{q} - \vec{v}_i\|}{z_i} \leq \tan(\beta^i) \right\} \tag{5.3}$$

where β^i is the half-angle of view of the camera.

To find the area per pixel function, $f(\vec{p}_i^c, \vec{q})$, consider the geometry in Figure 5-2. Let b be the focal length of the lens and $b^2/(b - z_i)^2$ be the area magnification factor

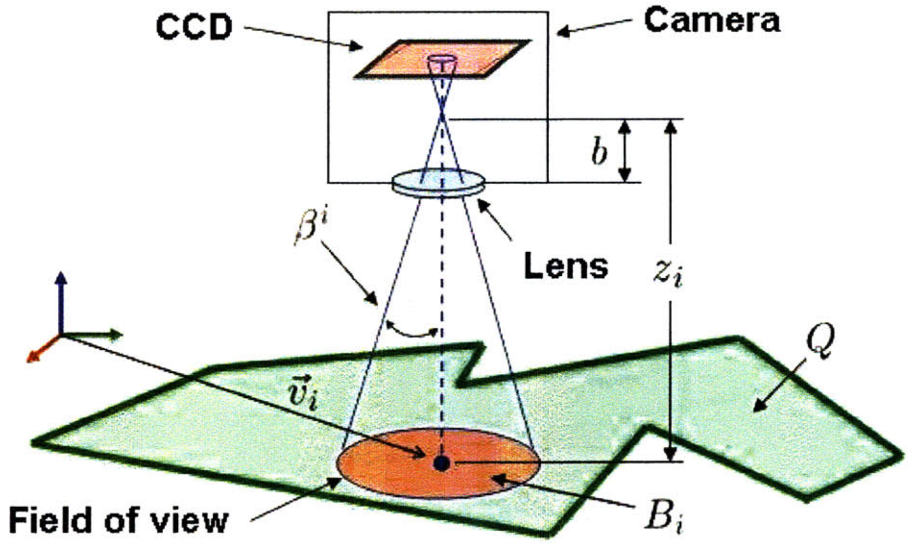


Figure 5-2: This figure shows optical and geometric parameters used in the coverage algorithm. For now, we are assuming a circular field of view.

as defined from classical optics [29]. Inside $\mathcal{B}_i$, the area/pixel is equal to the inverse of the area magnification factor times the area of one pixel. Define a to be the area of one pixel divided by the square of the focal length of the lens. We have,

$$f(\vec{p}_i^c, \vec{q}) = \begin{cases} a(b - z_i)^2 & \text{for} \quad \vec{q} \in \mathcal{B}_i \\ \infty & \text{otherwise,} \end{cases} \tag{5.4}$$

Outside of the field of view, there are no pixels, therefore the area per pixel is infinite (we will avoid dealing with infinite quantities in the multi-camera case). The optimal solution in this simple scenario is for $\vec{p}_i^c$ to be such that the field of view is the smallest ball that contains Q. However, with multiple cameras, the problem becomes more challenging.

5.1.2 Multiple Cameras

Consider a point $\vec{q}$ that appears in the image of n different cameras, such as in Figure 5-3. The number of pixels per area at that point is the sum of the pixels per area for each camera. If we assume all cameras are identical and thus use the same function

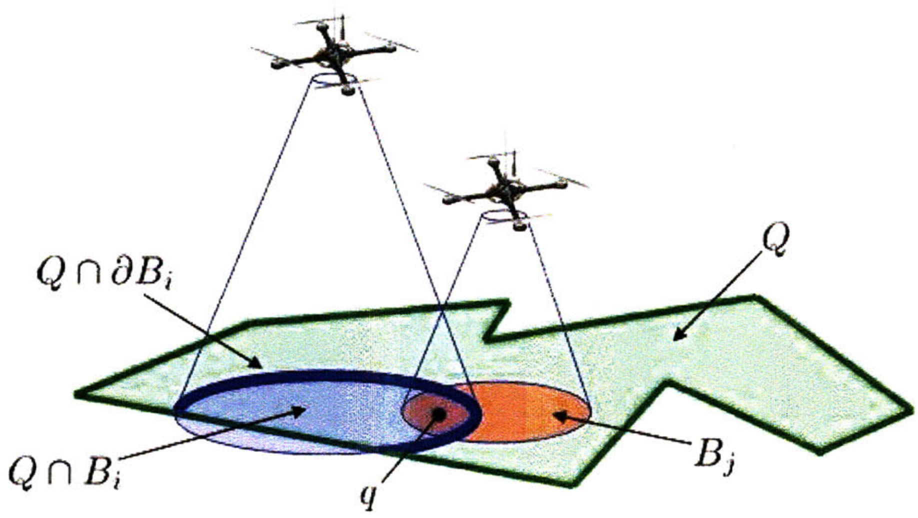


Figure 5-3: An environment can have overlapping fields of view from two or more cameras. The relevant quantities involved in characterizing this scenario are defined.

$f(\vec{p}_i^c, \vec{q})$, the area per pixel at point $\vec{q}$ is given by the *inverse* of the sum of the *inverse* of the area per pixel for each camera, or

$$\frac{\text{area}}{\text{pixel}} = \left(\sum_{i=1}^{n} f(\vec{p}_i^c, \vec{q})^{-1} \right)^{-1}, \tag{5.5}$$

where $\vec{p}_i^c$ is again the state of the ith camera. We emphasize that it is the *pixels per area* that sum because of the multiple cameras, not the *area per pixel*. In the overlap region, multiple pixels are observing the same area, resulting in the inverse of the sum of inverses. Incidentally, this is similar to sensor fusion when one combines the variances of multiple noisy measurements.

Finally, we introduce a prior area per pixel, $w \in (0, \infty)$. The interpretation of the prior is that there is some pre-existing photograph of the environment (e.g. an initial reconnaissance photograph) from which we can get a base-line area per pixel measurement. This pre-existing information can be arbitrarily vague such that w can be arbitrarily large, but this prior must exist. The prior also has the benefit of making the cost function finite for all robot positions, being combined with the camera sensors to get

$$\frac{\text{area}}{\text{pixel}} = \left(\sum_{i=1}^{n} f(\vec{p}_i^c, \vec{q})^{-1} + w^{-1} \right)^{-1}, \tag{5.6}$$

Let $\mathcal{N}_{\vec{q}} = \{i : \vec{q} \in \mathcal{B}_i\}$ be the set of indices of cameras for which $f(\vec{p}_i^c, \vec{q})$ is bounded. We can now write the area per pixel function as

$$h_{\mathcal{N}_{\vec{q}}}(\vec{p}_1^c, \ldots, \vec{p}_n^c, \vec{q}) = \left(\sum_{i \in \mathcal{N}_{\vec{q}}} f(\vec{p}_i^c, \vec{q})^{-1} + w^{-1} \right)^{-1}. \tag{5.7}$$

to give the cost function

$$\mathcal{H}(\vec{p}_1^c, \ldots, \vec{p}_n^c) = \int_Q h_{\mathcal{N}_{\vec{q}}}(\vec{p}_1^c, \ldots, \vec{p}_n^c, \vec{q}) \Phi(\vec{q})\, d\vec{q}. \tag{5.8}$$

which is valid for any area per pixel function $f(\vec{p}_i^c, \vec{q})$, and for any camera state space $\mathcal{P}^c$.

5.2 Distributed Control

We introduce the multi-camera optimization problem

$$\min_{(\vec{p}_1^c, \ldots, \vec{p}_n^c) \in \mathcal{P}^n} \mathcal{H}. \tag{5.9}$$

to minimize the total cost of the robot system. By taking the gradient of the cost function $\mathcal{H}(\vec{p}_1^c, \ldots, \vec{p}_n^c)$ with respect to a robot's position $\vec{p}_i^c$, we have

$$\frac{\partial \mathcal{H}}{\partial \vec{v}_i} = \int_{Q \cap \partial \mathcal{B}_i} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \frac{(\vec{q} - \vec{v}_i)}{\|q - \vec{v}_i\|} \Phi(\vec{q})\, d\vec{q}, \tag{5.10}$$

and

$$\frac{\partial \mathcal{H}}{\partial z_i} = \int_{Q \cap \partial \mathcal{B}_i} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \Phi(\vec{q}) \tan(\beta^i)\, d\vec{q} - \int_{Q \cap \mathcal{B}_i} \frac{2h_{\mathcal{N}_{\vec{q}}}^2}{a(b - z_i)^3} \Phi(\vec{q})\, d\vec{q}. \tag{5.11}$$

where $\mathcal{N}_{\vec{q}} \backslash \{i\}$ is the set of all indices in $\mathcal{N}_{\vec{q}}$ except for i. Refer to [52] for proof and remarks.

5.2.1 Control Law

In addition to knowledge about the environment, a robot calculating the gradient of its cost function needs state information about itself and other robots whose field of view overlaps with its own. This suggests a minimal network graph in which all robots i are connected to all other robots $j \in \mathcal{N}_i$, where $\mathcal{N}_i = \{j \mid Q \cap \mathcal{B}_i \cap \mathcal{B}_j \neq \emptyset, i \neq j\}$. One interpretation would be to assume that robots with overlapping views of view can communicate in a peer-to-peer fashion. For now we assume this topology, but as we discussed in Chapter 4 this assumption cannot always be realized in practice.

We propose a controller that moves a robot in the opposite direction of its locally calculated gradient component. For all robots, we have

$$\vec{u}_i^c = -k \frac{\partial \mathcal{H}}{\partial \vec{p}_i^c} \tag{5.12}$$

where $\vec{u}_i^c$ is the control input for robot i and $k \in (0, \infty)$ is a control gain. Assuming integrator dynamics for the robots,

$$\dot{\vec{p}}_i^c = \vec{u}_i^c, \tag{5.13}$$

we can prove the convergence of this controller to locally minimize the aggregate information per area. Refer to [52] for proof and remarks.

5.2.2 Rectangular Field of View

Until now we have assumed a circular field of view, which eliminates a rotational degree of freedom. Actual cameras have a rectangular CCD array, and therefore a rectangular field of view. Thus, we need to revisit the optimization problem (5.9) and recalculate for a rectangular field of view and a robot with a rotational degree of freedom.

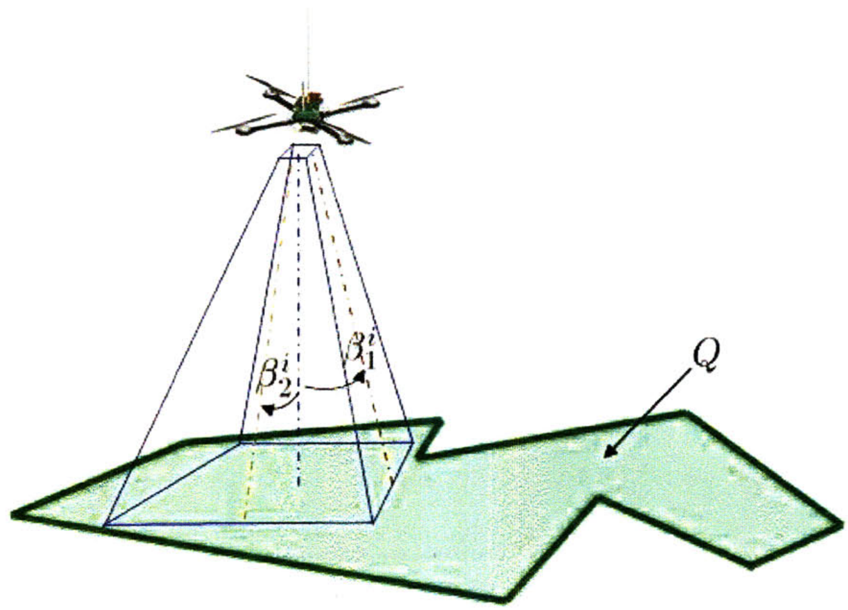


Figure 5-4: The geometry of a camera with a rectangular field of view is shown in this figure.

Let the state space of $\vec{p}_i^c = [\vec{v}_i^T \quad z_i \quad \psi_i]^T$ be $\mathcal{P}_c = \mathbb{R}^3 \times \S$, where ψ_i is the yaw angle. Define a rotation matrix

$$\mathbf{R}_c(\psi_i) = \begin{bmatrix} \cos(\psi_i) & \sin(\psi_i) \\ -\sin(\psi_i) & \cos(\psi_i) \end{bmatrix} \tag{5.14}$$

where $\mathbf{R}_c(\psi_i)\vec{q}$ rotates a vector $\vec{q}$ expressed in the global coordinate frame, to a coordinate frame aligned with the axes of the rectangular field of view. As is true for all rotation matrices, $\mathbf{R}_c(\psi_i)$ is orthogonal, meaning $\mathbf{R}_c(\psi_i)^T = \mathbf{R}_c(\psi_i)^{-1}$. Using this matrix, define the field of view of robot i to be

$$\mathcal{B}_i = \left\{\vec{q} : |\mathbf{R}_c(\psi_i)(\vec{q} - \vec{v}_i)| \le z_i \tan(\vec{\beta}^i)\right\} \tag{5.15}$$

where $\vec{\beta}^i = [\beta_1^i, \beta_2^i]^T$ is a vector of the camera's two half-view angles associated with two perpendicular edges of the rectangle, as shown in Figure 5-4. Here the $\le$ symbol applies element-wise such that all elements in the vector must satisfy the inequality.

We have to break up the boundary of the rectangle into each of its four edges. Let l_k be the kth edge, and define four outward-facing normal vectors $\vec{n}_k$, one associated with each edge, where $\vec{n}_1 = [1,0]^T$, $\vec{n}_2 = [0,1]^T$, $\vec{n}_3 = [-1,0]^T$, and $\vec{n}_4 = [0,-1]^T$.

With a rectangular field of view, the gradient of the cost function, $\mathcal{H}(\vec{p}_1^c, \ldots, \vec{p}_n^c)$, becomes

$$\frac{\partial \mathcal{H}}{\partial \vec{v}_i} = \sum_{k=1}^{4} \int_{Q \cap l_k} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \mathbf{R}_c(\psi_i)^T \vec{n}_k \Phi(\vec{q})\, d\vec{q} \tag{5.16}$$

$$\frac{\partial \mathcal{H}}{\partial z_i} = \sum_{k=1}^{4} \int_{Q \cap l_k} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \tan(\theta)^T \vec{n}_k \Phi(\vec{q})\, d\vec{q} - \int_{Q \cap \mathcal{B}_i} \frac{2h_{\mathcal{N}_{\vec{q}}}^2}{a(b-z_i)^3} \Phi(\vec{q})\, d\vec{q} \tag{5.17}$$

$$\frac{\partial \mathcal{H}}{\partial \psi_i} = \sum_{k=1}^{4} \int_{Q \cap l_k} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \cdot (\vec{q} - \vec{v}_i)^T \mathbf{R}_c(\psi_i + \pi/2)^T \vec{n}_k \Phi(\vec{q})\, d\vec{q} \tag{5.18}$$

Refer to [52] for proof and remarks.

5.2.3 Discrete Controllers

To be implemented on the embedded control module, the integrals in the controller must be computed using a discretized approximation. Let $\widehat{Q}$ be a discretized set of m points uniformly space $\Delta_{\widehat{Q}}$ units apart on a square grid. Given $\vec{p}_i^c$, let $\widehat{\partial \mathcal{B}_i}$ be a discretized sets of m points uniformly spaced $\Delta_{\widehat{\partial \mathcal{B}_i}}$ units apart on $\partial \mathcal{B}_i$. We select $\widehat{Q}$ and $\widehat{\partial \mathcal{B}_i}$ to best approximate Q and $\partial \mathcal{B}_i$, respectively. Using these sets, we can formulate discrete controllers that approximate Equation (5.12). Algorithms for circular and rectangular fields of view are given in Algorithms 2 and 3, respectively.

Proposition 2 (Controller Time Complexity - Circular Fields of View) *The time complexity of Algorithm 2 is*

$$T(n,m) = O(nm) \tag{5.19}$$

Algorithm 2 Discretized Controller for Circular Fields of View

k_v and k_z are lateral and vertical controller gains, respectively
Require: Robot i knows its state $\vec{p}_i^c$
Require: Robot i knows the environment Q
Require: Robot i knows the information per area function $\Phi(\vec{q})$.
Require: Robot i can communicate with all robots $j \in \mathcal{N}_i$.
loop
 Update $\vec{p}_j^c, \ \forall j \in N_i$
 $\frac{\widehat{\partial \mathcal{H}}}{\partial \vec{v}_i} \leftarrow \sum_{q \in Q \cap \widehat{\partial \mathcal{B}_i}} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \frac{(\vec{q}-\vec{v}_i)}{\|q-\vec{v}_i\|} \Phi(\vec{q}) \Delta_{\widehat{\partial \mathcal{B}_i}}$
 $\frac{\widehat{\partial \mathcal{H}}}{\partial z_i} \leftarrow \sum_{q \in Q \cap \widehat{\partial \mathcal{B}_i}} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \Phi(\vec{q}) \tan(\theta) \Delta_{\widehat{\partial \mathcal{B}_i}} - \sum_{q \in \widehat{Q} \cap \mathcal{B}_i} \frac{2h^2_{\mathcal{N}_{\vec{q}}}}{a(b-z_i)^3} \Phi(\vec{q}) \Delta^2_{\widehat{Q}}$
 $\vec{v}_i(t+\Delta t) \leftarrow \vec{v}_i(t) - k_v \frac{\widehat{\partial \mathcal{H}}}{\partial \vec{v}_i}$
 $z_i(t+\Delta t) \leftarrow z_i(t) - k_z \frac{\widehat{\partial \mathcal{H}}}{\partial z_i}$
end loop

Algorithm 3 Discretized Controller for Rectangular Fields of View

k_v, k_z, and k_ψ are lateral, vertical, and rotational controller gains, respectively
Require: Robot i knows its state $\vec{p}_i^c$
Require: Robot i knows the environment Q
Require: Robot i knows the information per area function $\Phi(\vec{q})$.
Require: Robot i can communicate with all robots $j \in \mathcal{N}_i$.
loop
 Update $\vec{p}_c^j, \ \forall j \in \mathcal{N}_i$
 $\frac{\widehat{\partial \mathcal{H}}}{\partial \vec{v}_i} \leftarrow \sum_{k=1}^4 \sum_{q \in Q \cap \widehat{l_k}} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \mathbf{R}_c(\psi_i)^T \vec{n}_k \Phi(\vec{q}) \, \Delta_{\widehat{\partial \mathcal{B}_i}},$
 $\frac{\widehat{\partial \mathcal{H}}}{\partial z_i} \leftarrow \sum_{k=1}^4 \sum_{q \in Q \cap \widehat{l_k}} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \tan(\theta)^T \vec{n}_k \Phi(\vec{q}) \, \Delta_{\widehat{\partial \mathcal{B}_i}} - \sum_{q \in \widehat{Q} \cap \mathcal{B}_i} \frac{2h^2_{\mathcal{N}_{\vec{q}}}}{a(b-z_i)^3} \Phi(\vec{q}) \, \Delta^2_{\widehat{Q}}$
 $\frac{\widehat{\partial \mathcal{H}}}{\partial \psi_i} \leftarrow \sum_{k=1}^4 \sum_{q \in Q \cap \widehat{l_k}} (h_{\mathcal{N}_{\vec{q}}} - h_{\mathcal{N}_{\vec{q}} \setminus \{i\}}) \cdot (\vec{q} - \vec{v}_i)^T \mathbf{R}_c(\psi_i + \pi/2)^T \vec{n}_k \Phi(\vec{q}) \, \Delta_{\widehat{\partial \mathcal{B}_i}}$
 $\vec{v}_i(t+\Delta t) \leftarrow \vec{v}_i(t) - k_v \frac{\widehat{\partial \mathcal{H}}}{\partial \vec{v}_i}$
 $z_i(t+\Delta t) \leftarrow z_i(t) - k_z \frac{\widehat{\partial \mathcal{H}}}{\partial z_i}$
 $\psi_i(t+\Delta t) \leftarrow \psi_i(t) - k_\psi \frac{\widehat{\partial \mathcal{H}}}{\partial \psi_i}$
end loop

where n is the number of robots and m is the number of elements in $\widehat{Q}$ and $\widehat{\partial\mathcal{B}_i}$.

Proof 2 *Discretizing* (5.10) *and* (5.11), *we have*

$$\sum_{j=1}^{m}\left(O(1)+\sum_{k=1}^{n}O(1)\right) \tag{5.20}$$

and

$$\sum_{j=1}^{m}\left(O(1)+\sum_{k=1}^{n}O(1)+\sum_{k=1}^{n-1}O(1)\right) \tag{5.21}$$

time, respectively. Summing these results gives $O(nm)$ time. Since all other loop operations are less than $O(n)$ time, Algorithm 2 is of $O(nm)$ time.

Proposition 3 (Controller Time Complexity - Rectangular Fields of View) *The time complexity of Algorithm 3 is*

$$T(n,m)=O(nm) \tag{5.22}$$

where n is the number of robots and m is the number of elements in $\widehat{Q}$ and $\widehat{\partial\mathcal{B}_i}$.

Proof 3 *Time complexity for discretizing* (5.16) *and* (5.17) *are identical to* (5.20) *and* (5.21), *respectively. Discretizing* (5.18), *we have*

$$\sum_{j=1}^{m}\left(O(1)+\sum_{k=1}^{n}O(1)\right) \tag{5.23}$$

time. Summing these results gives $O(nm)$ time. Since all other loop operations are less than $O(n)$ time, Algorithm 3 is of $O(nm)$ time.

Chapter 6

Robot Experiments

To demonstrate the functionality of our embedded control module for a team of AscTec Hummingbird flying quad-rotor robots, we conducted numerous experiments in both indoor and outdoor environments. We first implemented the optimal coverage algorithm presented in Chapter 5 indoors in distributed fashion. Three quad-rotor robots optimally positioned their downward facing cameras over a fixed environment. The robot team used the 2.4 Ghz Digi XBee-PRO radio modules to acquire position information from the motion capture system. In preparation for the outdoor extension of this algorithm, we then conducted a mobile ad hoc network experiment above the urban MIT campus using the same hardware configuration. State estimates were routed in multi-hop fashion from the robot team to a ground base station using a simple flooding scheme.

The limited communication range we experienced during the network experiment led to the selection of the 900 Mhz Digi XBee-XSC radio modules for outdoor flights. To accommodate the decrease in baud rate when compared to the XBee-PRO, we developed the location-based multi-hop algorithm described in Chapter 4. We then implemented this algorithm on five quad-rotor flying robots and four ground base stations to demonstrate network adaptability for a dynamic robot team. This same algorithm was used to allow state estimates to propagate during the outdoor deployment of our optimal coverage algorithm.

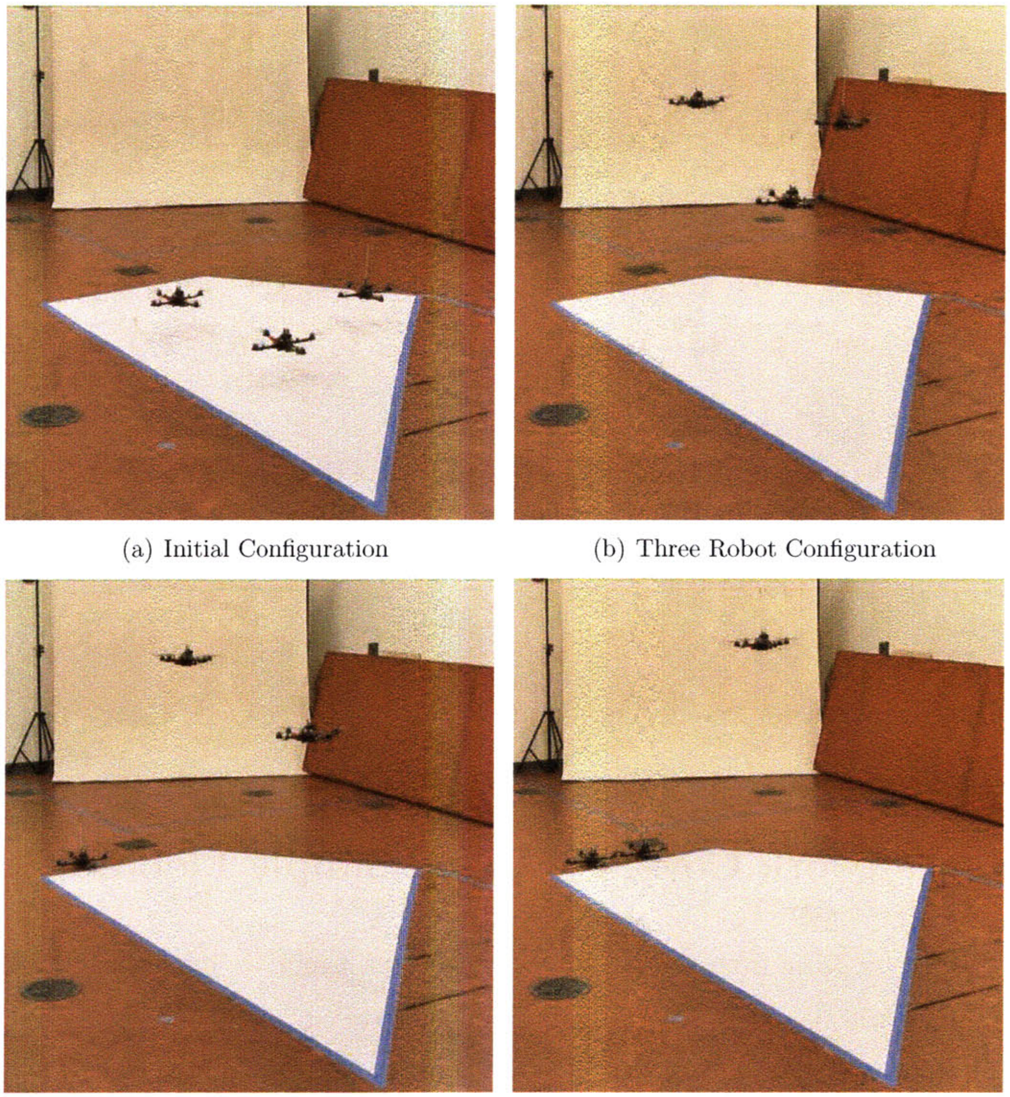
6.1 Optimal Coverage of an Indoor Environment

The optimal coverage algorithm for a circular field of view was implemented on our embedded control module, running asynchronously in a fully distributed fashion. The algorithm calculated the waypoints $\vec{c}_i(t)$ and $z_i(t)$ from Algorithm 2 at 1 Hz. This time-scale separation between the coverage algorithm and the low level PID controllers was required to approximate the integrator dynamics assumed in Equation (5.13). The camera parameters were set to $a = 10^{-6}$ and $b = 10^{-2}$ meters (which are typical for commercially available cameras), the field of view was $\beta^i = 35°$, the information per area was a constant $\phi(q) = 1$, the prior area per pixel was $w = 10^{-6}$ meters^2, and the control gain was $k = 10^{-5}$. The environment to be covered was a skewed rectangle, 3.7 meters across at its widest, shown in white in Figure 6.1.

To test the effectiveness of the algorithm and its robustness to robot failures, we conducted experiments as follows: 1) three robots moved to their optimal positions using the algorithm, 2) one robot was manually removed from the environment, and the remaining two were left to reconfigure automatically, 3) a second robot was removed from the environment and the last one was left to reconfigure automatically. Figure 6.1 shows photographs of a typical experiment at the beginning (Figure 6-1(a)), after the first stage (Figure 6-1(b)), after the second stage (Figure 6-1(c)), and after the third stage (Figure 6-1(d)).

The coverage cost of the robots over the course of the whole experiment, averaged over 19 experiments, is shown in Figure 6-2, where the error bars represent one standard deviation. Notice that when one robot is removed, the cost function momentarily increases, then decrease as the remaining robots find a new optimal configuration. The algorithm proved to be robust to the significant, highly nonlinear unmodeled aerodynamic effects of the robots, and to individual robot failures.

We repeated the above experiment a total of 20 times. Of these 19 were successful, while in one experiment two of the robots collided in mid air. The collision was caused by an unreliable gyroscopic sensor, not by a malfunction of the coverage algorithm. With appropriate control gain values, collisions are avoided by the algorithm's natural



(a) Initial Configuration (b) Three Robot Configuration

(c) Two Robot Configuration (d) One Robot Configuration

Figure 6-1: Frame shots from an experiment with three AscTec Hummingbird quadrotor robots are shown. After launching from the ground (Figure 6-1(a)), the three robots stabilize in an optimal configuration (Figure 6-1(b)). Then one robot is manually removed to simulate a failure, and the remaining two move to a new optimal position (Figure 6-1(c)). Finally a second robot is removed and the last one stabilizes at an optimal position (Figure 6-1(d)). The robots move so that their fields of view (which cannot be seen in the snapshots) cover the environment, represented by the white polygon.

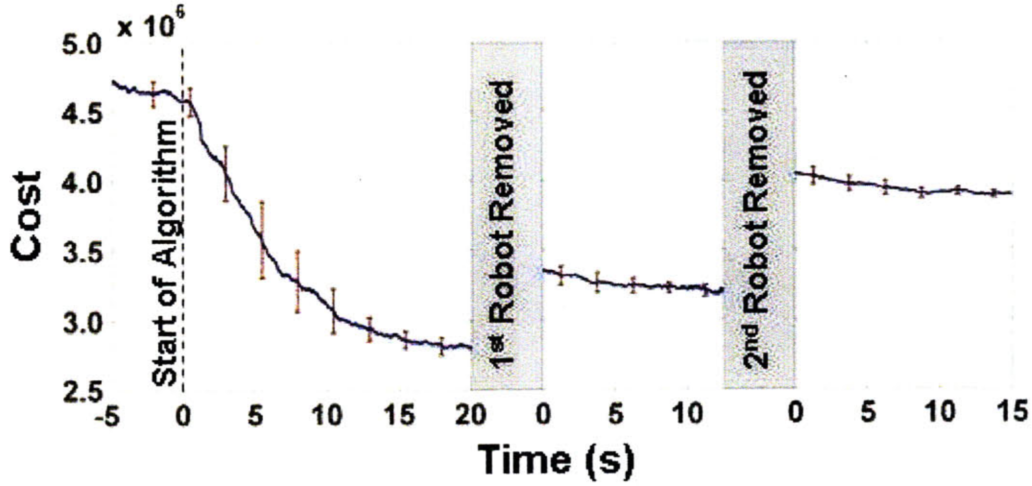


Figure 6-2: The cost function during the three stages of the experiment, averaged over 19 successful experiments, is shown. The error bars denote one standard deviation. The experiments demonstrate the performance of the algorithm, and its ability to adapt to unforeseen robot failures.

tendency for neighbors to repel one another.

6.2 Network Coverage Using Simple Flooding

Using the same team from the indoor coverage experiments, we implemented a simple flooding scheme using TDMA in an outdoor urban environment. Our embedded control module acquired GPS coordinates, altitude readings, and temperature measurements from the AutoPilot module at 4 Hz. During a robot's assigned time slot, the 2.4 GHz XBee-PRO module broadcasted the state estimates of all three robots in the system, where states for the other two robots were acquired from previously received broadcasts. Each robots state is described by an array containing five 32-bit integers (i.e. time stamp, GPS latitude, GPS longitude, altitude reading, and temperature measurement), resulting in a data size of 160 bits. Given that our wireless network could reliably run at a baud rate of 57.6 kbps, the maximum frame rate using Equation 4.1 was about 40 Hz - an order of magnitude faster than what is needed for our coverage algorithm.

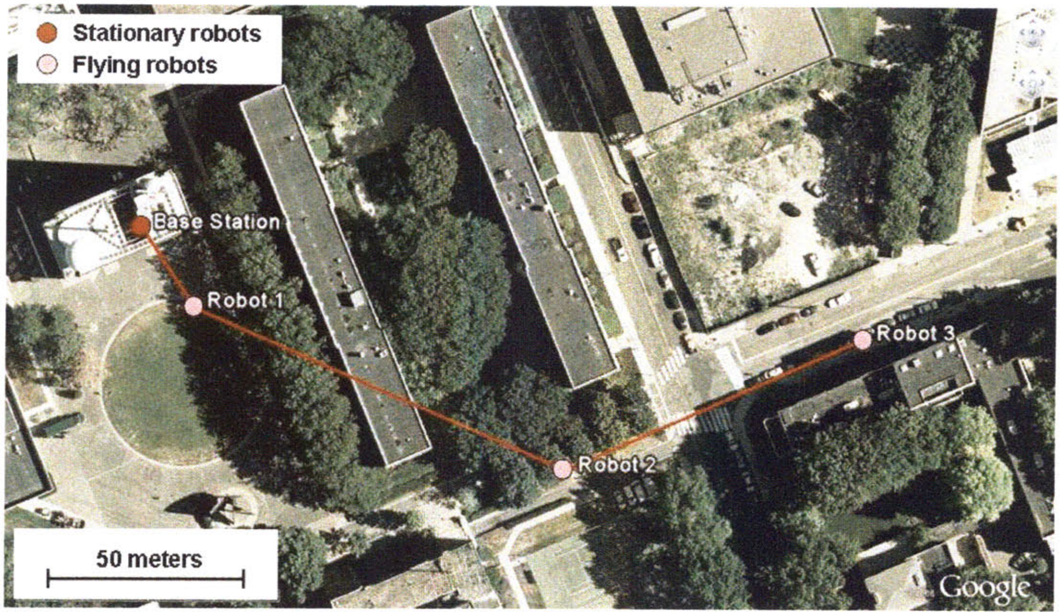


Figure 6-3: An example mobile ad hoc network graph from the quad-rotor flying robot experiment is plotted over an areal image. For a system of three robots, the simple flooding scheme was successful in routing state estimates back to the base station.

A base station monitored network activity as the robots were manually piloted to expand the spatial size of the network. Figure 6-3 shows a plot of the position estimates of the three robots as received by the base station using the full broadcast strategy described. The positions are laid over an aerial photograph of the environment. The robot temperature estimates were also recorded by the base station. Figure 6-4 plots temperature with respect to time for the three robots. In addition, the base station recorded how the estimates were received. If they were received directly from the originating source, the data point is labeled with an $\times$. Otherwise, the data traveled a multi-hop route to the base station and is labeled with an $\circ$.

6.3 Location-Based Algorithm

Although the simple flooding scheme in Section 6.2 was successful in routing state estimates through the robot team, we were disappointed in the communication range

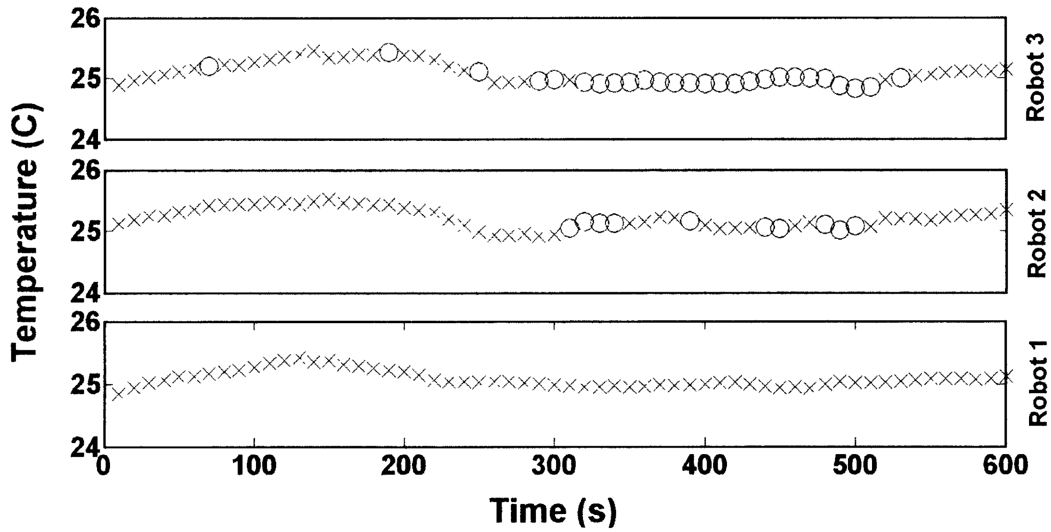


Figure 6-4: This figure shows acquired physical data from the experiment. Robot state estimates, as recorded by the base station, included onboard temperature. Estimates transferred directly to the base station from the originating robot are labeled with $\times$'s, while ones needing multiple hops are labeled with $\circ$'s.

obtained by the 2.4 Ghz Digi XBee-PRO radio modules. At best peer-to-peer communication links of about 100 m were maintained, although we experienced dropouts between robots located less than 50 meters from each other. To allow for large scale outdoor experiments, we decided to use the longer range 900 Mhz Digi XBee-XSC radio module. However, this hardware change results in the decrease of our wireless baud rate from about 57.6 kbps to 3 kbps. This motivated us to implement the location-based algorithm described in Chapter 4 to effectively multi-hop state estimates throughout the robot team.

This experimental setup was designed to emulate an initial configuration for five flying robots running the downward facing camera coverage controller. In addition, four stationary robots were used to monitor the system as base stations, thus creating a nine robot team. Since the coverage controller has a spatial dependence similar to the Voronoi coverage controller from Section 4.3.2, the same time slot length ($l = 3$), distance function (Euclidean), and α ($= 1$) were used. Figure 6-5 shows the network topology of a random deployment configuration prior to starting the coverage controller. Here we limited the communication range to 30 meters; this radio module

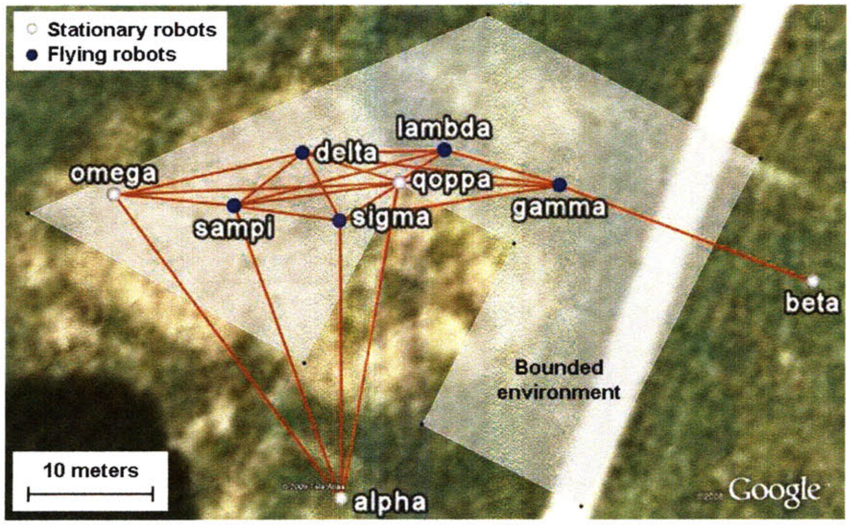


Figure 6-5: An example mobile ad hoc network graph from the quad-rotor flying robot experiment is plotted in Google Earth. For this nine robot system, the location-based algorithm routes state estimates through the entire team. The bounded environment from the downward facing camera coverage problem is also shown.

was able to produce links in excess of 100 meters in separate range tests.

Figure 6-6 plots the time stamp of the most current state estimates as received by the stationary robot beta, which can be considered the "worst case" receiver since it is the most remote robot in the team. As previously discussed, beta's own state estimate is always considered to be current. Estimates of other robots are updated as they are received by team broadcasts, whether directly from the originating robot or indirectly in a multi-hop fashion. Since closer robots are considered more important in the algorithm formulation, this results in their state estimates being more current with more frequent updates. This characteristic can be seen easily in Figure 6-7, where the state estimates of gamma and lambda are more important to beta than alpha and omega.

6.4 Optimal Coverage of an Outdoor Environment

We combined the optimal coverage algorithm in Chapter 5 and the location-based multi-hopping algorithm in Chapter 4 to form a completely autonomous system that

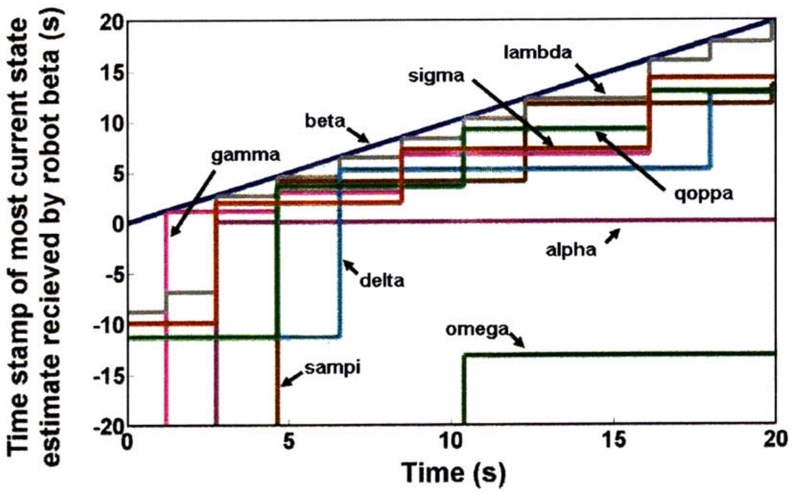


Figure 6-6: This plot shows the time stamp of the most current state estimates received by the stationary robot beta. Estimates of closer, more important robots are updated more frequently and tend to be more current, which validates the location-based algorithm.

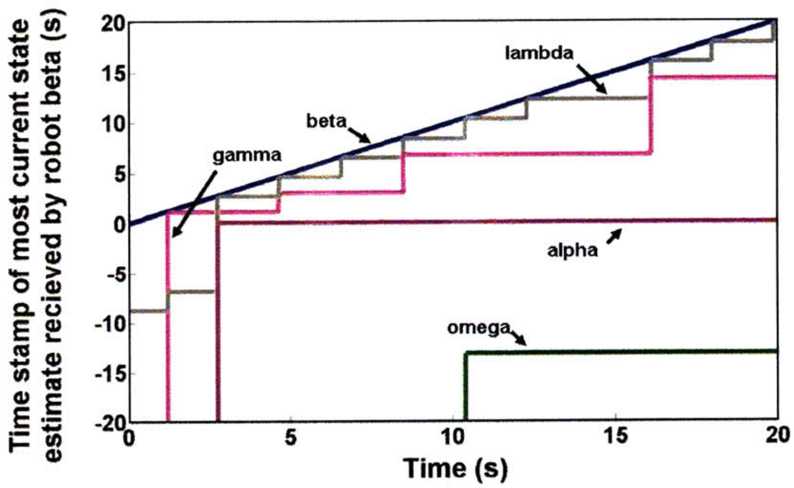


Figure 6-7: A simplified plot of Figure 6-5 is shown. From the frequency of the updates, we can conclude that gamma started the closest to beta but then moved away during the experiment. Once lambda moved into communication range with beta, state estimates of the team began to be routed through lambda.

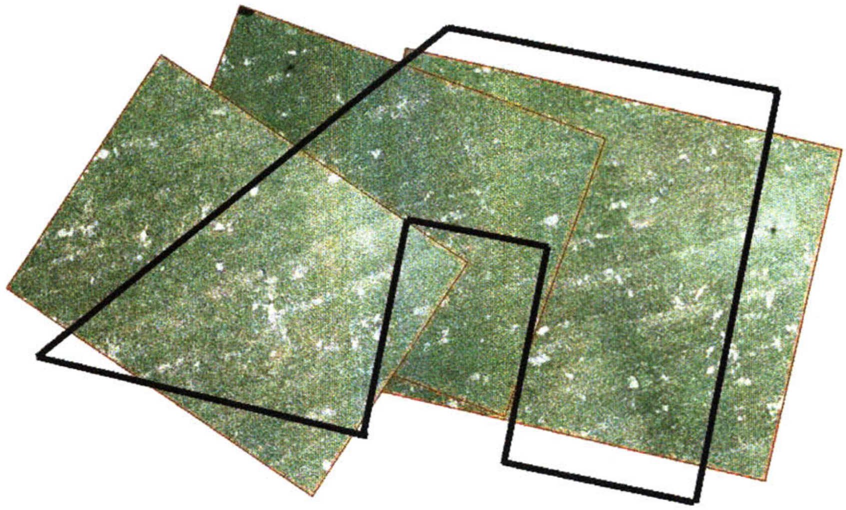


Figure 6-8: The embedded control modules are used to cover an outdoor environment. Off-the-shelf miniature camcorders recorded the field of view of each robot during the experiment.

can be deployed in an outdoor environment. Three quad-rotor flying robots were used to route state information to three stationary robots, forming a six robot team. Compared to the indoor experiment in Section 6.1, the environment was drastically different in two ways: 1) the skewed rectangle, which measured approximately 60 meters at its widest, was much larger; and 2) a square area was removed to create a non-convex environment. However, we were able to use the same camera parameters for the coverage algorithm as before, only this time a rectangular field of view was used with half angles of $\beta_1^i = 35°$ and $\beta_2^i = 26.25°$.

To assist in visualizing the team's coverage, we installed an inexpensive video camera on the bottom of each robot. Figure 6-8 shows an example snapshot of the team's coverage during the experiment. We noted that the mosaic did not fully cover the environment, however we believe this outcome is the result of three factors: 1) the environment area was discretized and may have not represented the perimeter of the skewed rectangle well; 2) there was considerable error in GPS readings, which may have prevent the robots from settling into an optimal configuration; and 3) due to limited amount of battery life, we may have not given the team enough time to fully cover the environment.

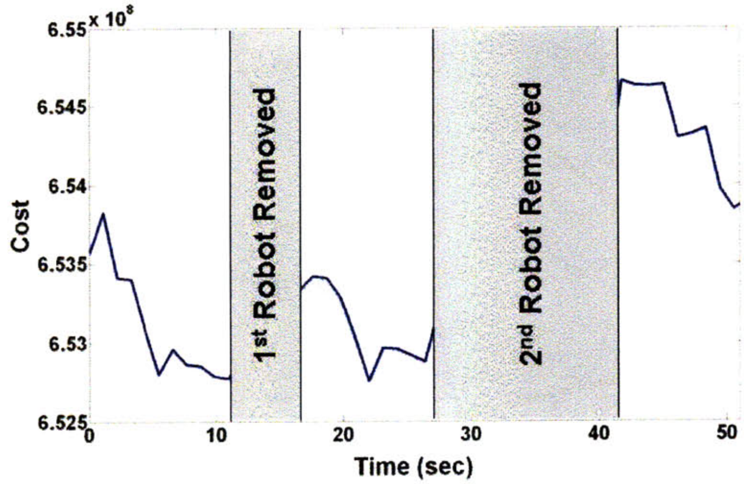


Figure 6-9: The cost function during the three stages of the outdoor experiment. Like the indoor experiment, we demonstrated the performance of the algorithm and its ability to adapt to unforeseen robot failures.

Figure 6-9 shows the cost function during the outdoor experiment. The three flying robots converged to an optimal configuration before one robot was manually removed from the system. This removal process was repeated after the two robot team converged to a new optimal configuration. The ability to adapt to unforeseen robot failures demonstrates the robustness of the algorithm even in outdoor environments. This experiment was repeated once more with similar results.

Chapter 7

Conclusions

In this thesis we discussed the distributed control of a high performance quad-rotor flying robot team. This research produced the design for an onboard embedded control module, with which we successfully deployed a distributed robot team in both an indoor and outdoor environment. In both environments we constructed a mobile ad hoc network infrastructure to propagate state estimates among the robots. Our system design was verified in numerous experiments that successfully showed the self-organization of the distributed team.

7.1 Lessons Learned

The process of designing, fabricating, programming, and implementing the embedded control module provided an extremely rewarding experience. The following summarizes some lessons learned during our research effort:

1. *Version Control Among All Robots* - Every computer scientist has (or will) learn the importance of version control. This task becomes exponentially more important when considering a distributed system. Careful bookkeeping is needed to insure all robot firmware is current and compatible. We made our biggest mistake when we assumed identical protocol among different firmware versions for the AscTec AutoPilot module. The resulting deployment produced a team

where only half of the robots operated correctly, even though all firmware for the embedded control module was current. This mistake could have been easily avoided (or easily debugged) if we ensured that all hardware in the system was up-to-date[1].

2. *Scaling Gradually* - Since all hardware and software was designed to scale automatically, we were often tempted to drastically increase the number of robots in the team between deployments. This action was usually less efficient (e.g. time, battery life, etc.) than gradually increasing the scale of the experiment. For example, consider our outdoor coverage experiment in Section 6.4. Even though it doesn't seem like a drastic increase, we wasted a good amount of time by increasing the number of robots from one to three. A two robot experiment would have given insight into deployment strategy for multiple robots. Instead, we had the added complexity of another robot member, which only complicated the situation and extended the setup time before the actual three robot experiment was conducted.

3. *Hardware-In-Loop Simulations* - We relied heavily on MATLAB and C code simulations to initially construct our distributed algorithms. However, we added another simulation step, hardware-in-loop (HIL), to further debug our code prior to the field experiments. By representing the quad-rotor flying robot as a virtual model on the embedded control module, we were able to closely monitor the performance of our algorithms running on the actual hardware. This setup, as shown in Figure 4-3, provided us with an effective tool for software integration and control parameter tuning.

7.2 Future Work

The primary motivation for developing the embedded control module was to support the development of a variety of high-dimensional (+3) distributed algorithms. The

[1]We would like to thank Daniel Gurdan from Ascending Technologies for helping us identify this incompatibility.

resulting system is advanced in overall functionality yet intuitive and user friendly. Currently we are developing a general user interface to be used by many distributed robot projects. These projects are not necessarily using quad-rotor flying robots - Yun et. al are using our system to propagate state estimates among assembly robots [40]. We hope this effort will form a testbed where a heterogeneous robot team can demonstrate distributed behavior; we are inspired by the work of Jon How et. al and the RAVEN testbed [35].

Concerning our location-based multi-hop algorithm from Chapter 4, we believe the simplicity of this strategy enables the cascading of additional location-based algorithms. The spatial reuse of time slots for robots separated by multiple hops can allow for virtually infinite robot team sizes and spatial coverage. In addition, our algorithm is not limited to TDMA; significant performance gains can be obtained by using different channel access method, such as code division multiple access (CDMA) and frequency division multiple access (FDMA).

Finally, we are actively pursuing more complex experiments for the optimal coverage algorithm with downward facing cameras from Chapter 5. Results for teams of 5+ robots will be published in the near future. We are also exploring the coverage of dynamically changing environments with respect to geometry of the boundary and the shape of the importance function. Our ultimate goal for this algorithm is to seamlessly incorporate robots of different dynamics and cameras of different optical properties.

Appendix A

Communication Protocol

The following tables outline the communication protocol used over the wireless network. Sections of the packets relating to byte count and checksum calculations are labeled. For all packets, the checksum section may be "looped;" for example, multiple state estimates may be sent within one state packet. The increase in length is accounted for by both the byte count and the checksum.

Table A.1: The state packet is sent between robots to update their state estimate lists. The first entry in the checksum loop contains the sender's estimate, which is assumed to be current and is used to synchronize teammates' clocks.

	Description	**Type**	**Bytes**	**Defined**	
	start string	uint8 [3]	3	“>*>”	
	packet indicator	uint8	1	’S’	
	byte count	uint32	4		
begin byte count	robot id	uint32	4		begin checksum
	time stamp	uint32	4	[μs]	
	x_i	int32	4	[mm]	
	y_i	int32	4	[mm]	
	z_i	int32	4	[mm]	
	ψ_i	int32	4	[mrad]	end checksum
end byte count	checksum	uint32	4	crc32	

Table A.2: The global packet provides position information for target robots during indoor operation. This information overrides all other methods of position acquisition. This packet has no effect during outdoor operation.

	Description	**Type**	**Bytes**	**Defined**	
	start string	uint8 [3]	3	“>*>”	
	packet indicator	uint8	1	'G'	
	byte count	uint32	4		
begin byte count	robot id	uint32	4		begin checksum
	time stamp	uint32	4	[μs]	
	x_i	int32	4	[mm]	
	y_i	int32	4	[mm]	
	z_i	int32	4	[mm]	
	r_i^x	int32	4	[mrad]	
	r_i^y	int32	4	[mrad]	
	r_i^z	int32	4	[mrad]	end checksum
end byte count	checksum	uint32	4	crc32	

Table A.3: The communication packet initializes TDMA communication among the robots in the team. The packet also determines which robots are in the team for high level control. Once sent, following packets must be sent in an assigned slot, even if originating from a base station.

	Description	Type	Bytes	Defined	
	start string	uint8 [3]	3	“>*>”	
	packet indicator	uint8	1	'C'	
	byte count	uint32	4		
begin byte count	robot id	uint32	4		begin checksum
	baud rate	int32	4	[bps]	
	slot length	int32	4	[# states]	end checksum
end byte count	checksum	uint32	4	crc32	

Table A.4: The environment packet builds the environment boundary polygon prior to high level control initialization. The importance function over the environment is assumed to be known by the robots, otherwise importance is assumed to be unity.

	Description	**Type**	**Bytes**	**Defined**	
	start string	uint8 [3]	3	“>*>”	
	packet indicator	uint8	1	’E’	
	byte count	uint32	4		
begin byte count	$x_{\partial Q}$	int32	4	[mm]	begin checksum
	$y_{\partial Q}$	int32	4	[mm]	end checksum
end byte count	checksum	uint32	4	crc32	

Table A.5: The algorithm packet initializes high level control for target robots. Team members are determined from a prior communication packet, otherwise robots assume there is no team.

	Description	Type	Bytes	Defined	
	start string	uint8 [3]	3	“>*>”	
	packet indicator	uint8	1	'A'	
	byte count	uint32	4		
begin byte count	robot id	uint32	4		calc checksum
end byte count	checksum	uint32	4	crc32	

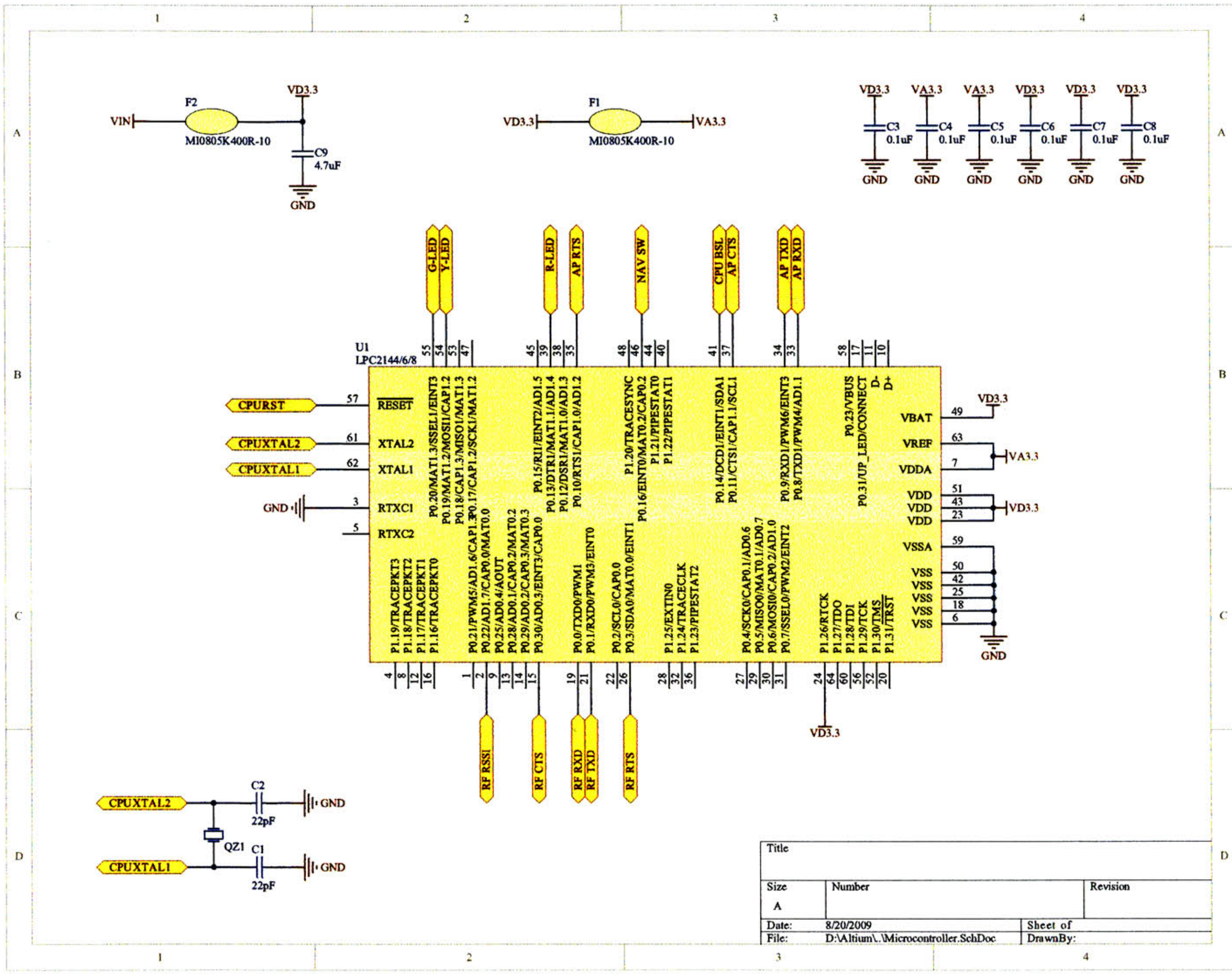
Table A.6: The dynamics packet initializes low level control for target robots. Current waypoint is set to first valid position acquired after packet is received.

	Description	**Type**	**Bytes**	**Defined**	
	start string	uint8 [3]	3	“>*>”	
	packet indicator	uint8	1	’D’	
	byte count	uint32	4		
begin byte count	robot id	uint32	4		calc checksum
end byte count	checksum	uint32	4	crc32	

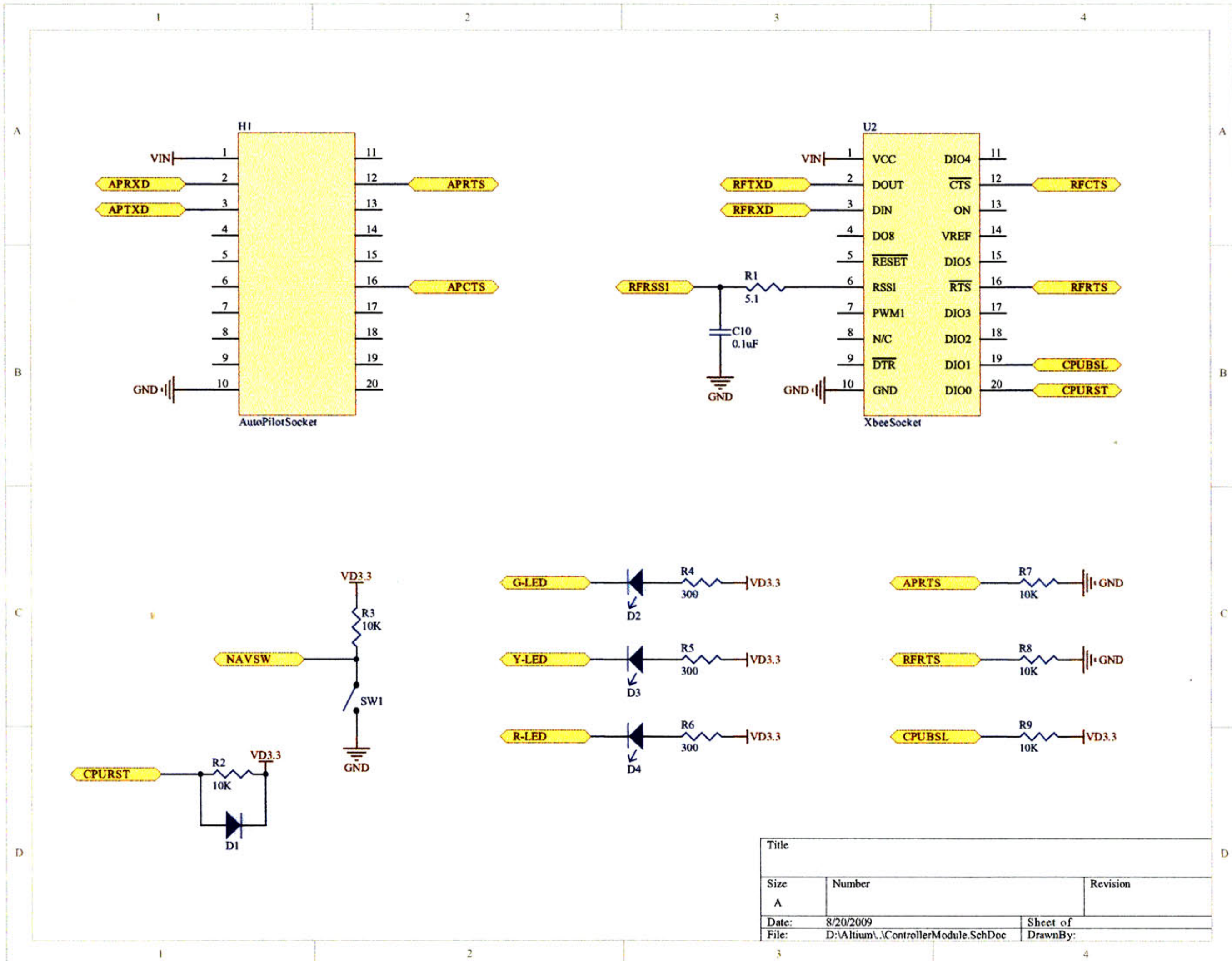
Appendix B

Electrical Schematics

The following figures contain the electrical schematics for the embedded control module. The first figure illustrates the installation of the NXP LPC2148 microcontroller. The second figure illustrates the port connections to the AutoPilot and radio modules, as well as the LED and navigation switch connections.



VIN
F2
MI0805K400R-10
VD3.3
C9
4.7uF
GND
VD3.3
F1
MI0805K400R-10
VA3.3
VD3.3
VA3.3
VA3.3
VD3.3
VD3.3
VD3.3
C3 0.1uF
C4 0.1uF
C5 0.1uF
C6 0.1uF
C7 0.1uF
C8 0.1uF
GND
G-LED
Y-LED
R-LED
AP RTS
NAV SW
CPU BSL
AP CTS
AP TXD
AP RXD
U1
LPC2144/6/8
CPURST
RESET
CPUXTAL2
XTAL2
CPUXTAL1
XTAL1
RTXC1
RTXC2
VBAT
VREF
VDDA
VDD
VSSA
VSS
P0.20/MAT1.3/SSEL1/EINT3
P0.19/MAT1.2/MOSI1/CAP1.2
P0.18/CAP1.3/MISO1/MAT1.3
P0.17/CAP1.2/SCK1/MAT1.2
P0.15/RI1/EINT2/AD1.5
P0.13/DTR1/MAT1.1/AD1.4
P0.12/DSR1/MAT1.0/AD1.3
P0.10/RTS1/CAP1.0/AD1.2
P1.20/TRACESYNC
P0.16/EINT0/MAT0.2/CAP0.2
P1.21/PIPESTAT0
P1.22/PIPESTAT1
P0.14/DCD1/EINT1/SDA1
P0.11/CTS1/CAP1.1/SCL1
P0.9/RXD1/PWM6/EINT3
P0.8/TXD1/PWM4/AD1.1
P0.23/VBUS
P0.31/UP_LED/CONNECT
D-
D+
P1.19/TRACEPKT3
P1.18/TRACEPKT2
P1.17/TRACEPKT1
P1.16/TRACEPKT0
P0.21/PWM5/AD1.6/CAP1.3
P0.22/AD1.7/CAP0.0/MAT0.0
P0.25/AD0.4/AOUT
P0.28/AD0.1/CAP0.2/MAT0.2
P0.29/AD0.2/CAP0.3/MAT0.3
P0.30/AD0.3/EINT3/CAP0.0
P0.0/TXD0/PWM1
P0.1/RXD0/PWM3/EINT0
P0.2/SCL0/CAP0.0
P0.3/SDA0/MAT0.0/EINT1
P1.25/EXTIN0
P1.24/TRACECLK
P1.23/PIPESTAT2
P0.4/SCK0/CAP0.1/AD0.6
P0.5/MISO0/MAT0.1/AD0.7
P0.6/MOSI0/CAP0.2/AD1.0
P0.7/SSEL0/PWM2/EINT2
P1.26/RTCK
P1.27/TDO
P1.28/TDI
P1.29/TCK
P1.30/TMS
P1.31/TRST
RF RSSI
RF CTS
RF RXD
RF TXD
RF RTS
C2
22pF
QZ1
C1
22pF
Title
Size
A
Number
Revision
Date: 8/20/2009
File: D:\Altium\..\Microcontroller.SchDoc
Sheet of
DrawnBy:


H1
AutoPilotSocket
VIN
APRXD
APTXD
GND
APRTS
APCTS
U2
XbeeSocket
VCC
DOUT
DIN
DO8
RESET
RSSI
PWM1
N/C
DTR
GND
DIO4
CTS
ON
VREF
DIO5
RTS
DIO3
DIO2
DIO1
DIO0
RFTXD
RFRXD
RFRSSI
R1
5.1
C10
0.1uF
RFCTS
RFRTS
CPUBSL
CPURST
VD3.3
R3
10K
NAVSW
SW1
CPURST
R2
10K
D1
G-LED
D2
R4
300
Y-LED
D3
R5
300
R-LED
D4
R6
300
APRTS
R7
10K
RFRTS
R8
10K
CPUBSL
R9
10K
Title
Size
A
Number
Revision
Date:
8/20/2009
Sheet of
File:
D:\Altium\..\ControllerModule.SchDoc
DrawnBy:

Appendix C

Source Code

The following pages show section of the DRL quad-rotor project relevant to this thesis. The main source file, which runs the high level distributed algorithm, is first shown. All other source code is shown in alphabetical order with respect to file name. We suggest that these pages are to be used for reference only.

```
/*
 * emb/Main.h
 *
 * main include for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_MAIN_H__

#ifndef __EMB_MAIN_H__
#define __EMB_MAIN_H__


// defines
#define MY_ROBOT_NAME ((byte_t *)"alpha")


// includes

#include "emb/Ap.h"
#include "emb/Alg.h"
```

```
#include "emb/Clock.h"
#include "emb/Comm.h"
#include "emb/Global.h"
#include "emb/Gps.h"
#include "emb/Imu.h"
#include "emb/Mhop.h"
#include "emb/Nav.h"
#include "emb/Status.h"
#include "emb/Types.h"
#include "lpc/Cntl.h"
#include "lpc/Eint.h"
#include "lpc/Tmr.h"
#include "lpc/Types.h"


// main struct

typedef struct
{
  lpcCntl_s Cntl;
  lpcTmr_s Tmr;

  embAp_s Ap;
  embAlg_s Alg;
  embClock_s Clock;
  embComm_s Comm;
  embEnv_s Env;
  embGlobal_s Global;
  embGps_s Gps;
  embImu_s Imu;
```

```
  embMhop_s Mhop;
  embNav_s Nav;
  embPidList_s PidList;
  embStateList_s StateList;
  embStatus_s Status;
}
  embMain_s;


// __EMB_MAIN_H__

#endif
```

```
/*
 * emb/Main.c
 *
 * main source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


#include "emb/Main.h"

#include "emb/Ap.h"
#include "emb/Alg.h"
#include "emb/Clock.h"
#include "emb/Comm.h"
#include "emb/Nav.h"
#include "emb/Status.h"
#include "lpc/Cntl.h"
#include "lpc/Eint.h"
#include "lpc/Tmr.h"
#include "lpc/Uart.h"
#include "lpc/Types.h"
#include "lpc/Lpc214x.h"
```

```c


// global variables

embMain_s MallocEmb;


// local defines

#define HIGH_LEVEL_MSEC_PERIOD 1000
#define SYS_TMR_NUM 0
#define RX_AP_PERIOD (CCLK/1000)
#define TX_AP_PERIOD (CCLK/33)
#define ALG_PERIOD (CCLK/1)


// local functions

void initEmbStruct(embMain_s *Emb);
void initEmb(embMain_s *Emb);
void configEmb(embMain_s *Emb);
void enableEmb(embMain_s *Emb);
void sysTmrVectAddr(void) __attribute__ ((interrupt("IRQ")));
void distributedAlg(embMain_s *Emb);



// main function for quadrotor project
int main(void)
{
```

```
  // local stack
  embMain_s *Emb;

  // create pointer to memory for emb structure
  Emb = &MallocEmb;

  // start embedded system
  initEmb(Emb);
  configEmb(Emb);
  enableEmb(Emb);

  // infinite while loop for distributed algorithm
  while(1)
    {
      embDistAlg(&Emb->Alg);
      lpcSleepMsecTmr(&Emb->Tmr,
                      ALG_PERIOD);
    }

  return(0);
}



// initialize embedded control
void initEmb(embMain_s *Emb)
{
  // initialize lpc control structure
  lpcInitCntl(&Emb->Cntl);

```

```
// initialize system timer
lpcInitTmr(&Emb->Tmr);

// assign embedded links
initEmbStruct(Emb);

// initialize AutoPilot
embInitAp(&Emb->Ap);

// initialize distributed algorithm
embInitAlg(&Emb->Alg);

// initialize system clock
embInitClock(&Emb->Clock);

// initialize comm
embInitComm(&Emb->Comm);

// initialize multi-hop algorithm
embInitEnv(&Emb->Env);

// initialize global position
embInitGlobal(&Emb->Global);

// initialize multi-hop algorithm
embInitMhop(&Emb->Mhop);

// initialize nav
embInitNav(&Emb->Nav);

```

```
  // initialize Pid
  embInitPid(&Emb->PidList);

  // initialize state
  embInitState(&Emb->StateList);

  // initialize status
  embInitStatus(&Emb->Status);
}



// config embedded control system
void configEmb(embMain_s *Emb)
{
  // configure lpc control structure

  // configure system timer
  lpcSetNumTmr(&Emb->Tmr,
               SYS_TMR_NUM);
  lpcSetVectAddrTmr(&Emb->Tmr,
                    (reg32_t)sysTmrVectAddr);

  // configure AutoPilot
  embConfigAp(&Emb->Ap);

  // configure distributed algorithm
  embConfigAlg(&Emb->Alg);

  // configure system clock
```

```
    embConfigClock(&Emb->Clock);

    // configure comm
    embConfigComm(&Emb->Comm);

    // configure multi-hop algorithm
    embConfigEnv(&Emb->Env);

    // configure global position
    embConfigGlobal(&Emb->Global);

    // configure multi-hop algorithm
    embConfigMhop(&Emb->Mhop);

    // configure nav
    embConfigNav(&Emb->Nav);

    // configure Pid
    embConfigPid(&Emb->PidList);

    // configure state
    embConfigState(&Emb->StateList);

    // configure status
    embConfigStatus(&Emb->Status);
}


// enable embedded control system
void enableEmb(embMain_s *Emb)
```

```
{
  // enable lpc control structure
  lpcEnableCntl(&Emb->Cntl);

  // enable two system timers
  lpcEnableTmr(&Emb->Tmr);

  // enable AutoPilot
  embEnableAp(&Emb->Ap);

  // enable distributed algorithm
  embEnableAlg(&Emb->Alg);

  // enable system clock
  embEnableClock(&Emb->Clock);

  // enable comm
  embEnableComm(&Emb->Comm);

  // enable multi-hop algorithm
  embEnableEnv(&Emb->Env);

  // enable global position
  embEnableGlobal(&Emb->Global);

  // enable multi-hop algorithm
  embEnableMhop(&Emb->Mhop);

  // enable nav
  embEnableNav(&Emb->Nav);
```

```
  // enable Pid
  embEnablePid(&Emb->PidList);

  // enable state
  embEnableState(&Emb->StateList);

  // enable status
  embEnableStatus(&Emb->Status);
}


// sets vector address for system timer

void sysTmrVectAddr(void)
{
  // local stack
  embMain_s *Emb;
  bool_t Read;

  // create linked emb
  Emb = &MallocEmb;

  // if RxComm match caused IR
  lpcReadIrMatchTmr(&Emb->Comm.Rx.Match,
                                  &Read);
  if(Read)
    {
      // rx comm handle
      embRxComm(&Emb->Comm);
```

```
    }

  // if TxComm match caused IR
  lpcReadIrMatchTmr(&Emb->Comm.Tx.Match,
                    &Read);
  if(Read)
    {
      // tx comm handle
      embTxComm(&Emb->Comm);

    }

  // if RxAp match caused IR
  lpcReadIrMatchTmr(&Emb->Ap.Rx.Match,
                    &Read);
  if(Read)
    {
      // tx ap handle
      embTxAp(&Emb->Ap);

    }

  // if TxAp match caused IR
  lpcReadIrMatchTmr(&Emb->Ap.Tx.Match,
                    &Read);
  if(Read)
    {
      // tx ap handle
      embTxAp(&Emb->Ap);

```

```
    }

  // reset vic
  lpcResetVicTmr(&Emb->Tmr);
}


// initialize embedded struct

void initEmbStruct(embMain_s *Emb)
{
  // alg
  Emb->Alg.Env = &Emb->Env;
  Emb->Alg.Global = &Emb->Global;
  Emb->Alg.StateList = &Emb->StateList;

  // ap
  Emb->Ap.Tmr = &Emb->Tmr;
  Emb->Ap.Gps = &Emb->Gps;
  Emb->Ap.StateList = &Emb->StateList;
  Emb->Ap.PidList = &Emb->PidList;
  Emb->Ap.Global = &Emb->Global;
  Emb->Ap.Imu = &Emb->Imu;
  Emb->Ap.Status = &Emb->Status;
  Emb->Ap.Nav = &Emb->Nav;
  Emb->Ap.Clock = &Emb->Clock;

  // comm
  Emb->Comm.Tmr = &Emb->Tmr;
  Emb->Comm.Status = &Emb->Status;
```

```
    Emb->Comm.StateList = &Emb->StateList;
    Emb->Comm.Env = &Emb->Env;
    Emb->Comm.PidList = &Emb->PidList;
    Emb->Comm.Global = &Emb->Global;
    Emb->Comm.Clock = &Emb->Clock;

    // nav
    Emb->Nav.Status = &Emb->Status;

    // statelist
    Emb->Mhop.StateList = &Emb->StateList;

    // global
    Emb->Global.StateList = &Emb->StateList;
    Emb->Global.PidList = &Emb->PidList;

    // clock
    Emb->Clock.Tmr = &Emb->Tmr;

    // assign my robot id
    Emb->StateList.State[EMB_MY_STATE].RobotId =
        embRobotIdState(MY_ROBOT_NAME);
}
```

```
/*
 * emb/Alg.h
 *
 * algorithm includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_ALG_H__

#ifndef __EMB_ALG_H__
#define __EMB_ALG_H__


// includes

#include "emb/Env.h"
#include "emb/Global.h"
#include "emb/State.h"
#include "emb/Types.h"
#include "lpc/Types.h"

```

```c
// defines

#define EMB_MAX_SIZE_ALG 10
#define EMB_MAX_BNDRY_SIZE_ALG 10
#define EMB_MAX_GRID_SIZE_ALG 10


// field of view struct

typedef struct
{
  pos_t Pos[2];
  int32_t Edge[2];
}
  embFov_s;

// field of view list

typedef struct
{
  int32_t HalfAngle[2];
  int32_t HalfEdge[2];

  embFov_s Fov[4];
}
  embFovList_s;


// algorithm struct
```

```
typedef struct
{
  int64_t PartH_PartP[4];
  int32_t GainInvH[4];

  int32_t OpticsInvA;
  int32_t OpticsB;
  int32_t PriorW;

  int32_t Nk[4][2];

  int32_t DeltaWayPosSat[4];
  int32_t DeltaWayNegSat[4];

  embFovList_s FovList[EMB_MAX_SIZE_STATE];

  embEnv_s *Env;
  embGlobal_s *Global;
  embStateList_s *StateList;
}
  embAlg_s;


// functions

void embInitAlg(embAlg_s *Alg);
void embConfigAlg(embAlg_s *Alg);
void embEnableAlg(embAlg_s *Alg);

```

```
void embClrAlg(embAlg_s *Alg);
void embDistAlg(embAlg_s *Alg);

// __EMB_ALG_H__

#endif
```

```c
/*
 * emb/Alg.c
 *
 * algorithm source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Alg.h"

#include "emb/Types.h"
#include "lpc/Math.h"
#include "lpc/Types.h"


// local defines

#define NUM_FOV_BNDRY_SEG 10
```

```
// local functions

void createFovEdges(embFov_s *Fov);
void calcFovList(embAlg_s *Alg);
void calcIntegrandArea(embAlg_s *Alg);
void calcIntegrandEdge(embAlg_s *Alg);
bool_t isPntInFov(const pos_t *Pnt,
                  const embFov_s *Fov);
bool_t isPntInEnv(const pos_t *Pnt,
                  const embEnv_s *Env);

// initialize algorithm

void embInitAlg(embAlg_s *Alg)
{ }


// configure algorithm

void embConfigAlg(embAlg_s *Alg)
{ }


// enable algorithm

void embEnableAlg(embAlg_s *Alg)
{ }


// distributed algorithm
```

```
void embDistAlg(embAlg_s *Alg)
{
  // local stack
  int32_t ii;
  pos_t Way[4];

  // clear PartH/PartP
  for(ii = 0;
      ii < sizeof(Alg->PartH_PartP)/sizeof(Alg->PartH_PartP
         [0]);
      ii++)
    {
      Alg->PartH_PartP[ii] = 0;
    }

  // calc new field of views based on states
  calcFovList(Alg);

  // for all grid points in environment
  calcIntegrandArea(Alg);

  // for all points on the field of view boundary
  calcIntegrandEdge(Alg);

  // update waypoints
  for(ii = 0;
      ii < sizeof(Way) / sizeof(Way[0]);
      ii++)
    {
```

```
      Way[ii] = -(int32_t)(Alg->PartH_PartP[ii] / Alg->GainInvH[ii]);
      Way[ii] = INT_MIN_MAX(Alg->DeltaWayNegSat[ii],
                            Alg->DeltaWayPosSat[ii],
                            Way[ii]);
      Way[ii] += Alg->Global->Way[ii];
    }
  embUpWayGlobal(Alg->Global,
                 Way);
}


// calc field of view list

void calcFovList(embAlg_s *Alg)
{
  // local stack
  int32_t ii;
  int32_t jj;
  int32_t kk;
  int32_t Cos;
  int32_t Sin;

  // calculate new fields of view
  for(ii = 0;
      ii < Alg->StateList->Size;
      ii++)
    {
      Cos = int32Cos(Alg->StateList->State[ii].Pos[EMB_YAW_STATE]);
```

```
Sin = int32Sin(Alg->StateList->State[ii].Pos[
    EMB_YAW_STATE]);

// use half angles to get edge vectors
for(jj = 0;
    jj < sizeof(Alg->FovList[0].HalfAngle)/sizeof(Alg->
        FovList[0].HalfAngle[0]);
    jj++)
  {
    Alg->FovList[ii].HalfEdge[jj] = int32Tan(Alg->
        FovList[ii].HalfAngle[jj]);
    Alg->FovList[ii].HalfEdge[jj] *= Alg->StateList->
        State[ii].Pos[EMB_Z_STATE];
    Alg->FovList[ii].HalfEdge[jj] >>= 10;;
  }

// calculate four vertices of field of view

Alg->FovList[ii].Fov[0].Pos[0] = Cos * Alg->FovList[ii
    ].HalfEdge[0];
Alg->FovList[ii].Fov[0].Pos[0] += Sin * Alg->FovList[ii
    ].HalfEdge[1];
Alg->FovList[ii].Fov[0].Pos[0] >>= 10;

Alg->FovList[ii].Fov[0].Pos[1] = Sin * Alg->FovList[ii
    ].HalfEdge[0];
Alg->FovList[ii].Fov[0].Pos[1] += -Cos * Alg->FovList[
    ii].HalfEdge[1];
Alg->FovList[ii].Fov[0].Pos[1] >>= 10;

```

```
Alg->FovList[ii].Fov[1].Pos[0] = Cos * Alg->FovList[ii].HalfEdge[0];
Alg->FovList[ii].Fov[1].Pos[0] += -Sin * Alg->FovList[ii].HalfEdge[1];
Alg->FovList[ii].Fov[1].Pos[0] >>= 10;

Alg->FovList[ii].Fov[1].Pos[1] = Sin * Alg->FovList[ii].HalfEdge[0];
Alg->FovList[ii].Fov[1].Pos[1] += Cos * Alg->FovList[ii].HalfEdge[1];
Alg->FovList[ii].Fov[1].Pos[1] >>= 10;

Alg->FovList[ii].Fov[2].Pos[0] = -Cos * Alg->FovList[ii].HalfEdge[0];
Alg->FovList[ii].Fov[2].Pos[0] += -Sin * Alg->FovList[ii].HalfEdge[1];
Alg->FovList[ii].Fov[2].Pos[0] >>= 10;

Alg->FovList[ii].Fov[2].Pos[1] = -Sin * Alg->FovList[ii].HalfEdge[0];
Alg->FovList[ii].Fov[2].Pos[1] += Cos * Alg->FovList[ii].HalfEdge[1];
Alg->FovList[ii].Fov[2].Pos[1] >>= 10;

Alg->FovList[ii].Fov[3].Pos[0] = -Cos * Alg->FovList[ii].HalfEdge[0];
Alg->FovList[ii].Fov[3].Pos[0] += Sin * Alg->FovList[ii].HalfEdge[1];
Alg->FovList[ii].Fov[3].Pos[0] >>= 10;

```

```
        Alg->FovList[ii].Fov[0].Pos[1] = -Sin * Alg->FovList[ii].HalfEdge[0];
        Alg->FovList[ii].Fov[0].Pos[1] += -Cos * Alg->FovList[ii].HalfEdge[1];
        Alg->FovList[ii].Fov[0].Pos[1] >>= 10;

        // create field of view edges
        createFovEdges(Alg->FovList[ii].Fov);

        // shift field of view under robot
        for(jj = 0;
            jj < sizeof(Alg->FovList[ii].Fov)/sizeof(Alg->FovList[ii].Fov[0]);
            jj++)
          {
            for(kk = 0;
                kk < sizeof(Alg->FovList[ii].Fov[0].Pos)/sizeof(Alg->FovList[ii].Fov[0].Pos[0]);
                kk++)
              {
                Alg->FovList[ii].Fov[jj].Pos[kk] += Alg->StateList->State[ii].Pos[kk];
              }
          }
    }
}


// integrand due to field of view area within environment

```

```
void calcIntegrandArea(embAlg_s *Alg)
{
  // local stack
  int32_t ii;
  int32_t jj;
  int64_t Integrand;
  int64_t h_Nq_i;
  int64_t h_Nq_j;

  // for all grid points in environment
  for(ii = 0;
      ii < Alg->Env->GridSize;
      ii++)
    {
      // if grid point is in my field of view
      if(isPntInFov(Alg->Env->Grid[ii].Pos,
                    Alg->FovList[EMB_MY_STATE].Fov))
        {
          // insert prior weight
          h_Nq_j = Alg->PriorW;

          // for all robots in team
          for(jj = 1;
              jj < Alg->StateList->Size;
              jj++)
            {
              // if grid point is in their field of view
              if(isPntInFov(Alg->Env->Grid[ii].Pos,
                            Alg->FovList[jj].Fov))
                {
```

```
                h_Nq_i = Alg->OpticsInvA;
                h_Nq_i /= (Alg->OpticsB - Alg->StateList->State[jj].Pos[EMB_Z_STATE]);
                h_Nq_i <<= 16;
                h_Nq_i /= (Alg->OpticsB - Alg->StateList->State[jj].Pos[EMB_Z_STATE]);
                h_Nq_j += h_Nq_i;
            }
        }

    // intermediate calculations
    h_Nq_i = Alg->OpticsInvA;
    h_Nq_i /= (Alg->OpticsB - Alg->StateList->State[EMB_MY_STATE].Pos[EMB_Z_STATE]);
    h_Nq_i <<= 16;
    h_Nq_i /= (Alg->OpticsB - Alg->StateList->State[EMB_MY_STATE].Pos[EMB_Z_STATE]);
    h_Nq_i += h_Nq_j;
    h_Nq_i = (1<<16) / h_Nq_i;

    // only affects Z position
    Integrand  = -2*Alg->OpticsInvA;
    Integrand *= Alg->Env->GridSpacing;
    Integrand *= Alg->Env->GridSpacing;
    Integrand *= h_Nq_i;
    Integrand *= h_Nq_i;
    Integrand /= (Alg->OpticsB - Alg->StateList->State[EMB_MY_STATE].Pos[EMB_Z_STATE]);
    Integrand /= (Alg->OpticsB - Alg->StateList->State[EMB_MY_STATE].Pos[EMB_Z_STATE]);
```

```
            Integrand /= (Alg->OpticsB - Alg->StateList->State[EMB_MY_STATE].Pos[EMB_Z_STATE]);

            // update summation of partials
            Alg->PartH_PartP[EMB_Z_STATE] += Integrand;
        }
    }
}


// integrand due to field of view edge within environment

void calcIntegrandEdge(embAlg_s *Alg)
{
  // local stack
  int32_t ii;
  int32_t jj;
  int32_t kk;
  int64_t h_Nq_i;
  int64_t h_Nq_j;
  int32_t SegLength[2];
  int32_t Cos;
  int32_t Sin;
  pos_t SegPnt[2];
  int32_t TempInt32[2];

  // create segments of field of view edge
  SegLength[0] = (Alg->FovList[EMB_MY_STATE].HalfEdge[1]>>9);
  SegLength[0] /= NUM_FOV_BNDRY_SEG;

```

```
SegLength[1] = (Alg->FovList[EMB_MY_STATE].HalfEdge[0]>>9);
SegLength[1] /= NUM_FOV_BNDRY_SEG;

Cos = int32Cos(Alg->StateList->State[EMB_MY_STATE].Pos[EMB_YAW_STATE]);
Sin = int32Sin(Alg->StateList->State[EMB_MY_STATE].Pos[EMB_YAW_STATE]);

// for all segments around field of view
for(ii = 0;
     ii < NUM_FOV_BNDRY_SEG;
     ii++)
  {
    // for all field of view edges
    for(jj = 0;
         jj < sizeof(Alg->FovList[EMB_MY_STATE].Fov) / sizeof(Alg->FovList[EMB_MY_STATE].Fov[0]);
         jj++)
      {
        // calculate coordinates of seg point
        for(kk = 0;
             kk < sizeof(Alg->FovList[EMB_MY_STATE].Fov[0].Edge) / sizeof(Alg->FovList[EMB_MY_STATE].Fov[0].Edge[0]);
             kk++)
          {
            SegPnt[kk]  = ii*Alg->FovList[EMB_MY_STATE].Fov[jj].Edge[kk];
            SegPnt[kk] /= NUM_FOV_BNDRY_SEG;
```

```
            SegPnt[kk] += Alg->FovList[EMB_MY_STATE].Fov[jj].Pos[kk];
          }

        // if seg point is in environment
        if(isPntInEnv(SegPnt,
                      Alg->Env))
          {
            // insert prior weight
            h_Nq_j = Alg->PriorW;

            // for all robots in team
            for(kk = 1;
                kk < Alg->StateList->Size;
                kk++)
              {
                // if seg point is in their field of view
                if(isPntInFov(SegPnt,
                              Alg->FovList[kk].Fov))
                  {
                    h_Nq_i = Alg->OpticsInvA;
                    h_Nq_i /= (Alg->OpticsB - Alg->StateList->State[kk].Pos[EMB_Z_STATE]);
                    h_Nq_i <<= 16;
                    h_Nq_i /= (Alg->OpticsB - Alg->StateList->State[kk].Pos[EMB_Z_STATE]);
                    h_Nq_j += h_Nq_i;
                  }
```

```
        }

        // intermediate calculations
        h_Nq_i = Alg->OpticsInvA;
        h_Nq_i /= (Alg->OpticsB - Alg->StateList->State[EMB_MY_STATE].Pos[EMB_Z_STATE]);
        h_Nq_i <<= 16;
        h_Nq_i /= (Alg->OpticsB - Alg->StateList->State[EMB_MY_STATE].Pos[EMB_Z_STATE]);
        h_Nq_i += h_Nq_j;
        h_Nq_i = (1<<16) / h_Nq_i;
        h_Nq_j = (1<<16) / h_Nq_j;

        // form common integrand terms
        TempInt32[0]  = h_Nq_i;
        TempInt32[0] -= h_Nq_j;
        TempInt32[0] *= SegLength[kk%2];

        // X effect
        TempInt32[1] = ((Cos*Alg->Nk[jj][0] - Sin*Alg->Nk[jj][1])>>10);
        Alg->PartH_PartP[EMB_X_STATE] += ((TempInt32[0]*TempInt32[1])>>10);

        // Y effect
        TempInt32[1] = ((Sin*Alg->Nk[jj][0] + Cos*Alg->Nk[jj][1])>>10);
        Alg->PartH_PartP[EMB_Y_STATE] += ((TempInt32[0]*TempInt32[1])>>10);

```

```
                // Z effect
                TempInt32[1] = ((INT_ABS(Alg->FovList[
                    EMB_MY_STATE].HalfAngle[0]*Alg->Nk[jj][0])
                                +INT_ABS(Alg->FovList[
                                    EMB_MY_STATE].HalfAngle[1]*
                                    Alg->Nk[jj][1]))>>10);
                Alg->PartH_PartP[EMB_Z_STATE] += ((TempInt32
                    [0]*TempInt32[1])>>10);

                // Yaw effect
                TempInt32[1] = ((SegPnt[0] - Alg->StateList->
                    State[EMB_MY_STATE].Pos[EMB_X_STATE])
                                *((-Sin*Alg->Nk[jj][0] - Cos*Alg
                                    ->Nk[jj][1])>>10)
                                +(SegPnt[1] - Alg->StateList->
                                    State[EMB_MY_STATE].Pos[
                                    EMB_Y_STATE])
                                *((Cos*Alg->Nk[jj][0] - Sin*Alg
                                    ->Nk[jj][1])>>10));
                Alg->PartH_PartP[EMB_YAW_STATE] -= ((TempInt32
                    [0]*TempInt32[1])>>10);
            }
        }
    }

}


// create edges from field of view points

```

```
void createFovEdges(embFov_s *Fov)
{
  // local stack
  int32_t ii;
  int32_t jj;

  // for all Fov points
  for(ii = 0;
      ii < 4;
      ii++)
    {
      for(jj = 0;
          jj < sizeof(Fov)/sizeof(Fov[0]);
          jj++)
        {
          Fov[ii].Edge[jj]  = Fov[(ii+1)%4].Pos[jj];
          Fov[ii].Edge[jj] -= Fov[ii].Pos[jj];
        }
    }
}



// is point in field of view?

bool_t isPntInFov(const pos_t *Pnt,
                  const embFov_s *Fov)
{
  // local stack
  int32_t ii;
```

```
int32_t Crossings;
int32_t T0;
int32_t T1;

// zero crossings
Crossings = 0;

// for all field of view edges
for(ii = 0;
     ii < sizeof(Fov) / sizeof(Fov[0]);
     ii++)
  {
     if(Fov[ii].Edge[0] != 0)
       {
          T0 = Pnt[0]-Fov[ii].Pos[0];
          T0 <<= 10;
          T0 /= Fov[ii].Edge[0];

          T1 = Fov[ii].Pos[1]-Pnt[1];
          T1 <<= 10;
          T1 += T0*Fov[ii].Edge[1];
       }
     else
       {
          T0=0;
          T1=0;
       }

     if((T0>0) && (T0<(1<<10)) && (T1>0))
       {
```

```
                Crossings++;
            }
        }

    // if odd num of crossings, return true
    return((bool_t)((Crossings%2)==1));
}


// is point in environment?

bool_t isPntInEnv(const pos_t *Pnt,
                  const embEnv_s *Env)
{
    // local stack
    int32_t ii;
    int32_t Crossings;
    int32_t T0;
    int32_t T1;

    // zero crossings
    Crossings = 0;

    // for all boundry entries
    for(ii = 0;
        ii < Env->BndrySize;
        ii++)
    {
        if(Env->Bndry[ii].Edge[0] != 0)
        {
```

```
            T0 = Pnt[0]-Env->Bndry[ii].Pos[0];
            T0 <<= 10;
            T0 /= Env->Bndry[ii].Edge[0];

            T1 = Env->Bndry[ii].Pos[1]-Pnt[1];
            T1 <<= 10;
            T1 += T0*Env->Bndry[ii].Edge[1];
          }
        else
          {
            T0=0;
            T1=0;
          }

        if((T0>0) && (T0<(1<<10)) && (T1>0))
          {
            Crossings++;
          }
      }

    // if odd num of crossings, return true
    return((bool_t)((Crossings%2)==1));
}
```

```
/*
 * emb/Ap.h
 *
 * AutoPilot includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_AP_H__

#ifndef __EMB_AP_H__
#define __EMB_AP_H__


// includes

#include "emb/Clock.h"
#include "emb/Env.h"
#include "emb/Gps.h"
#include "emb/Global.h"
#include "emb/Imu.h"
#include "emb/Mhop.h"
```

```
#include "emb/Nav.h"
#include "emb/Pid.h"
#include "emb/State.h"
#include "emb/Status.h"
#include "lpc/Types.h"
#include "lpc/Uart.h"
#include "lpc/Tmr.h"


// rx ap struct

typedef struct
{
  lpcMatchTmr_s Match;

  int32_t ParseIndex;
  uint32_t ByteCnt;
  crc16_t CalcCrc;
  crc16_t ReadCrc;
  byte_t PktId;
}
  embRxAp_s;


// tx ap struct

typedef struct
{
  lpcMatchTmr_s Match;

```

```
  crc16_t Crc;
  uint32_t ByteCnt;

  lpcBuf_s Buf;
  byte_t BufMem[LPC_TX_BUF_SIZE_UART];
}
  embTxAp_s;


// ap struct

typedef struct
{
  lpcTmr_s *Tmr;

  lpcUart_s Uart;

  embRxAp_s Rx;
  embTxAp_s Tx;

  embGps_s *Gps;
  embNav_s *Nav;
  embNav_s *Clock;
  embImu_s *Imu;
  embPidList_s *PidList;
  embStateList_s *StateList;
  embGlobal_s *Global;
  embStatus_s *Status;
}
  embAp_s;
```

```
// functions

void embInitAp(embAp_s *Ap);
void embConfigAp(embAp_s *Ap);
void embEnableAp(embAp_s *Ap);

void embInitRxAp(embAp_s *Ap);
void embConfigRxAp(embAp_s *Ap);
void embEnableRxAp(embAp_s *Ap);

void embInitTxAp(embAp_s *Ap);
void embConfigTxAp(embAp_s *Ap);
void embEnableTxAp(embAp_s *Ap);

void embRxAp(embAp_s *Ap);
void embTxAp(embAp_s *Ap);


// __EMB_AP_H__

#endif
```

```
/*
 * emb/Ap.c
 *
 * AutoPilot source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Ap.h"
#include "lpc/Cntl.h"
#include "lpc/Tmr.h"
#include "lpc/Types.h"
#include "lpc/Uart.h"


// local defines

#define AP_UART_NUM 1
#define AP_UART_BAUD_RATE 57600
#define AP_UART_FLOW_CNTL LPC_FLOW_CNTL_ON_UART
```

```
// local functions

void builtApStruct(embAp_s *Ap);
bool_t isUnreadByteInUart(embAp_s *Ap);


// initialize AutoPilot

void embInitAp(embAp_s *Ap)
{
  // initialize uart
  lpcInitUart(&Ap->Uart);

  // initialize rx ap
  embInitRxAp(Ap);

  // initialize tx ap
  embInitTxAp(Ap);

  // built ap struct
  builtApStruct(Ap);
}


// configure AutoPilot

void embConfigAp(embAp_s *Ap)
{
```

```
  // configure uart
  lpcSetNumUart(&Ap->Uart,
                AP_UART_NUM);

  lpcSetBaudRateUart(&Ap->Uart,
                     AP_UART_BAUD_RATE);

  lpcSetFlowCntlUart(&Ap->Uart,
                     AP_UART_FLOW_CNTL);

  // configure rx ap
  embConfigRxAp(Ap);

  // configure tx ap
  embConfigTxAp(Ap);
}


// enable AutoPilot

void embEnableAp(embAp_s *Ap)
{
  // enable uart
  lpcEnableUart(&Ap->Uart);

  // enable Tx ap
  embEnableRxAp(Ap);

  // enable Tx ap
  embEnableTxAp(Ap);
```

```
}


// builds default ap struct

void builtApStruct(embAp_s *Ap)
{ }
```

```
/*
 * emb/Clock.h
 *
 * clock includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_CLOCK_H__

#ifndef __EMB_CLOCK_H__
#define __EMB_CLOCK_H__


// includes

#include "emb/Types.h"
#include "lpc/Tmr.h"
#include "lpc/Types.h"


// clock struct
```

```c

typedef struct
{
  reg32_t Tc;
  clock_t Base;

  bool_t OverFlowChk;
  int32_t Accum;

  lpcTmr_s *Tmr;
}
  embClock_s;


// functions

void embInitClock(embClock_s *Clock);
void embConfigClock(embClock_s *Clock);
void embEnableClock(embClock_s *Clock);

clock_t embGetClock(embClock_s *Clock);
void embSyncClock(embClock_s *Clock,
                  clock_t Tic,
                  int32_t InvWeight);


// __EMB_CLOCK_H__

#endif
```

```
/*
 * emb/Clock.c
 *
 * clock source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Clock.h"
#include "emb/Types.h"
#include "lpc/Cntl.h"
#include "lpc/Tmr.h"
#include "lpc/Types.h"


// local defines

#define CCLK_DIV 59
#define DEF_BASE 0
#define CLOCK_OVER_FLOW (1<<31)
```

```
#define OVER_FLOW_VALUE 0xFFFFFFFF
// local functions
void  overFlowChk(embClock_s *Clock);
// initialize clock
void embInitClock(embClock_s *Clock)
{
  Clock->Base = DEF_BASE;
}
// configure clock
void embConfigClock(embClock_s *Clock)
{ }
// enable clock
void embEnableClock(embClock_s *Clock)
{ }
// get curr clock
```

```
clock_t embGetClock(embClock_s *Clock)
{
  // get tc
  lpcGetTcTmr(Clock->Tmr,
              &Clock->Tc);

  // check for overflow
  overFlowChk(Clock);

  // return clock
  return(Clock->Base + (Clock->Tc / CCLK_DIV));
}


// sync clock with received input

void embSyncClock(embClock_s *Clock,
                  clock_t Tic,
                  int32_t InvWeight)
{
  // local stack
  int32_t Drift;

  // get drift
  Drift = (int32_t)(Tic - embGetClock(Clock));
  Drift /= InvWeight;

  // adjust base and accumilator
  Clock->Base += Drift;
  Clock->Accum += Drift;
```

```
}


// check for tc overflow

void  overFlowChk(embClock_s *Clock)
{
  // if overflow possible
  if(Clock->Tc < CLOCK_OVER_FLOW)
    {
      if(Clock->OverFlowChk)
        {
          Clock->Base += (OVER_FLOW_VALUE / CCLK_DIV);
          Clock->OverFlowChk = false;
        }
    }

  // if not
  else
    {
      Clock->OverFlowChk = true;
    }
}
```

```
/*
 * emb/Comm.h
 *
 * communications includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_COMM_H__

#ifndef __EMB_COMM_H__
#define __EMB_COMM_H__


// includes

#include "emb/Clock.h"
#include "emb/Env.h"
#include "emb/Global.h"
#include "emb/Mhop.h"
#include "emb/Nav.h"
#include "emb/Pid.h"
```

```
#include "emb/State.h"
#include "emb/Status.h"
#include "lpc/Types.h"
#include "lpc/Uart.h"
#include "lpc/Tmr.h"


// rx comm struct

typedef struct
{
  lpcMatchTmr_s Match;

  int32_t ParseIndex;
  uint32_t ByteCnt;
  crc32_t CalcCrc;
  crc32_t ReadCrc;
  byte_t PktId;
}
  embRxComm_s;


// tx comm struct

typedef struct
{
  lpcMatchTmr_s Match;

  crc32_t Crc;
  uint32_t ByteCnt;
```

```

  lpcBuf_s Buf;
  byte_t BufMem[LPC_TX_BUF_SIZE_UART];
}
  embTxComm_s;


// comm struct

typedef struct
{
  lpcTmr_s *Tmr;

  lpcUart_s Uart;

  embRxComm_s Rx;
  embTxComm_s Tx;

  embClock_s *Clock;
  embEnv_s *Env;
  embGlobal_s *Global;
  embMhop_s *Mhop;
  embNav_s *Nav;
  embPidList_s *PidList;
  embStateList_s *StateList;
  embStatus_s *Status;

  bool_t InitLlCntl;
  bool_t InitHlCntl;
  bool_t InitLand;
```

```
    bool_t InitHome;
    bool_t InitEnv;
}
    embComm_s;


// functions

void embInitComm(embComm_s *Comm);
void embConfigComm(embComm_s *Comm);
void embEnableComm(embComm_s *Comm);

void embInitRxComm(embComm_s *Comm);
void embConfigRxComm(embComm_s *Comm);
void embEnableRxComm(embComm_s *Comm);

void embInitTxComm(embComm_s *Comm);
void embConfigTxComm(embComm_s *Comm);
void embEnableTxComm(embComm_s *Comm);

void embRxComm(embComm_s *Comm);
void embTxComm(embComm_s *Comm);


// __EMB_COMM_H__

#endif
```

```
/*
 * emb/Comm.c
 *
 * communications source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Comm.h"
#include "lpc/Cntl.h"
#include "lpc/Tmr.h"
#include "lpc/Types.h"
#include "lpc/Uart.h"


// local defines

#define COMM_UART_NUM 0
#define COMM_UART_BAUD_RATE 57600
#define COMM_UART_FLOW_CNTL LPC_FLOW_CNTL_ON_UART
```

```
// local functions

void builtCommStruct(embComm_s *Comm);
bool_t isUnreadByteInUart(embComm_s *Comm);


// initialize comm

void embInitComm(embComm_s *Comm)
{
  // initialize uart
  lpcInitUart(&Comm->Uart);

  // initialize rx comm
  embInitRxComm(Comm);

  // initialize tx comm
  embInitTxComm(Comm);

  // built comm struct
  builtCommStruct(Comm);
}


// configure comm

void embConfigComm(embComm_s *Comm)
{
```

```
  // configure uart
  lpcSetNumUart(&Comm->Uart,
                COMM_UART_NUM);

  lpcSetBaudRateUart(&Comm->Uart,
                     COMM_UART_BAUD_RATE);

  lpcSetFlowCntlUart(&Comm->Uart,
                     COMM_UART_FLOW_CNTL);

  // configure rx comm
  embConfigRxComm(Comm);

  // configure tx comm
  embConfigTxComm(Comm);
}


// enable comm

void embEnableComm(embComm_s *Comm)
{
  // enable uart
  lpcEnableUart(&Comm->Uart);

  // enable Tx comm
  embEnableRxComm(Comm);

  // enable Tx comm
  embEnableTxComm(Comm);
```

```
}


// builds default comm struct

void builtCommStruct(embComm_s *Comm)
{
  Comm->InitLlCntl = false;
  Comm->InitHlCntl = false;
  Comm->InitLand = false;
  Comm->InitHome = false;
  Comm->InitEnv = false;
}
```

```
/*
 * emb/Crc16.h
 *
 * crc16 includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_CRC16_H__

#ifndef __EMB_CRC16_H__
#define __EMB_CRC16_H__


// includes

#include "emb/Types.h"
#include "lpc/Types.h"


// defines

```

```
#define EMB_CLR_CRC16 0xFF


// functions

void embUpCrc16(crc16_t *Crc,
                byte_t Byte);


// __EMB_CRC16_H__

#endif
```

```
/*
 * emb/Crc16.c
 *
 * crc16 source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Crc16.h"
#include "emb/Types.h"
#include "lpc/Types.h"


// updates crc16

void embUpCrc16(crc16_t *Crc,
                byte_t Byte)
{
  // update
  Byte ^= ((*Crc) & 0xFF);
```

```
    Byte ^= (Byte << 4);

    *Crc >>= 8;
    *Crc &= 0xFF;
    *Crc |= (((crc16_t)Byte) << 8);
    *Crc ^= (((byte_t)(Byte)) >> 4);
    *Crc ^= (((crc16_t)(Byte)) << 3);
}
```

```
/*
 * emb/Crc32.h
 *
 * crc32 includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_CRC32_H__

#ifndef __EMB_CRC32_H__
#define __EMB_CRC32_H__


// includes

#include "emb/Types.h"
#include "lpc/Types.h"


// defines

```

```
#define EMB_CLR_CRC32 0x00000000


// functions

void embUpCrc32(crc32_t *Crc,
                byte_t Byte);


// __EMB_CRC32_H__

#endif
```

```
/*
 * emb/Crc32.c
 *
 * crc32 source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Crc32.h"
#include "emb/Types.h"
#include "lpc/Types.h"


// local defines

#define CRC32_POLY 0xEDB88320
#define CRC32_FLIP 0xFFFFFFFF


// updates crc32
```

```

void embUpCrc32(crc32_t *Crc,
                byte_t Byte)
{
  // local stack
  crc32_t Temp;
  int32_t ii;

  // update
  *Crc ^= CRC32_FLIP;
  Temp = (*Crc ^ Byte);
  Temp &= 0xff;

  for(ii = 0;
      ii < 8;
      ii++)
    {
      if((Temp & 1) == 1)
        {
          Temp >>= 1;
          Temp ^= CRC32_POLY;
        }
      else
        {
          Temp >>= 1;
        }
    }

  *Crc >>= 8;
  *Crc ^= Temp;
```

```
    *Crc ^= CRC32_FLIP;
}
```

```c
/*
 * emb/Env.h
 *
 * environment includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_ENV_H__

#ifndef __EMB_ENV_H__
#define __EMB_ENV_H__


// includes

#include "emb/Types.h"
#include "lpc/Types.h"


// defines

```

```
#define EMB_MAX_SIZE_ENV 10
#define EMB_MAX_BNDRY_SIZE_ENV 10
#define EMB_MAX_GRID_SIZE_ENV 10


// boundry struct

typedef struct
{
  pos_t Pos[2];
  int32_t Edge[2];
}
  embBndry_s;


// grid struct

typedef struct
{
  pos_t Pos[2];
  int32_t Imp;
}
  embGrid_s;


// environmnet struct

typedef struct
{
  int32_t BndrySize;
```

```
    embBndry_s Bndry[EMB_MAX_BNDRY_SIZE_ENV];

    int32_t GridSize;
    int32_t GridSpacing;
    embGrid_s Grid[EMB_MAX_GRID_SIZE_ENV];
}
    embEnv_s;


// functions

void embInitEnv(embEnv_s *Env);
void embConfigEnv(embEnv_s *Env);
void embEnableEnv(embEnv_s *Env);

void embClrEnv(embEnv_s *Env);


// __EMB_ENV_H__

#endif
```

```c
/*
 * emb/Env.c
 *
 * environment source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Env.h"
#include "emb/Types.h"
#include "lpc/Types.h"


// initialize environment

void embInitEnv(embEnv_s *Env)
{
  // clear env
  embClrEnv(Env);
}
```

```c


// configure environment

void embConfigEnv(embEnv_s *Env)
{ }


// enable environment

void embEnableEnv(embEnv_s *Env)
{ }


// clear environment

void embClrEnv(embEnv_s *Env)
{
  // local stack
  int32_t ii;
  int32_t jj;

  // clear sizes
  Env->BndrySize = 0;
  Env->GridSize = 0;

  // clear bndry
  for(ii = 0;
        ii < EMB_MAX_BNDRY_SIZE_ENV;
        ii++)
```

```
    {
      for(jj = 0;
          jj < sizeof(Env->Bndry[0].Pos)/sizeof(Env->Bndry
              [0].Pos[0]);
          jj++)
        {
          Env->Bndry[ii].Pos[jj] = 0;
        }

      for(jj = 0;
          jj < sizeof(Env->Bndry[0].Edge)/sizeof(Env->Bndry
              [0].Edge[0]);
          jj++)
        {
          Env->Bndry[ii].Edge[jj] = 0;
        }
    }

  // clear grid
  for(ii = 0;
      ii < EMB_MAX_GRID_SIZE_ENV;
      ii++)
    {
      Env->Grid[ii].Imp = 0;

      for(jj = 0;
          jj < sizeof(Env->Grid[0].Pos)/sizeof(Env->Grid[0].
              Pos[0]);
          jj++)
        {
```

```
            Env->Grid[ii].Pos[jj] = 0;
        }
    }
}


// create edges from acquired bndry positions

void embCreateBndryEdgesEnv(embEnv_s *Env)
{
  // local stack
  int32_t ii;
  int32_t jj;

  // for all bndry points
  for(ii = 0;
      ii < Env->BndrySize;
      ii++)
    {
      for(jj = 0;
          jj < sizeof(Env->Bndry[0].Pos)/sizeof(Env->Bndry[0].Pos[0]);
          jj++)
        {
          Env->Bndry[ii].Edge[jj]  = Env->Bndry[(ii+1)%Env->BndrySize].Pos[jj];
          Env->Bndry[ii].Edge[jj] -= Env->Bndry[ii].Pos[jj];
        }
    }
}
```

```
/*
 * emb/Global.h
 *
 * global position includes for quadrotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_GLOBAL_H__

#ifndef __EMB_GLOBAL_H__
#define __EMB_GLOBAL_H__


// includes

#include "emb/Pid.h"
#include "emb/State.h"
#include "emb/Types.h"
#include "lpc/Types.h"


```

```
// global struct

typedef struct
{
  clock_t TimeStamp;

  pos_t So3[6];
  pos_t Yaw;

  pos_t Way[4];
  pos_t OldWay[4];
  bool_t WayActive;
  pos_t Err[4];

  int32_t R_b[9];

  embPidList_s *PidList;
  embStateList_s *StateList;
}
  embGlobal_s;


// functions

void embInitGlobal(embGlobal_s *Global);
void embConfigGlobal(embGlobal_s *Global);
void embEnableGlobal(embGlobal_s *Global);

void embUpGlobal(embGlobal_s *Global,
                 const embGlobal_s *NewGlobal);
```

```
void embUpWayGlobal(embGlobal_s *Global,
                    const pos_t *Way);
void embUpPidFromGlobal(const embGlobal_s *Global);
void embUpStateFromGlobal(const embGlobal_s *Global);


// __EMB_GLOBAL_H__

#endif
```

```c
/*
 * emb/Global.c
 *
 * global pos source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Global.h"
#include "emb/State.h"
#include "emb/Types.h"
#include "lpc/Math.h"
#include "lpc/Types.h"


// initialize global position

void embInitGlobal(embGlobal_s *Global)
{ }

```

```

// configure global position

void embConfigGlobal(embGlobal_s *Global)
{ }


// enable global position

void embEnableGlobal(embGlobal_s *Global)
{ }


// update waypoint position

void embUpWayGlobal(embGlobal_s *Global,
                    const pos_t *Way)
{
  // local stack
  int32_t ii;

  // update waypoint
  for(ii = 0;
      ii < sizeof(Global->Way)/sizeof(Global->Way[0]);
      ii++)
    {
      if(Global->WayActive)
        {
          Global->OldWay[ii] = Global->Way[ii];
        }
```

```
      Global->Way[ii] = Way[ii];
    }
  Global->WayActive = false;
}


// update global position

void embUpGlobal(embGlobal_s *Global,
                 const embGlobal_s *NewGlobal)
{
  // local stack
  int32_t Psi;
  int32_t SinPsi;
  int32_t CosPsi;
  int32_t T_c;
  int32_t ii;

  // update time stamp
  Global->TimeStamp = NewGlobal->TimeStamp;

  // update so3
  for(ii = 0;
      ii < sizeof(NewGlobal->So3)/sizeof(NewGlobal->So3[0]);
      ii++)
    {
      Global->So3[ii] = NewGlobal->So3[ii];
    }

  // matrix params
```

```
Psi    = int32Pow(Global->So3[3],
                   2);
Psi   += int32Pow(Global->So3[4],
                   2);
Psi   += int32Pow(Global->So3[5],
                   2);
Psi    = int32Sqrt(Psi);

CosPsi = int32Cos(Psi);
SinPsi = int32Sin(Psi);

T_c = (1<<10) - CosPsi;

// normalize so3 angles
for( ii = 3;
     ii < 6;
     ii++)
  {
    Global->So3[ii] <<= 10;
    Global->So3[ii] /= Psi;
  }

// rotational matrix
Global->R_b[0] =  ((((T_c*Global->So3[4]*Global->So3[3]) >>
    10) + (Global->So3[5]*SinPsi)) >> 10);
Global->R_b[1] =  ((((T_c*Global->So3[4]*Global->So3[4]) >>
    20) + CosPsi));
Global->R_b[2] =  ((((T_c*Global->So3[4]*Global->So3[5]) >>
    10) - (Global->So3[3]*SinPsi)) >> 10);

```

```
    Global->R_b[3] =  ((((T_c*Global->So3[3]*Global->So3[3]) >>
        20) + CosPsi));
    Global->R_b[4] =  ((((T_c*Global->So3[3]*Global->So3[4]) >>
        10) - (Global->So3[5]*SinPsi)) >> 10);
    Global->R_b[5] =  ((((T_c*Global->So3[3]*Global->So3[5]) >>
        10) + (Global->So3[4]*SinPsi)) >> 10);

    Global->R_b[6] = -((((T_c*Global->So3[5]*Global->So3[3]) >>
        10) - (Global->So3[4]*SinPsi)) >> 10);
    Global->R_b[7] = -((((T_c*Global->So3[5]*Global->So3[4]) >>
        10) + (Global->So3[3]*SinPsi)) >> 10);
    Global->R_b[8] = -((((T_c*Global->So3[5]*Global->So3[5]) >>
        20) + CosPsi));

    // update yaw
    Global->Yaw = int32Atan2(-Global->R_b[0],
                             Global->R_b[3]);

    // update way error
    for(ii = 0;
        ii < 3;
        ii++)
      {
        Global->Err[ii] = Global->Way[ii] - Global->So3[ii];
      }
    Global->Err[3] = Global->Way[3] - Global->Yaw;
}
```

```
void embUpStateFromGlobal(const embGlobal_s *Global)
{
  // local stack
  int32_t ii;

  // update my state and prepare body coord
  for(ii = 0;
      ii < 3;
      ii++)
    {
      Global->StateList->State[EMB_MY_STATE].Pos[ii] = Global
          ->So3[ii];
    }
  Global->StateList->State[EMB_MY_STATE].Pos[3] = Global->Yaw
    ;
}


void embUpPidFromGlobal(const embGlobal_s *Global)
{
  // local stack
  int32_t ii;
  int32_t jj;

  // update my body coords
  for(ii = 0;
      ii < 3;
      ii++)
    {
      // store prev coords
```

```
            Global->PidList->Pid[ii].PosB[1] = Global->PidList->Pid[ii].PosB[0];
            Global->PidList->Pid[ii].ErrB[1] = Global->PidList->Pid[ii].ErrB[0];

            // zero curr coords
            Global->PidList->Pid[ii].PosB[0] = 0;
            Global->PidList->Pid[ii].ErrB[0] = 0;

            // calc curr coords
            for(jj = 0;
                jj < 3;
                jj++)
              {
                Global->PidList->Pid[ii].PosB[0] += Global->R_b[3*ii+jj]*Global->So3[jj];
                Global->PidList->Pid[ii].ErrB[0] += Global->R_b[3*ii+jj]*Global->Err[jj];
              }

            Global->PidList->Pid[ii].PosB[0] >>= 10;
            Global->PidList->Pid[ii].ErrB[0] >>= 10;
        }
    Global->PidList->Pid[3].PosB[0] = -Global->Yaw;
    Global->PidList->Pid[3].ErrB[0] = -Global->Err[3];
}
```

```
/*
 * emb/Nav.h
 *
 * navigation includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_NAV_H__

#ifndef __EMB_NAV_H__
#define __EMB_NAV_H__


// includes

#include "emb/Status.h"
#include "emb/Types.h"
#include "lpc/Eint.h"
#include "lpc/Types.h"


```

```
// navigation flight enum

typedef enum
  {
    NAV_Indoor = true,
    NAV_Outdoor = false
  }
  embFlight_e;


// nav struct

typedef struct
{
  lpcEint_s Eint;
  embFlight_e Flight;

  embStatus_s *Status;
}
  embNav_s;


// functions

void embInitNav(embNav_s *Nav);
void embConfigNav(embNav_s *Nav);
void embEnableNav(embNav_s *Nav);


// __EMB_NAV_H__
```

```
#endif
```

```c
/*
 * emb/Nav.c
 *
 * navigation source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Nav.h"
#include "emb/Status.h"
#include "emb/Types.h"
#include "lpc/Eint.h"
#include "lpc/Types.h"


// local defines

#define NAV_EINT_PIN 7


```

```
// global pointer

embNav_s *LinkedNav;


// local functions

void initNavStruct(embNav_s *Nav);
void navEintVectAddr(void) __attribute__ ((interrupt("IRQ")))
    ;


// initialize navigation

void embInitNav(embNav_s *Nav)
{
  // create internal link for interrupt
  LinkedNav = Nav;

  // initialize external interrupts
  lpcInitEint(&Nav->Eint);
}


// configure navigation

void embConfigNav(embNav_s *Nav)
{
  // configure navigation external interrupt
  lpcSetPinEint(&Nav->Eint,
```

```
                    NAV_EINT_PIN);
  lpcSetVectAddrEint(&Nav->Eint,
                    (reg32_t)navEintVectAddr);
}


// enable navigation

void embEnableNav(embNav_s *Nav)
{
  // enable external interrupts
  lpcEnableEint(&Nav->Eint);
}


// eint handle

void navEintVectAddr(void)
{
  // local stack
  embNav_s *Nav;

  // assigned linked nav
  Nav = LinkedNav;

  // change nav state
  lpcReadPinEint(&Nav->Eint,
                    (bool_t *)&Nav->Flight);

  switch(Nav->Flight)
```

```
        {
        case(NAV_Indoor):
            Nav->Status->Green.Speed = LED_Fast;
            break;
        case(NAV_Outdoor):
            Nav->Status->Green.Speed = LED_Slow;
            break;
        default:
            break;
        }

    // clear interrupt request
    lpcClearIrEint(&Nav->Eint);

    // reset vic
    lpcResetVicEint(&Nav->Eint);
}
```

```
/*
 * emb/RxAp.c
 *
 * rx apunications source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Clock.h"
#include "emb/Ap.h"
#include "emb/Crc16.h"
#include "emb/Gps.h"
#include "emb/Imu.h"
#include "emb/State.h"
#include "emb/Status.h"
#include "emb/Types.h"
#include "lpc/Cntl.h"
#include "lpc/Math.h"
#include "lpc/Types.h"
#include "lpc/Uart.h"
```

```
// local defines

#define RX_AP_PERIOD (CCLK/10000)
#define RX_MATCH_SLOT 2

#define CRC_RST EMB_CLR_CRC16

#define AP_RANGE 30000

#define START_STR ">*>"
#define BYTE_CNT_MIN 0
#define BYTE_CNT_MAX 100
#define STATE_SENDER_INDEX 0
#define RST_PARSE_INDEX 0

#define GPS_PKT_ID 0x29
#define IMU_PKT_ID 0x03

#define START_STR_PARSE_INDEX 1
#define BYTE_CNT_PARSE_INDEX (START_STR_PARSE_INDEX+1)
#define PKT_ID_PARSE_INDEX (BYTE_CNT_PARSE_INDEX+sizeof(
    int16_t))
#define GPS_PKT_PARSE_INDEX 1000
#define IMU_PKT_PARSE_INDEX 2000


// local functions

```

```c
void initRxStructAp(embAp_s *Ap);
void discardApCrc(embAp_s *Ap);
bool_t isApCrcValid(embAp_s *Ap);
bool_t isInApRange(embAp_s *Ap,
                   embState_s *State);
bool_t isUnreadByteInApUart(embAp_s *Ap);
void parseApByteCnt(embAp_s *Ap);
void parseApGpsPkt(embAp_s *Ap);
void parseApImuPkt(embAp_s *Ap);
void parseApPktId(embAp_s *Ap);
void parseApStartStr(embAp_s *Ap);


// init rx ap

void embInitRxAp(embAp_s *Ap)
{
  // build rx structure
  initRxStructAp(Ap);

  // initialize sys match timer
  lpcInitMatchTmr(&Ap->Rx.Match,
                  Ap->Tmr);
}


// config rx ap

void embConfigRxAp(embAp_s *Ap)
{
```

```
  // configure rx match
  lpcSetSlotMatchTmr(&Ap->Rx.Match,
                     RX_MATCH_SLOT);
}


// enable rx ap

void embEnableRxAp(embAp_s *Ap)
{
  // enable rx match
  lpcEnableMatchTmr(&Ap->Rx.Match);

  // reset rx match
  lpcResetMatchTmr(&Ap->Rx.Match,
                   CCLK);
}


// receive ap handle

void embRxAp(embAp_s *Ap)
{
  // while unread bytes in buf
  while(isUnreadByteInApUart(Ap))
    {
      // update parse index and figure out which case
      switch(++(Ap->Rx.ParseIndex))
        {
          // if start string index
```

```
            case(START_STR_PARSE_INDEX):
                parseApStartStr(Ap);
                break;

                // if byte count index
            case(BYTE_CNT_PARSE_INDEX):
                parseApByteCnt(Ap);
                break;

                // if packet id index
            case(PKT_ID_PARSE_INDEX):
                parseApPktId(Ap);
                break;

                // if environment packet index
            case(GPS_PKT_PARSE_INDEX):
                parseApGpsPkt(Ap);
                break;

                // if homing index
            case(IMU_PKT_PARSE_INDEX):
                parseApImuPkt(Ap);
                break;
            }
    }

// reset rx ap match
lpcResetMatchTmr(&Ap->Rx.Match,
                 RX_AP_PERIOD);

```

```c
  // clear rx ap match IR
  lpcClrIrMatchTmr(&Ap->Rx.Match);
}


// builds rx ap struct

void initRxStructAp(embAp_s *Ap)
{ }


// checks if calculated crc is same as read one

bool_t isApCrcValid(embAp_s *Ap)
{
  // local stack
  int32_t ii;
  byte_t ReadByte;
  bool_t DummyBool;
  int32_t BytesRecv;

  // unread packet for crc
  lpcOffsetReadUart(&Ap->Uart,
                    -Ap->Rx.ByteCnt);

  // reset crcs
  Ap->Rx.CalcCrc = CRC_RST;
  Ap->Rx.ReadCrc = CRC_RST;

  // calculate crc of packet
```

```
  for( ii = 0;
       ii < Ap->Rx.ByteCnt - sizeof(Ap->Rx.CalcCrc);
       ii++)
    {
      lpcReadByteUart(&Ap->Uart,
                      &ReadByte,
                      &DummyBool);
      embUpCrc16(&Ap->Rx.CalcCrc,
                 ReadByte);
    }

  // read crc from packet
  LPC_READ_UART(Ap->Uart,
                Ap->Rx.ReadCrc,
                BytesRecv);

  // return match comparison
  return((bool_t)(Ap->Rx.CalcCrc == Ap->Rx.ReadCrc));
}


// check for valid start string at beginning of packet

void parseApStartStr(embAp_s *Ap)
{
  // local stack
  int32_t ii;
  bool_t ValidStartStr;
  bool_t DummyBool;
  byte_t ReadByte;
```

```
  // unread bytes for start string comparison
  lpcOffsetReadUart(&Ap->Uart,
                    -(sizeof(START_STR)-1));

  // verify start string
  for(ii = 0, ValidStartStr = true;
      ii < sizeof(START_STR)-1;
      ii++)
    {
      lpcReadByteUart(&Ap->Uart,
                      &ReadByte,
                      &DummyBool);

      ValidStartStr = ValidStartStr && ((bool_t)(ReadByte == START_STR[ii]));
    }

  // reset parse if not valid start string
  if(!ValidStartStr)
    {
      Ap->Rx.ParseIndex = RST_PARSE_INDEX;
    }
}


// parse byte count from packet

void parseApByteCnt(embAp_s *Ap)
{
```

```
  // local stack
  int32_t BytesRecv;

  // read byte count from uart
  lpcOffsetReadUart(&Ap->Uart,
                    -sizeof(Ap->Rx.ByteCnt));
  LPC_READ_UART(Ap->Uart,
                Ap->Rx.ByteCnt,
                BytesRecv);
}


// parse packet id from packet

void parseApPktId(embAp_s *Ap)
{
  // local stack
  bool_t DummyBool;
  bool_t ValidByteCnt;

  // read byte count from uart
  lpcOffsetReadUart(&Ap->Uart,
                    -sizeof(byte_t));
  lpcReadByteUart(&Ap->Uart,
                  &Ap->Rx.PktId,
                  &DummyBool);

  // check if valid byte count
  ValidByteCnt = (bool_t)((Ap->Rx.ByteCnt >= BYTE_CNT_MIN) &&
                          (Ap->Rx.ByteCnt <= BYTE_CNT_MAX));
```

```
// set parse index if valid byte count
if(ValidByteCnt)
  {
    // assign corresponding parse index for valid packet id
    switch(Ap->Rx.PktId)
      {
        // if gps packet
      case(GPS_PKT_ID):
        Ap->Rx.ParseIndex = GPS_PKT_PARSE_INDEX;
        Ap->Rx.ParseIndex -= Ap->Rx.ByteCnt;
        break;

        // if imu packet
      case(IMU_PKT_ID):
        Ap->Rx.ParseIndex = IMU_PKT_PARSE_INDEX;
        Ap->Rx.ParseIndex -= Ap->Rx.ByteCnt;
        break;

        // reset if not valid packet
      default:
        Ap->Rx.ParseIndex = RST_PARSE_INDEX;
        break;
      }
  }

// reset parse index if not valid byte count
else
  {
    Ap->Rx.ParseIndex = RST_PARSE_INDEX;
```

```
    }
}


// parse gps packet

void parseApGpsPkt(embAp_s *Ap)
{
  // local stack
  int32_t BytesRecv;

  // if valid checksum
  if(isApCrcValid(Ap))
    {
      // unread packet for gps information
      lpcOffsetReadUart(&Ap->Uart,
                                -Ap->Rx.ByteCnt);

      // read data
      LPC_READ_UART(Ap->Uart,
                          Ap->Gps->Latitude,
                          BytesRecv);

      LPC_READ_UART(Ap->Uart,
                          Ap->Gps->Longitude,
                          BytesRecv);

      LPC_READ_UART(Ap->Uart,
                          Ap->Gps->Height,
                          BytesRecv);
```

```
    LPC_READ_UART(Ap->Uart,
                  Ap->Gps->SpeedX,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Gps->SpeedY,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Gps->Heading,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Gps->HorizontalAccuracy,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Gps->VerticalAccuracy,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Gps->SpeedAccuracy,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Gps->NumSV,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
```

```
                        Ap->Gps->Status,
                        BytesRecv);

            LPC_READ_UART(Ap->Uart,
                        Ap->Gps->BestEstimateLatitude,
                        BytesRecv);

            LPC_READ_UART(Ap->Uart,
                        Ap->Gps->BestEstimateLongitude,
                        BytesRecv);

            LPC_READ_UART(Ap->Uart,
                        Ap->Gps->BestEstimateSpeedX,
                        BytesRecv);

            LPC_READ_UART(Ap->Uart,
                        Ap->Gps->BestEstimateSpeedY,
                        BytesRecv);

            // discard trailing checksum
            discardApCrc(Ap);
        }

    // reset parse index
    Ap->Rx.ParseIndex = RST_PARSE_INDEX;
}


// parse imu packet

```

```
void parseApImuPkt(embAp_s *Ap)
{
  // local stack
  int32_t BytesRecv;

  // if valid checksum
  if(isApCrcValid(Ap))
    {
      // unread packet for imu information
      lpcOffsetReadUart(&Ap->Uart,
                        -Ap->Rx.ByteCnt);

      // read data
      LPC_READ_UART(Ap->Uart,
                    Ap->Imu->AngleNick,
                    BytesRecv);

      LPC_READ_UART(Ap->Uart,
                    Ap->Imu->AngleRoll,
                    BytesRecv);

      LPC_READ_UART(Ap->Uart,
                    Ap->Imu->AngleYaw,
                    BytesRecv);

      LPC_READ_UART(Ap->Uart,
                    Ap->Imu->AngVelNick,
                    BytesRecv);

      LPC_READ_UART(Ap->Uart,
```

```
                        Ap->Imu->AngVelRoll,
                        BytesRecv);

        LPC_READ_UART(Ap->Uart,
                        Ap->Imu->AngVelYaw,
                        BytesRecv);

        LPC_READ_UART(Ap->Uart,
                        Ap->Imu->CalibAccX,
                        BytesRecv);

        LPC_READ_UART(Ap->Uart,
                        Ap->Imu->CalibAccY,
                        BytesRecv);

        LPC_READ_UART(Ap->Uart,
                        Ap->Imu->CalibAccZ,
                        BytesRecv);

        LPC_READ_UART(Ap->Uart,
                        Ap->Imu->AccX,
                        BytesRecv);

        LPC_READ_UART(Ap->Uart,
                        Ap->Imu->AccY,
                        BytesRecv);

        LPC_READ_UART(Ap->Uart,
                        Ap->Imu->AccZ,
                        BytesRecv);
```

```
    LPC_READ_UART(Ap->Uart,
                  Ap->Imu->AccAngleNick,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Imu->AccAngleRoll,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Imu->AbsAcc,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Imu->Hx,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Imu->Hy,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Imu->Hz,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
                  Ap->Imu->MagHeading,
                  BytesRecv);

    LPC_READ_UART(Ap->Uart,
```

```
                    Ap->Imu->SpeedX,
                    BytesRecv);

    LPC_READ_UART(Ap->Uart,
                    Ap->Imu->SpeedY,
                    BytesRecv);

    LPC_READ_UART(Ap->Uart,
                    Ap->Imu->SpeedZ,
                    BytesRecv);

    LPC_READ_UART(Ap->Uart,
                    Ap->Imu->Height,
                    BytesRecv);

    LPC_READ_UART(Ap->Uart,
                    Ap->Imu->dHeight,
                    BytesRecv);

    LPC_READ_UART(Ap->Uart,
                    Ap->Imu->dHeightRef,
                    BytesRecv);

    LPC_READ_UART(Ap->Uart,
                    Ap->Imu->HeightRef,
                    BytesRecv);

    // discard trailing checksum
    discardApCrc(Ap);
}
```

```c

  // reset parse index
  Ap->Rx.ParseIndex = RST_PARSE_INDEX;
}


// discard trailing checksum

void discardApCrc(embAp_s *Ap)
{
  // local stack
  crc16_t Crc;
  int32_t BytesRecv;

  // discard
  LPC_READ_UART(Ap->Uart,
                Crc,
                BytesRecv);
}


// check if in ap range

bool_t isInApRange(embAp_s *Ap,
                   embState_s *State)
{
  // local stack
  int32_t Dist;

  // calc distance to sender
```

```
  Dist = int32Dist(Ap->StateList->State[EMB_MY_STATE].Pos,
                   State->Pos,
                   3);

  // if in ap range, return true, else false
  if(Dist < AP_RANGE)
    {
      return(true);
    }
  else
    {
      return(false);
    }
}

// see if unread bytes still in uart

bool_t isUnreadByteInApUart(embAp_s *Ap)
{
  // local stack
  bool_t ByteRecv;
  byte_t DummyByte;

  // function
  lpcReadByteUart(&Ap->Uart,
                  &DummyByte,
                  &ByteRecv);

  // return bool
  return(ByteRecv);
```

```
}
```

```
/*
 * emb/RxComm.c
 *
 * rx communications source code for the drl quad-rotor
     project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Clock.h"
#include "emb/Comm.h"
#include "emb/Crc32.h"
#include "emb/State.h"
#include "emb/Status.h"
#include "emb/Types.h"

#include "lpc/Cntl.h"
#include "lpc/Math.h"
#include "lpc/Types.h"
#include "lpc/Uart.h"
```

```
// local defines

#define RX_COMM_PERIOD (CCLK/10000)
#define RX_MATCH_SLOT 0

#define CRC_RST EMB_CLR_CRC32

#define COMM_RANGE 30000

#define START_STR ">*>"
#define BYTE_CNT_MIN 0
#define BYTE_CNT_MAX 100
#define STATE_SENDER_INDEX 0
#define RST_PARSE_INDEX 0

#define HL_CNTL_PKT_ID 'A'
#define GLOBAL_PKT_ID 'G'
#define LAND_PKT_ID 'L'
#define WAY_PKT_ID 'W'
#define LL_CNTL_PKT_ID 'D'
#define STATE_PKT_ID 'S'
#define ENV_PKT_ID 'E'
#define HOME_PKT_ID 'H'

#define LL_CNTL_PKT_LOOP_CNT 4
#define LAND_PKT_LOOP_CNT 4
#define HOME_PKT_LOOP_CNT 4
#define STATE_PKT_LOOP_CNT 24
```

```
#define ENV_PKT_LOOP_CNT 8
#define WAY_PKT_LOOP_CNT 20
#define GLOBAL_PKT_LOOP_CNT 32
#define HL_CNTL_PKT_LOOP_CNT 4

#define START_STR_PARSE_INDEX 1
#define PKT_ID_PARSE_INDEX (START_STR_PARSE_INDEX+1)
#define BYTE_CNT_PARSE_INDEX (PKT_ID_PARSE_INDEX+sizeof(
    int32_t))
#define HL_CNTL_PKT_PARSE_INDEX 1000
#define GLOBAL_PKT_PARSE_INDEX 2000
#define LAND_PKT_PARSE_INDEX 3000
#define WAY_PKT_PARSE_INDEX 4000
#define LL_CNTL_PKT_PARSE_INDEX 5000
#define STATE_PKT_PARSE_INDEX 6000
#define ENV_PKT_PARSE_INDEX 7000
#define HOME_PKT_PARSE_INDEX 8000


// local functions

void initRxStruct(embComm_s *Comm);
void discardCrc(embComm_s *Comm);
bool_t isCrcValid(embComm_s *Comm);
bool_t isInCommRange(embComm_s *Comm,
                         embState_s *State);
bool_t isUnreadByteInUart(embComm_s *Comm);
void parseByteCnt(embComm_s *Comm);
void parseEnvPkt(embComm_s *Comm);
void parseGlobalPkt(embComm_s *Comm);
```

```
void parseHlCntlPkt(embComm_s *Comm);
void parseHomePkt(embComm_s *Comm);
void parseLandPkt(embComm_s *Comm);
void parseLlCntlPkt(embComm_s *Comm);
void parsePktId(embComm_s *Comm);
void parseStartStr(embComm_s *Comm);
void parseStatePkt(embComm_s *Comm);
void parseWayPkt(embComm_s *Comm);
void syncClocks(embComm_s *Comm,
                embState_s *State);
void upStateList(embComm_s *Comm,
                 embState_s *State);


// init rx comm

void embInitRxComm(embComm_s *Comm)
{
  // build rx structure
  initRxStruct(Comm);

  // initialize sys match timer
  lpcInitMatchTmr(&Comm->Rx.Match,
                  Comm->Tmr);
}


// config rx comm

void embConfigRxComm(embComm_s *Comm)
```

```
{
  // configure rx match
  lpcSetSlotMatchTmr(&Comm->Rx.Match,
                     RX_MATCH_SLOT);
}


// enable rx comm

void embEnableRxComm(embComm_s *Comm)
{
  // enable rx match
  lpcEnableMatchTmr(&Comm->Rx.Match);

  // reset rx match
  lpcResetMatchTmr(&Comm->Rx.Match,
                   CCLK);
}


// receive comm handle

void embRxComm(embComm_s *Comm)
{
  // while unread bytes in buf
  while(isUnreadByteInUart(Comm))
    {
      // update parse index and figure out which case
      switch(++(Comm->Rx.ParseIndex))
        {
```

```
    // if start string index
    case(START_STR_PARSE_INDEX):
      parseStartStr(Comm);
      break;

    // if packet id index
    case(PKT_ID_PARSE_INDEX):
      parsePktId(Comm);
      break;

    // if byte count index
    case(BYTE_CNT_PARSE_INDEX):
      parseByteCnt(Comm);
      break;

    // if high level control packet index
    case(HL_CNTL_PKT_PARSE_INDEX):
      parseHlCntlPkt(Comm);
      break;

    // if position packet index
    case(GLOBAL_PKT_PARSE_INDEX):
      parseGlobalPkt(Comm);
      break;

    // if land request packet index
    case(LAND_PKT_PARSE_INDEX):
      parseLandPkt(Comm);
      break;

```

```
        // if waypoint packet index
      case(WAY_PKT_PARSE_INDEX):
        parseWayPkt(Comm);
        break;

        // if low level control packet index
      case(LL_CNTL_PKT_PARSE_INDEX):
        parseLlCntlPkt(Comm);
        break;

        // if state packet index
      case(STATE_PKT_PARSE_INDEX):
        parseStatePkt(Comm);
        break;

        // if environment packet index
      case(ENV_PKT_PARSE_INDEX):
        parseEnvPkt(Comm);
        break;

        // if homing index
      case(HOME_PKT_PARSE_INDEX):
        parseHomePkt(Comm);
        break;
      }
  }

// reset rx comm match
lpcResetMatchTmr(&Comm->Rx.Match,
                 RX_COMM_PERIOD);
```

```c

  // clear rx comm match IR
  lpcClrIrMatchTmr(&Comm->Rx.Match);
}


// builds rx comm struct

void initRxStruct(embComm_s *Comm)
{ }


// checks if calculated crc is same as read one

bool_t isCrcValid(embComm_s *Comm)
{
  // local stack
  int32_t ii;
  byte_t ReadByte;
  bool_t DummyBool;
  int32_t BytesRecv;

  // unread packet for crc
  lpcOffsetReadUart(&Comm->Uart,
                    -Comm->Rx.ByteCnt);

  // reset crcs
  Comm->Rx.CalcCrc = CRC_RST;
  Comm->Rx.ReadCrc = CRC_RST;

```

```
  // calculate crc of packet
  for(ii = 0;
        ii < Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc);
        ii++)
     {
        lpcReadByteUart(&Comm->Uart,
                        &ReadByte,
                        &DummyBool);
        embUpCrc32(&Comm->Rx.CalcCrc,
                   ReadByte);
     }

  // read crc from packet
  LPC_READ_UART(Comm->Uart,
                Comm->Rx.ReadCrc,
                BytesRecv);

  // return match comparison
  return((bool_t)(Comm->Rx.CalcCrc == Comm->Rx.ReadCrc));
}


// check for valid start string at beginning of packet

void parseStartStr(embComm_s *Comm)
{
  // local stack
  int32_t ii;
  bool_t ValidStartStr;
  bool_t DummyBool;
```

```
  byte_t ReadByte;

  // unread bytes for start string comparison
  lpcOffsetReadUart(&Comm->Uart,
                    -(sizeof(START_STR)-1));

  // verify start string
  for(ii = 0, ValidStartStr = true;
      ii < sizeof(START_STR)-1;
      ii++)
    {
      lpcReadByteUart(&Comm->Uart,
                      &ReadByte,
                      &DummyBool);

      ValidStartStr = ValidStartStr && ((bool_t)(ReadByte == START_STR[ii]));
    }

  // reset parse if not valid start string
  if(!ValidStartStr)
    {
      Comm->Rx.ParseIndex = RST_PARSE_INDEX;
    }
}


// parse byte count from packet

void parseByteCnt(embComm_s *Comm)
```

```
{
  // local stack
  int32_t BytesRecv;
  bool_t ValidByteCnt;

  // read byte count from uart
  lpcOffsetReadUart(&Comm->Uart,
                    -sizeof(Comm->Rx.ByteCnt));
  LPC_READ_UART(Comm->Uart,
                Comm->Rx.ByteCnt,
                BytesRecv);

  // check if valid byte count
  ValidByteCnt = (bool_t)((Comm->Rx.ByteCnt >= BYTE_CNT_MIN)
    &&
                          (Comm->Rx.ByteCnt <= BYTE_CNT_MAX))
                          ;

  // set parse index if valid byte count
  if(ValidByteCnt)
    {
      // assign corresponding parse index for valid packet id
      switch(Comm->Rx.PktId)
        {
          // if high level control packet
        case(HL_CNTL_PKT_ID):
          Comm->Rx.ParseIndex = HL_CNTL_PKT_PARSE_INDEX;
          Comm->Rx.ParseIndex -= Comm->Rx.ByteCnt;
          break;

```

```
        // if global position packet
        case(GLOBAL_PKT_ID):
          Comm->Rx.ParseIndex = GLOBAL_PKT_PARSE_INDEX;
          Comm->Rx.ParseIndex -= Comm->Rx.ByteCnt;
          break;

        // if land request packet
        case(LAND_PKT_ID):
          Comm->Rx.ParseIndex = LAND_PKT_PARSE_INDEX;
          Comm->Rx.ParseIndex -= Comm->Rx.ByteCnt;
          break;

        // if waypoint packet
        case(WAY_PKT_ID):
          Comm->Rx.ParseIndex = WAY_PKT_PARSE_INDEX;
          Comm->Rx.ParseIndex -= Comm->Rx.ByteCnt;
          break;

        // if low level control packet
        case(LL_CNTL_PKT_ID):
          Comm->Rx.ParseIndex = LL_CNTL_PKT_PARSE_INDEX;
          Comm->Rx.ParseIndex -= Comm->Rx.ByteCnt;
          break;

        // if state packet
        case(STATE_PKT_ID):
          Comm->Rx.ParseIndex = STATE_PKT_PARSE_INDEX;
          Comm->Rx.ParseIndex -= Comm->Rx.ByteCnt;
          break;

```

```
        // if environment packet
      case(ENV_PKT_ID):
        Comm->Rx.ParseIndex = ENV_PKT_PARSE_INDEX;
        Comm->Rx.ParseIndex -= Comm->Rx.ByteCnt;
        break;

        // if homing packet
      case(HOME_PKT_ID):
        Comm->Rx.ParseIndex = HOME_PKT_PARSE_INDEX;
        Comm->Rx.ParseIndex -= Comm->Rx.ByteCnt;
        break;

        // reset if not valid packet
      default:
        Comm->Rx.ParseIndex = RST_PARSE_INDEX;
        break;
      }
    }

  // reset parse index if not valid byte count
  else
    {
      Comm->Rx.ParseIndex = RST_PARSE_INDEX;
    }
}


// parse packet id from packet

void parsePktId(embComm_s *Comm)
```

```
{
  // local stack
  bool_t DummyBool;;

  // read byte count from uart
  lpcOffsetReadUart(&Comm->Uart,
                    -sizeof(byte_t));
  lpcReadByteUart(&Comm->Uart,
                  &Comm->Rx.PktId,
                  &DummyBool);
}


// parse low level init packet

void parseL1CntlPkt(embComm_s *Comm)
{
  // local stack
  robotId_t TargetRobotId;
  int32_t BytesRecv;
  int32_t ii;

  // if valid checksum
  if(isCrcValid(Comm))
    {
      // unread packet for low level control
      lpcOffsetReadUart(&Comm->Uart,
                        -Comm->Rx.ByteCnt);

      // for packet loop entries
```

```
      for(ii = 0;
          ii < (Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc))
            / LL_CNTL_PKT_LOOP_CNT;
          ii++)
        {
          // read robot id
          LPC_READ_UART(Comm->Uart,
                        TargetRobotId,
                        BytesRecv);

          // if me, initialize low level control
          if(TargetRobotId == Comm->StateList->State[
            EMB_MY_STATE].RobotId)
            {
              Comm->InitL1Cntl = true;
            }
        }

      // discard trailing checksum
      discardCrc(Comm);
    }

  // reset parse index
  Comm->Rx.ParseIndex = RST_PARSE_INDEX;
}


// parse high level init packet

void parseHlCntlPkt(embComm_s *Comm)
```

```
{
  // local stack
  robotId_t TargetRobotId;
  int32_t BytesRecv;
  int32_t ii;

  // if valid checksum
  if(isCrcValid(Comm))
    {
      // unread packet for high level control
      lpcOffsetReadUart(&Comm->Uart,
                        -Comm->Rx.ByteCnt);

      // for all entries in packet loop
      for(ii = 0;
          ii < (Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc))
             / HL_CNTL_PKT_LOOP_CNT;
          ii++)
        {
          // read target robot id
          LPC_READ_UART(Comm->Uart,
                        TargetRobotId,
                        BytesRecv);

          // if me, initialize high level control
          if(TargetRobotId == Comm->StateList->State[
             EMB_MY_STATE].RobotId)
            {
              Comm->InitHlCntl = true;
            }
```

```
            }

        // discard trailing checksum
        discardCrc(Comm);
    }

  // reset parse index
  Comm->Rx.ParseIndex = RST_PARSE_INDEX;
}


// parse landing packet

void parseLandPkt(embComm_s *Comm)
{
  // local stack
  robotId_t TargetRobotId;
  int32_t BytesRecv;
  int32_t ii;

  // if valid checksum
  if(isCrcValid(Comm))
    {
      // unread packet for landing request
      lpcOffsetReadUart(&Comm->Uart,
                        -Comm->Rx.ByteCnt);

      // for all entries in packet loop
      for(ii = 0;
```

```
              ii < (Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc))
                / LAND_PKT_LOOP_CNT;
              ii++)
            {
              // read robot id
              LPC_READ_UART(Comm->Uart,
                            TargetRobotId,
                            BytesRecv);

              // if me, initialize landing
              if(TargetRobotId == Comm->StateList->State[
                EMB_MY_STATE].RobotId)
                {
                  Comm->InitLand = true;
                }
            }

          // discard trailing checksum
          discardCrc(Comm);
        }

  // reset parse index
  Comm->Rx.ParseIndex = RST_PARSE_INDEX;
}


// parse homing packet

void parseHomePkt(embComm_s *Comm)
{
```

```
// local stack
robotId_t TargetRobotId;
int32_t BytesRecv;
int32_t ii;

// if valid checksum
if(isCrcValid(Comm))
  {
    // unread packet for homing request
    lpcOffsetReadUart(&Comm->Uart,
                      -Comm->Rx.ByteCnt);

    // for all entires in packet loop
    for(ii = 0;
        ii < (Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc))
          / HOME_PKT_LOOP_CNT;
        ii++)
      {
        // read robot id
        LPC_READ_UART(Comm->Uart,
                      TargetRobotId,
                      BytesRecv);

        // if me, initialize homing routine
        if(TargetRobotId == Comm->StateList->State[
          EMB_MY_STATE].RobotId)
          {
            Comm->InitHome = true;
          }
      }
```

```

      // discard trailing checksum
      discardCrc(Comm);
    }

  // reset parse index
  Comm->Rx.ParseIndex = RST_PARSE_INDEX;
}


// parse state packet

void parseStatePkt(embComm_s *Comm)
{
  // local stack
  embState_s State;
  bool_t CommRange;
  int32_t ii;
  int32_t jj;
  int32_t BytesRecv;

  // if valid checksum
  if(isCrcValid(Comm))
    {
      // unread packet for state information
      lpcOffsetReadUart(&Comm->Uart,
                        -Comm->Rx.ByteCnt);

      // for all entires in packet loop
      for(ii = 0, CommRange = false;
```

```
      ii < (Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc))
        / STATE_PKT_LOOP_CNT;
      ii++)
    {
      // read robot id
      LPC_READ_UART(Comm->Uart,
                    State.RobotId,
                    BytesRecv);

      // read time stamp
      LPC_READ_UART(Comm->Uart,
                    State.TimeStamp,
                    BytesRecv);

      // read state positions
      for(jj = 0;
          jj < sizeof(State.Pos)/sizeof(State.Pos[0]);
          jj++)
        {
          LPC_READ_UART(Comm->Uart,
                        State.Pos[jj],
                        BytesRecv);
        }

      // if info of sender, will only work for index==0
      if(ii == STATE_SENDER_INDEX)
        {
          // see if sender was in comm range
          CommRange = isInCommRange(Comm,
                                    &State);
```

```
                    // if in comm range, sync clocks
                    if(CommRange)
                      {
                        /*
                        embSyncClock(Comm->Clock,
                                          State.TimeStamp + Comm->
                                            Latency,
                                          Comm->Rx.Lambda);
                        */
                      }
                  }

                // if in comm range, update state list
                if(CommRange)
                  {
                    upStateList(Comm,
                                &State);
                  }
              }

            // discard trailing checksum
            discardCrc(Comm);
          }

    // reset parse index
    Comm->Rx.ParseIndex = RST_PARSE_INDEX;
}
```

```
// parse global position packet

void parseGlobalPkt(embComm_s *Comm)
{
  // local stack
  robotId_t TargetRobotId;
  embGlobal_s Global;
  int32_t ii;
  int32_t jj;
  int32_t BytesRecv;

  // if valid checksum
  if(isCrcValid(Comm))
    {
      Comm->Status->Yellow.Speed = LED_Slow;

      // unread packet for state information
      lpcOffsetReadUart(&Comm->Uart,
                        -Comm->Rx.ByteCnt);

      // for all entires in packet loop
      for(ii = 0;
          ii < (Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc))
            / GLOBAL_PKT_LOOP_CNT;
          ii++)
        {
          // read robot id
          LPC_READ_UART(Comm->Uart,
                        TargetRobotId,
                        BytesRecv);
```

```
        // read time stamp
        LPC_READ_UART(Comm->Uart,
                      Global.TimeStamp,
                      BytesRecv);

        // read state positions
        for(jj = 0;
            jj < sizeof(Global.So3)/sizeof(Global.So3[0]);
            jj++)
          {
            LPC_READ_UART(Comm->Uart,
                          Global.So3[jj],
                          BytesRecv);
          }

        // if my global state, keep it
        if(TargetRobotId == Comm->StateList->State[
            EMB_MY_STATE].RobotId)
          {
            Comm->Status->Yellow.Speed = LED_Flkr;

            // update global
            embUpGlobal(Comm->Global,
                        &Global);

            // update state
            embUpStateFromGlobal(Comm->Global);

            // update pid
```

```
                embUpPidFromGlobal(Comm->Global);
              }
          }

        // discard trailing checksum
        discardCrc(Comm);
      }

    // reset parse index
    Comm->Rx.ParseIndex = RST_PARSE_INDEX;
}


// parse environment initialization packet

void parseEnvPkt(embComm_s *Comm)
{
    // local stack
    embEnv_s Env;
    int32_t ii;
    int32_t jj;
    int32_t BytesRecv;

    // if valid checksum
    if(isCrcValid(Comm))
      {
        // unread packet for environment information
        lpcOffsetReadUart(&Comm->Uart,
                          -Comm->Rx.ByteCnt);

```

```
      // clear environment information
      embClrEnv(Comm->Env);

      // for all entires in packet loop
      for(ii = 0;
          (ii < (Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc)) / ENV_PKT_LOOP_CNT) && (ii < EMB_MAX_BNDRY_SIZE_ENV);
          ii++)
        {
          // read environment entry
          for(jj = 0;
              jj < sizeof(Env.Bndry[0].Pos)/sizeof(Env.Bndry[0].Pos[0]);
              jj++)
            {
              LPC_READ_UART(Comm->Uart,
                            Env.Bndry[ii].Pos[jj],
                            BytesRecv);
            }

          // initialize environment
          Comm->InitEnv = true;
        }

      // discard trailing checksum
      discardCrc(Comm);
    }

  // reset parse index
```

```
  Comm->Rx.ParseIndex = RST_PARSE_INDEX;
}


// parse waypoint packet

void parseWayPkt(embComm_s *Comm)
{
  // local stack
  robotId_t TargetRobotId;
  int32_t BytesRecv;
  pos_t Way[4];
  int32_t ii;
  int32_t jj;

  // if valid checksum
  if(isCrcValid(Comm))
    {
      // unread packet for landing request
      lpcOffsetReadUart(&Comm->Uart,
                        -Comm->Rx.ByteCnt);

      // for all entries in packet loop
      for(ii = 0;
          ii < (Comm->Rx.ByteCnt - sizeof(Comm->Rx.CalcCrc))
            / WAY_PKT_LOOP_CNT;
          ii++)
        {
          // read robot id
          LPC_READ_UART(Comm->Uart,
```

```
                            TargetRobotId,
                            BytesRecv);

            // read environment entry
            for(jj = 0;
                jj < sizeof(Way)/sizeof(Way[0]);
                jj++)
              {
                LPC_READ_UART(Comm->Uart,
                                 Way[jj],
                                 BytesRecv);
              }

            // if for me, write to waypoint struct
            if(TargetRobotId == Comm->StateList->State[
              EMB_MY_STATE].RobotId)
              {
                embUpWayGlobal(Comm->Global,
                                 Way);
              }
          }

        // discard trailing checksum
        discardCrc(Comm);
    }

  // reset parse index
  Comm->Rx.ParseIndex = RST_PARSE_INDEX;
}

```

```
// discard trailing checksum

void discardCrc(embComm_s *Comm)
{
  // local stack
  crc32_t Crc;
  int32_t BytesRecv;

  // discard
  LPC_READ_UART(Comm->Uart,
                Crc,
                BytesRecv);
}


// check if in comm range

bool_t isInCommRange(embComm_s *Comm,
                     embState_s *State)
{
  // local stack
  int32_t Dist;

  // calc distance to sender
  Dist = int32Dist(Comm->StateList->State[EMB_MY_STATE].Pos,
                   State->Pos,
                   3);

  // if in comm range, return true, else false
```

```
  if(Dist < COMM_RANGE)
    {
      return(true);
    }
  else
    {
      return(false);
    }
}


// updates state list

void upStateList(embComm_s *Comm,
                 embState_s *State)
{
  // local stack
  int32_t Index;
  int32_t ii;
  bool_t IsValid;
  int32_t TimeStampDiff;

  // find robot index
  IsValid = embFindState(Comm->StateList,
                         State->RobotId,
                         &Index);

  // if valid and not my state
  if((IsValid) && (Index != EMB_MY_STATE))
    {
```

```
      // if more current, update list
      TimeStampDiff = State->TimeStamp - Comm->StateList->State[Index].TimeStamp;

      if(TimeStampDiff > 0)
        {
          Comm->StateList->State[Index].TimeStamp = State->TimeStamp;
          for(ii = 0;
              ii < sizeof(State->Pos)/sizeof(State->Pos[0]);
              ii++)
            {
              Comm->StateList->State[Index].Pos[ii] = State->Pos[ii];
            }
        }
    }
}


// see if unread bytes still in uart

bool_t isUnreadByteInUart(embComm_s *Comm)
{
  // local stack
  bool_t ByteRecv;
  byte_t DummyByte;

  // function
  lpcReadByteUart(&Comm->Uart,
```

```
                    &DummyByte,
                    &ByteRecv);

  // return bool
  return(ByteRecv);
}
```

```
/*
 * emb/TxAp.c
 *
 * tx ap source code for drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Clock.h"
#include "emb/Ap.h"
#include "emb/Crc16.h"
#include "emb/Nav.h"
#include "lpc/Buf.h"
#include "lpc/Cntl.h"
#include "lpc/Math.h"


// local defines

#define CRC_RST EMB_CLR_CRC32
```

```
#define TX_MATCH_SLOT 3
#define TX_AP_PERIOD (CCLK)

#define START_STR ">*>"
#define STATE_PKT_ID 'S'
#define STATE_PKT_LOOP_CNT 10

#define LAT_ORIGIN    423599810
#define LONG_ORIGIN -710891290

#define HEIGHT_ORIGIN 3000

#define LAT_LENGTH 11
#define LONG_LENGTH 8


// local functions

void calcApBufCrc(embAp_s *Ap);
void initTxStructAp(embTxAp_s *Tx);
void sendState(embAp_s *Ap);
void sendWay(embAp_s *Ap);
void sendCmd(embAp_s *Ap);

// initialize tx ap

void embInitTxAp(embAp_s *Ap)
{
  // initialize tx ap struct
```

```
  initTxStructAp(&Ap->Tx);

  // initialize tx ap buf
  lpcInitBuf(&Ap->Tx.Buf);

  // initialize sys match timer
  lpcInitMatchTmr(&Ap->Tx.Match,
                  Ap->Tmr);
}


// configure tx ap

void embConfigTxAp(embAp_s *Ap)
{
  // assign read buf
  lpcAssignBuf(&Ap->Tx.Buf,
               Ap->Tx.BufMem,
               LPC_TX_BUF_SIZE_UART);

  // configure tx match
  lpcSetSlotMatchTmr(&Ap->Tx.Match,
                     TX_MATCH_SLOT);
}


// enable tx ap

void embEnableTxAp(embAp_s *Ap)
{
```

```
  // enable read buf
  lpcEnableBuf(&Ap->Tx.Buf);

  // enable tx match tmr
  lpcEnableMatchTmr(&Ap->Tx.Match);

  // reset tx ap match
  lpcResetMatchTmr(&Ap->Tx.Match,
                   CCLK);
}


// main tx ap handle

void embTxAp(embAp_s *Ap)
{
  // local stack

  // action based on navigation setting
  switch(Ap->Nav->Flight)
    {
      // if indoor flight, send commands
    case(NAV_Indoor):
      sendCmd(Ap);
      break;

      // if outdoor flight, send ways
    case(NAV_Outdoor):
      sendWay(Ap);
      break;
```

```
    }

  // reset tx ap match
  lpcResetMatchTmr(&Ap->Tx.Match,
                   TX_AP_PERIOD);

  // clear tx ap match IR
  lpcClrIrMatchTmr(&Ap->Tx.Match);
}


// calculate crc of buffer

void calcApBufCrc(embAp_s *Ap)
{
  // local stack
  byte_t Read;
  bool_t ByteRecv;
  int32_t OffsetSize;

  // offset non crc portion of packet
  OffsetSize  = sizeof(START_STR) - 1;
  OffsetSize += sizeof(byte_t);
  OffsetSize += sizeof(Ap->Tx.ByteCnt);

  lpcOffsetReadBuf(&Ap->Tx.Buf,
                   OffsetSize);

  // reset Crc
  Ap->Tx.Crc = EMB_CLR_CRC16;
```

```
  // calc Crc of buf
  lpcReadByteBuf(&Ap->Tx.Buf,
                 &Read,
                 &ByteRecv);

  while(ByteRecv)
    {
      embUpCrc16(&Ap->Tx.Crc,
                 Read);
      lpcReadByteBuf(&Ap->Tx.Buf,
                     &Read,
                     &ByteRecv);
      OffsetSize++;
    }

  // unread buf
  lpcOffsetReadBuf(&Ap->Tx.Buf,
                   -OffsetSize);
}


// initialize tx structure

void initTxStructAp(embTxAp_s *Tx)
{ }


// send way over ap uart

```

```
void sendWay(embAp_s *Ap)
{
  // local stack
  uint8_t  WpNumber = 1;
  uint8_t  Dummy1 = 0;
  uint16_t Dummy2 = 0;
  uint8_t  Properties = ((0x01)|(0x02)|(0x04)|(0x10));
  uint8_t  MaxSpeed = 20;
  uint16_t Time = 0;
  uint16_t PosAcc = 2500;
  int16_t  ChkSum;
  int32_t  X;
  int32_t  Y;
  int32_t  Yaw = 0;
  int32_t  Height = 0;

  // Position calculation
  X = (Ap->Global->Way[0]/LONG_LENGTH)+LONG_ORIGIN;
  Y = (Ap->Global->Way[1]/LAT_LENGTH)+LAT_ORIGIN;
  Yaw = -(180000*Ap->Global->Way[3])/INT_PI;
  Height = Ap->Global->Way[2];


  // Checksum calcualtion
  ChkSum = (0xAAAA) + (int16_t)Yaw + (int16_t)Height + (int16_t)Time +
    (int16_t)X + (int16_t)Y + (int16_t)MaxSpeed + (int16_t)PosAcc +
    (int16_t)Properties + (int16_t)WpNumber;

```

```c

  // write way cmd to buffer
  LPC_WRITE_BUF(Ap->Tx.Buf,">*>ws");
  LPC_WRITE_BUF(Ap->Tx.Buf,WpNumber);
  LPC_WRITE_BUF(Ap->Tx.Buf,Dummy1);
  LPC_WRITE_BUF(Ap->Tx.Buf,Dummy2);
  LPC_WRITE_BUF(Ap->Tx.Buf,Properties);
  LPC_WRITE_BUF(Ap->Tx.Buf,MaxSpeed);
  LPC_WRITE_BUF(Ap->Tx.Buf,Time);
  LPC_WRITE_BUF(Ap->Tx.Buf,PosAcc);
  LPC_WRITE_BUF(Ap->Tx.Buf,ChkSum);
  LPC_WRITE_BUF(Ap->Tx.Buf,X);
  LPC_WRITE_BUF(Ap->Tx.Buf,Y);
  LPC_WRITE_BUF(Ap->Tx.Buf,Yaw);
  LPC_WRITE_BUF(Ap->Tx.Buf,Height);

  // write buf to uart
  lpcWriteBufToUart(&Ap->Uart,
                    &Ap->Tx.Buf);
}

// send way over ap uart

void sendCmd(embAp_s *Ap)
{
  // local stack
  int16_t  ChkSum;
  int16_t Property = 0x02;
  uint16_t TempShort = 0x0F;

```

```c
    // Checksum calcualtion
    ChkSum = (0xAAAA);
    ChkSum += (int16_t)Ap->PidList->Input[0];
    ChkSum += (int16_t)Ap->PidList->Input[1];
    ChkSum += (int16_t)Ap->PidList->Input[2];
    ChkSum += (int16_t)Ap->PidList->Input[3];
    ChkSum += (int16_t)TempShort;

    // write way cmd to buffer
    LPC_WRITE_BUF(Ap->Tx.Buf,">*>p");
    LPC_WRITE_BUF(Ap->Tx.Buf,Property);
    LPC_WRITE_BUF(Ap->Tx.Buf,">*>di");
    LPC_WRITE_BUF(Ap->Tx.Buf,Ap->PidList->Input[0]);
    LPC_WRITE_BUF(Ap->Tx.Buf,Ap->PidList->Input[1]);
    LPC_WRITE_BUF(Ap->Tx.Buf,Ap->PidList->Input[2]);
    LPC_WRITE_BUF(Ap->Tx.Buf,Ap->PidList->Input[3]);
    LPC_WRITE_BUF(Ap->Tx.Buf,TempShort);
    LPC_WRITE_BUF(Ap->Tx.Buf,ChkSum);


    // write buf to uart
    lpcWriteBufToUart(&Ap->Uart,
                      &Ap->Tx.Buf);
}
```

```c
/*
 * emb/TxComm.c
 *
 * tx communication source code for drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Clock.h"
#include "emb/Comm.h"
#include "emb/Crc32.h"
#include "emb/Nav.h"
#include "lpc/Buf.h"
#include "lpc/Cntl.h"


// local defines

#define CRC_RST EMB_CLR_CRC32

```

```c
#define TX_MATCH_SLOT 1
#define TX_COMM_PERIOD (CCLK)

#define START_STR ">*>"
#define STATE_PKT_ID 'S'
#define STATE_PKT_LOOP_CNT 10

#define FAKE_STR "abc_"


// local functions

void calcBufCrc(embComm_s *Comm);
void initTxStruct(embTxComm_s *Tx);
void sendState(embComm_s *Comm);


// initialize tx comm

void embInitTxComm(embComm_s *Comm)
{
  // initialize tx comm struct
  initTxStruct(&Comm->Tx);

  // initialize tx comm buf
  lpcInitBuf(&Comm->Tx.Buf);

  // initialize sys match timer
  lpcInitMatchTmr(&Comm->Tx.Match,
                  Comm->Tmr);
```

```
}


// configure tx comm

void embConfigTxComm(embComm_s *Comm)
{
  // assign read buf
  lpcAssignBuf(&Comm->Tx.Buf,
               Comm->Tx.BufMem,
               LPC_TX_BUF_SIZE_UART);

  // configure tx match
  lpcSetSlotMatchTmr(&Comm->Tx.Match,
                     TX_MATCH_SLOT);
}


// enable tx comm

void embEnableTxComm(embComm_s *Comm)
{
  // enable read buf
  lpcEnableBuf(&Comm->Tx.Buf);

  // enable tx match tmr
  lpcEnableMatchTmr(&Comm->Tx.Match);

  // reset tx comm match
  lpcResetMatchTmr(&Comm->Tx.Match,
```

```
                              CCLK);
}


// main tx comm handle

void embTxComm(embComm_s *Comm)
{
  // local stack

  // action based on navigation setting
  switch(Comm->Nav->Flight)
    {
      // if indoor flight, do nothing
    case(NAV_Indoor):
      break;

      // if outdoor flight, send states
    case(NAV_Outdoor):
      sendState(Comm);
      break;
    }

  // reset tx comm match
  lpcResetMatchTmr(&Comm->Tx.Match,
                              TX_COMM_PERIOD);

  // clear tx comm match IR
  lpcClrIrMatchTmr(&Comm->Tx.Match);
}
```

```


// calculate crc of buffer

void calcBufCrc(embComm_s *Comm)
{
  // local stack
  byte_t Read;
  bool_t ByteRecv;
  int32_t OffsetSize;

  // offset non crc portion of packet
  OffsetSize  = sizeof(START_STR) - 1;
  OffsetSize += sizeof(byte_t);
  OffsetSize += sizeof(Comm->Tx.ByteCnt);

  lpcOffsetReadBuf(&Comm->Tx.Buf,
                   OffsetSize);

  // reset Crc
  Comm->Tx.Crc = EMB_CLR_CRC32;

  // calc Crc of buf
  lpcReadByteBuf(&Comm->Tx.Buf,
                 &Read,
                 &ByteRecv);

  while(ByteRecv)
    {
      embUpCrc32(&Comm->Tx.Crc,
```

```
                        Read);
          lpcReadByteBuf(&Comm->Tx.Buf,
                        &Read,
                        &ByteRecv);
          OffsetSize++;
      }

    // unread buf
    lpcOffsetReadBuf(&Comm->Tx.Buf,
                        -OffsetSize);
}


// initialize tx structure

void initTxStruct(embTxComm_s *Tx)
{ }


// send state over comm uart

void sendState(embComm_s *Comm)
{
    // local stack
    int32_t ii;
    int32_t jj;

    // calc byte count
    Comm->Tx.ByteCnt = sizeof(Comm->StateList->State[0].RobotId
        );
```

```
Comm->Tx.ByteCnt += sizeof(Comm->StateList->State[0].
    TimeStamp);
Comm->Tx.ByteCnt += sizeof(Comm->StateList->State[0].Pos);
Comm->Tx.ByteCnt *= Comm->Mhop->SlotLength;
Comm->Tx.ByteCnt += sizeof(Comm->Tx.Crc);

// write start string
lpcWriteBuf(&Comm->Tx.Buf,
            (byte_t *)START_STR,
            sizeof(START_STR)-1);

// write state packet id
lpcWriteByteBuf(&Comm->Tx.Buf,
                (byte_t)STATE_PKT_ID);

// write byte count
LPC_WRITE_BUF(Comm->Tx.Buf,
              Comm->Tx.ByteCnt);

// get my time stamp
Comm->StateList->State[0].TimeStamp = embGetClock(Comm->
    Clock);

// write states
for(ii = 0;
    ii < Comm->Mhop->SlotLength;
    ii++)
  {
    LPC_WRITE_BUF(Comm->Tx.Buf,
```

```
                    Comm->StateList->State[Comm->StateList->SendIndex[ii]].RobotId);
        LPC_WRITE_BUF(Comm->Tx.Buf,
                    Comm->StateList->State[Comm->StateList->SendIndex[ii]].TimeStamp);
        for(jj = 0;
            jj < sizeof(Comm->StateList->State[0].Pos) / sizeof(Comm->StateList->State[0].Pos[0]);
            jj++)
        {
            LPC_WRITE_BUF(Comm->Tx.Buf,
                        Comm->StateList->State[Comm->StateList->SendIndex[ii]].Pos[jj]);
        }
    }

    // calc crc
    calcBufCrc(Comm);

    // write crc to buf
    LPC_WRITE_BUF(Comm->Tx.Buf,
                Comm->Tx.Crc);

    // write buf to uart
    lpcWriteBufToUart(&Comm->Uart,
                    &Comm->Tx.Buf);
}
```

```
/*
 * emb/State.h
 *
 * state includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_STATE_H__

#ifndef __EMB_STATE_H__
#define __EMB_STATE_H__


// includes

#include "emb/Types.h"
#include "lpc/Types.h"


// defines

```

```
#define EMB_MY_STATE 0
#define EMB_MAX_SIZE_STATE 10

#define EMB_X_STATE 0
#define EMB_Y_STATE 1
#define EMB_Z_STATE 2
#define EMB_YAW_STATE 3

// state struct

typedef struct
{
  clock_t TimeStamp;
  robotId_t RobotId;
  pos_t Pos[4];
}
  embState_s;


// state list struct

typedef struct
{
  int32_t Size;
  embState_s State[EMB_MAX_SIZE_STATE];
  int32_t SendIndex[EMB_MAX_SIZE_STATE];
}
  embStateList_s;


```

```c
// functions

void embInitState(embStateList_s *StateList);
void embConfigState(embStateList_s *StateList);
void embEnableState(embStateList_s *StateList);

bool_t embFindState(const embStateList_s *StateList,
                    const robotId_t RobotId,
                    int32_t *Index);
robotId_t embRobotIdState(robotName_t RobotName);


// __EMB_STATE_H__

#endif
```

```
/*
 * emb/State.c
 *
 * state source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/State.h"
#include "emb/Types.h"

#include "lpc/Math.h"
#include "lpc/Types.h"


// local defines

#define MAX_ROBOT_NAME 7
#define ROBOT_NAME_BASE 27

```

```
// initialize state

void embInitState(embStateList_s *StateList)
{ }


// configure state

void embConfigState(embStateList_s *StateList)
{ }


// enable state

void embEnableState(embStateList_s *StateList)
{ }


// returns robot id from robot name

robotId_t embRobotIdState(robotName_t RobotName)
{
  // local stack
  robotId_t RobotId;
  int ii;

  // zero robot id
  RobotId = 0;

```

```
  // calc robot id
  for(ii = 0;
      (RobotName[ii] != '\0') && (ii < MAX_ROBOT_NAME);
      ii++)
    {
      RobotId += ((robotId_t)(RobotName[ii]-'a'+1))*int32Pow(ROBOT_NAME_BASE, ii);
    }

  // return id
  return(RobotId);
}


// finds robot name among state list

bool_t embFindState(const embStateList_s *StateList,
                    const robotId_t RobotId,
                    int32_t *Index)
{
  // find in state list
  for(*Index = 0;
      (StateList->State[*Index].RobotId != RobotId) && ((*Index) < StateList->Size);
      (*Index)++);

  if((*Index) == EMB_MAX_SIZE_STATE)
    {
      return(false);
    }
```

```
    else
      {
        return(true);
      }
}
```

```
/*
 * emb/Status.h
 *
 * status includes for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_STATUS_H__

#ifndef __EMB_STATUS_H__
#define __EMB_STATUS_H__


// includes

#include "lpc/Types.h"
#include "lpc/Tmr.h"


// led speed enum

```

```
typedef enum
  {
    LED_On,
    LED_Slow,
    LED_Med,
    LED_Fast,
    LED_Flkr
  }
  embLedSpeed_e;


// led struct

typedef struct
{
  lpcMatchTmr_s Match;
  embLedSpeed_e Speed;
}
  embLed_s;


// status struct

typedef struct
{
  lpcTmr_s Tmr;

  embLed_s Green;
  embLed_s Yellow;
  embLed_s Red;
```

```
}
  embStatus_s;


// functions

void embInitStatus(embStatus_s *Status);
void embConfigStatus(embStatus_s *Status);
void embEnableStatus(embStatus_s *Status);


// __EMB_STATUS_H__

#endif
```

```
/*
 * emb/Status.c
 *
 * status source code for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// includes

#include "emb/Status.h"
#include "lpc/Cntl.h"
#include "lpc/Tmr.h"
#include "lpc/Types.h"


// local defines

#define STATUS_TMR_NUM 1

#define GREEN_MATCH_SLOT 3
#define GREEN_MATCH_PIN 20
```

```
#define YELLOW_MATCH_SLOT 2
#define YELLOW_MATCH_PIN 19

#define RED_MATCH_SLOT 1
#define RED_MATCH_PIN 13

#define ON_PERIOD (CCLK/100)
#define OFF_PERIOD CCLK
#define SLOW_PERIOD CCLK
#define MED_PERIOD (CCLK/4)
#define FAST_PERIOD (CCLK/16)


// global pointer

embStatus_s *LinkedStatus;


// local functions

void manageLedState(embLed_s *Led,
                    lpcTmr_s *Tmr);
void statusTmrVectAddr(void) __attribute__ ((interrupt("IRQ")
    ));


// initialize status

void embInitStatus(embStatus_s *Status)
```

```
{
  // link status struct
  LinkedStatus = Status;

  // initialize status timer
  lpcInitTmr(&Status->Tmr);

  // initialize green led
  Status->Green.Speed = LED_On;
  lpcInitMatchTmr(&Status->Green.Match,
                  &Status->Tmr);

  // initialize yellow led
  Status->Yellow.Speed = LED_On;
  lpcInitMatchTmr(&Status->Yellow.Match,
                  &Status->Tmr);

  /*
  // initialize red led
  Status->Red.Speed = LED_On;
  lpcInitMatchTmr(&Status->Red.Match,
                  &Status->Tmr);
  */
}


// configure status

void embConfigStatus(embStatus_s *Status)
{
```

```
    // configure status timer
    lpcSetNumTmr(&Status->Tmr,
                 STATUS_TMR_NUM);
    lpcSetVectAddrTmr(&Status->Tmr,
                      (reg32_t)statusTmrVectAddr);

    // configure green led
    lpcSetSlotMatchTmr(&Status->Green.Match,
                       GREEN_MATCH_SLOT);
    lpcSetPinMatchTmr(&Status->Green.Match,
                      GREEN_MATCH_PIN);

    // configure yellow led
    lpcSetSlotMatchTmr(&Status->Yellow.Match,
                       YELLOW_MATCH_SLOT);
    lpcSetPinMatchTmr(&Status->Yellow.Match,
                      YELLOW_MATCH_PIN);

    /*
    // configure red led
    lpcSetSlotMatchTmr(&Status->Red.Match,
                       RED_MATCH_SLOT);
    lpcSetPinMatchTmr(&Status->Red.Match,
                      RED_MATCH_PIN);
    */
}


// enable status

```

```
void embEnableStatus(embStatus_s *Status)
{
  // enable status timer
  lpcEnableTmr(&Status->Tmr);

  // enable green led
  lpcSetExtMatchTmr(&Status->Green.Match);
  lpcEnableMatchTmr(&Status->Green.Match);
  lpcResetMatchTmr(&Status->Green.Match,
                   CCLK);

  // enable Yellow led
  lpcSetExtMatchTmr(&Status->Yellow.Match);
  lpcEnableMatchTmr(&Status->Yellow.Match);
  lpcResetMatchTmr(&Status->Yellow.Match,
                   CCLK);

  /*
  // enable Red led
  lpcSetExtMatchTmr(&Status->Red.Match);
  lpcEnableMatchTmr(&Status->Red.Match);
  lpcResetMatchTmr(&Status->Red.Match,
  CCLK);
  */
}


// timer management of leds

void manageLedState(embLed_s *Led,
```

```
                            lpcTmr_s *Tmr)
{
  // local stack
  bool_t Read;
  reg32_t TcOffset;

  // adjust match tc depending on set speed
  switch(Led->Speed)
    {
      // case on
    case(LED_On):
      TcOffset = ON_PERIOD;
      break;

      // case flkr
    case(LED_Flkr):
      lpcReadPinMatchTmr(&Led->Match,
                              &Read);
      if(Read)
        {
          TcOffset = ON_PERIOD;
        }
      else
        {
          TcOffset = OFF_PERIOD;
        }
      break;

      // case slow
    case(LED_Slow):
```

```
        TcOffset = SLOW_PERIOD;
        break;

        // case medium
      case(LED_Med):
        TcOffset = MED_PERIOD;
        break;

        // case fast
      case(LED_Fast):
        TcOffset = FAST_PERIOD;
        break;

        // default
      default:
        TcOffset = CCLK;
        break;
      }

  // reset match
  lpcResetMatchTmr(&Led->Match,
                   TcOffset);
}


// timer vector address handle

void statusTmrVectAddr(void)
{
  // local stack
```

```
embStatus_s *Status;
bool_t Read;

// create status link
Status = LinkedStatus;

// if green match caused IR
lpcReadIrMatchTmr(&Status->Green.Match,
                  &Read);
if(Read)
  {
    manageLedState(&Status->Green,
                   &Status->Tmr);
    lpcClrIrMatchTmr(&Status->Green.Match);
  }

// if yellow match caused IR
lpcReadIrMatchTmr(&Status->Yellow.Match,
                  &Read);
if(Read)
  {
    manageLedState(&Status->Yellow,
                   &Status->Tmr);
    lpcClrIrMatchTmr(&Status->Yellow.Match);
  }

/*
// if red match caused IR
lpcReadIrMatchTmr(&Status->Red.Match,
                  &Read);
```

```
    if(Read)
      {
        manageLedState(&Status->Red,
                       &Status->Tmr);
        lpcClrIrMatchTmr(&Status->Red.Match);
      }
    */

    // reset vic
    lpcResetVicTmr(&Status->Tmr);
}
```

```
/*
 * emb/Types.h
 *
 * type definitions for the drl quad-rotor project
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __EMB_TYPES_H__

#ifndef __EMB_TYPES_H__
#define __EMB_TYPES_H__


// position type

#ifndef pos_t
#define pos_t int32_t
#endif


// AutoPilot command input type
```

```
#ifndef cmd_t
#define cmd_t int16_t
#endif


// robot id type

#ifndef robotId_t
#define robotId_t uint32_t
#endif


// clock type

#ifndef clock_t
#define clock_t uint32_t
#endif


// robot name type

#ifndef robotName_t
#define robotName_t byte_t*
#endif


// crc16 type

#ifndef crc16_t
```

```
#define crc16_t uint16_t
#endif


// crc32 type

#ifndef crc32_t
#define crc32_t uint32_t
#endif


// __EMB_TYPES_H__

#endif
```

```c
/*
 * lpc/Buf.h
 *
 * buffer include for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_BUF_H__
#ifndef __LPC_BUF_H__
#define __LPC_BUF_H__


// includes

#include "lpc/Types.h"


// write buf macro

#define LPC_WRITE_BUF(A,B)                    \
  lpcWriteBuf(&A, (byte_t *)&B, sizeof(B))
```

```
// buffer struct

typedef struct
{
  byte_t *Ptr;
  int32_t WriteIndex;
  int32_t ReadIndex;
  int32_t Size;
}
  lpcBuf_s;


// functions

int32_t lpcInitBuf(lpcBuf_s *Buf);
err32_t lpcEnableBuf(lpcBuf_s *Buf);

err32_t lpcAssignBuf(lpcBuf_s *Buf,
                     byte_t *Ptr,
                     int32_t Size);
err32_t lpcWriteBuf(lpcBuf_s *Buf,
                    const byte_t *Write,
                    const int32_t Size);
err32_t lpcReadBuf(lpcBuf_s *Buf,
                   byte_t *Read,
                   const int32_t Size,
                   int32_t *BytesRecv);
err32_t lpcWriteByteBuf(lpcBuf_s *Buf,
```

```
                                const byte_t Write);
err32_t lpcReadByteBuf(lpcBuf_s *Buf,
                               byte_t *Read,
                               bool_t *ByteRecv);
err32_t lpcOffsetReadBuf(lpcBuf_s *Buf,
                                 int32_t Offset);


// __LPC_BUF_H__

#endif
```

```c
/*
 * lpc/Buf.c
 *
 * buffer source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


#include "lpc/Buf.h"

#include "lpc/Err.h"
#include "lpc/Types.h"



#define DEF_BUF_PTR (byte_t *)NULL
#define DEF_WRITE_INDEX 0
#define DEF_READ_INDEX 0



// initialize buffer
```

```
int32_t lpcInitBuf(lpcBuf_s *Buf)
{
  // init buf structure
  Buf->Ptr = DEF_BUF_PTR;
  Buf->WriteIndex = DEF_WRITE_INDEX;
  Buf->ReadIndex = DEF_READ_INDEX;

  // return successful
  return(LPC_SUCC);
}



// enable buffer
err32_t lpcEnableBuf(lpcBuf_s *Buf)
{
  // return successful
  return(LPC_SUCC);
}



err32_t lpcAssignBuf(lpcBuf_s *Buf,
                     byte_t *Ptr,
                     int32_t Size)
{
  Buf->Ptr = Ptr;
  Buf->Size = Size;

  // return successful
```

```
  return(LPC_SUCC);
}




// read unread elements upto write index
err32_t lpcWriteBuf(lpcBuf_s *Buf,
                    const byte_t *Write,
                    const int32_t Size)
{
  // local stack
  int32_t ii;
  int32_t BytesSent;

  // check if valid size
  if((Size < 0) || (Size > Buf->Size))
    {
      return(LPC_BUF_SIZE_ERR);
    }

  // do write
  for(BytesSent = 0, ii = Buf->WriteIndex;
      BytesSent < Size;
      BytesSent++)
    {
      Buf->Ptr[ii++] = Write[BytesSent];
      ii %= Buf->Size;
    }

```

```
  // update write index
  Buf->WriteIndex = ii;

  // return successful
  return(LPC_SUCC);
}



// read unread elements upto write index
err32_t lpcReadBuf(lpcBuf_s *Buf,
                   byte_t *Read,
                   const int32_t Size,
                   int32_t *BytesRecv)
{
  // local stack
  int32_t ii;

  // check if valid size
  if((Size < 0) || (Size > Buf->Size))
    {
      return(LPC_BUF_SIZE_ERR);
    }

  // do read
  for(*BytesRecv = 0, ii = Buf->ReadIndex;
      (*BytesRecv < Size) && (ii != Buf->WriteIndex);
      (*BytesRecv)++)
    {
      Read[*BytesRecv] = Buf->Ptr[ii++];
```

```
      ii %= Buf->Size;
    }

  // update read index
  Buf->ReadIndex = ii;

  // return successful
  return(LPC_SUCC);
}



// read unread elements upto write index
err32_t lpcWriteByteBuf(lpcBuf_s *Buf,
                        const byte_t Write)
{
  Buf->Ptr[Buf->WriteIndex] = Write;

  // update write index
  Buf->WriteIndex++;
  Buf->WriteIndex %= Buf->Size;

  // return successful
  return(LPC_SUCC);
}



// read unread elements upto write index
err32_t lpcReadByteBuf(lpcBuf_s *Buf,
```

```
                              byte_t *Read,
                              bool_t *ByteRecv)
{
  // zero recv
  *ByteRecv = false;

  // do read
  if(Buf->ReadIndex != Buf->WriteIndex)
    {
      *Read = Buf->Ptr[Buf->ReadIndex];

      Buf->ReadIndex++;
      Buf->ReadIndex %= Buf->Size;

      *ByteRecv = true;
    }

  // return successful
  return(LPC_SUCC);
}


// offset read index
err32_t lpcOffsetReadBuf(lpcBuf_s *Buf,
                                int32_t Offset)
{
  // check if valid size
  if((Offset < -Buf->Size) || (Offset > Buf->Size))
    {
      return(LPC_BUF_SIZE_ERR);
```

```
    }

  // offset read index
  Buf->ReadIndex += Buf->Size;
  Buf->ReadIndex += Offset;
  Buf->ReadIndex %= Buf->Size;

  // return successful
  return(LPC_SUCC);
}
```

```c
/*
 * lpc/Cntl.h
 *
 * low level control include for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_CNTL_H__

#ifndef __LPC_CNTL_H__
#define __LPC_CNTL_H__


// includes

#include "lpc/Irq.h"
#include "lpc/Types.h"


// control defines

```

```c
#define FOSC 14745600
#define MSEL 4
#define PSEL 2
#define CCLK (FOSC*MSEL)
#define FCCO (2*PSEL*CCLK)


// ctrl struct

typedef struct
{
  lpcIrq_s Irq;
}
  lpcCntl_s;


// function declarations

err32_t lpcInitCntl(lpcCntl_s *Cntl);
err32_t lpcEnableCntl(lpcCntl_s *Cntl);


// __LPC_CNTL_H__

#endif
```

```c
/*
 * lpc/Cntl.c
 *
 * low level control source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


#include "lpc/Cntl.h"

#include "lpc/Err.h"
#include "lpc/Irq.h"
#include "lpc/Lpc214x.h"
#include "lpc/Types.h"


// local defines
#define ENABLE_FIO_0 (1<<0)
#define ENABLE_FIO_1 (1<<1)
#define ENABLE_PLL (1<<0)
```

```
#define CONNECT_PLL (1<<1)
#define MASK_PLL_LOCK_STAT (1<<10)
#define BUS_SAME_AS_CLK (1<<0)
#define DISABLE_MAM 0
#define FETCH_CYCLE_6 ((1<<2)|(1<<1))
#define ENABLE_MAM (1<<0)
#define PSEL_1 0
#define PSEL_2 (1<<5)
#define PSEL_4 (1<<6)
#define PSEL_8 ((1<<6)|(1<<5))



// local function declarations
void feedSeq(void);


// init low level controller
err32_t lpcInitCntl(lpcCntl_s *Cntl)
{
  // set system control and status flags to high speed GPIO
  SCS = ENABLE_FIO_0 | ENABLE_FIO_1;

  // return success
  return(LPC_SUCC);
}
```

```
// enable low level controller
err32_t lpcEnableCntl(lpcCntl_s *Cntl)
{
  // configure the PLL
  switch(PSEL)
    {
    case(2):
      PLLCFG = (PSEL_2|(MSEL-1));
      break;
    case(4):
      PLLCFG = (PSEL_4|(MSEL-1));
      break;
    case(8):
      PLLCFG = (PSEL_8|(MSEL-1));
      break;
    default:
      PLLCFG = (PSEL_1|(MSEL-1));
      break;
    }

  // enable the PLL
  PLLCON = ENABLE_PLL;

  // perform correct feed sequence for PLL changes
  feedSeq();

  // wait for PLL status to change
  while((PLLSTAT & MASK_PLL_LOCK_STAT)==0);

  // connect the PLL
```

```
    PLLCON = ENABLE_PLL | CONNECT_PLL;

    // perform correct feed sequence for PLL changes
    feedSeq();

    // set VPB to be same as processor clock speed
    VPBDIV = BUS_SAME_AS_CLK;

    // disable memory accelerator module
    MAMCR = DISABLE_MAM;

    // configure memory accelerator module timing control
    MAMTIM = FETCH_CYCLE_6;

    // enable memory accelerator module
    MAMCR = ENABLE_MAM;

    // initialize irq
    lpcInitIrq(&Cntl->Irq);

    // enable irq
    lpcEnableIrq(&Cntl->Irq);

    // return success
    return(LPC_SUCC);
}



// perform correct feed sequence for PLL changes
```

```
void feedSeq(void)
{
  PLLFEED = (0xAA);
  PLLFEED = (0x55);
}
```

```c
/*
 * lpc/Eint.h
 *
 * external interrupt control source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_EINT_H__

#ifndef __LPC_EINT_H__
#define __LPC_EINT_H__


// includes

#include "lpc/Types.h"
#include "lpc/Vic.h"


// external interrupt struct

```

```c
typedef struct
{
  int32_t Num;
  int32_t Pin;
  reg32_t VectAddr;
  lpcVic_s Vic;
}
  lpcEint_s;


// function declarations

err32_t lpcInitEint(lpcEint_s *Eint);
err32_t lpcEnableEint(lpcEint_s *Eint);

err32_t lpcSetPinEint(lpcEint_s *Eint,
                      const int32_t Pin);
err32_t lpcSetVectAddrEint(lpcEint_s *Eint,
                           const reg32_t VectAddr);
err32_t lpcResetVicEint(const lpcEint_s *Eint);
err32_t lpcClearIrEint(const lpcEint_s *Eint);
err32_t lpcReadPinEint(const lpcEint_s *Eint,
                       bool_t *Read);


// __LPC_EINT_H__

#endif
```

```
/*
 * lpc/Eint.c
 *
 * external interrupt source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */



#include "lpc/Eint.h"

#include "lpc/Err.h"
#include "lpc/Irq.h"
#include "lpc/Lpc214x.h"
#include "lpc/Types.h"



// local defines
#define DEF_PIN 0


```

```
// initialize external interrupt controller
err32_t lpcInitEint(lpcEint_s *Eint)
{
  // assign vector address
  Eint->VectAddr = (reg32_t)NULL;

  // set default pin
  Eint->Pin = DEF_PIN;

  // initialize vector interrupt controller
  lpcInitVic(&Eint->Vic);

  // return success
  return(LPC_SUCC);
}


// enable external interrupt controller
err32_t lpcEnableEint(lpcEint_s *Eint)
{
  // check pin
  switch(Eint->Pin)
    {
      // EINT0
    case(1):
    case(16):
      Eint->Num = 0;
      break;
```

```
        // EINT1
    case(3):
    case(14):
        Eint->Num = 1;
        break;

        // EINT2
    case(7):
    case(15):
        Eint->Num = 2;
        break;

        // EINT3
    case(9):
    case(20):
    case(30):
        Eint->Num = 3;
        break;

        // error
    default:
        return(LPC_EINT_PIN_ERR);
        break;
    }

// configure pinsel
PINSEL0 &= ~((1<<(Eint->Pin<<1)) | (1<<((Eint->Pin<<1)+1)))
    ;
switch(Eint->Pin)
```

```
{
  // PINSEL 11
case(1):
case(3):
case(7):
case(9):
  PINSEL0 |= (1<<(Eint->Pin<<1));
  PINSEL0 |= (1<<((Eint->Pin<<1)+1));
  break;

case(20):
  PINSEL1 |= (1<<((Eint->Pin-16)<<1));
  PINSEL1 |= (1<<(((Eint->Pin-16)<<1)+1));
  break;

  // PINSEL 10
case(14):
case(15):
  PINSEL0 &= ~(1<<(Eint->Pin<<1));
  PINSEL0 |= (1<<((Eint->Pin<<1)+1));
  break;

case(30):
  PINSEL1 &= ~(1<<((Eint->Pin-16)<<1));
  PINSEL1 |= (1<<(((Eint->Pin-16)<<1)+1));
  break;

  // PINSEL 01
case(16):
  PINSEL1 &= ~(1<<(((Eint->Pin-16)<<1)+1));
```

```
        PINSEL1 |= (1<<((Eint->Pin-16)<<1));
        break;

        // error
      default:
        return(LPC_EINT_PIN_ERR);
        break;
      }

  // assign vic type
  lpcSetTypeVic(&Eint->Vic,
                LPC_EINT_0_VIC + Eint->Num);

  // assign vic vect addr
  lpcSetVectAddrVic(&Eint->Vic,
                    Eint->VectAddr);

  // Set ext mode, polarity, and enable eint
  EXTMODE |= (1<<(Eint->Num));

  // enable eint vic
  lpcEnableVic(&Eint->Vic);

  // Clear interrupts
  lpcClearIrEint(Eint);

  // return success
  return(LPC_SUCC);
}

```

```
err32_t lpcClearIrEint(const lpcEint_s *Eint)
{
  // Clear interrupts
  EXTINT = (1<<(Eint->Num));

  // return success
  return(LPC_SUCC);
}


err32_t lpcResetVicEint(const lpcEint_s *Eint)
{
  VICVectAddr = LPC_CLR_REG;

  // return success
  return(LPC_SUCC);
}



err32_t lpcSetPinEint(lpcEint_s *Eint,
                      const int32_t Pin)
{
  Eint->Pin = Pin;

  // return success
  return(LPC_SUCC);
}
```

```
err32_t lpcSetVectAddrEint(lpcEint_s *Eint,
                                const reg32_t VectAddr)
{
  Eint->VectAddr = VectAddr;

  // return success
  return(LPC_SUCC);
}



err32_t lpcReadPinEint(const lpcEint_s *Eint,
                           bool_t *Read)
{
  // read pin state
  *Read = (bool_t)((FIO0PIN & (1 << Eint->Pin)) != 0);

  // return success
  return(LPC_SUCC);
}
```

```c
/*
 * lpc/Err.h
 *
 * error definitions for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_ERR_H__

#ifndef __LPC_ERR_H__
#define __LPC_ERR_H__


// error definitions

#define LPC_SUCC 0
#define LPC_NUM_IRQ_SLOTS_ERR -1
#define LPC_BUF_SIZE_ERR -2
#define LPC_VECT_ADDR_ERR -3
#define LPC_VECT_CNTL_ERR -4
#define LPC_TMR_MATCH_IR_SLOT_ERR -5
```

```
#define LPC_UART_WRITE_ERR -6
#define LPC_UART_READ_ERR -7
#define LPC_EINT_PIN_ERR -7


// __LPC_ERR_H__

#endif
```

```
/*
 * lpc/Irq.h
 *
 * interrupt request include for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_IRQ_H__

#ifndef __LPC_IRQ_H__
#define __LPC_IRQ_H__


// includes

#include "lpc/Types.h"


// irq struct

typedef struct
```

```
{
  // dummy cpsr
  unsigned Cpsr;
}
  lpcIrq_s;


// function declarations

err32_t lpcInitIrq(lpcIrq_s *Irq);
err32_t lpcEnableIrq(lpcIrq_s *Irq);

err32_t lpcDisableIrq(lpcIrq_s *Irq);
err32_t lpcRestoreIrq(lpcIrq_s *Irq);


// __LPC_IRQ_H__

#endif
```

```
/*
 * lpc/Irq.c
 *
 * interrupt source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


#include "lpc/Irq.h"

#include "lpc/Err.h"
#include "lpc/Lpc214x.h"
#include "lpc/Types.h"


static inline unsigned __get_cpsr(void);
static inline void __set_cpsr(unsigned Cpsr);
void defVectAddr(void) __attribute__ ((interrupt("IRQ")));


err32_t lpcDisableIrq(lpcIrq_s *Irq)
{
```

```
    unsigned _cpsr;
    _cpsr = __get_cpsr();
    __set_cpsr(_cpsr | (1<<7));
    return(LPC_SUCC);
}

err32_t lpcRestoreIrq(lpcIrq_s *Irq)
{
    unsigned _cpsr;
    _cpsr = __get_cpsr();
    __set_cpsr((_cpsr & ~(1<<7)) | (Irq->Cpsr & (1<<7)));
    Irq->Cpsr = _cpsr;
    return(LPC_SUCC);
}

err32_t lpcEnableIrq(lpcIrq_s *Irq)
{
    unsigned _cpsr;
    _cpsr = __get_cpsr();
    __set_cpsr(_cpsr & ~(1<<7));
    Irq->Cpsr = _cpsr;
    return(LPC_SUCC);
}

static inline unsigned __get_cpsr(void)
{
    unsigned Cpsr;
    asm volatile (" mrs  %0, cpsr" : "=r" (Cpsr) : /* no inputs */ );
    return(Cpsr);
```

```c
}

static inline void __set_cpsr(unsigned Cpsr)
{
    asm volatile (" msr  cpsr, %0" : /* no outputs */ : "r" (Cpsr) );
}

void defVectAddr(void)
{
    VICVectAddr = 0;
    while(1);
}

err32_t lpcInitIrq(lpcIrq_s *Irq)
{
    // Clear all IRQs and FIQs
    VICIntEnClr =  LPC_SET_REG;
    VICIntSelect = LPC_CLR_REG;

    // Set default interrupt handler
    VICDefVectAddr = (reg32_t)defVectAddr;

    return(LPC_SUCC);
}
```

```
/*
 * lpc/Math.h
 *
 * verified math functions for lpc microcontrollers
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


//  __LPC_MATH_H__

#ifndef __LPC_MATH_H__
#define __LPC_MATH_H__


// includes

#include "lpc/Types.h"


// mathmatical constants

#define INT_PI 3216
```

```
#define INT_2_PI 6432
#define INT_PI_2 1608


// math macros

#define INT_MIN(A, B) (((A)<(B))?(A):(B))
#define INT_MAX(A, B) (((A)>(B))?(A):(B))
#define INT_MIN_MAX(A, B, C) (INT_MIN((B),INT_MAX((A),(C))))
#define INT_SIGN(A) (((A)>(0))?(1):(((A)<(0))?(-1):(0)))
#define INT_ABS(A) (((A)<(0))?(-A):(A))


// functions

int32_t int32Min(const int32_t *Array,
                 const int32_t Size);
int32_t int32Max(const int32_t *Array,
                 const int32_t Size);
int32_t int32Sum(const int32_t *Array,
                 const int32_t Size);
int32_t int32Dist(const int32_t *Pos1,
                  const int32_t *Pos2,
                  const int32_t Dim);
int32_t int32Sqrt(const int32_t Num);
int32_t int32Cos(int32_t Mrad);
int32_t int32Sin(int32_t Mrad);
int32_t int32Tan(const int32_t Mrad);
int32_t int32Atan(const int32_t Z);
int32_t int32Atan2(const int32_t Y,
```

```
                          const int32_t X);
int32_t int32Mrad(int32_t Mrad);
int32_t int32Pow(const int32_t Base,
                 const int32_t Exp);
int64_t int64Pow(const int64_t Base,
                 const int32_t Exp);
int64_t int64Sqrt(const int64_t Num);
int64_t int64Min(const int64_t *Array,
                 const int32_t Size);
int64_t int64Max(const int64_t *Array,
                 const int32_t Size);
int64_t int64Sum(const int64_t *Array,
                 const int32_t Size);


// __LPC_MATH_H__

#endif
```

```
/*
 * lpc/Math.c
 *
 * verified math functions for lpc microcontrollers
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */

// include files
#include "lpc/Math.h"

#include "lpc/Types.h"


// finds min value and corresponding index in an integer array
int32_t int32Min(const int32_t *Array,
                 const int32_t Size)
{
  // local stack
  int32_t ii;
  int32_t Index;

```

```
  // function
  for(ii = 0, Index = 0;
      ii < Size;
      ii++)
    {
      if(Array[ii] < Array[Index])
        {
          Index = ii;
        }
    }

  return(Index);
}

// Finds max value and corresponding index in an integer
   array
int32_t int32Max(const int32_t *Array,
                 const int32_t Size)
{
  // local stack
  int32_t ii;
  int32_t Index;

  // function
  for(ii = 0, Index = 0;
      ii < Size;
      ii++)
    {
      if(Array[ii] > Array[Index])
        {
```

```
                Index = ii;
            }
        }

    return(Index);
}

// calculates sum of integer array
int32_t int32Sum(const int32_t *Array,
                  const int32_t Size)
{
    // local stack
    int32_t ii;
    int32_t Sum;

    // function
    for(ii = 0, Sum = 0;
        ii < Size;
        ii++)
      {
        Sum += Array[ii];
      }

    return(Sum);
}

// calculates distance between two position integer arrays
int32_t int32Dist(const int32_t *Pos1,
                  const int32_t *Pos2,
                  const int32_t Dim)
```

```
{
  // local stack
  int32_t ii;
  int64_t Sum;

  // function
  for(ii = 0, Sum = 0;
      ii < Dim;
      ii++)
    {
      Sum += int64Pow((int64_t)(Pos1[ii]-Pos2[ii]),
                      2);
    }

  return(int64Sqrt(Sum));
}

// calcuates integer square root of integer
int32_t int32Sqrt(const int32_t Num)
{
  // local stack
  int32_t Delta;
  int32_t Pow;
  int32_t Sqrt;

  // function
  for(Delta = (1<<15), Sqrt = 0;
      Delta > 0;
      Delta >>= 1)
    {
```

```
        Pow = int32Pow(Delta + Sqrt,
                       2);
        if(Pow <= Num)
          {
            Sqrt += Delta;
          }
      }

  return(Sqrt);
}

// calculates "milli"cos of milliradians
int32_t int32Cos(int32_t Mrad)
{
  // local stack
  int32_t C0, C2, C4;

  // normalize milliradians
  Mrad = int32Mrad(Mrad);

  // function
  if(INT_ABS(Mrad) > INT_PI_2)
    {
      return(-int32Cos(Mrad + INT_PI));
    }

  C0 = (1<<10);
  C2 = (Mrad * Mrad) >> 10;
  C4 = (Mrad * ((Mrad * C2) >> 10)) >> 10;

```

```
  return(C0 - (C2/2) + (C4/24));
}

// Calculates "milli"sin of milliradians
int32_t int32Sin(int32_t Mrad)
{
  // local stack
  int32_t S1,S3,S5;

  // normalize milliradians
  Mrad = int32Mrad(Mrad);

  // function
  if(INT_ABS(Mrad) > INT_PI_2)
    {
      return(-int32Sin(Mrad + INT_PI));
    }

    S1 = Mrad;
    S3 = (Mrad * ((Mrad * S1) >> 10)) >> 10;
    S5 = (Mrad * ((Mrad * S3) >> 10)) >> 10;

    return(S1 - (S3/6) + (S5/120));
}

// Calculates "milli"tan of milliradians
int32_t int32Tan(const int32_t Mrad)
{
  return((int32Sin(Mrad)<<10)/int32Cos(Mrad));
}
```

```
// Calculates "milli" arctan of "milli" ratio
int32_t int32Atan(const int32_t Z)
{
  // local stack
  int32_t A1, A3, A5;

  // function
  if(INT_ABS(Z) > (1<<10))
    {
      return(INT_SIGN(Z)*INT_PI_2 - int32Atan((1 << 20) / Z))
        ;
    }

  A1 = Z;
  A3 = (Z * ((Z * A1) >> 10)) >> 10;
  A5 = (Z * ((Z * A3) >> 10)) >> 10;

  return(A1 - (A3/3) + (A5/5));
}



// Calculates "milli" arctan2 of "milli" ratio
int32_t int32Atan2(const int32_t Y,
                   const int32_t X)
{
  // local stack
  int32_t Phi;
  int32_t Atan2;
```

```
// function
switch(Y)
  {
    // Y == 0
  case(0):
    if(X < 0)
      {
        Atan2 = INT_PI;
      }
    else
      {
        Atan2 = 0;
      }
    break;

    // Y != 0
  default:
    if(X == 0)
      {
        Atan2 = INT_PI_2*INT_SIGN(Y);
      }
    else
      {
        Phi  = int32Pow(X,2);
        Phi += int32Pow(Y,2);
        Phi  = int32Sqrt(Phi);
        Phi += X;
        Phi  = (Y << 10) / Phi;
        Atan2 = int32Atan(Phi);
```

```
            Atan2 <<= 1;
          }
        break;
      }

  return(Atan2);
}

// normalizes milliradians into (INT_PI,INT_PI] range
int32_t int32Mrad(int32_t Mrad)
{
  // function
  while(INT_ABS(Mrad) > INT_PI)
    {
      Mrad -= INT_2_PI * INT_SIGN(Mrad);
    }

  return(Mrad);
}

// calculates integer power of an integer base
int32_t int32Pow(const int32_t Base,
                 const int32_t Exp)
{
  // local stack
  int32_t Pow;
  int32_t ii;

  // function
  for(ii = 0, Pow = 1;
```

```
        ii < Exp;
        ii++)
     {
        Pow *= Base;
     }

  return(Pow);
}


// calculates integer power of an integer base
int64_t int64Pow(const int64_t Base,
                 const int32_t Exp)
{
  // local stack
  int32_t ii;
  int64_t Pow;

  // function
  for(ii = 1, Pow = 1;
        ii < Exp;
        ii++)
     {
        Pow *= Base;
     }

  return(Pow);
}
```

```
// calcuates integer square root of integer
int64_t int64Sqrt(const int64_t Num)
{
  // local stack
  int64_t Delta;
  int64_t Pow;
  int64_t Sqrt;

  // function
  for(Delta = (1<<31), Sqrt = 0;
      Delta > 0;
      Delta >>= 1)
    {
      Pow = int64Pow(Delta + Sqrt,
                     2);
      if(Pow <= Num)
        {
          Sqrt += Delta;
        }
    }

  return(Sqrt);
}


// finds min value and corresponding index in an integer array
int64_t int64Min(const int64_t *Array,
                 const int32_t Size)
{
```

```
  // local stack
  int32_t ii;
  int32_t Index;

  // function
  for(ii = 0, Index = 0;
      ii < Size;
      ii++)
    {
      if(Array[ii] < Array[Index])
        {
          Index = ii;
        }
    }

  return(Index);
}

// Finds max value and corresponding index in an integer array
int64_t int64Max(const int64_t *Array,
                 const int32_t Size)
{
  // local stack
  int32_t ii;
  int32_t Index;

  // function
  for(ii = 0, Index = 0;
      ii < Size;
```

```
        ii++)
      {
        if(Array[ii] > Array[Index])
          {
            Index = ii;
          }
      }

  return(Index);
}

// calculates sum of integer array
int64_t int64Sum(const int64_t *Array,
                 const int32_t Size)
{
  // local stack
  int32_t ii;
  int64_t Sum;

  // function
  for(ii = 0, Sum = 0;
      ii < Size;
      ii++)
    {
      Sum += Array[ii];
    }

  return(Sum);
}
```

```
/*
 * lpc/Tmr.h
 *
 * timer include for lpc214x microcontrollers
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_TMR_H__

#ifndef __LPC_TMR_H__
#define __LPC_TMR_H__


// includes

#include "lpc/Vic.h"
#include "lpc/Types.h"


// defines

```

```
#define LPC_RESETTING_ON_TMR (1<<1)
#define LPC_RESETTING_OFF_TMR 0
#define LPC_STOPPING_ON_TMR (1<<2)
#define LPC_STOPPING_OFF_TMR 0


// timer struct

typedef struct
{
  lpcVic_s Vic;
  reg32_t VectAddr;
  int32_t Num;
  reg32_t BaseAddr;
  int32_t ClosedIrSlots;
}
  lpcTmr_s;


// match sub-structure for timer

typedef struct
{
  lpcTmr_s *Tmr;
  reg32_t Resetting;
  reg32_t Stopping;
  reg32_t IrSlot;
  reg32_t BaseAddr;
  reg32_t Tc;
  int32_t Pin;
```

```
}
  lpcMatchTmr_s;


// function

err32_t lpcInitTmr(lpcTmr_s *Tmr);
err32_t lpcEnableTmr(lpcTmr_s *Tmr);

err32_t lpcGetTcTmr(const lpcTmr_s *Tmr,
                    reg32_t *Tc);
err32_t lpcInitMatchTmr(lpcMatchTmr_s *Match,
                        lpcTmr_s *Tmr);
err32_t lpcEnableMatchTmr(lpcMatchTmr_s *Match);
err32_t lpcResetMatchTmr(lpcMatchTmr_s *Match,
                         const reg32_t TcOffset);
err32_t lpcSetTcMatchTmr(lpcMatchTmr_s *Match,
                         const reg32_t Tc);
err32_t lpcSetSlotMatchTmr(lpcMatchTmr_s *Match,
                           const int32_t Slot);
err32_t lpcClrIrMatchTmr(lpcMatchTmr_s *Match);
err32_t lpcReadIrMatchTmr(lpcMatchTmr_s *Match,
                          bool_t *Read);
err32_t lpcResetVicTmr(const lpcTmr_s *Tmr);
err32_t lpcSetNumTmr(lpcTmr_s *Tmr,
                     const int32_t Num);
err32_t lpcSetVectAddrTmr(lpcTmr_s *Tmr,
                          const reg32_t VectAddr);
err32_t lpcSetExtMatchTmr(const lpcMatchTmr_s *Match);
err32_t lpcSetPinMatchTmr(lpcMatchTmr_s *Match,
```

```
                                 const int32_t Pin);
err32_t lpcReadPinMatchTmr(const lpcMatchTmr_s *Match,
                                  bool_t *Read);
err32_t lpcSleepMsecTmr(const lpcTmr_s *Tmr,
                               uint32_t Msec);


// __LPC_TMR_H__

#endif
```

```c
/*
 * lpc/Tmr.c
 *
 * timer source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */



#include "lpc/Tmr.h"

#include "lpc/Cntl.h"
#include "lpc/Err.h"
#include "lpc/Lpc214x.h"
#include "lpc/Types.h"
#include "lpc/Vic.h"



// defines
#define ENABLE_COUNTER (1<<0)
#define RESET_COUNTER (1<<1)
```

```
#define MIN_MATCH_IR_SLOT 0
#define MAX_MATCH_IR_SLOT 3
#define CLR_MCR ((1<<2)|(1<<1)|(1<<0))
#define DEF_RESETTING LPC_RESETTING_OFF_TMR
#define DEF_STOPPING LPC_STOPPING_OFF_TMR
#define ENABLE_MCR (1<<0)
#define DEF_TMR 0


/* The PINSEL0 register controls the functions of the pins as per the
   settings listed in Table 63. The direction control bit in the
   IO0DIR register is effective only when the GPIO function is
   selected for a pin. For other functions, direction is controlled
   automatically. */

#define PIN_SEL(A)                                    \
  LPC_REG((A) + PINSEL_BASE_ADDR)





/* Interrupt Register. The IR can be written to clear interrupts. The
   IR can be read to identifiy which of the eight possible interrut
```

```
    sources are pending. */

#define T_IR(A)                                  \
  LPC_REG((A) + 0x00)



/* Timer Control Register. The TCR is used to control the timer counter
    functions.  The Timer Counter can be disabled or reset throught the
    TCR */

#define T_TCR(A)                                 \
  LPC_REG((A) + 0x04)



/* Timer Counter. The 32-bit TC is incremented every PR+1 cycles of
    PCLK. The TC is controlled through the TCR. */

#define T_TC(A)                                  \
  LPC_REG((A) + 0x08)



/* Prescale Register. The Prescale Counter (below) is R/W equal to
    this value, the next clock increments the TC and clears the PC. */
```

```
#define T_PR(A)                                          \
  LPC_REG((A) + 0x0C)



/* Prescale Counter. The 32-bit PC is a counter which is incremented
   to the value stored in PR. When the value in PR is reached, the TC
   is incremented and the PC is cleared. The PC is observable and
   controllable through the bus interface. */

#define T_PC(A)                                          \
  LPC_REG((A) + 0x10)



/* Match Control Register. The MCR is used to control if an interrupt
   is generated and if the TC is reset when a Match occurs. */

#define T_MCR(A)                                         \
  LPC_REG((A) + 0x14)



```

```
/* Match Register 0. MR0 can be enabled through the MCR to reset the
   TC, stop both the TC and PC, and/or generate an interrupt every
   time MR0 matches the TC. */

#define T_MR(A)                                          \
  LPC_REG((A) + 0x18)


/* The External Match Register provides both control and status of the
   external match pins MAT(0-3). */

#define T_EMR(A)                                         \
  LPC_REG((A) + 0x3C)




// initialize timer
err32_t lpcInitTmr(lpcTmr_s *Tmr)
{
  // assign vector address
  Tmr->VectAddr = (reg32_t)NULL;

  // default timer number
  Tmr->Num = DEF_TMR;

  // default interrupt register int
```

```
  Tmr->ClosedIrSlots = LPC_CLR_REG;

  // initialize vector interrupt controller
  lpcInitVic(&Tmr->Vic);

  // return success
  return(LPC_SUCC);
}



// enable timer
err32_t lpcEnableTmr(lpcTmr_s *Tmr)
{
  // assign vic type
  lpcSetTypeVic(&Tmr->Vic,
                LPC_TMR_0_VIC + Tmr->Num);

  // assign vic vect addr
  lpcSetVectAddrVic(&Tmr->Vic,
                    Tmr->VectAddr);

  // enable tmr vic
  lpcEnableVic(&Tmr->Vic);

  // calc base addr
  Tmr->BaseAddr = TMR0_BASE_ADDR + Tmr->Num*(TMR1_BASE_ADDR -
      TMR0_BASE_ADDR);

```

```
  // clear interrupts
  T_IR(Tmr->BaseAddr) = LPC_SET_REG;

  // enable counters
  T_TCR(Tmr->BaseAddr) = ENABLE_COUNTER;

  // reset counters
  T_TCR(Tmr->BaseAddr) |= RESET_COUNTER;
  T_TCR(Tmr->BaseAddr) &= ~RESET_COUNTER;

  // return success
  return(LPC_SUCC);
}



// get current timer clock count
err32_t lpcGetTcTmr(const lpcTmr_s *Tmr,
                    reg32_t *Tc)
{
  // set TC
  *Tc = T_TC(Tmr->BaseAddr);

  // return success
  return(LPC_SUCC);
}


// initialize match for timer
err32_t lpcInitMatchTmr(lpcMatchTmr_s *Match,
```

```
                        lpcTmr_s *Tmr)
{
  // link timer structure
  Match->Tmr = Tmr;

  // set default resetting
  Match->Resetting = DEF_RESETTING;


  // set default stopping
  Match->Stopping = DEF_STOPPING;

  // return success
  return(LPC_SUCC);
}



// enable match for timer
err32_t lpcSetSlotMatchTmr(lpcMatchTmr_s *Match,
                           const int32_t Slot)
{
  Match->IrSlot = Slot;

  // return success
  return(LPC_SUCC);
}



```

```
// enable match for timer
err32_t lpcEnableMatchTmr(lpcMatchTmr_s *Match)
{
  // if no open match ir slot, return error
  if((Match->Tmr->ClosedIrSlots & (1<<Match->IrSlot)) != 0)
    {
      return(LPC_TMR_MATCH_IR_SLOT_ERR);
    }

  // close selected ir slot
  Match->Tmr->ClosedIrSlots |= (1<<Match->IrSlot);

  // assign base address
  Match->BaseAddr = TMR0_BASE_ADDR;
  Match->BaseAddr += Match->Tmr->Num*(TMR1_BASE_ADDR - TMR0_BASE_ADDR);
  Match->BaseAddr += Match->IrSlot*(0x04);

  // set stopping and/or resetting
  T_MCR(Match->Tmr->BaseAddr) |= ((Match->Resetting | Match->Stopping)<<(3 * Match->IrSlot));

  // return success
  return(LPC_SUCC);
}



// enable match for timer
err32_t lpcResetMatchTmr(lpcMatchTmr_s *Match,
```

```
                        const reg32_t TcOffset)
{
  // local stack
  reg32_t Tc;

  // reset match control register
  T_MCR(Match->Tmr->BaseAddr) &= ~(CLR_MCR << (3*Match->
     IrSlot));

  // get current tc
  lpcGetTcTmr(Match->Tmr,
              &Tc);

  // set match tc
  lpcSetTcMatchTmr(Match,
                   Tc + TcOffset);

  // set match register
  T_MR(Match->BaseAddr) = Match->Tc;

  // enable interrupts for match control register
  T_MCR(Match->Tmr->BaseAddr) |= (ENABLE_MCR << (3*Match->
     IrSlot));

  // return success
  return(LPC_SUCC);
}
```

```c
// read current match pin state
err32_t lpcReadPinMatchTmr(const lpcMatchTmr_s *Match,
                           bool_t *Read)
{
  // read pin state
  *Read = (bool_t)((FIO0PIN & (1 << Match->Pin)) != 0);

  // return success
  return(LPC_SUCC);
}




err32_t lpcSetPinMatchTmr(lpcMatchTmr_s *Match,
                          const int32_t Pin)
{
  // set pin
  Match->Pin = Pin;

  // scan through possible pins
  switch(Pin)
    {
      // 0b01 enabled
    case(19):
    case(20):
      PINSEL1 &= ~(1 << (((Pin-16)<<1)+1));
      PINSEL1 |= (1 << ((Pin-16)<<1));
      break;

```

```
      // 0b10 enabled
    case(3):
    case(5):
    case(12):
    case(13):
      PINSEL0 &= ~(1 << (Pin<<1));
      PINSEL0 |= (1 << ((Pin<<1)+1));
      break;

    case(16):
      PINSEL1 &= ~(1 << ((Pin-16)<<1));
      PINSEL1 |= (1 << (((Pin-16)<<1)+1));
      break;

      // 0b11 enabled
    case(17):
    case(18):
    case(22):
    case(28):
    case(29):
      PINSEL1 |= (1 << ((Pin-16)<<1));
      PINSEL1 |= (1 << (((Pin-16)<<1)+1));
      break;

      // else
    default:
      break;
    }

  // return success
```

```
  return(LPC_SUCC);
}





err32_t lpcSetExtMatchTmr(const lpcMatchTmr_s *Match)
{
  T_EMR(Match->Tmr->BaseAddr) |= ((1<<Match->IrSlot) |
                                  (1<<(4+2*Match->IrSlot)) |
                                  (1<<(5+2*Match->IrSlot)));

  // return success
  return(LPC_SUCC);
}




// set match register for timer
err32_t lpcSetTcMatchTmr(lpcMatchTmr_s *Match,
                         const reg32_t Tc)
{
  // set register
  Match->Tc = Tc;

  // return success
  return(LPC_SUCC);
}
```

```
// set match register for timer
err32_t lpcClrIrMatchTmr(lpcMatchTmr_s *Match)
{
  // clear register
  T_IR(Match->Tmr->BaseAddr) = (1<<Match->IrSlot);

  // return success
  return(LPC_SUCC);
}



// set match register for timer
err32_t lpcReadIrMatchTmr(lpcMatchTmr_s *Match,
                           bool_t *Read)
{
  // set register
  *Read = (bool_t)((T_IR(Match->Tmr->BaseAddr) & (1<<Match->
      IrSlot)) != 0);

  // return success
  return(LPC_SUCC);
}



err32_t lpcResetVicTmr(const lpcTmr_s *Tmr)
```

```
{
  VICVectAddr = LPC_CLR_REG;

  // return success
  return(LPC_SUCC);
}


err32_t lpcSetNumTmr(lpcTmr_s *Tmr,
                     const int32_t Num)
{
  Tmr->Num = Num;

  // return success
  return(LPC_SUCC);
}



err32_t lpcSetVectAddrTmr(lpcTmr_s *Tmr,
                          const reg32_t VectAddr)
{
  Tmr->VectAddr = VectAddr;

  // return success
  return(LPC_SUCC);
}


err32_t lpcSleepMsecTmr(const lpcTmr_s *Tmr,
```

```
                                    uint32_t Msec)
{
  // local stack
  reg32_t Tc;
  reg32_t StartTc;

  // convert Msec to counts
  Msec *= (CCLK/1000);

  // get start tc
  lpcGetTcTmr(Tmr,
              &StartTc);

  // sleep
  do
    {
      lpcGetTcTmr(Tmr,
                  &Tc);
      Tc -= StartTc;
    }
  while(Tc < Msec);

  // return success
  return(LPC_SUCC);
}
```

```c
/*
 * lpc/Types.h
 *
 * variable definitions for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_TYPES_H__

#ifndef __LPC_TYPES_H__
#define __LPC_TYPES_H__


// defines to modify registers

#define LPC_REG(ADDR) (*(volatile unsigned long *)(ADDR))
#define LPC_CLR_REG 0x00000000
#define LPC_SET_REG 0xffffffff


// null type
```

```
#ifndef NULL
#define NULL (void *)0x00
#endif


// bool types

#ifndef false
#define false 0
#endif

#ifndef true
#define true 1
#endif


// 8-bit general byte type

#ifndef byte_t
#define byte_t unsigned char
#endif


// 8-bit general bool type

#ifndef bool_t
#define bool_t char
#endif

```

```
// 8-bit signed char type

#ifndef char8_t
#define char8_t char
#endif


// 32-bit register type

#ifndef reg32_t
#define reg32_t volatile unsigned long
#endif


// 32-bit error type

#ifndef err32_t
#define err32_t long
#endif


// 8-bit signed integer type

#ifndef int8_t
#define int8_t char
#endif


// 8-bit unsigned integer type
```

```
#ifndef uint8_t
#define uint8_t unsigned char
#endif


// 16-bit signed integer type

#ifndef int16_t
#define int16_t short
#endif


// 16-bit unsigned integer type

#ifndef uint16_t
#define uint16_t unsigned short
#endif


// 32-bit signed integer type

#ifndef int32_t
#define int32_t long
#endif


// 32-bit unsigned integer type

#ifndef uint32_t
```

```
#define uint32_t unsigned long
#endif


// 64-bit signed integer type

#ifndef int64_t
#define int64_t long long
#endif


// 64-bit unsigned integer type

#ifndef uint64_t
#define uint64_t unsigned long long
#endif


// __LPC_TYPES_H__

#endif
```

```
/*
 * lpc/Uart.h
 *
 * uart include for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_UART_H__

#ifndef __LPC_UART_H__
#define __LPC_UART_H__


// includes

#include "lpc/Types.h"
#include "lpc/Buf.h"
#include "lpc/Vic.h"


// defines
```

```
#define LPC_RX_BUF_SIZE_UART 256
#define LPC_TX_BUF_SIZE_UART 256
#define LPC_FLOW_CNTL_ON_UART true
#define LPC_FLOW_CNTL_OFF_UART false


// macro to read buffer

#define LPC_READ_UART(A, B, C)                    \
  lpcReadUart(&A, (byte_t *)&B, sizeof(B), &C)


// buffer struct

typedef struct
{
  lpcBuf_s RxBuf;
  lpcBuf_s TxBuf;

  byte_t RxBufMem[LPC_RX_BUF_SIZE_UART];
  byte_t TxBufMem[LPC_TX_BUF_SIZE_UART];

  int32_t Baudrate;
  bool_t FlowCntl;
  reg32_t BaseAddr;

  reg32_t VectAddr;
  lpcVic_s Vic;
  int32_t Num;
```

```
}
  lpcUart_s;


// functions

err32_t lpcInitUart(lpcUart_s *Uart);
err32_t lpcEnableUart(lpcUart_s *Uart);

err32_t lpcSetVectAddrUart(lpcUart_s *Uart,
                           const reg32_t VectAddr);
err32_t lpcSetNumUart(lpcUart_s *Uart,
                      const int32_t Num);
err32_t lpcSetBaudRateUart(lpcUart_s *Uart,
                           const int32_t BaudRate);
err32_t lpcSetFlowCntlUart(lpcUart_s *Uart,
                           const bool_t FlowCntl);
err32_t lpcSendUart(lpcUart_s *Uart);
err32_t lpcRecvUart(lpcUart_s *Uart);
err32_t lpcWriteUart(lpcUart_s *Uart,
                     const byte_t *Buf,
                     const int32_t Size);
err32_t lpcReadUart(lpcUart_s *Uart,
                    byte_t *Buf,
                    const int32_t Size,
                    int32_t *BytesRecv);
err32_t lpcReadByteUart(lpcUart_s *Uart,
                        byte_t *Read,
                        bool_t *ByteRecv);
err32_t lpcOffsetReadUart(lpcUart_s *Uart,
```

```
                                 int32_t Offset);
err32_t lpcWriteBufToUart(lpcUart_s *Uart,
                                 lpcBuf_s *Buf);


// __LPC_UART_H__

#endif
```

```c
/*
 * lpc/Uart.c
 *
 * uart source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */

#include "lpc/Uart.h"

#include "lpc/Types.h"
#include "lpc/Cntl.h"
#include "lpc/Err.h"
#include "lpc/Lpc214x.h"


#define DEF_BAUDRATE 38400
#define DEF_UART_NUM 0
#define THRE_WRITE_SIZE 14



```

```
/* The U_RBR is the top byte of the UART Rx FIFO. The top byte of the
   Rx FIFO contains the oldest character received and can be read via
   the bus interface. The LSB (bit 0) represents the oldest received
   data bit. If the character received is less than 8 bits, the unused
   MSBs are padded with zeroes. */

#define U_RBR(A)                                    \
  LPC_REG((A) + 0x00)


/*  The U_THR is the top byte of the UART0 TX FIFO. The top byte is
    the newest character in the TX FIFO and can be written via the bus
    interface. The LSB represents the first bit to transmit. */

#define U_THR(A)                                    \
  LPC_REG((A) + 0x00)


/* The UART Divisor Latch is part of the UART Fractional Baud Rate
   Generator and holds the value used to divide the clock supplied by
```

```
the fractional prescaler in order to produce the baud rate clock,
which must be 16x the desired baud rate (Equation 1). The U_DLL and
U_DLM registers together form a 16 bit divisor where U0DLL contains
the lower 8 bits of the divisor and U0DLM contains the higher 8
bits of the divisor. A 0x0000 value is treated like a 0x0001 value
as division by zero is not allowed.The Divisor Latch Access Bit
(DLAB) in U_LCR must be one in order to access the UART Divisor
Latches. */

#define U_DLL(A)                                    \
  LPC_REG((A) + 0x00)

#define U_DLM(A)                                    \
  LPC_REG((A) + 0x04)


/* The U_IER is used to enable UART interrupt sources. */

#define U_IER(A)                                    \
  LPC_REG((A) + 0x04)


```

```
/* The U_IIR provides a status code that denotes the priority and
   source of a pending interrupt. The interrupts are frozen during an
   U_IIR access. If an interrupt occurs during an U_IIR access, the
   interrupt is recorded for the next U_IIR access. */

#define U_IIR(A)                                        \
  LPC_REG((A) + 0x08)


/* The U_FCR controls the operation of the UART Rx and TX FIFOs. */

#define U_FCR(A)                                        \
  LPC_REG((A) + 0x08)


/* The U_LCR determines the format of the data character that is to be
   transmitted or received. */

#define U_LCR(A)                                        \
  LPC_REG((A) + 0x0C)


/* The U_LSR is a read-only register that provides status information
   on the UART TX and RX blocks. */
```

```c

#define U_LSR(A)                                         \
  LPC_REG((A) + 0x14)



lpcUart_s *LinkedUart[2];


void uartVectAddr(void)  __attribute__ ((interrupt("IRQ")));



err32_t lpcInitUart(lpcUart_s *Uart)
{
  Uart->Baudrate = DEF_BAUDRATE;

  Uart->Num = DEF_UART_NUM;

  Uart->VectAddr = (reg32_t)uartVectAddr;

  lpcInitBuf(&Uart->RxBuf);
  lpcInitBuf(&Uart->TxBuf);
  lpcInitVic(&Uart->Vic);

  lpcAssignBuf(&Uart->RxBuf,
               Uart->RxBufMem,
               sizeof(Uart->RxBufMem));
  lpcAssignBuf(&Uart->TxBuf,
               Uart->TxBufMem,
```

```
                    sizeof(Uart->TxBufMem));



  // return successful
  return(LPC_SUCC);
}


err32_t lpcEnableUart(lpcUart_s *Uart)
{
  int32_t PinReg;
  reg32_t DivReg;
  reg32_t DummyReg;

  Uart->BaseAddr = UART0_BASE_ADDR + Uart->Num*(
      UART1_BASE_ADDR - UART0_BASE_ADDR);

  PinReg = Uart->Num<<4;

  DivReg = (CCLK)/(Uart->Baudrate<<4);

  LinkedUart[Uart->Num] = Uart;

  lpcSetTypeVic(&Uart->Vic,
                    LPC_UART_0_VIC + Uart->Num);

  lpcSetVectAddrUart(Uart,
                    (reg32_t)uartVectAddr);

```

```
// Disable Uart interrupt enable (UIER)
U_IER(Uart->BaseAddr) = LPC_CLR_REG;

// Clear UIIR, URBR, ULSR
DummyReg = U_IIR(Uart->BaseAddr);
DummyReg = U_RBR(Uart->BaseAddr);
DummyReg = U_LSR(Uart->BaseAddr);

// Set Register 0 for TX and RX
PINSEL0 &= ~(1<<(PinReg+3));
PINSEL0 |=  (1<<(PinReg+2));
PINSEL0 &= ~(1<<(PinReg+1));
PINSEL0 |=  (1<<(PinReg));

// Configure line control register (ULCR)
U_LCR(Uart->BaseAddr) = (1<<7) | (1<<1) | (1<<0);

// 9.3.2 Uart0 Divisor Latch Registers (UDLL)
U_DLL(Uart->BaseAddr) = DivReg & (0xFF);

// 9.3.2 Uart0 Divisor Latch Registers (UDLM)
U_DLM(Uart->BaseAddr) = (DivReg >> 8) & (0xFF);

// 9.3.9 Uart0 Line Control Register (ULCR)
U_LCR(Uart->BaseAddr) = (1<<1) | (1<<0);

// 9.3.8 Uart FIFO Control Register (UFCR)
U_FCR(Uart->BaseAddr) = (1<<0);

// Enable Uart Vic and Buf
```

```
    lpcEnableBuf(&Uart->RxBuf);
    lpcEnableBuf(&Uart->TxBuf);
    lpcEnableVic(&Uart->Vic);

    // Enable Uart Interrupt Enable Register (UIER)
    U_IER(Uart->BaseAddr) = (1<<1) | (1<<0);

    return(LPC_SUCC);
}



err32_t lpcSetVectAddrUart(lpcUart_s *Uart,
                           const reg32_t VectAddr)
{
    lpcSetVectAddrVic(&Uart->Vic,
                      VectAddr);

    return(LPC_SUCC);
}



err32_t lpcSetNumUart(lpcUart_s *Uart,
                      const int32_t Num)
{
    Uart->Num = Num;

    // return success
    return(LPC_SUCC);
```

```
}



err32_t lpcSetBaudRateUart(lpcUart_s *Uart,
                           const int32_t BaudRate)
{
  Uart->Baudrate = BaudRate;

  // return success
  return(LPC_SUCC);
}



err32_t lpcSetFlowCntlUart(lpcUart_s *Uart,
                           const bool_t FlowCntl)
{
  Uart->FlowCntl = FlowCntl;

  // return success
  return(LPC_SUCC);
}



void uartVectAddr(void)
{
  lpcUart_s *Uart;
  byte_t DummyByte;
```

```
reg32_t UIIR;

Uart = LinkedUart[0];
UIIR = U_IIR(Uart->BaseAddr);

if( (UIIR & (1<<0)) != 0 )
  {
    Uart = LinkedUart[1];
    UIIR = U_IIR(Uart->BaseAddr);
  }

// Figure out which port caused interrupt
while( (UIIR & (1<<0)) == 0 )
  {
    // Check type of interrupt
    switch( UIIR & ( (1<<3) | (1<<2) | (1<<1) ) )
      {
        // Case receive line status
      case( (1<<2) | (1<<1) ):
        DummyByte = U_LSR(Uart->BaseAddr);
        break;

        // Case received data available
      case( (1<<2) ):
        lpcRecvUart(Uart);
        break;

        // Case character time-out indicator
      case( (1<<3) | (1<<2) ):
        DummyByte = U_RBR(Uart->BaseAddr);
```

```
          break;

          // Case THRE interrupt
        case( (1<<1) ):
          lpcSendUart(Uart);
          break;

        default:
          DummyByte = U_LSR(Uart->BaseAddr);
          DummyByte = U_RBR(Uart->BaseAddr);
          break;
        }

      UIIR = U_IIR(Uart->BaseAddr);
    }

  // Clear vector addr
  VICVectAddr = 0;
}


err32_t lpcSendUart(lpcUart_s *Uart)
{
  // local stack
  int32_t ii;
  bool_t RecvByte;
  byte_t ScratchByte;

  // if transmitter is empty
  if((U_LSR(Uart->BaseAddr) & (1<<5)) != 0)
```

```
    {
      for(ii = 0;
          ii < THRE_WRITE_SIZE;
          ii++)
        {
          // read byte from tx uart buf
          lpcReadByteBuf(&Uart->TxBuf,
                         &ScratchByte,
                         &RecvByte);

          // break if no available data in uart buf
          if(!RecvByte)
            {
              break;
            }

          // else write byte to transmitter
          else
            {
              U_THR(Uart->BaseAddr) = ScratchByte;
            }
        }
    }

  // return successful
  return(LPC_SUCC);
}


err32_t lpcRecvUart(lpcUart_s *Uart)
```

```
{
  // local stack
  byte_t ScratchByte;

  // while receiver data available
  while((U_LSR(Uart->BaseAddr) & (1<<0)) != 0)
    {
      // receive byte from receiver
      ScratchByte = U_RBR(Uart->BaseAddr);

      // write byte to uart buf
      lpcWriteByteBuf(&Uart->RxBuf,
                      ScratchByte);
    }

  // return successful
  return(LPC_SUCC);
}



err32_t lpcWriteUart(lpcUart_s *Uart,
                     const byte_t *Buf,
                     const int32_t Size)
{
  // check if valid size
  if((Size < 0) || (Size > Uart->TxBuf.Size))
    {
      return(LPC_UART_WRITE_ERR);
    }
```

```

  // write to uart buf
  lpcWriteBuf(&Uart->TxBuf,
              Buf,
              Size);

  // see if empty
  lpcSendUart(Uart);

  // return successful
  return(LPC_SUCC);
}


err32_t lpcWriteBufToUart(lpcUart_s *Uart,
                          lpcBuf_s *Buf)
{
  // local stack
  bool_t ByteRecv;
  byte_t Read;

  // write buf to uart buf
  lpcReadByteBuf(Buf,
                 &Read,
                 &ByteRecv);

  while(ByteRecv)
    {
      lpcWriteByteBuf(&Uart->TxBuf,
                      Read);
```

```
      lpcReadByteBuf(Buf,
                     &Read,
                     &ByteRecv);
    }

  // see if empty
  lpcSendUart(Uart);

  // return successful
  return(LPC_SUCC);
}



err32_t lpcReadUart(lpcUart_s *Uart,
                    byte_t *Buf,
                    const int32_t Size,
                    int32_t *BytesRecv)
{
  // check if valid size
  if((Size < 0) || (Size > Uart->RxBuf.Size))
    {
      return(LPC_UART_READ_ERR);
    }

  // read to uart buf
  lpcReadBuf(&Uart->RxBuf,
             Buf,
             Size,
             BytesRecv);
```

```
  // return successful
  return(LPC_SUCC);
}


err32_t lpcReadByteUart(lpcUart_s *Uart,
                        byte_t *Read,
                        bool_t *ByteRecv)
{
  // read to uart buf
  lpcReadByteBuf(&Uart->RxBuf,
                 Read,
                 ByteRecv);

  // return successful
  return(LPC_SUCC);
}


err32_t lpcOffsetReadUart(lpcUart_s *Uart,
                          int32_t Offset)
{
  // move read index back
  lpcOffsetReadBuf(&Uart->RxBuf,
                   Offset);

  // return successful
  return(LPC_SUCC);
}
```

```
/*
 * lpc/Vic.h
 *
 * vector interrupt control source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */


// __LPC_VIC_H__

#ifndef __LPC_VIC_H__
#define __LPC_VIC_H__


// includes

#include "lpc/Types.h"


// defines

#define LPC_ARM_CORE_0_VIC 2
```

```
#define LPC_ARM_CORE_1_VIC 3
#define LPC_TMR_0_VIC 4
#define LPC_TMR_1_VIC 5
#define LPC_UART_0_VIC 6
#define LPC_UART_1_VIC 7
#define LPC_PWM_0_VIC 8
#define LPC_I2C_0_VIC 9
#define LPC_SPI_0_VIC 10
#define LPC_SPI_1_VIC 11
#define LPC_PLL_VIC 12
#define LPC_RTC_VIC 13
#define LPC_EINT_0_VIC 14
#define LPC_EINT_1_VIC 15
#define LPC_EINT_2_VIC 16
#define LPC_EINT_3_VIC 17
#define LPC_AD_0_VIC 18
#define LPC_I2C_1_VIC 19
#define LPC_BOD_VIC 20
#define LPC_AD_1_VIC 21
#define LPC_USB_VIC 22


// vic struct

typedef struct
{
  reg32_t BaseAddr;
  int32_t IrqSlot;
  reg32_t VectAddr;
  reg32_t Type;
```

```
}
  lpcVic_s;


// function

err32_t lpcInitVic(lpcVic_s *Vic);
err32_t lpcEnableVic(lpcVic_s *Vic);

err32_t lpcSetVectAddrVic(lpcVic_s *Vic,
                          const reg32_t VectAddr);
err32_t lpcSetTypeVic(lpcVic_s *Vic,
                      const reg32_t Type);


// __LPC_VIC_H__

#endif
```

```c
/*
 * lpc/Vic.c
 *
 * vector interrupt controller source code for lpc214x
 *
 * Brian J. Julian
 *
 * bjulian{at}mit{dot}edu
 *
 * Version 0.1
 *
 * 31 March 2009
 *
 */

#include "lpc/Vic.h"

#include "lpc/Err.h"
#include "lpc/Lpc214x.h"
#include "lpc/Types.h"



// local defines
#define NUM_IRQ_SLOTS 16
#define ENABLE_VECT_CNTL (1<<5)
#define DEF_VIC_TYPE 0
```

```
/* Vector Address Register. When an IRQ interrupt occurs, the R/W IRQ
   service routine can read this register and jump to the value
   read. */

#define VIC_VECT_ADDR(A)                                \
  LPC_REG((A) + 0x100)



/* Vector control 0 register. Vector Control Registers 0-15 each R/W
   control one of the 16 vectored IRQ slots. Slot 0 has the highest
   priority and slot 15 the lowest. */

#define VIC_VECT_CNTL(A)                                \
  LPC_REG((A) + 0x200)



// global register to keep track of closed irq slots
reg32_t ClosedIrqSlots = LPC_CLR_REG;


// initialize vic
err32_t lpcInitVic(lpcVic_s *Vic)
{
  // set default vector address
```

```
  Vic->VectAddr = (reg32_t)NULL;

  // set default vector control
  Vic->Type = DEF_VIC_TYPE;

  // return success
  return(LPC_SUCC);
}



// enable vic
err32_t lpcEnableVic(lpcVic_s *Vic)
{
  // if not valid vector control type, return error
  if(Vic->Type == DEF_VIC_TYPE)
    {
      return(LPC_VECT_CNTL_ERR);
    }

  // find open vectored irq slot
  for(Vic->IrqSlot = 0;
      (ClosedIrqSlots & (1<<Vic->IrqSlot)) != 0;
      Vic->IrqSlot++);

  // if no irq slot is open, return error
  if(Vic->IrqSlot >= NUM_IRQ_SLOTS)
    {
      return(LPC_NUM_IRQ_SLOTS_ERR);
    }
```

```

  // close selected irq slot
  ClosedIrqSlots |= (1<<Vic->IrqSlot);

  // assign base register
  Vic->BaseAddr = VIC_BASE_ADDR + Vic->IrqSlot*(0x04);

  // assign vector address
  VIC_VECT_ADDR(Vic->BaseAddr) = (reg32_t)Vic->VectAddr;

  // enable vector control
  VIC_VECT_CNTL(Vic->BaseAddr) = (Vic->Type |
      ENABLE_VECT_CNTL);

  // enable interrupt request to contribute to irq
  VICIntEnable = (1<<Vic->Type);

  // return success
  return(LPC_SUCC);
}




err32_t lpcSetVectAddrVic(lpcVic_s *Vic,
                           const reg32_t VectAddr)
{
  Vic->VectAddr = VectAddr;

  // return success
```

```
  return(LPC_SUCC);
}


err32_t lpcSetTypeVic(lpcVic_s *Vic,
                      const reg32_t Type)
{
  Vic->Type = Type;;

  // return success
  return(LPC_SUCC);
}
```

Bibliography

[1] Autopilot - ascending technologies gmbh.

[2] Hummingbird - ascending technologies gmbh.

[3] Starmac - stanford testbed of autonomous rotorcraft for multi-agent control.

[4] M. Achtelik, A. Bachrach, R. He, S. Prentice, and N. Roy. Autonomous navigation and exploration of a quadrotor helicopter in gps-denied indoor environments. In *IARC First Symposium on Indoor Flight Issues*, 2009.

[5] M. Achtelik, A. Bachrach, R. He, S. Prentice, and N. Roy. Stereo vision and laser odometry for autonomous helicopters in gps-denied indoor environments. In *Proceedings of the International Society for Optical Engineering*, 2009.

[6] E. Altug, J. P. Ostrowski, and R. Mahony. Control of a quadrotor helicopter using visual feedback. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pages 72–77, 2002.

[7] E. Altug, J. P. Ostrowski, and C. J. Taylor. Quadrotor control using dual camera visual feedback. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 4294–4299, September 2003.

[8] A. Bachrach, R. He, and N. Roy. Autonomous flight in unstructured and unknown indoor environments. In *Robotics Science and Systems Workshop on Autonomous Flying Vehicles - Fundamentals and Applications*, 2009.

[9] R. W. Beard, T. W. McLain, D. B. Nelson, D. Kingston, and D. Johanson. Decentralized cooperative aerial surveillance using fixed-wing minature uavs. *Proceedings of the IEEE*, 94(7):1306–1324, July 2006.

[10] J. Black and T. Ellis. Multi camera image tracking. *Image and Vision Computing*, 11:1256–1267, 2006.

[11] Y. Bouktir, M. Haddad, and T. Chettibi. Trajectory planning for a quadrotor helicopter. In *Proceedings of the 16th Mediterranean Conference on Control and Automation*, pages 1258–1263, June 2008.

[12] F. Bullo, J. Cortes, and S. Martinez. *Distributed Control of Robotic Networks.* June 2008. Manuscript preprint. Electronically available at http://coordinationbook.info.

[13] Q. Cai and J. K. Aggarwal. Tracking human motion in structured environments using a distributed-camera systemsing a distributed-camera system. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(12):1241–1247, 1999.

[14] J. R. Clapper, J. J. Young, J. E. Cartwright, and J. G. Grimes. Unmanned systems roadmap 2007-2032. Technical report, Secretary of Defense, DoD, December 2007.

[15] R. Collins, A. J. Lipton, H. Fujiyoshi, and T. Kanade. Algorithms for cooperative multisensor surveillance. *Proceedings of the IEEE*, 89(10):1456–1477, 2001.

[16] J. Cortes, S. Martinez, T. Karatas, and F. Bullo. Coverage control for mobile sensing networks. *IEEE Transactions on Robotics and Automation*, 20(2):243–255, April 2004.

[17] Y. Cui, K. A. Hua, and A. Phillips. Leveraging 1-hop neighborhood knowledge for efficient flooding in wireless ad hoc networks. In *Proceedings of the 24th IEEE International Performance, Computing, and Communications Conference*, pages 347–354, April 2005.

[18] Z. Drezner. *Facility Location: A Survey of Applications and Methods.* Springer Series in Operations Research. Springer-Verlag, New York, 1995.

[19] J. Escareno, S. Salazar-Cruz, and R. Lozano. Embedded control of a four-rotor uav. In *Proceedings of the American Control Conference*, 2006.

[20] S. G. Fowers, D.-J. Lee, B. J. Tippetts, K. D. Lillywhite, A. W. Dennis, and J. K. Archibald. Vision aided stabilization and the development of a quad-rotor micro uav. In *Proceedings of the International Symposium on Computational Intelligence in Robotics and Automation*, pages 143–148, June 2007.

[21] R. Franz, M. Milam, and J. Hauser. Applied receding horizon control of the caltech ducted fan. In *Proceedings of the American Control Conference*, 2002.

[22] A. Ganguli, J. Cortes, and F. Bullo. Maximizing visibility in nonconvex polygons: Nonsmooth analysis and gradient algorithm design. In *Proceedings of the American Control Conference*, pages 792–797, Portland, OR, June 2005.

[23] M. F. Godwin, S. C. Spry, and J. K. Hedrick. *A Distributed System for Collaboration and Control of UAV Groups: Experiments and Analysis*, volume 588 of *Lecture Notes in Economics and Mathematical Systems*, chapter 9, pages 139–156. Springer Berlin Heidelberg, March 2007.

[24] D. Gurdan, J. Stumpf, M. Achtelik, K.-M. Doth, G. Hirzinger, and D. Rus. Energy-efficient autonomous four-rotor flying robot controlled at 1 khz. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 361–366, April 2007.

[25] Z. J. Haas. A new routing protocol for the reconfigurable wireless networks. In *Proceedings of the IEEE 6th International Conference on Universal Personal Communications Record*, volume 2, pages 562–566, October 1997.

[26] T. Hamel, R. Mahony, R. Lozano, and J. Ostrowski. Dynamic modelling and configuration stabilization for an x4-flyer. In *Proceedings of the 15th IFAC World Congress*, 2002.

[27] J. Hauser, S. Sastry, and G. Meyer. Nonlinear control design for slightly non-minimum phase systems: application to v/stol aircraft. *Automatica*, 28(4):665–679, 1992.

[28] R. He, S. Prentice, and N. Roy. Planning in information space for a quadrotor helicopter in a gps-denied environment. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1814–1820, May 2008.

[29] E. Hecht. *Optics.* Addison Wesley, Reading, MA, 3 edition, 1998.

[30] G. M. Hoffmann, D. G. Rajnarayan, S. L. Waslander, D. Dostal, J. S. Jang, and C. J. Tomlin. The stanford testbed of autonomous rotorcraft for multi agent control (starmac). In *Proceedings of the 23rd Digital Avionics Systems Conference*, 2004.

[31] G. M. Hoffmann and C. J. Tomlin. Decentralized cooperative collision avoidance for acceleration constrained vehicles. In *Proceedings of the 47th IEEE Conference on Decision and Control*, pages 4357–4363, December 2008.

[32] G. M. Hoffmann and C. J. Tomlin. Autonomous multi-agent search and rescue: Algorithms and experiments. In *Robotics Science and Systems Workshop on Autonomous Flying Vehicles - Fundamentals and Applications*, 2009.

[33] G. M. Hoffmann, S. L. Wasl, and C. J. Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, August 2007.

[34] O. Holland, J. Woods, R. DeNardi, and A. Clark. Beyond swarm intelligence: The ultraswarm. In *Proceedings of the IEEE Swarm Intelligence Symposium*, 2005.

[35] J. P. How, B. Bethke, A. Frank, D. Dale, and J. Vian. Real-time indoor autonomous vehicle test environment. *IEEE Control Systems Magazine*, 28(2):51–64, April 2008.

[36] J. P. How, E. King, and Y. Kuwata. Flight demonstrations of cooperative control for uav teams. In *Proceedings of the AIAA 3rd Unmanned Unlimited Technical Conference*, Chicago, IL, September 2004.

[37] B. J. Julian, M. Schwager, M. Angermann, and D. Rus. Location-based algorithm for multi-hopping state estimates within a distributed robot team. In *Field and Service Robotics*, Cambridge, MA, July 2009.

[38] C. Kemp. *Visual Control of a Miniature Quad-Rotor Helicopter.* Doctoral thesis, Churchill College, University of Cambridge, 2006.

[39] S. M. Khan and M. Shah. Consistent labeling of tracked objects in multiple cameras with overlapping fields of view. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(10):1355–1360, 2003.

[40] Seung kook Yun, David Alan Hjelle, Hod Lipson, and Daniela Rus. Planning the reconfiguration of grounded truss structures with truss climbing robots that carry truss elements. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Kobe, Japan, May 2009.

[41] S. Martinez, J. Cortes, and F. Bullo. Motion coordination with distributed information. *IEEE Control Systems Magazine*, 27(4):75–88, 2007.

[42] Meister, N. Frietsch, C. Ascher, and G. F. Trommer. Adaptive path planning for a vtol-uav. In *Proceedings of the IEEE/ION Position, Location, and Navigation Symposium*, pages 1252–1259, May 2008.

[43] A. Mokhtari and A. Benallegue. Dynamic feedback controller of euler angles and wind parameters estimation for a quadrotor unmanned aerial vehicle. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2004.

[44] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 151–162, 1999.

[45] Philips. *Volume 1: LPC214x User Manual*, July 2006.

[46] L. C. A. Pimenta, V. Kumar, R. C. Mesquita, and G. A. S. Pereira. Sensing and coverage for a network of heterogeneous robots. In *Proceedings of the IEEE Conference on Decision and Control*, Cancun, Mexico, December 2008.

[47] P. Pounds, R. Mahony, J. Gresham, P. Corke, and J. Roberts. Towards dynamically-favourable quad-rotor aerial robots. In *Proceedings of the Australian Conference on Robotics and Automation*, 2004.

[48] P. Pounds, R. Mahony, P. Hynes, and J. Roberts. Design of a four-rotor aerial robot. In *Proceedings of the Australian Conference on Robotics and Automation*, 2002.

[49] R. Rao, V. Kumar, and C. J. Taylor. Planning and control of mobile robots in image space from overhead cameras. In *Proceedings of the International Conference on Robotics and Automation*, Barcelona, Spain, April 18-22 2005.

[50] A. Ryan, X. Xiao, S. Rathinam, J. Tisdale, M. Zennaro, D. Caveney, R. Sengupta, and J. K. Hedrick. A modular software infrastructure for distributed control of collaborating uavs. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, August 2006.

[51] S. Salazar-Cruz, A. Palomino, and R. Lozano. Trajectory tracking for a four rotor mini-aircraft. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 2505–2510, December 2005.

[52] M. Schwager, B. J. Julian, and D. Rus. Optimal coverage for multiple hovering robots with downward facing cameras. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2009.

[53] M. Schwager, D. Rus, and J. J. Slotine. Decentralized, adaptive coverage control for networked robots. *International Journal of Robotics Research*, 28(3):357–375, March 2009.

[54] D. Shim, H. Chung, H. J. Kim, and S. Sastry. Autonomous exploration in unknown urban environments for unmanned aerial vehicles. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, 2005.

[55] H. Shim, T. J. Koo, F. Hoffmann, and S. Sastry. A comprehensive study of control design for an autonomous helicopter. In *Proceedings of the 37th IEEE Conference on Decision and Control*, 1998.

[56] M.-T. Sun and T.-H. Lai. Location aided broadcast in wireless ad hoc network systems. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, volume 2, pages 597–602, March 2002.

[57] J. W. R. Taylor. *Jane's Pocket Book of Remotely Piloted Vehicles: Robot Aircraft Today.* Collier Books, New York, third edition, 1977.

[58] R. Teo, J. S. Jang, and C. J. Tomlin. Automated multiple uav flight: The stanford dragonfly uav program. In *Proceedings of the 43rd IEEE Conference on Decision and Control*, Paradise Island, Bahamas, 2004.

[59] G. P. Tournier, M. Valenti, J. P. How, and E. Feron. Estimation and control of a quadrotor vehicle using monocular vision and moire patterns. In *Proceedings of the AIAA Guidance, Navigation and Control Conference*, August 2006.