Final Documentation:
The Development of an Autonomous Hovercraft System

Team Calvin

Matthew Buckle
Mario Chiu
Jeffrey Spieldenner
Clement Suhendra

Table of Contents

Introduction

(1) The Problem

The overall goal of this project was to design and construct a swarm of three hovercrafts which would advance along a predefined path of four waypoint beacons. The swarm would progress by following a leader hovercraft, which would be responsible for following the waypoints, and avoiding collisions with each other. Multiple questions needed to be pondered before progressing with the project: How would the hovercrafts recognize the waypoint beacons? How would they navigate towards the beacons? How would they know when they reached a given beacon? What would be used as waypoints? The further we delved into these questions, the more complex our system became.

Two main ideas were discussed to solve the problem of waypoint signal recognition and control of the hovercraft: using a system of two Telos motes on either side of the hovercraft in a Master and Slave combination sending a signal to a series of relays which would control the thrust motors, or using a system of two Chipcon receivers attached to a microcontroller board with an H-bridge controlling each motor. Both of these strategies have their positives and negatives. The Telos motes are relatively simple to use, but are expensive and can only control the hovercraft motors via a series of switches (thus limiting the ways in which the hovercraft can be controlled). The Chipcon receivers and microcontroller system is inexpensive and allows more versatility in controlling the motors of the hovercraft, but learning how to use a microcontroller is slow and tedious work. After becoming frustrated with the control limitations offered by the Telos Mote system, we decided to move forward with the customizable options that microcontroller system offered.

Despite the fact that we abandoned the idea of using the Telos motes as the receivers on the hovercraft that would be used to control the thrust motors, we decided to continue to use them as the waypoint beacons from which the hovercraft system would receive its various signals. Although the motes were not the ideal instruments to control the hovercraft due to the limitations mentioned above, they had multiple properties which made them perfectly suited to be the waypoint beacons. The ease of (re)programming the strength and period of the signals being sent, their small size, and the LEDs embedded on the board (which we could program to give a visual of when a given signal was being transmitted) were all properties which would prove to be beneficial as we began to test our system.

(2) System Requirements

The hovercraft system must…

- be rigid and stable
- be able to track its target
- be able to determine if the beacon is to its right or to its left
- not collide with other hovercrafts or waypoint beacons
- be able to differentiate between attract and repel signals
- recognize when it has reached the target beacon
- know what to do if the next beacon's signal is not in range
- be able to vary the speed of its thrust motors

The hovercraft system must be rigid and stable

Any system that can be expected to perform at its peak in an outdoor setting must be able to withstand the wear and tear that such a setting can be expected to produce. As such, care had to be taken to ensure that our hovercraft system was durable enough to be able to maintain its functionality in a less than ideal setting, and yet still be light enough to be able to hover well.

There are two main components that make up the body of a hovercraft: the chassis and the skirt. Even with the most careful of planning, the hovercraft system will occasional collide into foreign objects, thus its chassis must be able to withstand these impacts and yet still continue on its mission. Even though it is hovering off the ground, the skirt will still occasionally come into contact with a course surface, thus it must not be able to tear easily. Given these requirements, we decided to create the chassis out of corrugated plastic sheets, a material that is very rigid and yet still lightweight. Our first skirts were made out of a lightweight vinyl material; however, we were unable to find the same product to produce more skirts. Due to this, we created our other skirts out of heavy duty garbage bags, which performed similarly to the vinyl skirts when attached to the hovercrafts.

The hovercraft system must be able to track its target

Whether it is the leader tracking a waypoint beacon, or the pursuers following the leader, the hovercrafts must know what their specific target is and must be able to move towards it. To achieve this goal, we implemented an "attractive" signal in the waypoint beacons and in the leader's Chipcons that the hovercrafts could receive and track.

The hovercraft system must be able to determine if the beacon is to its right or to its left

Given that the hovercraft is expected to be an autonomous system, it must be able to steer itself in the direction of the targeted waypoint beacon or leader hovercraft. The first step in achieving this is determining from which direction the signal is coming. This was accomplished by installing a Chipcon receiver on each side of the hovercraft and by placing a parabolic aluminum shield behind each Chipcon. These aluminum shields serve two purposes: to focus the signal on the receiver that is closest to the beacon, and to weaken the signal received by the beacon that is further away from the beacon (as the

signal has to pass through at least one shield, depending on the positioning of the beacon in relation to the hovercraft).

The hovercraft system must not collide with other hovercrafts or waypoint beacons

For any swarm to operate at its peak efficiency, care must be taken to ensure that the individuals who make up the swarm avoid coming into unwanted physical contact with each other and with their targets. As such, we installed "repellant" signals[1] in the waypoint beacons and in the all of hovercrafts' Chipcons. After receiving these signals, the hovercrafts were programmed to take evasive action in order to avoid a collision.

The hovercraft system must be able to differentiate between attract and repel signals

Since there could be two distinct and opposite signals being received by the Chipcon at any given time, the hovercraft system must be able to differentiate between them. To achieve this, a variable called "beacon_type" was placed into the packet[2]. Whenever a signal is received by a Chipcon, it immediately determines whether the beacon type is attractive or repellant, and makes its control decisions based upon this determination.

The hovercraft system must recognize when it has reached the target beacon

For the hovercraft swarm to progress from beacon to beacon, the leader must be able to recognize when it has successfully approached its current target beacon. This is accomplished through the use of the waypoint beacons' repellant signals as well as the waypoint beacons' individual identification numbers. The intelligence of the leader hovercraft starts by looking for beacon number one. When it receives a user-defined number of repel signals from this beacon, it "knows" that it has successfully reached the target, and moves on to the next waypoint.

The hovercraft system must know what to do if the next beacon's signal is not in range

As mentioned above, when the leader hovercraft reaches the repel signal of the target beacon, it immediately begins to look for the next waypoint. However, what if the next waypoint's signal is not in range? In this case, the hovercraft will remain at its current target beacon. When the repel signal is in range, it will move away from the beacon, and when the repel signal is out of range, it will move back towards it. One possible addition that can be made is to have the hovercraft look for any attractive signal in range and react accordingly instead of moving around a single beacon.

The hovercraft system must be able to vary the speed of its thrust motors

[1] The repellant signal is significantly weaker than the attractive signal.
[2] The "packet" is the data transmitted by the Telos waypoint beacons and the Chipcon. Data in the packet includes the beacon_id (the number of the beacon), Beacon_type (attractive or repellant), and a timestamp (used to ensure that the signal being compared by the right and left Chipcon on any given hovercraft was sent at the same time).

One of the major benefits of using the Chipcon / Microcontroller / H-bridge subsystem versus the Telos / Switch subsystem is the more fine-tuned control that is possible. The motors can move forward, backwards, and have varying speeds via relatively simple software commands. By taking advantage of the speed control offered by the H-bridges, we can have the hovercraft move faster while far from the beacon, and slower as it approaches. This level of speed control will allow the hovercraft to maintain stability[3] while maneuvering around the target beacon.

(3) High Level Description

The successful completion of our project had the following goals at the time of conception:

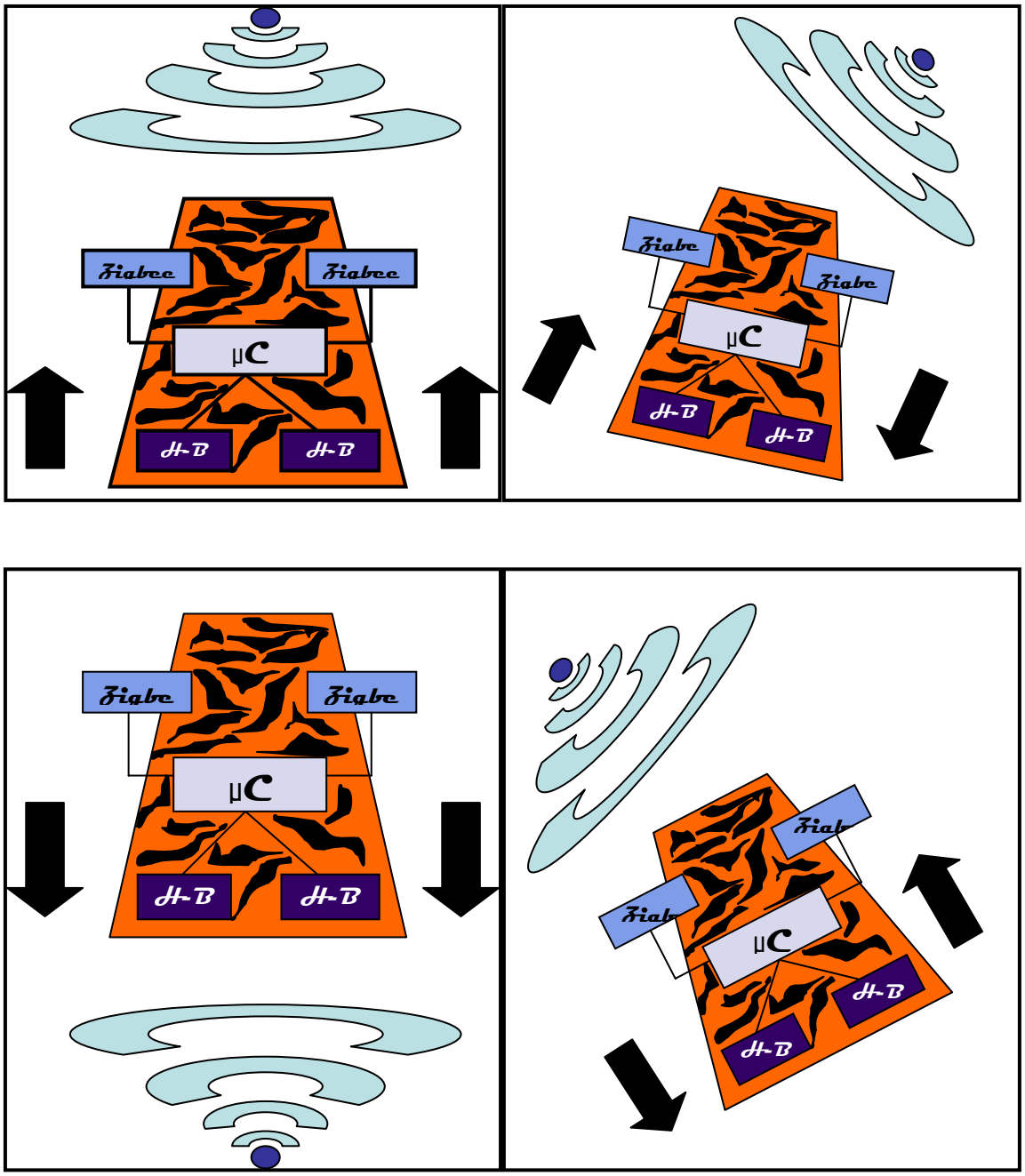
- Waypoint navigation system
- Three-point control
- Creation of an H-bridge to allow for linear control
- Navigation of waypoint system by two independent hovercrafts
- Obtaining materials for and construction of two more hovercrafts
- Navigation of waypoint system by four hovercrafts working as a swarm
- Transfer of system from one which uses Telos motes to one that utilizes Zigbee transmitters and microcontrollers
- Possible change from two thrust fan system to a single fan with a controllable rudder

The following will describes the goals individually and includes a short assessment of the goal.

Waypoint Navigation System

The purpose of this goal was to allow a single hovercraft to traverse a series of over two transmitting beacons (waypoints). As all of the beacons send their signals throughout the test, the hovercraft needs to be able to recognize the signals sent by the current waypoint that it is looking for. It will know when it has reached the waypoint when it receives a repel signal, and then will begin to look for the next one. The process starts over from the beginning once the final waypoint has been reached. We successfully completed this goal.

[3] Since the hovercraft is a low friction system, it loses less momentum when the thrust is turned off. Thus, while turning at high speeds, the hovercraft will make very wide turns – not what we want when it is approaching the target beacon.



Three Point Control

In this goal, the control of the hovercraft was be improved. Previously, the control scheme was a relatively simple "Bang-bang" control, in which the hovercraft turns in the direction of whichever receiving beacon (Master on the left side, Slave on the right side) receives a stronger signal from the waypoint transmitter. In three point control, the both the left and the right thrust fans will be fired almost simultaneously (the current circuitry does not allow for both to be on at the same time) if the transmitter is directly in front of the hovercraft. This goal was superseded in importance by the H-Bridge control, discussed below.

Creation of H-bridge to Allow for Linear Control

Through the creation of an H-bridge circuit, we are able to vastly improve the control of the hovercrafts. Rather than simply having Bang-bang or Three-point control (which have obvious limitations), the hovercrafts will be able to head towards the beacon with a much more control, less oscillations in motion and in a tunable manner. We successfully completed this goal.

Navigation of Waypoint System by Two Independent Hovercrafts

To allow two (or more) independent hovercrafts to successfully navigate the waypoints, we must ensure that no collisions take place between them as they move around the testing area. In order to prevent this, we installed a repellant beacon in one (or both) of the motes on the hovercraft which will allow other hovercrafts to recognize when it is approaching another hovercraft and thus avoid colliding with it. This goal works but not reliably so we will be leaving out for our demonstration.

Obtaining Materials for and Construction of Two More Hovercrafts

In order to proceed with further testing, it was necessary to build more hovercrafts. As our previous source for materials (www.hovercraftmodels.com) has temporarily gone out of business, we must use other suppliers to provide us with the individual materials needed to construct them. We successfully found our materials for the hovercrafts we built.

Navigation of Waypoint System by Four Hovercrafts Working as a Swarm

The end of this goal was to have four hovercrafts traversing the waypoint system in unison. To achieve this, we needed to designate one hovercraft as the leader, and have the other three follow it from beacon to beacon. An attractive signal sent from one of the motes on the leader will allow the other three to successfully follow it. This goal ended up being outside the scope of our project as demanded by our customer Dr. Bauer.

Replacement of Telos motes with Zigbee Transmitters and Microcontrollers

One important goal for this project was to remove the reliance on Telos motes, which are very expensive and have shown to be unreliable. The replacement was to be a microcontroller board with a couple of Zigbee transmitters. We accomplished this by creating a board with a PIC18F4620 microcontroller and two CC2420 transceiver chips.

Implementation of a Single Thrust Fan / Rudder Combination

With the implementation of an H-bridge, it is possible to exchange our current thrust fan configuration with one that can more properly utilize the benefits of the H-bridge. By limiting the hovercraft to one thrust fan, the longevity of the battery should

improve. However, during our project we decided that this had no obvious benefits to the performance of the system, given that battery life was no longer an issue.

Detailed Project Description

(1) System Theory of Operation

The overall Autonomous Hovercraft System is made up of four separate, but integrated, subsystems: the microcontroller board, the H-bridge, the hovercraft, and the waypoint beacons. Each of these four subsystems provides a vital role in the success of the Autonomous Hovercraft System. The microcontroller board contains both Chipcon transmitters, which are responsible for receiving packets from the beacons, and the microcontroller, which is responsible for taking the data received by both Chipcons, making control decisions based upon this information, and sending these control decisions to the two H-bridges. The H-bridges take the command sent by the microcontroller and directly manipulate the speed and direction of the two thrust fans. The hovercraft houses the above two subsystems, as well as the two thrust motors and the lift motor. The beacon subsystem is responsible for sending the signals that the autonomous hovercraft is tracking. While the complexity of each of the four subsystems is not the same, the success of the entire project is reliant on smooth transitions between each of them.

The entire Autonomous Hovercraft System begins with the beacon subsystem. Each beacon has three key components: the period[4], the attractive signal strength, and the repellant signal strength[5]. The smaller the time the period is, the more exact the Autonomous Hovercraft System can be in tracking the target beacon. If the period is too small, however, and there are too many beacons in range sending signals, the airwaves can become cluttered and the Chipcons can lose packets. If there are few signals being sent in the immediate area, a period of 100 ms works well, but if it is a more complex system of beacons and hovercrafts, a period of 200 or 250 ms is more appropriate.

The hovercraft subsystem is responsible for housing the microcontroller and H-bridge subsystems, as well as the thrust and lift motors. The body consists of corrugated plastic, a strong but lightweight material that is easy to cut and shape into the pieces that are needed. The skirt can either be made out of a lightweight vinyl material or heavy duty trash bags, either work equally well. The microcontroller board is mounted in a slit towards the front of the hovercraft chassis: the microcontroller itself and most of the circuitry are hidden inside of the body of the hovercraft, while the two Chipcon boards and their respective aluminum shielding are exposed on the exterior. The H-bridges, on the other hand, are more towards the back the hovercraft. Like the circuitry of the

[4] The period value is inputted in milliseconds
[5] Signal strength values are inputted as integers ranging from 0 (weakest) to 31 (strongest).

microcontroller board, they are hidden inside the body of the hovercraft. All that is visible are the wires that extend from the H-bridges to the respective thrust motors that they control, located on a platform raised about the body. The lift fan is also located inside the body, in between the microcontroller board and the H-bridges.

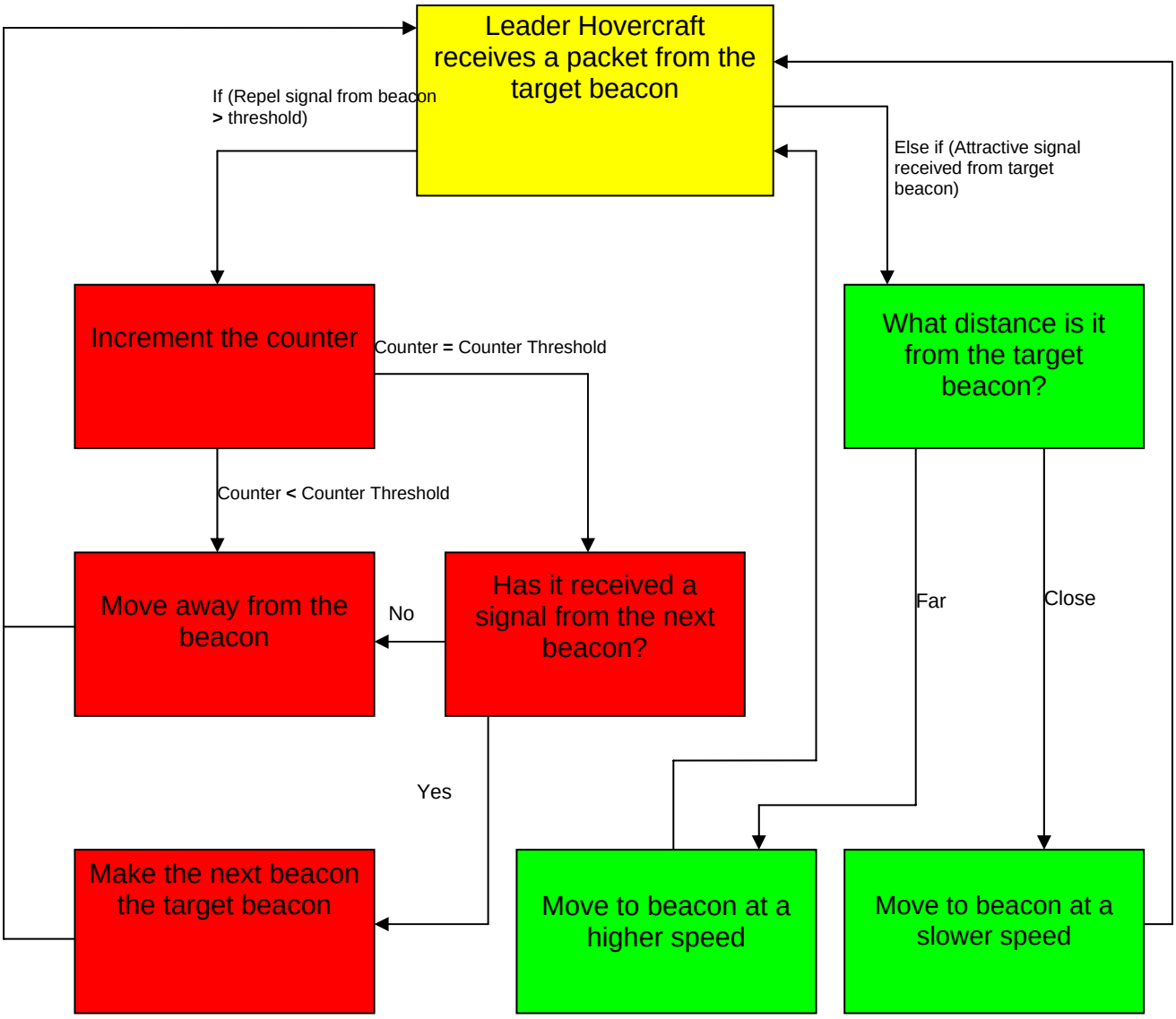
The microcontroller subsystem does the brunt of the work of the Autonomous Hovercraft System. With dimensions of 2.25" by 12", it is long enough so that both of the Chipcon receivers and their aluminum shielding can protrude from the body of the hovercraft, while the remainder of the circuitry remains comfortably inside. When a beacon transmits a signal, both Chipcon receivers obtain the packet, and an interrupt is fired. When this occurs, the microcontroller retrieves the data (beacon type, beacon id, timestamp, and RSSI[6]) from both of the packets, decides whether the hovercraft should move towards or away from the beacon depending on the strength of the repel signal received, decides which direction to turn by comparing the RSSI values of both of the Chipcons, and, if the RSSI values are very close (plus or minus 2 dbm) decides how fast to move based upon the strength of the RSSI value.

The H-bridge is the final subsystem of the Autonomous Hovercraft System. After deciding on the direction and speed of the hovercraft, the microcontroller calls a function and brings the H-bridge into play. The H-bridge, mounted in the rear of the body of the hovercraft, receives the commands sent by the microcontroller and directly controls the movement (forward, reverse, or stopped) and speed (via the duty cycle) of its specific motor. The hovercraft moves in the direction specified by both of the H-bridges, and the entire process is repeated as the next set of packets is received by the Chipcons.

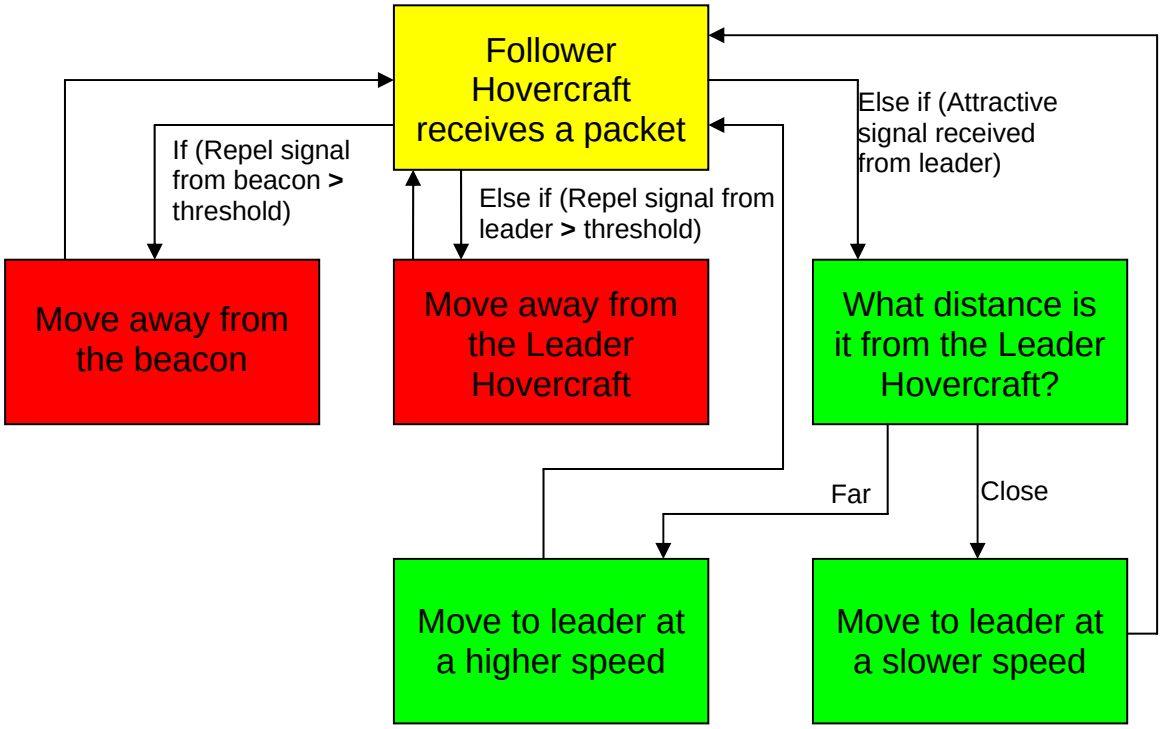
[6] Signal strength of the packet

(2) System Block Diagram

Block Diagram of the Leader Hovercraft



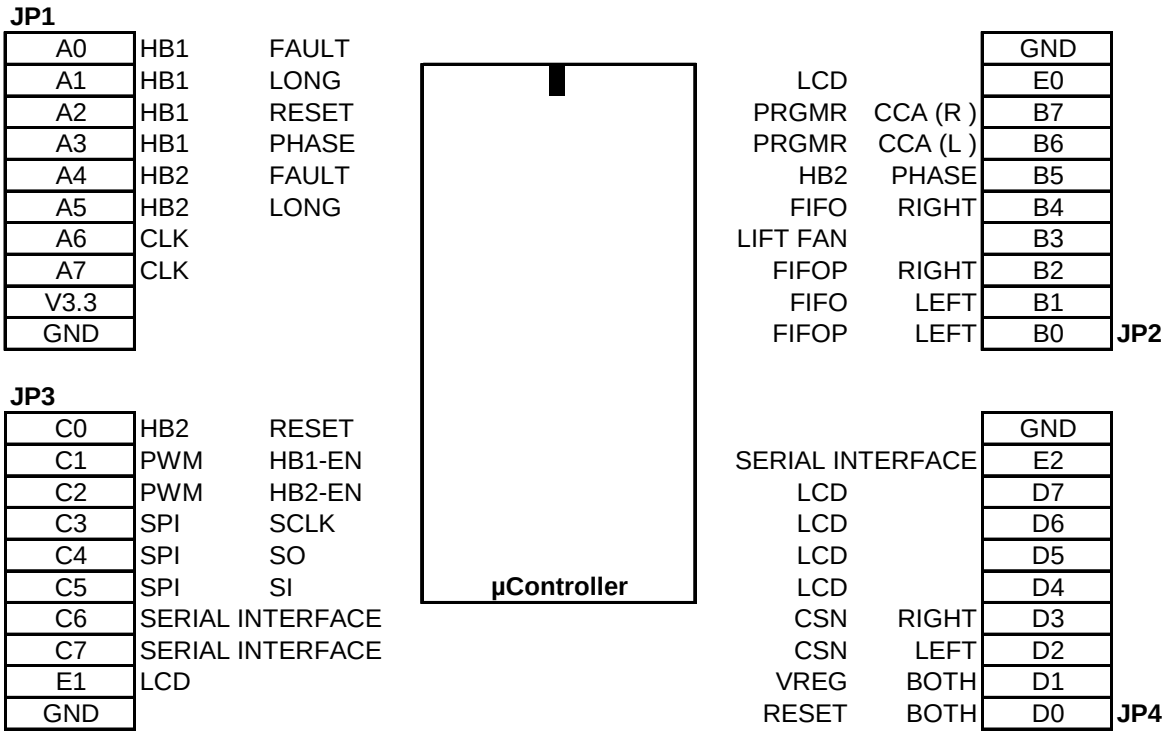
Block Diagram of the Follower Hovercrafts



(3) The Microcontroller Board Subsystem

The microcontroller used in our board was Microchip's PIC18F4620. We chose this microcontroller because of our familiarity with it, resulting from the first semester Senior Design class and the tasks we carried out in the class. Additionally, it has 35 input/output pins which we found to be fitting under the design we were pursuing. It is connected to an external 10 MHz crystal oscillator that provides the clock timing.

The functionality of the board includes having a serial interface which we use to communicate to the computer via the HyperTerminal program. This is meant mostly for troubleshooting and testing. The board also has an LCD display which is also meant for troubleshooting and testing, especially in a testing environment where connecting the board to a computer is not feasible. For the radio communications we use two CC2420, by Chipcon, and interface to them via the SPI functionality of the microcontroller. The CC2420 communicate via Zigbee (IEEE 802.15.4) at the 2.4 GHz band. The microcontroller also interfaces with the two H-bridges that drive the thrust motor on the hovercraft. Below is a schematic of what the pin connections look like on the board itself. This diagram may come in handy when troubleshooting using a digital analyzer or a scope.



Microcontroller and Code

The microcontroller is programmed via an external programmer, provided by Dr. Schafer, which loads all the code into the flash memory in the microcontroller. We used the SourceBoost software to code, compile and link our code and Microchip software is used to program the board. Our board is programmed in the same manner as the board use by Dr. Schafer in his Senior Design class. All the necessary software is available in all the computers found in the learning center.

Our code is structured so that we have a main file and several libraries that the main file refers to. Our main file is called hovercraft.c and it contains the algorithm that we want the hovercraft to run, and anything directly related to it. EESDlib.c contains all the routines that deal with the serial interface and the LCD display. Our version of this file is built upon the version provided by Dr. Schafer for the Senior Design class, but most of the serial interface code was written by Team Calvin. Hbridelib.c contains all the functions used to control both H-bridges. Both PWM and pin settings for the H-Bridges are defined here and speed and directional control happens with the functions included in this file. cc2420lib.c contains all the functions that deal with the configuration and use of the CC2420 transceiver chips. It also contains the code that deals with setting up the SPI in master mode to and to use this interface. cc2420.h contains all the definitions used in the c file, including definitions obtained from the CC2420 datasheet.

In order to find the correct settings for a microcontroller function, be it interrupts, timers, SPI interface or I/O the first step is exploring the PIC18F4620 datasheet. All the information necessary to set up any capability is found there and the simplest way to do this is by first looking for a particular function and finding a table like the example

below. This table list all the registers associated with the SPI interface, and it serves as a guide to keep exploring the datasheet to configure the capability correctly.

TABLE 17-2: REGISTERS ASSOCIATED WITH SPI™ OPERATION

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	49
PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	52
PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	52
IPR1	PSPIP(1)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	52
TRISA	TRISA7(2)	TRISA6(2)	PORTA Data Direction Control Register						52
TRISC	PORTC Data Direction Control Register								52
SSPBUF	SSP Receive Buffer/Transmit Register								50
SSPCON1	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	50
SSPSTAT	SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF	50

Legend: Shaded cells are not used by the MSSP in SPI mode.

Note 1: These bits are unimplemented on 28-pin devices and read as '0'.

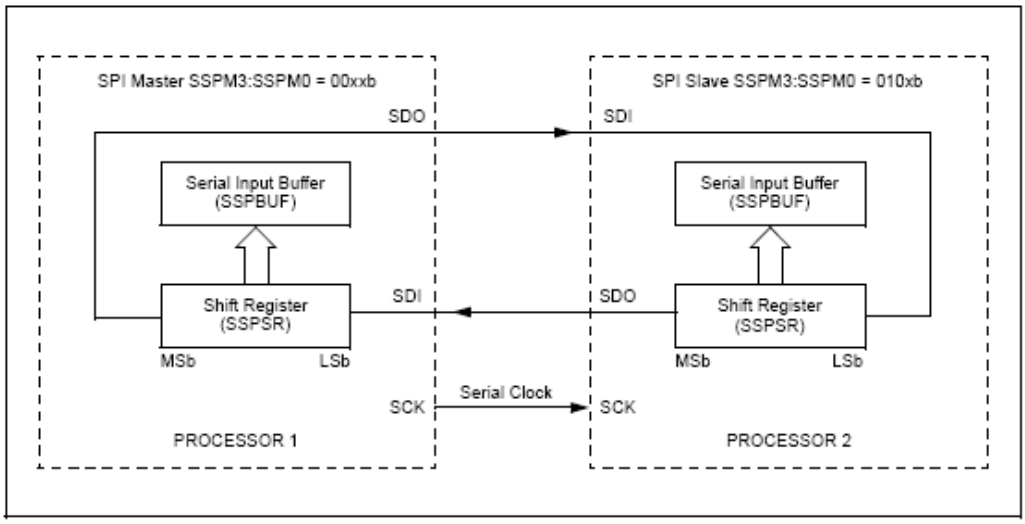
2: PORTA<7:6> and their direction bits are individually configured as port pins based on various primary oscillator modes. When disabled, these bits read as '0'.

A few important considerations addressed in the code setting up interrupts triggered by timers and by certain I/O pins, configuring all the pins as inputs or outputs as necessary, obtaining a PWM signal with a variable duty-cycle and configuring the SPI and serial interfaces used in by other devices.

SPI Interface

The Serial Peripheral Interface is a type of communications interface that is used by the CC2420 and that the PIC18F4620 has capability of handling. It works under a Master/Slave configuration where the Master initiates the communications and the slave responds to these prompts. The communication is handled by four I/O pins: serial clock (SCK), serial output (SO), serial input (SI) and a chip select (CSN). The clock is used to synchronize the two devices involved, the input and output transmit the bits and the chip select activates the slave and prompts the communication to take place. Below is a diagram of how the SPI works in the PIC18F4620. In our board both CC2420s interface with the microcontroller via the SPI, CSN pins are enabled-low.

FIGURE 17-2: SPI™ MASTER/SLAVE CONNECTION



As shown in the diagram above, all SPI communications are handled by the SSPBUF register. All messages to be sent are to be loaded here and all messages received are read here as well. As soon as one byte is sent out from the SSPBUF, the slave loads another byte into the SSPBUF. This byte can either be an acknowledgement of communication or a piece of data, depending on the device and what is sent to it.

CC2420

The CC2420 by Chipcon is a single chip ZigBee system compliant with the IEEE 802.15.4 set of standards. It is built for low power, low voltage and low data rate wireless applications. As mentioned earlier, it interfaces with the microcontroller via SPI and we felt that it would be easier to interface with this chip than with an alternative that used a serial interface. SPI timing considerations are well illustrated and addressed in the CC2420 datasheet.

The most basic commands of the CC2420 are carried out with what is called ‘command strobes’. They are single-byte commands that are use to initialize and configure the CC2420 as well as to carry out functions such as transmission. Immediately after a command strobe is issue a status byte is return. However, the status byte does not reflect the current command strobe, only the previous ones. The SNOP command, which does nothing else than return a status byte is useful here. For further information as to the content of the status byte, refer to the CC2420 datasheet, page 29. The list of command strobes as defined in the CC2420 datasheet is shown below.

Address	Register	Register type	Description
0x00	SNOP	S	No Operation (has no other effect than reading out status-bits)
0x01	SXOSCON	S	Turn on the crystal oscillator (set XOSC16M_PD = 0 and BIAS_PD = 0)
0x02	STXCAL	S	Enable and calibrate frequency synthesizer for TX; Go from RX / TX to a wait state where only the synthesizer is running.
0x03	SRXON	S	Enable RX
0x04	STXON	S	Enable TX after calibration (if not already performed) Start TX in-line encryption if SPI_SEC_MODE ≠ 0
0x05	STXONCCA	S	If CCA indicates a clear channel: Enable calibration, then TX. Start in-line encryption if SPI_SEC_MODE ≠ 0 else do nothing
0x06	SRFOFF	S	Disable RX/TX and frequency synthesizer
0x07	SXOSCOFF	S	Turn off the crystal oscillator and RF
0x08	SFLUSHRX	S	Flush the RX FIFO buffer and reset the demodulator. Always read at least one byte from the RXFIFO before issuing the SFLUSHRX command strobe
0x09	SFLUSHTX	S	Flush the TX FIFO buffer
0x0A	SACK	S	Send acknowledge frame, with pending field cleared.
0x0B	SACKPEND	S	Send acknowledge frame, with pending field set.
0x0C	SRXDEC	S	Start RXFIFO in-line decryption / authentication (as set by SPI_SEC_MODE)
0x0D	STXENC	S	Start TXFIFO in-line encryption / authentication (as set by SPI_SEC_MODE), without starting TX.

The CC2420 also contains registers that store configuration information, among other things. Through these registers you can do things such as choosing the channel to be used for transmission (there are 11 channels available). All register store two bytes of information and most of these registers are both readable and writable. A comprehensive list of registers and their individual functions is given in the CC2420 datasheet, starting on page 63.

The only two registers that are not two bytes long are the TXFIFO and the RXFIFO. These are 128-byte FIFO registers (the first byte in is first byte out) and handle the transmission and reception payloads. They are interfaced in the same manner that the rest of the register but they have a few extra considerations to be taken into account. For the RXFIFO, when a message is received the FIFO output pin in the microcontroller goes high indicating that there is a packet in the register. If the length of the packet exceeds a threshold number of bytes that is configurable in a register, then the FIFOP flag goes up. In our code, this threshold is set to be the same length of the packets expected and an interrupt flag goes up as soon as the packet is completely in the register. The first byte out always indicates the length of the packet stored in the register and the next-to-last byte indicates the RSSI value in 2's complement. In our code we decided to disregard the negative sign of the dB readings coming from the CC2420 for simplicity's sake.

The TXFIFO requires that the first byte written into is the length byte and that the subsequent bytes comply with this length. If this is not the case a TX_UNDERFLOW is reported in the status byte. As soon as the register is loaded with a packet, this can be

transmitted issuing a STXON or a STXONCCA command. The main difference between these two is that the latter will only transmit the packet if there are no other transmissions taking place on the same channel (Clear Channel Assessment is successful). If STXONCCA is not successful this can be read from the status byte received after transmission. It is always recommended that the channel be assessed before any transmissions. This can also be done by checking the CCA output pin in the CC2420 before any transmission; however we decided to go with the first strategy for our code.

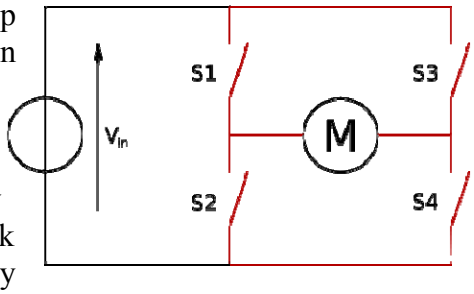
In order to have packet transmission compliant with the IEEE 802.15.4 set of standards we had to implement the CSM-CA algorithm outline in the documentation of the standard and illustrated by Figure 61 in that document. This algorithm basically involves waiting a random number of back-off periods (time units) and to retry transmission up to 4 times before considering the transmission a failure. For each new try, the range of random back-off periods to wait grows. In our code this is implemented by using timer0 as the timer that controls the waiting time between tries.

(4) The H-Bridge Subsystem

The system that we inherited from the summer project does not have a speed control only directional control, which we strongly believe to be one of the causes why the hovercraft is unstable. A series connection of 3 to 4 diodes is the only limiting factor for the battery power to fully power the motor. Even with this setup, the speed of the hovercraft is still considered to be fast and powerful. The directional control is based on two relays circuits, which switch from one motor to the other. The switching is based on the Telos motes reading of the beacons. The default switch is on the right motor. When the readings of the signals of the beacons are the same, the hovercraft has to go forward by switching the right and left motors back and forward. When it is stronger on the right side, the hovercraft will solely turn on the right motor until the signals on the left side are stronger than the right side, and so on.

As mentioned in the Introduction section, the H-Bridge subsystem provides a better control of the motors. Not only that the H-Bridge offers a better directional control, but also speed control that is not available on the previous system. To do so, we make several numbers of decisions, such as the number of motors and parts (Power Mosfet vs. Relays) to be used in the new system.

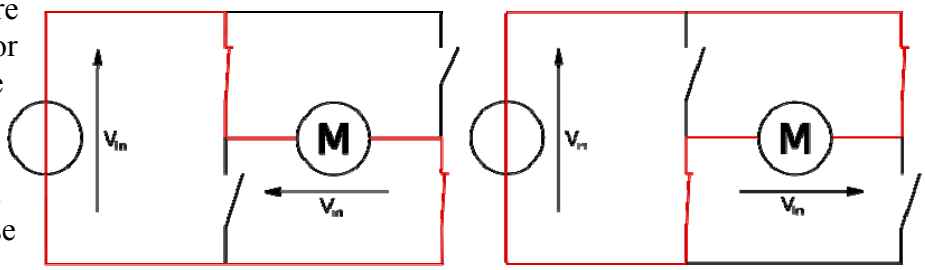
One of the options that we could have taken is using a servo to control one motor with a rudder. Instead of doing this, we decide to keep the two motors. This decision is made based on research on the parts and the old hovercraft body design. The trade off that we take into account is to have more weight in the system, with the two motors. While we save ourselves from creating a new body design for the one motor option, which we think would be more troublesome. Creating a new body



design is not in our critical path. Thus going back to our proposal that we presented in the fall, our goal for this project is to improve performance, which can be completed by pairing a motor with an H-Bridge circuit.

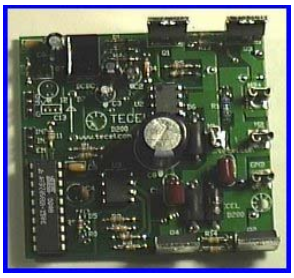
What is an H-Bridge? A circuit diagram resembles the letter "H", consisting of four-transistors. The load is the horizontal line, connected between two pairs of intersecting lines. It is very common in DC motor-drive applications where switches are used in the "vertical" branches of the "H" to control the direction of current flow, and thus the rotational direction of the motor.

An H-Bridge circuit offers a lot more than the two relays circuits. H-Bridge can run a motor not only forward, but also in reverse. In the early process of deciding whether we are pairing each motor with an H-Bridge circuit or not, we find that with this forward and reverse command



abilities, it would be better for us to use these abilities. By having one H-Bridge per motor, we are expecting a better performance in a way that the hovercraft can make a smoother, faster, and sharper turns. This is achieved by putting one motor forward and the other reverse. We then combine this performance with the speed control, so that the turns performed would not cause instability of the system. For example, for a right turn, we turn on the right motor and put the left motor on reverse. With a lower speed, we would be able to manage the torque created in a sudden turn. The directional algorithm of the new system is still the same as the previous system. Decisions made by the microcontroller are dependent to the reading of signals' strength received by the Chipcon radio chips. A forward action, both motors turn forward, is due to the same signal strength received by the left and right Chipcon chips.

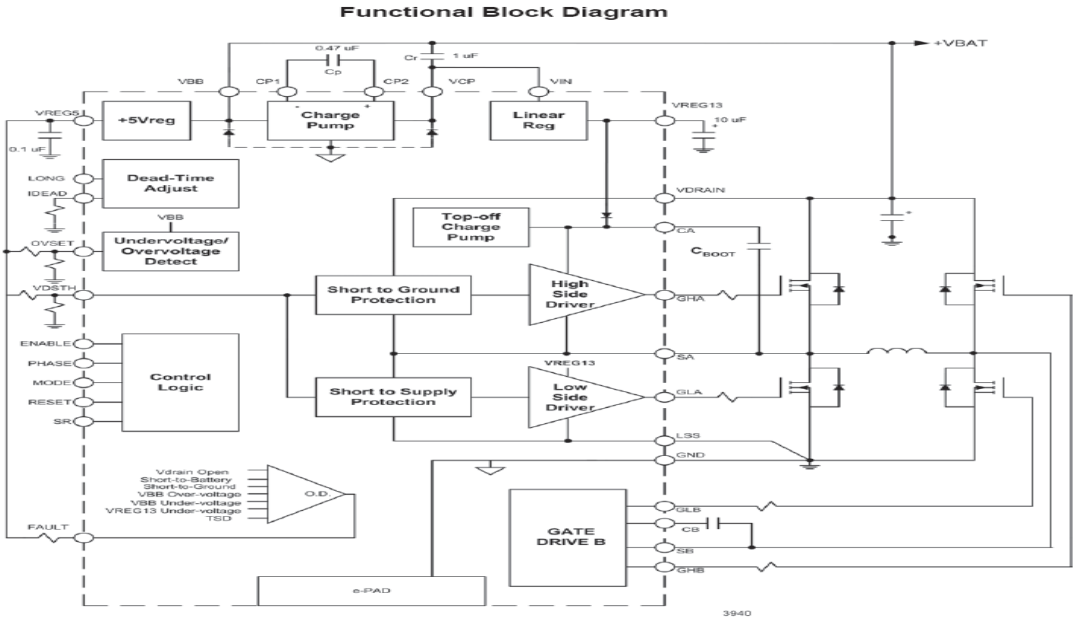
We think the best additional feature that H-Bridge circuits offers to our project is the speed control. Not being able to control the speed on the previous system is a big disadvantage. This feature is achieved by firstly sending PWM signal to the motor, provided by the microcontroller, and secondly inputting the percentage of duty cycle. The higher the duty cycle, the faster the motor will go. A 100% duty cycle is equivalent to what the previous system has, full power, not using PWM signal. By using low duty cycle (10 – 20%), the hovercraft becomes more stable. For our final boards, we find that in certain frequencies (over 1000 Hz) in our setup, we need to have a jumpstart. It is sending a 100% duty signal for a couple milliseconds to start the motor and then dropping it to a duty cycle less than 50%. In inputting the values, refer to the formulas of 'pr2' in the microcontroller data sheet.



In the process of creating of our own board, we started by making a purchase of a 10-A H-Bridge from Tecel.com to help us in getting a big picture idea. This part works very well in a way that we can run the motor bi-directionally. What we then realize is this Tecel board does not use an H-Bridge driver, to drive the MOSFETs a programmable logic driver is used. From the recommendation from Dr. Schafer, I start looking at H-Bridge driver parts from Allegro Microsystem Inc. They offer two different parts: a half-bridge driver, A3946, and a full-bridge driver, A3940. The full-bridge part becomes our choice, because we want the capability to have the motor to run forward and reverse, an upgrade from the inherited system.

After building and testing the A3940 circuit on the datasheet in a breadboard (see next page), I find this part to work and satisfy the requirements for the system. The only problem in my testing environment is that there is not enough current to run the motor from this circuit in the breadboard, but I can hear the motor is spinning on the inside. The breadboard has a fuse of 1A, and from previous testing the motor at least need 7A. The values of resistors and capacitors in the block diagram can be found in the datasheet. Thus Dr. Schafer helps me in this testing process by building a PCB board so that I can run the motor with enough current.

This subsystem involves both software and hardware. The software is not as complicated as the Chipcon subsystem. For the most part, the software for this subsystem engages on setting values of the applicable registers internally in the microcontroller and sending logic 0 or 1 to the H-Bridge board. The hardware for this subsystem is consists of a full-bridge driver (A3940, mentioned above), four-MOSFET, a number resistors and capacitors. The microcontroller board and the H-Bridge boards are connected with a bundle of wire with 10-pin Molex pin connectors on the three boards



Software

Generating PWM Signal

This Microchip controller capability is implemented by setting the values of the duty cycle register (CCPRxL and CCPxCON), timer control register (T2CON), and PWM period register (PR2). CCPRxL and PR2 values are calculated from the PWM duty cycle and period formulas in the PIC datasheet (Matlab code to calculate these values, the function can be found in the Appendices section). In our final code, the calculation is not implemented in the code. I use the Matlab code to obtain the values of pr2 and CCPRxL and CCPxCON, and use the values in the duty cycle function. PIC18F4620 is able to provide two PWM outputs in pin c1 and c2. Keep in mind that the output of CCP1CON is in c2, while CCP2CON is c1.

EQUATION 15-1:

PWM Period = [(PR2) + 1] • 4 • TOSC • (TMR2 Prescale Value)

EQUATION 15-2:

PWM Duty Cycle = (CCPRxL:CCPxCON<5:4>) • TOSC • (TMR2 Prescale Value)

For more details, this information can be found in Chapter 15, Capture/Compare/PWM modules.

```
//1. Setting the pwm period by writing to the PR2 register = 0x9C (hex)
      pr2 = 156;

//2. Setting the pwm duty cycle by writing to the CCPRxL register and
CCPxCON<5:4> bits in this case CCPR1L & CCP1CON (with 40% duty cycle);
PWM Duty Cycle = 4.0192e-4 s

     ccpr1l = 125 >> 2;
     ccpr2l = 376 >> 2;
     //ccpr1l = 00011111b;             //for 20% duty cycle
     //ccpr2l = 01011110b;             //for 60% duty cycle

     //writing to CCP1CON
//5. Configuring the CCPx module for PWM operation (for PWM mode: 11xx)
     ccp1con.3 = 1;
     ccp1con.2 = 1;
     ccp1con.1 = 0;
     ccp1con.0 = 0;

     //writing to CCP2CON
     ccp2con.3 = 1;
     ccp2con.2 = 1;
     ccp2con.1 = 0;
     ccp2con.0 = 0;

//3. make the CCPx pin an output by clearing the appropriate tris bit
     trisc.2 = 0;       //the output of CCP1 is c2, not c1
     trisc.1 = 0;       //the output of CCP2 is c1

//4. set the TMR2 prescale value, then enable Timer2 by writing to
T2CON (TMR2 = 16 (00 = 1; 01 = 4; 1x = 16)
     t2con.1 = 1;                    //prescale value
     t2con.0 = 0;                    //prescale value

     t2con.2 = 1;                    //turning on timer2
```

Sending signals to A3940

Control Logic

PHASE	ENABLE	MODE	SR	GLA	GLB	GHA	GHB	SA	SB	Mode of Operation
0	1	X	X	1	0	0	1	Lo	Hi	Reverse
0	0	0	1	0	1	1	0	Hi	Lo	Fast decay, SR enabled
0	0	1	1	1	1	0	0	Lo	Lo	Slow decay, braking mode
0	0	0	0	0	0	0	0	Z	Z	Fast decay, coast
0	0	1	0	1	0	0	0	Lo	Z	Slow decay, SR disabled
1	1	X	X	0	1	1	0	Hi	Lo	Forward
1	0	0	1	1	0	0	1	Lo	Hi	Fast decay, SR enabled
1	0	1	1	1	1	0	0	Lo	Lo	Slow decay, braking mode
1	0	0	0	0	0	0	0	Z	Z	Fast decay, coast
1	0	1	0	0	1	0	0	Z	Lo	Slow decay, SR disabled

NOTES: All faults will coast the motor, i.e., GHA = GHB = GLA = GLB = 0 to switch off all bridge MOSFETs.
X = Indicates a "don't care".
Z = Indicates a high-impedance state.

Fault Responses

Fault Mode	RESET	FAULT	CP Reg.	VREG13	VREG5	GHx	GLx
No Fault	1	0	ON	ON	ON	–	–
Short-to-Battery①②	1	1	ON	ON	ON	0	0
Short-to-Ground①③	1	1	ON	ON	ON	0	0
Open Bridge (V_{DRAIN})①④	1	1	ON	ON	ON	0	0
V_{REG13} Undervoltage	1	1	ON	ON⑤	ON	0⑥	0⑥
V_{BB} Overvoltage	1	1	ON	ON	ON	0	0
V_{BB} Undervoltage	1	1	OFF	OFF	ON⑤	0⑥	0⑥
Thermal Shutdown	1	1	OFF	OFF	ON⑤	0⑥	0⑥
Sleep	0	1	OFF	OFF	OFF	Z	Z

RESET. Control input to put device into minimum power consumption mode and to clear latched faults. Logic "1" enables the device; logic "0" triggers the sleep mode. It is internally pulled down via 50 k Ω resistor.

ENABLE. Logic "1" enables direct control of the output drivers via the PHASE input, as in PWM controls, and ignores the MODE and SR inputs. It is internally pulled down via 50 k Ω resistor.

MODE. Logic input to set the current decay mode. Logic "1" (slow-decay mode) switches off the high-side MOSFET in response to a PWM "off" command. Logic "0" (fast-decay mode) switches off both the high-side and low-side MOSFETs. It is internally pulled down via 50 k Ω resistor.

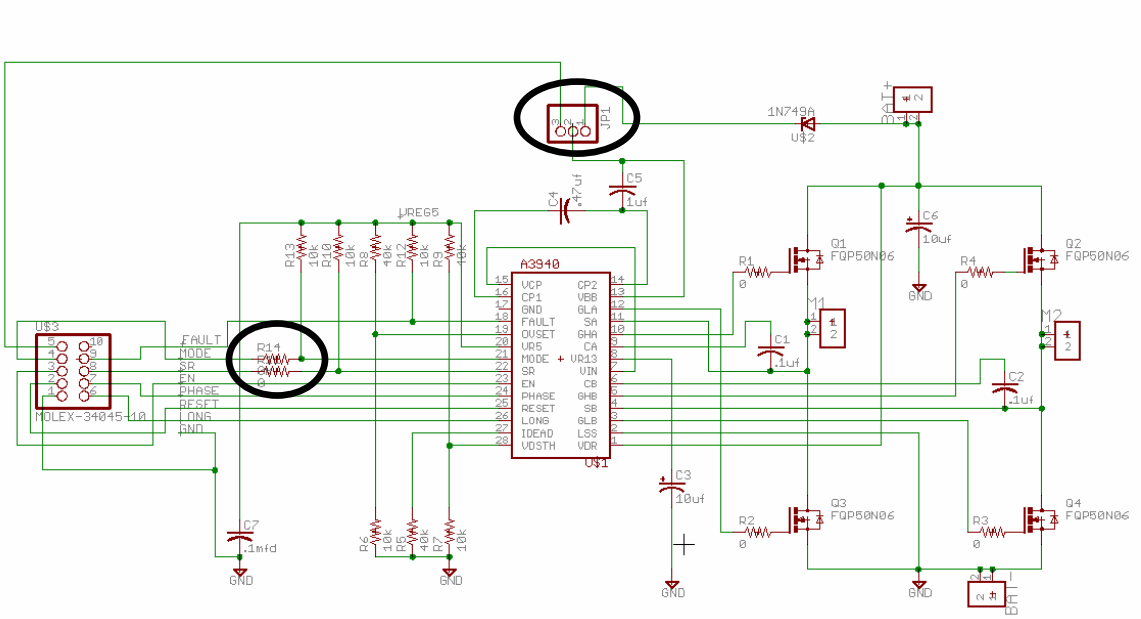
PHASE. Motor direction control. When logic is "1", it enables gate drive outputs GHA and GLB by allowing current flow from SA to SB. When logic is "0", it enables GHB and GLA allowing current flow from SB to SA. It is internally pulled down via 50 kΩ resistor.

SR. When logic "1", enables synchronous rectification; logic "0" disables the synchronous rectification. It is internally pulled down via 50 k Ω resistor.

FAULT. Open drain, diagnostic logic output signal. When logic is "1", it indicates that one or more fault conditions have occurred. Use an external pull-up resistor to VREG5 or to digital controller. Internally causes a coast when asserted. See also Functional Description, next page.

LONG. When logic is "1", it selects long dead time between GHx and GLx transitions of same phase. When logic is "0", it selects short dead times. It is internally pulled down via 50 k Ω resistor.

The possible ten modes of operations from the Truth Table are more than enough for the performance that we want for the hovercraft. Hence with the limited amount of pins that we are using and with the advice of Dr. Schafer, SR and MODE are set to be high by leaving of the resistors, to open the connection to these input pins. The schematic below is based on the functional block diagram above, with an addition of jumpers so that user can choose to power the chip using the battery power from the microcontroller or from the motor's battery.



As a result, the control input signals that we are sending from the microcontroller board are ENABLE (PWM signal), PHASE (directional), RESET (enabling the device), FAULT (to avoid fault), and LONG (to set for a short dead time). For having two H-Bridge boards, we need a total of 10 pins from the microcontroller.

The commands algorithms are:

Forward: PHASE = 1; RESET = 1; FAULT = 0
Reverse: PHASE = 0; RESET = 1; FAULT = 0
Brake: Set Dutycyle = 0%

Note: The pin connected to FAULT is supposed to be set as input, instead of output. When these pins are set as inputs, a FAULT occurred. From checking the voltages on the board and comparing it to the values on the Fault Responses table, I find that the possible cause of this fault response is VREG13 undervoltage. The voltage measured on that pin in this setup is 7 V, which is below the minimum of 12.6 V stated on the datasheet. By setting FAULT as an output from the microcontroller to the H-Bridge board, the fault response is no more, although the VREG13 is still 7V. Although it sounds impossible to set a value on an output pin, this setup works fairly well.

Hardware (schematics of both boards can be found in the Appendices section)

Part Description	Quantity (per board)
Spade Plug	4
MOSFET N-Channel 60V, 50A	4
Resistor 0 ohm	4
Resistor 10K ohm	5
Resistor 39K ohm	3
Capacitor 1 microfarad	1
Capacitor 0.1 microfarad	3
Capacitor 0.47 microfarad	1
Double Row PCB Headers (Right Angle)	1
Dual Row Micro-fit Connectors (10-pin)	1

Setup for the 10-pin Molex connection between the H-Bridge board and the microcontroller board

Pin	Dr. Schafer's H-Bridge	Microcontroller Board
1	GROUND	GROUND
2	RESET	PHASE
3	ENABLE	RESET
4	MODE	LONG
5	V_{BB}	*N/C*
6	LONG	ENABLE
7	PHASE	*N/C*
8	SR	*N/C*
9	FAULT	FAULT
10	*N/C*	V_{BB}

How the Molex pins are connected (H-Bridge's pin → Microcontroller's)

H-Bridge	Microcontroller
1	1
2	3
3	6
4	*N/C* (5 or 7 or 8)
5	10
6	4
7	2
8	*N/C* (5 or 7 or 8)
9	9
10	*N/C* (5 or 7 or 8)

Microcontroller board Pin Assignments for the H-Bridge (where the signals coming from):

	Right Motor	**Left Motor**
PHASE	B5 (pin 38)	A3 (pin 5)
FAULT	A4 (pin 6)	A0 (pin 2)
RESET	C0 (pin 15)	A2 (pin 4)
ENABLE	C2 (pin 17)	C1 (pin 16)

Subsystem Testing

To ensure the functionality of the subsystem, different testing is performed. On the hardware side, the 9-wire connections are tested using the functionality of Ohmmeter on a DMM. The purpose of this testing is to make sure that the headers are placed correctly on the slots, so then there won't be signal missing in transmission. This same test is also performed to ensure the connection on the microcontroller board. Since there are two sets of 10-pin connectors, there are two sets of the signals being sent. By doing this, I find out the set of signals is going to the left or right connector. There is no software-focused testing, other than the SourceBoost Builder and Compiler.

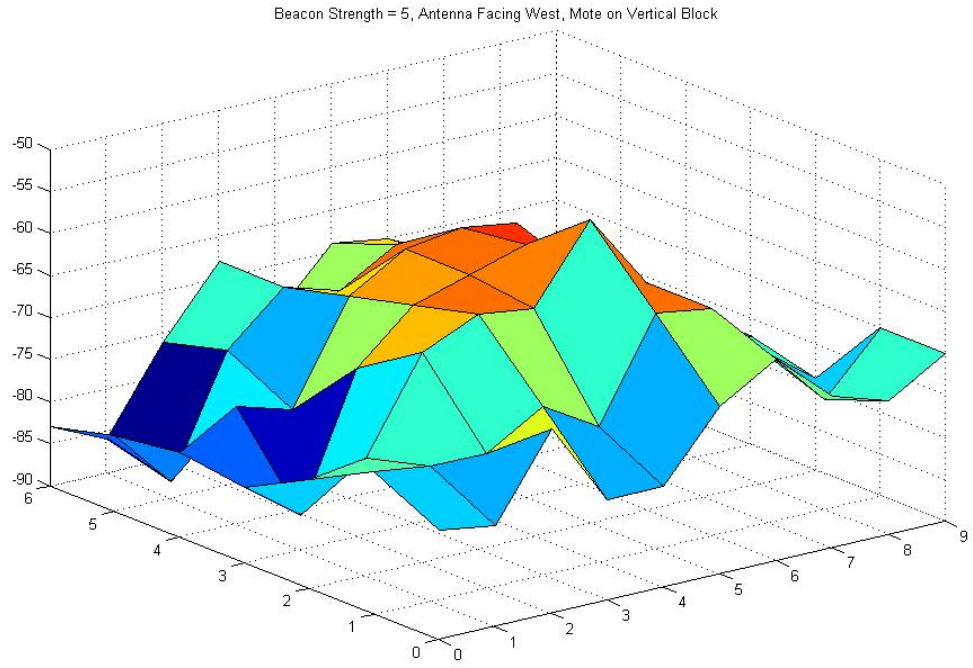
The actual test of the software is when I integrate the hardware with the software. For input to the program, the hyperterminal is used. The microcontroller is connected through a serial cable connection. Values such as duty cycle for speed control and set of phase, reset, and fault for directional control can be set through the hyperterminal. For the first couple of times, it is necessary to focus on the polarity of the motor. If you command the motor to go forward and one or both of them are going on reverse, it means that you either connect the wrong polarity of the motors to the board or switch the phase values that are being sent to the motors. One more thing to keep in mind: the same identical motor won't run the same in the same setting and sometimes the motors need a jumpstart. This can be done by starting with high percentage of duty cycle (above 50%) for less than a second, then drop the percentage of duty cycle to achieve a more stable system.

System Integration Testing

(1) How Did We Test It?

When working with wireless technologies, the testing environment is a very important factor to consider in order achieving optimal results. Small areas with several walls, such as a classroom or small lab, are bad testing environments for two reasons. First, it doesn't give much room for the hovercraft to maneuver, and second, there is large probably of signal interference due to reflections. For this reason, we decided to take out

testing to the Stepan Center in order to test in a larger environment and minimize reflections.



(2) Did Our Testing Verify the Requirements?

After testing the hovercrafts it was apparent that some of the requirements were fulfilled, but others were not. All of the subsystems functioned properly independent of the other, but when combined there were some issues. We believe that these issues consisted mostly of software uncertainties that deal with timing and interrupts. This is one possible reason why the overall system worked sometimes, but crashed at other times. So the end result was that our system functioned inconsistently and there are still a few kinks to be worked out.

However, we did demonstrate that we were able to build a stable and rigid hovercraft that traveled fairly straight when both motors were commanded to move forward. This proves that our weight distribution throughout the hovercraft was fairly consistent. We also proved that the Chipcon radios in conjunction with the microcontroller serve as a very consistent means with which to read and detect packets sent from a remote Telos beacon. These measurements were more precise than the measurements taken by the hovercraft fitted with Telos motes from last semester.

Also, we were able to show that with one H-bridge controlling one motor we were able to provide much more control over the hovercraft than we ever could have had with the switching circuitry of last semester. Last semester we were only able to achieve “bang-bang” control by rapidly switching ON and OFF the motors to go straight, but now

we are able to run both motors simultaneously which allows for a more precise beacon locating machine.

Overall, the algorithms which prove the requirements are shown to be successful in that the software is able to display information which verifies correct recognition of packets and the steps that should be taken in order to accomplish the subsequent task. However, the software was inconsistent in the manner in which it made the hovercraft physically accomplish these goals. In other words, the "brain" was unable to make its "legs" walk.

Users / Installation Manual

(1) Installation

The creation of the hovercraft and installation of the electronics is fairly straightforward. The stencil for the hovercraft base, chassis, and motor platform were originally taken from an online hovercraft company at www.hovercraftmodels.com. However, we deviated slightly from their design in order to accommodate our needs. For example, we had to construct the chassis in such a way as to fit the width of the microcontroller board and still have enough room to have the Chipcon antennae project from the side. Also, the company's design utilizes a servo-mechanical system for one thrust fan for propulsion, but we changed to two thrust fans which we could more easily control using an H-bridge circuit. The custom design and scaled measurements were documented and are listed as an attachment.

After the proper plastic pieces have been cut out in accordance with the specified measurements, they can take shape with the assistance of plastic screws to hold the respective pieces together in a nice rigid structure. Note: the locations for the screws are indicated on the hovercraft design drawings. Next, the skirt may be attached to the underside of the hovercraft base using the same screws. The skirt can be purchased from the website as part of a kit, but an alternative that works just as well is an extra-strength plastic trash bag that should be cut to an appropriate size (a little longer and wider than the hovercraft) and glued together with proper epoxy.

Finally, the motors may be attached. The lift motor is attached in the center of the Top Deck of the base, and the two thrust motors are attached to the top of the motor platform as indicated by the screw holes of the design drawings.

The microcontroller board is inserted under the "hood" of the hovercraft where it fits nicely in designated slots on both sides of the chassis. Make sure that there is enough room at the ends of the board to mount the shielding for the Chipcon radio. There are pre-drilled holes on the board for mounting, however, it might be necessary to drill larger holes in order to accommodate the larger screws needed for the right-angle mounting

piece. The H-bridge boards can be mounted on the deck of the hovercraft base directly beneath the motors in order to keep symmetry and better center of gravity.

(2) Setup

Once the equipment is properly installed, you can begin preparing the hovercraft for testing by placing the lithium polymer batteries on the hovercraft. One battery supplies the lift fan, and the other supplies the two thrust motors. Keep in mind that you would like to keep a well-balanced center of gravity, so positioning of the batteries are very important to the dynamics of the system. Also, make sure that the batteries are securely fastened to the hovercraft to avoid sliding of the components during operation.

Connect the terminals of the lift fan battery to a switch, and connect the terminals of the thrust fan battery to each H-bridge board. Also, at this time you can connect the 10-pin molex connectors from the H-bridge to the respective microcontroller board connector. And if you haven't already done so, attach the Chipcon radios to each side of the microcontroller board with the antenna facing closest to the edge of the board.

When you are ready to begin programming, insert the 9V battery onto the microcontroller board and attach the programmer connector to the appropriate jumper on the board. Also, if you would like to use the HyperTerminal for testing, then you will need to also attach the serial port cable to the board. Open the SourceBoost software program on your computer to write C code to be downloaded onto the microcontroller board.

(3) Is It Working?

Once the components are properly installed and set-up, you are then ready to begin testing. Press the Reset button on the microcontroller board to start the program over and check the results. If you observe strange results or if nothing happens, then check to make sure that the programmer connector is attached in the correct orientation. Also, it is a good idea to check that you are getting the proper supply voltage from the battery on the microcontroller board. Otherwise, review your code to make sure your initializations and functions are setup correctly.

If you have soldered your own board, then double-check the solder-joints to make sure that you are not shorting out some of the connections. Use a DMM if necessary to check voltages, resistances, et cetera.

(4) Hovercrafting for Dummies

In order to become an expert in the art of "hovercrafting", it is essential that you familiarize yourself with the microcontroller in all its glory. Thoroughly complete the Task Assignments given throughout the semester in order to gain a better understanding of the processes involved, and it will help to serve as a tutorial to learning the ropes.

Download the microcontroller datasheet from the web to gain some insight into which pins are responsible for which actions and how they can be used. And while you're at it, download the datasheets for other major components to understand how the pieces work together to create the whole system.

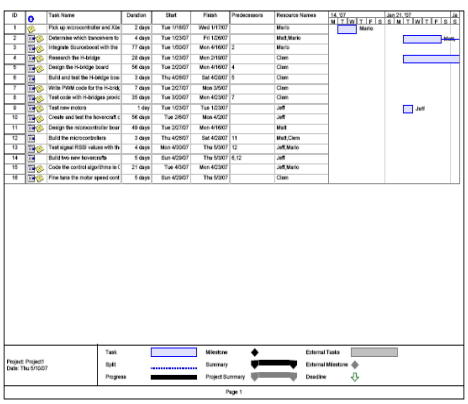
Conclusion

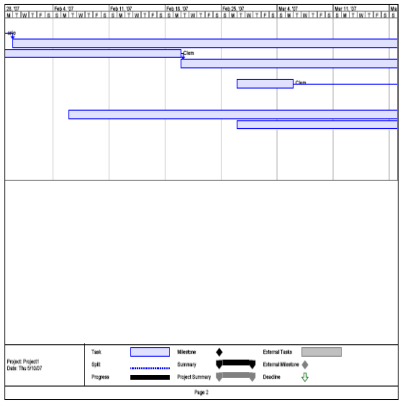
Going into the project, we had targeted a goal of having three swarming hovercrafts traversing a series of four waypoints. To achieve this, we planned on building a microcontroller board with two Zigbee transceivers which would control the thrust fans by sending commands to two H-bridges that were based upon the differences in received signal strength by the Zigbee transceivers. To accomplish this, we split the project into two main subsystems – the microcontroller and the H-bridge subsystem. The majority of the semester was spent researching how to build and program both devices, and by the end of the term, we had both subsystems working independently of each other. We when tried to run the Hovercraft system, however, we ran into problems.

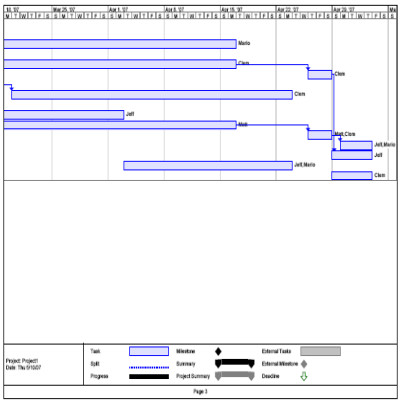
For some reason, when the microcontroller and H-bridge subsystems were connected, the Zigbee transceivers would arbitrarily stop receiving packets at varying times and the system would freeze. When disconnected, the microcontroller subsystem would successfully execute all of our code. The better part of a week was spent trying to determine the cause of the lock-up, but we were unsuccessful in achieving the goal that we had marked for ourselves at the beginning of the year. Given the tests of the two individual subsystems, we hold that this new system will perform much better than the old hovercraft system - which relied on bang-bang control – once the bug is fixed. We hope that the information provided in this report has included enough information so that whoever follows us in working on this project can quickly learn from our successes and our failures, and determine the communication problem between the H-bridge and the microcontroller subsystems.

Appendix

(1) The Gantt Chart



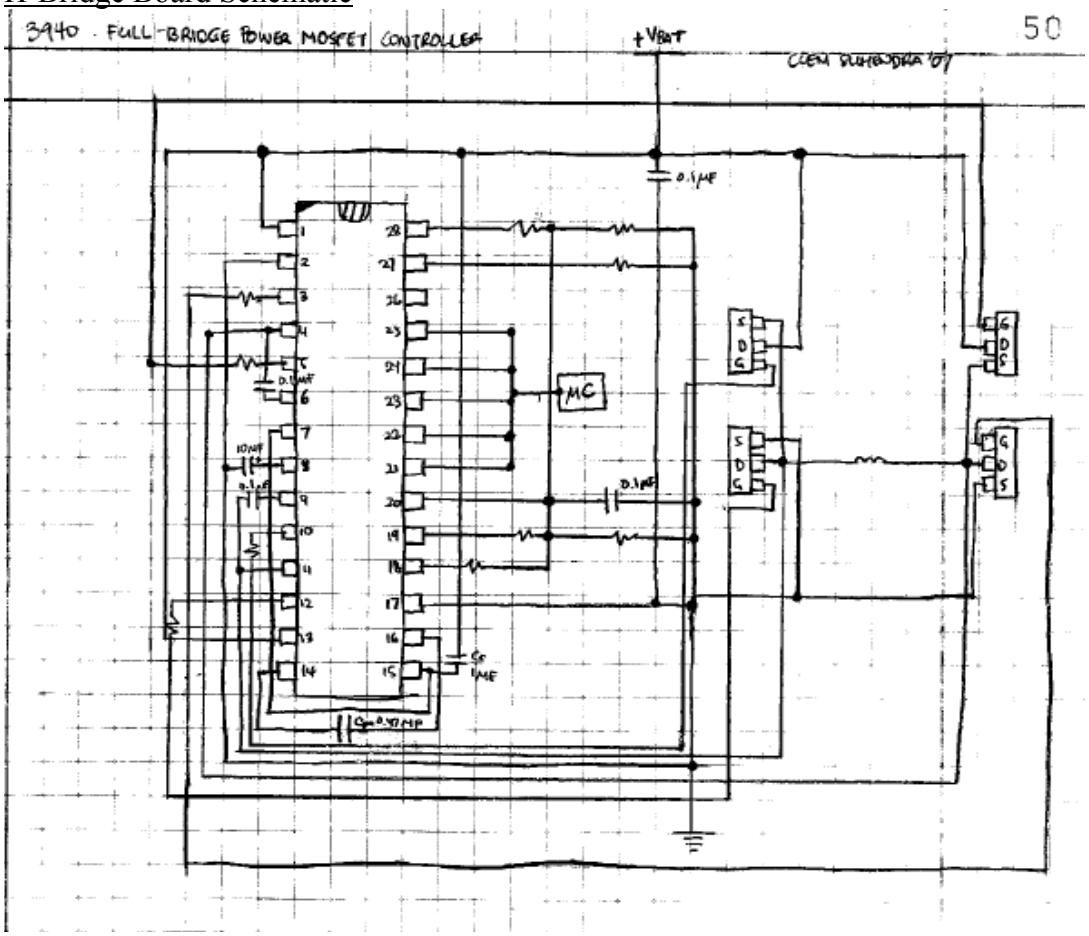




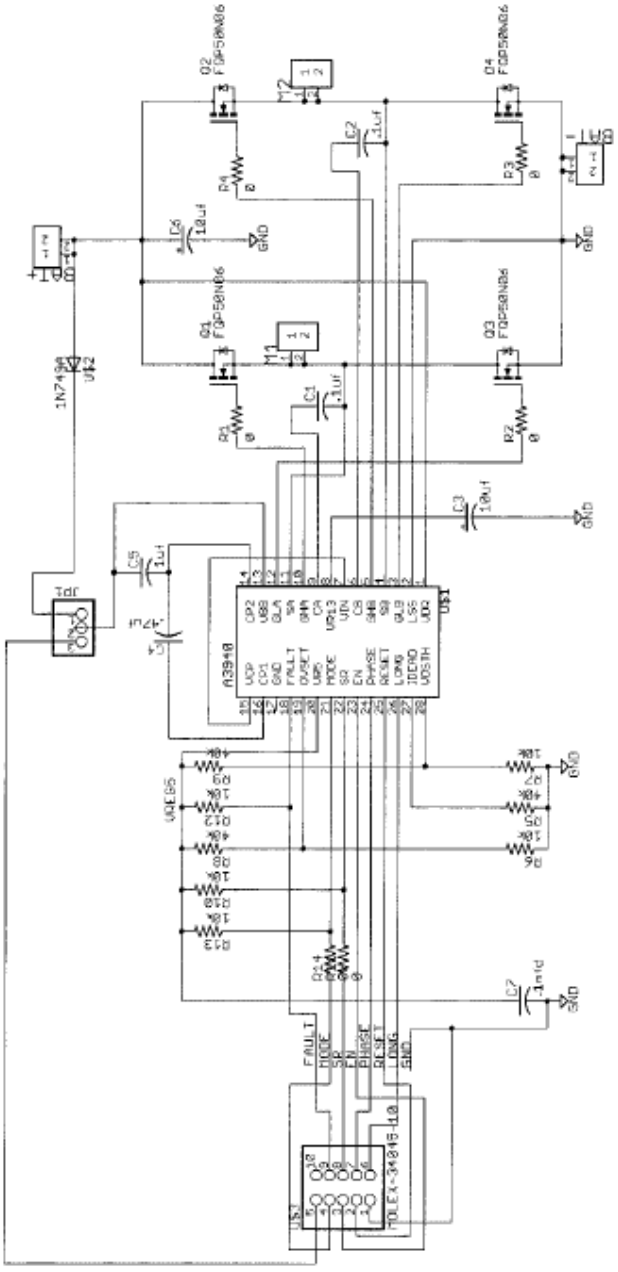
The complete file can be found on the website or the file bucket CD.

(2) Hardware

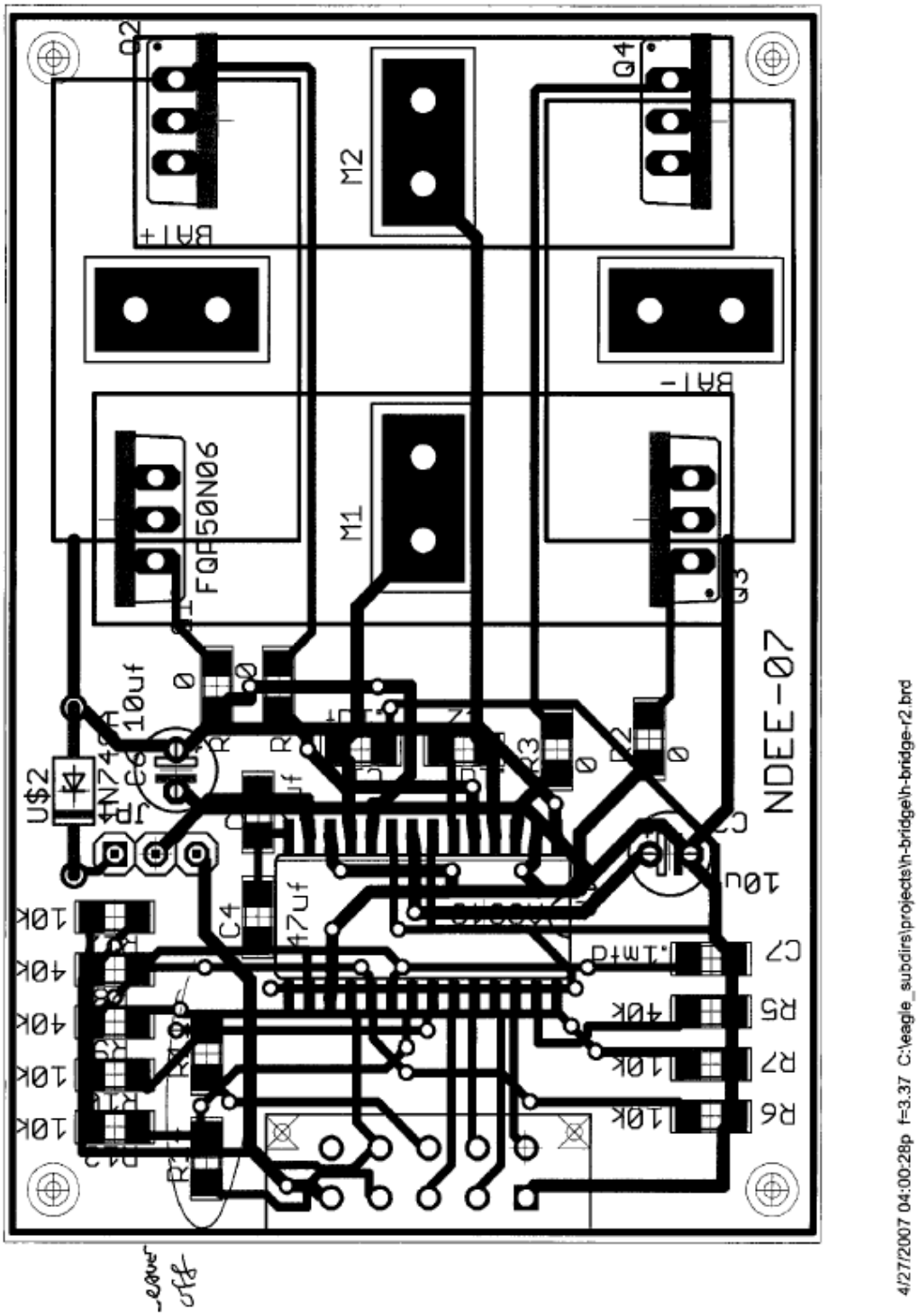
H-Bridge Board Schematic



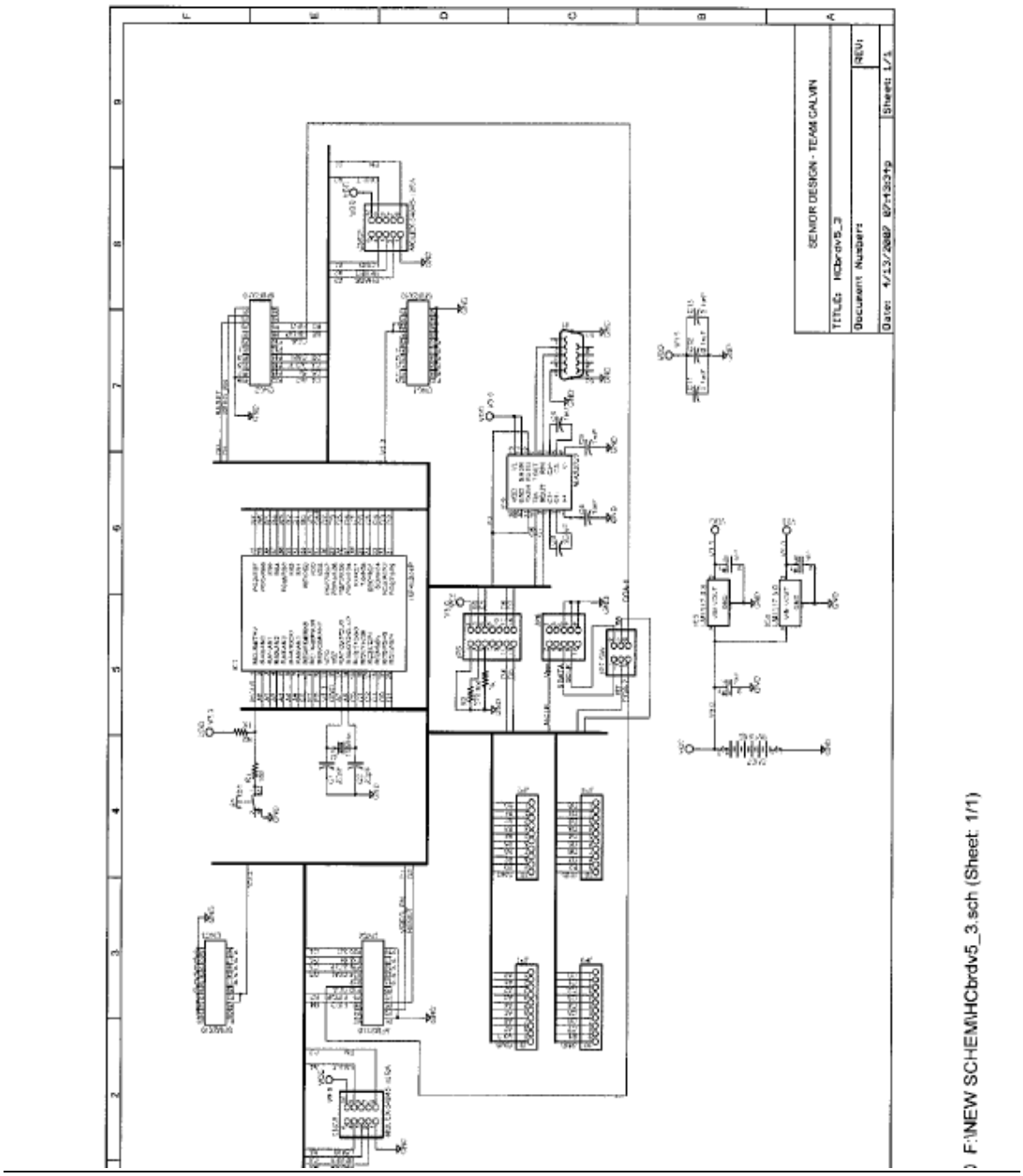
H-Bridge Schematic



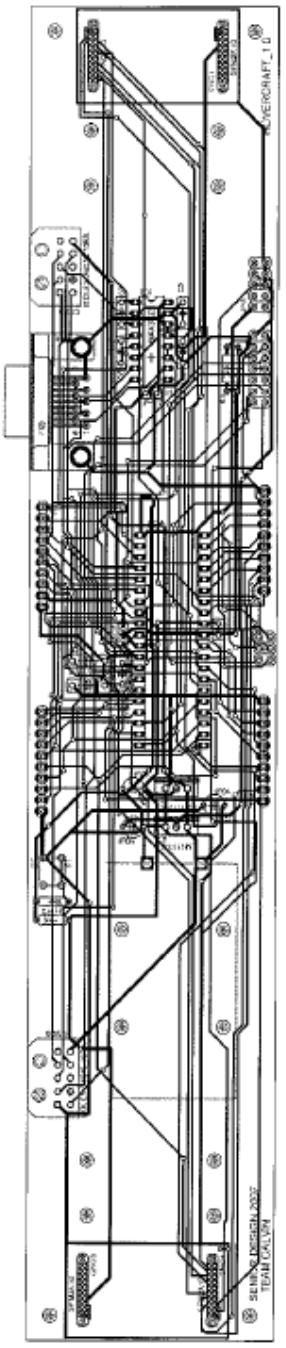
H-Bridge Board Schematic



Microcontroller Board Schematic

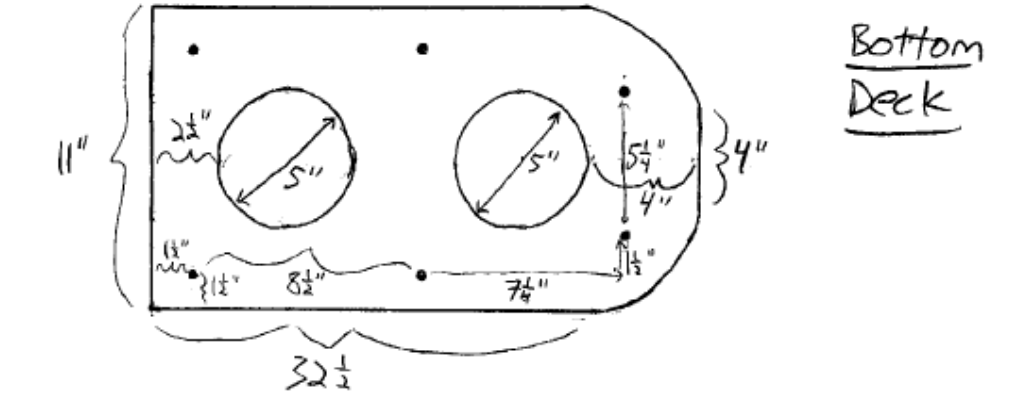


Microcontroller Board Design

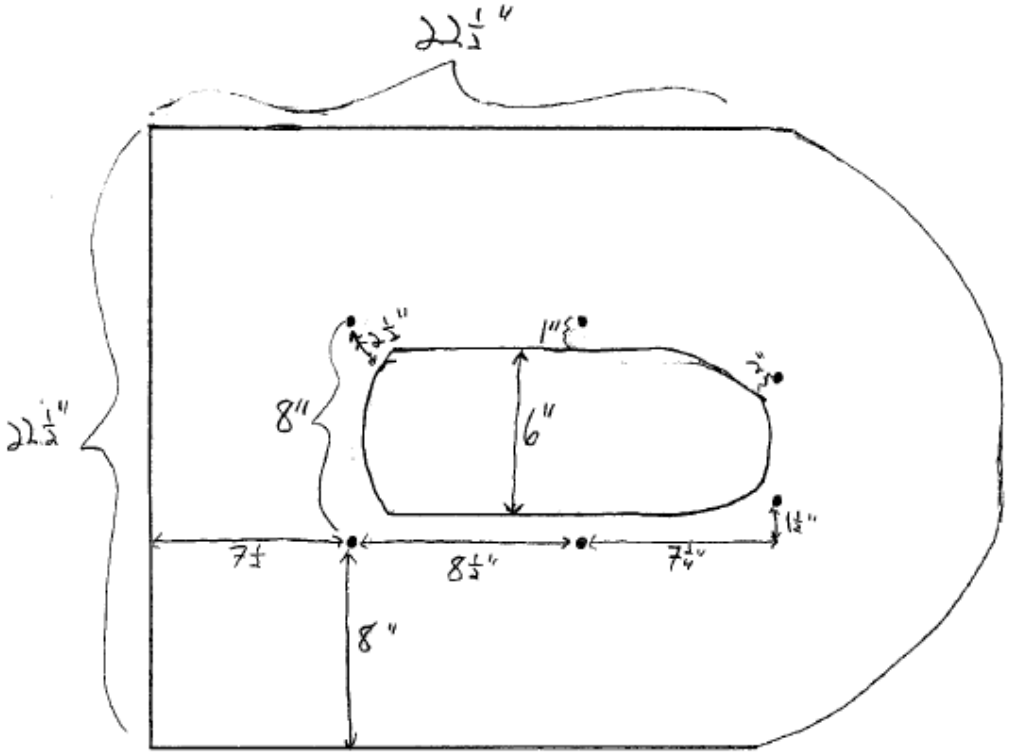


' 10:40:14a f=0.80 F:\NEW SCHEM\HCbrdv5_3.brd

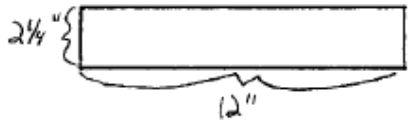
Hovercraft Body Design



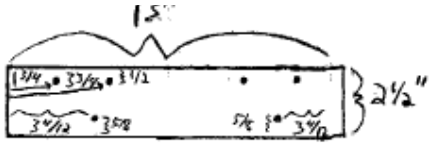
Bottom Deck



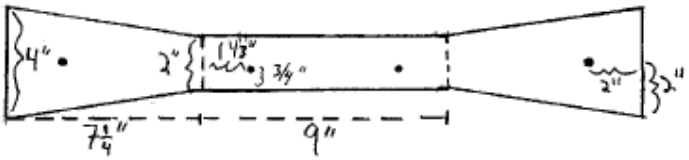
Skirt



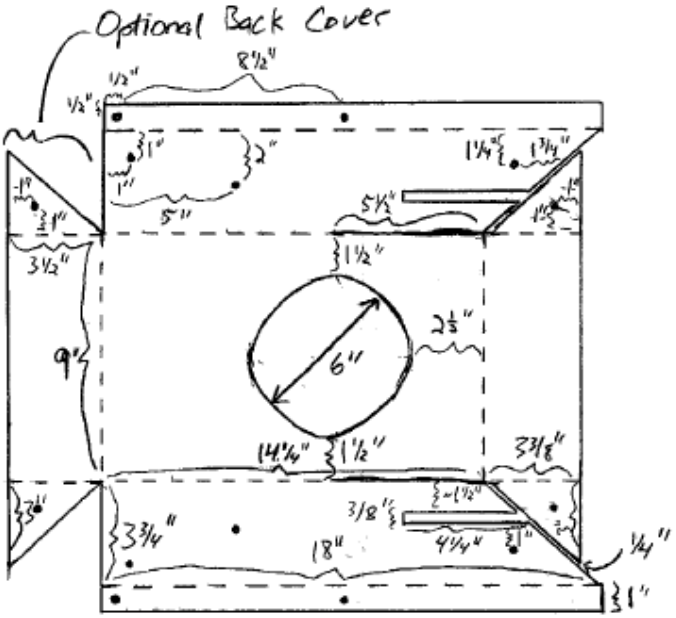
Microcontroller Mount



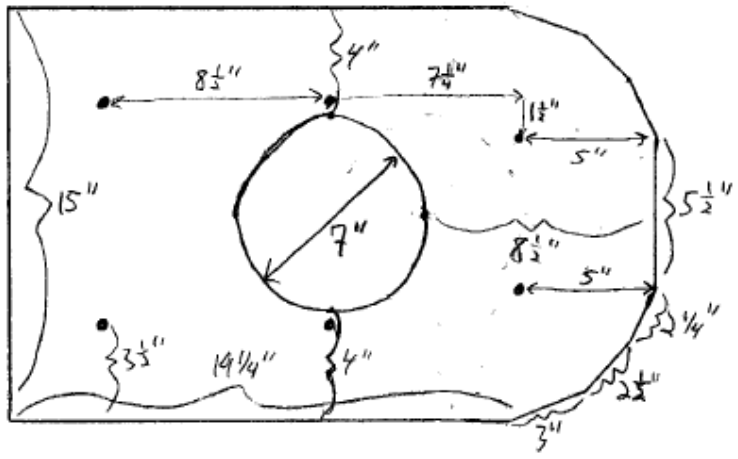
Motor Platform



Spoiler



Chassis



Top Deck

(3) Software

Hovercraft.c

H:\Senior Design\Second semester\Code\final\hovercraft.c 1

```
#include    <system.h>

#include "EESD.h"
#include "cc2420.h"
#include "Hbridge.h"

#pragma DATA _CONFIG1H, _OSC_HS_1H //10 mhz
#pragma DATA _CONFIG2H, _WDT_OFF_2H
#pragma DATA _CONFIG4L, _LVP_OFF_4L
#pragma DATA _CONFIG3H, _MCLRE_ON_3H


#pragma CLOCK_FREQ 10000000

extern bit l_receive; //semaphore for the left chipcon receiving a packet
extern bit r_receive; //semaphore for the right chipcon receiving a packet
extern short direction;

typedef struct payload
    {
        short id;
        short type;
        short timestamp;
        short rssi;
    } Payload;

Payload SetPayloadToZero(Payload);

void main(void)
{
    // Initializes SPI and serial port
    SPI init();
    serial init();
    LCD init();
    CC2420 init();
    delay ms(200);
    intrpt init();
    hbridge_init();

    // Defines the speeds used (scale 0-9) and the difference in dbm that
    // indicates that the beacon is in the center
    #define dbm diff            3
    #define turn speed          6
    #define straight_speed      6

    short counter = 0;
    short temp;
    char* packet;
    Payload l_payload, r_payload;
    bool repel_rcvd = false;
    putstring("\r\nTest program");

    // clears the semaphores
    l_receive = 0;
    r_receive = 0;


    // The code block below provides a current rush to the motors
    // that is necessary to get the motors running in the very beginning
    forward(6);
    delay_ms(250);
    delay_ms(250);
    //--------------------------
    while(1)
    {
```

```
        if (l_receive == 1) // Packet received, left semanphore
        {

            packet = receive_pkt_l();    // Receives packet
            l_receive = 0;               //Clears the semaphore

            l_payload.id = packet[0];
            l_payload.type = packet[1];
            l_payload.timestamp = packet[2];
            l_payload.rssi = packet[3];
        }

        if (r_receive == 1) // Packet received, right semanphore
        {
            //brake();
            packet = receive_pkt_r();    // Receives packet
            r_receive = 0;               //Clears the semaphore

            r_payload.id = packet[0];
            r_payload.type = packet[1];
            r_payload.timestamp = packet[2];
            r_payload.rssi = packet[3];

        }

        /*-----------------------------------------------------
        ---------- The negative sign in front of the ---------
        ---------- rssi values was dropped for ---------------
        ---------- simplicity's sake. Smaller rssi -----------
        ---------- values are indeed stronger signals. -------
        -----------------------------------------------------*/


        if (r_payload.timestamp == l_payload.timestamp) // Check if timestamps are the
    same
        {
            if (r_payload.type == 0xBB && l_payload.type == 0xBB)    // Check if repel
            {
                // Check for repel threshold value
                if (r_payload.rssi <= REPEL_THRESH || l_payload.rssi <= REPEL_THRESH)
                {
                repel_rcvd = true;
                counter = counter + 1;
                    // If center repel
                    if((l_payload.rssi - r_payload.rssi) >= -dbm_diff && (l_payload.rssi -
     r_payload.rssi) <= dbm_diff)
                    {
                        putstring("\r\n      Center Repel");
                        putdec(l_payload.rssi);
                        putstring(" vs ");
                        putdec(r_payload.rssi);

                        if(direction!=4) reverse(straight_speed);

                        LCD_clear();
                        LCD_printf("   --->Repel<---");
                    }

                    // If right repel
                    else if(l_payload.rssi < r_payload.rssi)
                    {
                        putstring("\r\nLeft Repel: ");
                        putdec(l_payload.rssi);
                        putstring(" vs ");
                        putdec(r_payload.rssi);
```

```
                        if(direction!=3) turn_left(turn_speed);

                        LCD clear();
                        LCD_printf("           Repel--->");


                    }
                    // If left repel
                    else if(l_payload.rssi > r_payload.rssi)
                    {
                        putstring("\r\n               Right Repel");
                        putdec(l_payload.rssi);
                        putstring(" vs ");
                        putdec(r_payload.rssi);

                        if(direction!=2) turn_right(turn_speed);

                        LCD clear();
                        LCD_printf("<---Repel");
                    }

                    if (counter >= COUNTER_THRESH)
                    {
                    counter = 0;
                    //move to the next waypoint beacon
                    putstring("\r\nNext Beacon");

                    LCD setpos(5,0);
                    LCD_printf("Beacon: ");
                    }
                // Set the Payload variables to zero
                l_payload = SetPayloadToZero(l_payload);
                r_payload = SetPayloadToZero(r_payload);
                }

                // If not BB type, then not repel signal
                else
                {
                repel_rcvd = false;
                }

            }
            // If attract and not repel signal received
            else if ((r_payload.type == 0xAA && l_payload.type == 0xAA) && repel_rcvd ==
false)
            {
                // If center attract
                if ((l_payload.rssi - r_payload.rssi) >= -dbm_diff && (l_payload.rssi -
r_payload.rssi) <= dbm_diff)
                {
                    putstring("\r\n      Center Attract");
                    putdec(l_payload.rssi);
                    putstring(" vs ");
                    putdec(r_payload.rssi);


                    if(direction!=1) forward(straight_speed);

                    LCD_clear();
                    LCD_printf("  --->Attract<---");
                }

                // If right attract
                else if(l_payload.rssi < r_payload.rssi)
                {
                    putstring("\r\nLeft Attract");
                    putdec(l_payload.rssi);
```

```
                    putstring(" vs ");
                    putdec(r_payload.rssi);

                    if(direction!=2) turn_right(turn_speed);

                    LCD_clear();
                    LCD_printf("         Attract--->");
                }

                // If left attract
                else if (l_payload.rssi > r_payload.rssi)
                {
                    putstring("\r\n               Right Attract");
                    putdec(l_payload.rssi);
                    putstring(" vs ");
                    putdec(r_payload.rssi);

                    if(direction!=3) turn_left(turn_speed);

                    LCD clear();
                    LCD_printf("<---Attract");


                }
            // Clear Payload variables
            l payload = SetPayloadToZero(l payload);
            r_payload = SetPayloadToZero(r_payload);
            }
        }
        else        // If the timestamps are not the same an error has occured
        {
            putstring("Error!");

            l payload = SetPayloadToZero(l payload); // Resets payload variables
            r_payload = SetPayloadToZero(r_payload);

            l receive = 0;              // Clears semaphores
            r receive = 0;

            CSN r = 0;                  // Sets receive state of the CC2420 and flushes
            temp =SSPsend(SXOSCON);     // the RXFIFO
            temp =SSPsend(STXCAL);
            temp =SSPsend(SRXON);
            temp =SSPsend(SFLUSHRX);
            temp =SSPsend(SFLUSHRX);
            CSN_r =1;

            CSN l = 0;
            temp =SSPsend(SXOSCON);
            temp =SSPsend(STXCAL);
            temp =SSPsend(SRXON);
            temp =SSPsend(SFLUSHRX);
            temp =SSPsend(SFLUSHRX);
            CSN_l = 1;


        }


    }
}


Payload SetPayloadToZero(Payload package)
{
    package.id = 0;
    package.type = 0;
```



```
    package.timestamp = 0;
    package.rssi = 0;

    return package;
}
```

HBridgelib.c

H:\Senior Design\Second semester\Code\final\Hbridgelib.c 1

```
// Code updated 5/1/07

/***************************
 * This program contains: *
 * H-Bridge control fncs  *
 ***************************/
#include    <system.h>
#include    "EESD.h"
#include    "Hbridge.h"
short direction = 0; // 0 = brake, 1 = fwd, 2= rgt, 3 =lft, 4 =rvrs


// Sets H-bridge intial settings

void hbridge_init(void)
{
    unsigned short num;

    //**********************
    //**start of pwm code**
    //**********************
    //1. setting the pwm period by writing to the PR2 register
    pr2 = 156;                  //pwm frequency of 995 Hz
    //pr2 = 60;
    //writing to CCP1CON
    //ccp1con.5 = 0;
    //ccp1con.4 = 1;
    //5. configuring the CCPx module for PWM operation (for PWM mode: 1100)
    ccp1con.3 = 1;
    ccp1con.2 = 1;
    ccp1con.1 = 0;
    ccp1con.0 = 0;

    //writing to CCP2CON
    //ccp2con.5 = 0;
    //ccp2con.4 = 0;
    //5. configuring the CCPx module for PWM operation (for PWM mode: 1100 -- CCP mode,
    not EECP)
    ccp2con.3 = 1;
    ccp2con.2 = 1;
    ccp2con.1 = 0;
    ccp2con.0 = 0;

    //3. making the CCPx pin an output by clearing the appropriate tris bit
    trisc.2 = 0;    //the output of CCP1 is c2
    trisc.1 = 0;    //the output of CCP2 is c1

    //4. setting the TMR2 prescale value, then enable Timer2 by writing to T2CON
    //TMR2 = 16 (00 = 1; 01 = 4; 1x = 16)
    t2con.1 = 1;            //prescale value
    t2con.0 = 0;            //prescale value

    t2con.2 = 1;            //turning on timer2


    //directional variables initialization


    num = 0;
    ccpr1l = num >> 2;  //is this necessary? for now yes.
    ccpr2l = num >> 2;


    //setting the pins to be output values to PHASE and RESET
    //for Right Motor
    trisb.5 = 0;        //for PHASE
    trisc.0 = 0;        //for RESET
```

```
    //for Left Motor
    trisa.3 = 0;        //for PHASE
    trisa.2 = 0;        //for RESET

    //setting the pins to be input values
    trisa.4 = 0;        //for FAULT
    trisa.0 = 0;        //for FAULT


    lata.5 = 0;
    trisa.1 = 0;
    lata.1 = 0;

    /*if porta.0 == 1{
        lata.2 = 0;
        delay_ms(250);
        lata.2 = 1;}
    else if porta.4 == 1{
        latc.0 = 0;
        delay_ms(250);
        latc.0 = 1;}
    else {}
    */
    latc.0 = 0;     //initialization for RESET
    lata.2 = 0;     //initialization for RESET

    //brake();          //initialization so reset is set 1(???)
    return;
}

/*****************************
 * The following functions*
 * deal with directional  *
 * control of the HC      *
 *****************************/


void frwd_mR (void)
{
    latb.5 = 1;     //initialization for PHASE
    lata.4 = 0;     //initialization for FAULT
    latc.0 = 1;     //initialization for RESET
}

void frwd_mL (void)
{
    lata.3 = 0;     //initialization for PHASE
    lata.0 = 1;     //initialization for FAULT
    lata.2 = 1;     //initialization for RESET
}

void rvrs_mR (void)
{
    latb.5 = 0;     //initialization for PHASE
    lata.4 = 0;     //initialization for FAULT
    latc.0 = 1;     //initialization for RESET
}

void rvrs_mL (void)
{
    lata.3 = 0;     //initialization for PHASE
    lata.0 = 0;     //initialization for FAULT
    lata.2 = 1;     //initialization for RESET
}

void brk_mR (void)
{
```

```
    latb.5 = 1;      //initialization for PHASE
    lata.4 = 0;      //initialization for FAULT
    latc.0 = 1;      //initialization for RESET
}

void brk_mL (void)
{
    lata.3 = 1;      //initialization for PHASE
    lata.0 = 0;      //initialization for FAULT
    lata.2 = 1;      //initialization for RESET
}




// Input is a number, output is a corresponding
// PWM setting

unsigned short pwmconvert(unsigned short in1)
{
    unsigned short num;
    /*
    if (in1 == 0)
        num = 24;    //10% dtcyc
    else if (in1 == 1)
        num = 48;    //20% dtcyc
    else if (in1 == 2)
        num = 73;    //30% dtcyc
    else if (in1 == 3)
        num = 97;    //40% dtcyc
    else if (in1 == 4)
        num = 122;   //50% dtcyc
    else if (in1 == 5)
        num = 146;   //60% dtcyc
    else if (in1 == 6)
        num = 170;   //70% dtcyc
    else if (in1 == 7)
        num = 195;   //80% dtcyc
    else if (in1 == 8)
        num = 219;   //90% dtcyc
    else if (in1 == 9)
        num = 244;   //100% dtcyc
    return num;
    */
    if (in1 == 0)
        num = 62;    //10% dtcyc
    else if (in1 == 1)
        num = 125;   //20% dtcyc
    else if (in1 == 2)
        num = 188;   //30% dtcyc
    else if (in1 == 3)
        num = 251;   //40% dtcyc
    else if (in1 == 4)
        num = 314;   //50% dtcyc
    else if (in1 == 5)
        num = 376;   //60% dtcyc
    else if (in1 == 6)
        num = 439;   //70% dtcyc
    else if (in1 == 7)
        num = 502;   //80% dtcyc
    else if (in1 == 8)
        num = 565;   //90% dtcyc
    else if (in1 == 9)
        num = 628;   //100% dtcyc
    return num;
```

```
}

// The following functions take as input the speed
// as defined in pwmconvert()


void forward (unsigned short speed)
{

    unsigned short num;

    num = pwmconvert(speed);

    ccpr1l = num >> 2;
    ccpr2l = num >> 2;


    frwd_mL();
    frwd_mR();
    direction = 1;

}

void reverse (unsigned short speed)
{

    unsigned short num;

    num = pwmconvert(speed);

    ccpr1l = num >> 2;
    ccpr2l = num >> 2;


    rvrs_mL();
    rvrs_mR();
    direction = 4;

}

void brake (void)
{
    brk_mL();
    brk_mR();
    //ccpr1l = 0 >> 2;
    //ccpr2l = 0 >> 2;
    direction = 0;
}

void turn_right (unsigned short speed)
{

    unsigned short num;

    num = pwmconvert(speed);

    ccpr1l = num >> 2;
    ccpr2l = num >> 2;

    frwd_mL();
    rvrs_mR();
    direction = 2;

}

void turn_left (unsigned short speed)
{
```

```
    unsigned short num;

    num = pwmconvert(speed);

    ccpr1l = num >> 2;
    ccpr2l = num >> 2;

    frwd_mR();
    rvrs_mL();

    direction = 3;
}
```

cc240.h

```
H:\Senior Design\Second semester\Code\final\cc2420.h                    1
#include    <system.h>

// PIN definitions
#define FIFO_l          portb.1   // FIFO pin
#define FIFOP_l         portb.0   // FIFOP pin
#define CCA_l           portb.6   // Clear Channel Assessment pin
#define CSN_l           latd.2    // Chip select

#define FIFO_r          portb.4   // Right FIFO pin
#define FIFOP_r         portb.2   // FIFOP pin
#define CCA_r           portb.7   // Clear Channel Assessment pin
#define CSN_r           latd.3    // Chip select

// Timer interrupts
#define CSMATMR         intcon.5 // CSMA algorithm timer

// Command Strobes
#define SNOP            0x00 // No Operation (has no other effect than reading out status-
    bits)
#define SXOSCON         0x01 // Turn on the crystal oscillator (set XOSC16M_PD = 0 and
    BIAS_PD = 0)
#define STXCAL          0x02 // Enable and calibrate frequency synthesizer for TX;Go from
    RX/TX to a wait state where only the synthesizer is running.
#define SRXON           0x03 // Enable RX
#define STXON           0x04 // Enable TX after calibration (if not already performed).
    Start TX in-line encryption if SPI SEC MODE != 0
#define STXONCCA        0x05 // If CCA indicates a clear channel:Enable calibration, then
    TX.Start in-line encryption if SPI SEC MODE != 0 else do nothing
#define SRFOFF          0x06 // Disable RX/TX and frequency synthesizer
#define SXOSCOFF        0x07 // Turn off the crystal oscillator and RF
#define SFLUSHRX        0x08 // Flush the RX FIFO buffer and reset the demodulator. Always
    read at least one byte from the RXFIFO before issuing the SFLUSHRX command strobe
#define SFLUSHTX        0x09 // Flush the TX FIFO buffer
#define SACK            0x0A // Send acknowledge frame, with pending field cleared.
#define SACKPEND        0x0B // Send acknowledge frame, with pending field set.
#define SRXDEC          0x0C //  S Start RXFIFO in-line decryption / authentication (as
    set by SPI SEC MODE)
#define STXENC          0x0D // Start TXFIFO in-line encryption / authentication (as set
    by SPI SEC MODE), without starting TX.
#define SAES            0x0E // AES Stand alone encryption strobe. SPI SEC MODE is not
    required to be 0, but the encryption module must be idle. If not, the strobe is
    ignored.

// Register Adresses
#define MAIN_REG            0x10 // Main Control Register
#define MDMCTRL0_REG        0x11 // Modem Control Register 0
#define MDMCTRL1_REG        0x12 // Modem Control Register 1
#define RSSI_REG            0x13 // RSSI and CCA Status and Control register
#define SYNCWORD_REG        0x14 // Synchronisation word control register
#define TXCTRL_REG          0x15 // Transmit Control Register
#define RXCTRL0_REG         0x16 // Receive Control Register 0
#define RXCTRL1_REG         0x17 // Receive Control Register 1
#define FSCTRL_REG          0x18 // Frequency Synthesizer Control and Status Register
#define SECCTRL0_REG        0x19 // Security Control Register 0
#define SECCTRL1_REG        0x1A // Security Control Register 1
#define BATTMON_REG         0x1B // Battery Monitor Control and Status Register
#define IOCFG0_REG          0x1C // Input / Output Control Register 0
#define IOCFG1_REG          0x1D // Input / Output Control Register 1
#define MANFIDL_REG         0x1E // Manufacturer ID, Low 16 bits
#define MANFIDH_REG         0x1F // Manufacturer ID, High 16 bits
#define FSMTC_REG           0x20 // Finite State Machine Time Constants
#define MANAND_REG          0x21 // Manual signal AND override register
#define MANOR_REG           0x22 // Manual signal OR override register
#define AGCCTRL_REG         0x23 // AGC Control Register
#define AGCTST0_REG         0x24 // AGC Test Register 0
#define AGCTST1_REG         0x25 // AGC Test Register 1
#define AGCTST2_REG         0x26 // AGC Test Register 2
```

```
#define FSTST0_REG          0x27 // Frequency Synthesizer Test Register 0
#define FSTST1_REG          0x28 // Frequency Synthesizer Test Register 1
#define FSTST2_REG          0x29 // Frequency Synthesizer Test Register 2
#define FSTST3_REG          0x2A // Frequency Synthesizer Test Register 3
#define RXBPFTST_REG        0x2B // Receiver Bandpass Filter Test Register
#define FSMSTATE_REG        0x2C // Finite State Machine State Status Register
#define ADCTST_REG          0x2D // ADC Test Register
#define DACTST_REG          0x2E // DAC Test Register
#define TOPTST_REG          0x2F // Top Level Test Register
#define RESERVED_REG        0x30 // Reserved for future use control / status register
#define TXFIFO_REG          0x3E // Transmit FIFO Byte Register
#define RXFIFO_REG          0x3F // Receiver FIFO Byte Register

// IEEE 802.15.4 Definitions
#define BEinit          3
#define NBinit          0
#define BEMax           5
#define NBMax           4

// Algorithm definitions
#define REPEL_THRESH    55
#define COUNTER_THRESH   5

// Algorithm variables
//bit l_receive; //defined in the header instead of the individual C files...
//bit r_receive; //do the values from cc2420lib need to be passed any special way to the
    values in hovercraft.c?

void interrupt(void);
void SPI_init(void);
void CC2420_init(void);
void intrpt_init(void);
char SSPsend(char address);

unsigned short read_reg_l(unsigned short address);
unsigned short read_reg_r(unsigned short address);

void write_reg_l(unsigned short address, unsigned short data);
void write_reg_r(unsigned short address, unsigned short data);

char* read_rxfifo_l(void);
char* read_rxfifo_r(void);

void write_txfifo_l(char* payload);
void write_txfifo_r(char* payload);

char* read_txfifo_l(void);
char* read_txfifo_r(void);

char* receive_pkt_l(void);
char* receive_pkt_r(void);

void send_packet_l(char* packet);
void send_packet_r(char* packet);

void CSMA_CA_l(void);
void CSMA_CA_r(void);

void timer_320us(int delay);
```

cc240lib.c

H:\Senior Design\Second semester\Code\final\cc2420lib.c 1

```
/*************************************************************
 *      This file contains the required routines             *
 *      required to communicate with the CC2420              *
 *************************************************************/

#include    <system.h>
#include    "cc2420.h"
#include    <rand.h>
#include    "EESD.h"


int BE, NB, counter;
bit failure, send_side;
volatile bit  CSMA_int@INTCON.2;
volatile bit  FIFOP_left@INTCON.1;
volatile bit  FIFOP_right@INTCON3.1;


volatile bit mssp@PIR1.3;   // SSPIF: MSSP Interrupt Flag --> trans/recp complete or
    waiting
bit l_receive = 0; //semaphore for the left chipcon receiving a packet
bit r_receive = 0; //semaphore for the right chipcon receiving a packet

void interrupt(void)
{

    if (FIFOP_left == 1)
    {
        FIFOP_left = 0;
        l_receive = 1;
    }

    if (FIFOP_right == 1)
    {
        FIFOP_right = 0;
        r_receive = 1;

    }
    /*
    if (CSMA_int == 1)
    {
        counter++;
        CSMA_int = 0;             //clear interrupt flag
        CSMATMR = 0; // Timer0 interrupt DISABLED

        if(send_side == 0)
        {
            CSMA_CA_l();
        }
        if(send_side == 1)
        {
            CSMA_CA_r();
        }
    }*/

    return;
}

void SPI_init(void)
{

    adcon1 = 0x0f;  // Set all I/O pins to digital
    trisd.0 = 0;    // RESET the chipcon
    latd.0 = 0;
    delay_ms(250);
    latd.0 = 1;
```

```
// INTCON: Interrupt Control Register
    intcon.7 = 0;       // GIE: Global Interrupt Enable
    intcon.6 = 0;       // PEIE: Peripheral Interrupt Enable

// PIE1.3 Peripheral Interrupt Enable Register
    pie1.3 = 0;         // SSPIE: MSSP Interrupt Enable

// IPR1.3: Peripheral Interrupt Priority Register
    ipr1.3 = 0;         // SSPIP: MSSP Interrupt Priority

// SSPCON1 control register for SPI mode
    sspcon1.7 = 0;      // WCOL: Write Collision Detect
    sspcon1.6 = 0;      // SSPOV: Receive Overflow Indicator --> Slave Mode Only
    sspcon1.5 = 1;      // SSPEN: Sync. Serial Port Enable
    sspcon1.4 = 0;      // CKP: Clock Polarity Select
    sspcon1.3 = 0;      // SSPM (3:0) Mode Select --> 0010 = FOSC/64
    sspcon1.2 = 0;
    sspcon1.1 = 1;
    sspcon1.0 = 0;

// SSPSTAT status register for SPI mode
    sspstat.7 = 1;      // SMP: Sample bit
    sspstat.6 = 1;      // CKE: SPI Clock Select bit


    trisc.5 = 0; // SDO, serial data out
    trisc.3 = 0; // SCK, serial clock
    trisc.4 = 1; // SDI, serial data in

    trisd.1 = 0; // Voltage regulator
    latd.1 = 1;

    trisd.2 = 0; // CSn_l (chip select)
    latd.2 = 1;
    trisd.3 = 0; // CSn_r (chip select)
    latd.3 = 1;


// FIFOs and the rest as input pins

    trisb.1 = 1; // FIFO_l
    trisb.0 = 1; // FIFOP_l
    trisb.6 = 1; // CCA_l

    trisb.4 = 1; // FIFO_r
    trisb.2 = 1; // FIFOP_r
    trisb.7 = 1; // CCA_r

    //debug
    trisa.5 = 0;

    return;
}

void CC2420_init(void)
{
    unsigned short ans;


    putstring("\r\nRight Status: "); // Configures right CC2420
    CSN_r = 0;
    ans = SSPsend(SNOP);
    ans = SSPsend(SXOSCON);
        puthex(ans);
    ans = SSPsend(STXCAL);
        puthex(ans);
```

```
    ans = SSPsend(SRXON);
        puthex(ans);
    ans = SSPsend(SNOP);     // Expected state: 0x46
        puthex(ans);
    CSN_r = 1;

    ans = read_reg_r(RXFIFO_REG);
    CSN_r = 0;
    ans = SSPsend(SFLUSHRX);
    ans = SSPsend(SFLUSHRX);
    CSN_r = 1;


    putstring("\r\nLeft Status: "); // Configures left CC2420
    CSN_l = 0;
    ans = SSPsend(SNOP);
    ans = SSPsend(SXOSCON);
        puthex(ans);
    ans = SSPsend(STXCAL);
        puthex(ans);
    ans = SSPsend(SRXON);
        puthex(ans);
    ans = SSPsend(SNOP);     // Expected state: 0x46
        puthex(ans);
    CSN_l = 1;

    ans = read_reg_l(RXFIFO_REG); // Reads RXFIFO before flushing it (refer to CC2420
    datasheet)
    CSN_l = 0;
    ans = SSPsend(SFLUSHRX);
    ans = SSPsend(SFLUSHRX);
    CSN_l = 1;


    write_reg_l(FSCTRL_REG, 0x6565);     // Set to Channel 11
    write_reg_l(IOCFG0_REG, 25);         // Set FIFOP threshold length
    write_reg_r(FSCTRL_REG, 0x6565);
    write_reg_r(IOCFG0_REG, 25);
}

void intrpt_init(void)
{
    // INTCON bits
    intcon.7 = 1; // Global Interrupt enabled
    intcon.6 = 1; // Peripheral Interrupt enabled
    intcon.5 = 0; // Timer0 interrupt DISABLED
    intcon.4 = 1; // External Interrupt 0 bit enabled

    // INTCON2 bit
    intcon2.6 = 1; // External Int 0 on rising edge(FIFOP_l)
    intcon2.4 = 1; // External Int 2 on rising edge(FIFOP_r)

    // INTCON3 bit
    intcon3.4 = 1; // External Interrupt 2 bit enabled


    // TIMER0 used as CSMA timer
    t0con.7 = 1;
    t0con.6 = 1; // Timer0 set to be 8-bit
    t0con.5 = 0;
    t0con.3 = 0;
    t0con.2 = 1; // prescale factor 1:256
    t0con.1 = 1;
    t0con.0 = 1;

    return;
}
```

```
/**********************************
 *  SPI read/write routine        *
 **********************************/
char SSPsend(char address)
{
    sspbuf = address;   // load data
    while(mssp == 0){}  // wait until transmition is complete
    mssp = 0;           // reset flag

    return sspbuf;
}

/**********************************
 *  Reads an 8-bit register       *
 **********************************/
unsigned short read_reg_l(unsigned short address)
{
    unsigned short data;
    char temp;

    address = address | 01000000b; // 6-bit address and read command
    CSN_l = 0;          // chip select
    temp = SSPsend(address);
    temp = SSPsend(0);
    data = temp;
    data = data << 8;   // Shift data
    temp = SSPsend(0);
    data = data | temp;
    CSN_l = 1;      //chip select high

    return data;
}
unsigned short read_reg_r(unsigned short address)
{
    unsigned short data;
    char temp;

    address = address | 01000000b; // 6-bit address and read command
    CSN_r = 0;          // chip select
    temp = SSPsend(address);
    temp = SSPsend(0);
    data = temp;
    data = data << 8;   // Shift data
    temp = SSPsend(0);
    data = data | temp;
    CSN_r = 1;      //chip select high

    return data;
}


/**********************************
 *  Writes to an 8-bit register   *
 **********************************/
void write_reg_l(unsigned short address, unsigned short data)
{
    char temp;
    unsigned short data1, data2;

    data1 = ((data >> 8) & 0xFF);
    data2 = (data & 0xFF);  // Separates 16-bit data to 2 8-bit
    CSN_l = 0;              // chip select
    temp = SSPsend(address);
    temp = SSPsend(data1);
    temp = SSPsend(data2);
    CSN_l = 1;          //chip select high
```

```
    return;
}
void write_reg_r(unsigned short address, unsigned short data)
{
    char temp;
    unsigned short data1, data2;

    data1 = ((data >> 8) & 0xFF);
    data2 = (data & 0xFF);  // Separates 16-bit data to 2 8-bit
    CSN_r = 0;              // chip select
    temp = SSPsend(address);
    temp = SSPsend(data1);
    temp = SSPsend(data2);
    CSN_r = 1;          //chip select high

    return;
}

/**********************************
 *  Reads message from RXFIFO     *
 **********************************/
char* read_rxfifo_1(void)
{

    unsigned short n, null, length;
    unsigned short address, rssi, fcs;
    unsigned long sfd;
    char msg[19];


    address = 0x3f | 01000000b;


    CSN_1 = 0;  // chip select
    null = SSPsend(address);
    length = SSPsend(0);
    msg[0] = length;
    n=1;

    if(length == 17)
    {
        for( n=1; n<=17; n++)
        {
            msg[n] = SSPsend(0);
        }
        CSN_1 = 1; //chip select high

    }
    else    // If the packet is not of the desired length then ignore
    {
        CSN_1 = 1; //chip select high

        for( n=1; n<=17; n++)
        {
            msg[n] = 0x0E;
        }
    }

    return msg;
}
char* read_rxfifo_r(void)
{

    unsigned short n, null, length;
    unsigned short address, temp;
    unsigned long sfd;
```

```
    char msg[19];


    address = 0x3f | 01000000b;


    CSN_r = 0;  // chip select
    null = SSPsend(address);
    length = SSPsend(0);
    msg[0] = length;
    n=1;

    if(length == 17)
    {
        for( n=1; n<=17; n++)
        {
            msg[n] = SSPsend(0);
        }
        CSN_r = 1; //chip select high

    }
    else
    {
        CSN_r = 1; //chip select high

        for( n=1; n<=17; n++)
        {
            msg[n] = 0xEE;
        }
    }



    return msg;
}

/**********************************
 *  Receives packet and beacon,   *
 *  type, RSSI and timestamp      *
 *  information                   *
 **********************************/
char* receive_pkt_r(void)
{

    unsigned short  payload, length, rssi;
    unsigned short rssi_val, beacon_type, beacon_id, timestamp;
    unsigned long sfd;
    char packet[4];
    char* msg;

    msg = read_rxfifo_r();
    length = msg[0];
    payload = length - 10;
    rssi = length-1;

    beacon_type = msg[payload+5];
    beacon_id = msg[payload+3];
    timestamp = msg[payload+7];

        // Get signed 2's comp RSSI value (assumption: packet is 15 bytes long)
    rssi_val = msg[rssi];

    if (rssi_val >= 128)
    {
        rssi_val = 256 - rssi_val;
        rssi_val = rssi_val + 45;  // dB offset: -45
    }
```

```
    else
    {
        // dB offset: -45
        rssi_val = 45 - rssi_val;
    }


    if (length != 17)
    {
        beacon_type = 0xEE;
        beacon_id = 0x0;
        timestamp = 0x0;
        rssi_val = 0;
    }

    packet[0] = beacon_id;
    packet[1] = beacon_type;
    packet[2] = timestamp;
    packet[3] = rssi_val;


    return packet;
}

char* receive_pkt_1(void)
{

    unsigned short  payload, length, rssi;
    unsigned short rssi_val, beacon_type, beacon_id, timestamp;
    unsigned long sfd;
    char packet[4];
    char* msg;

    msg = read_rxfifo_1();
    length = msg[0];
    payload = length - 10;
    rssi = length-1;

    beacon_id = msg[payload+3];
    beacon_type = msg[payload+5];
    timestamp = msg[payload+7];

        // Get signed 2's comp RSSI value (assumption: packet is 15 bytes long)
    rssi_val = msg[rssi];

    if (rssi_val >= 128)
    {
        rssi_val = 256 - rssi_val;
        rssi_val = rssi_val + 45;  // dB offset: -45
    }
    else
    {
        // dB offset: -45
        rssi_val = 45 - rssi_val;
    }

    if (length != 17)
    {
        beacon_type = 0xEE;
        beacon_id = 0;
        timestamp = 0;
        rssi_val = 0;
    }

    packet[0] = beacon_id;
    packet[1] = beacon_type;
    packet[2] = timestamp;
```

```
    packet[3] = rssi_val;


    return packet;
}

/**********************************
 *  Writes message from TXFIFO    *
 **********************************/

void write_txfifo_l(char* payload)
{
    unsigned short length, txfifo_ad, null;
    int n;

    txfifo_ad = 0x3E;
    length = payload[0];

    CSN_l = 0;
    null = SSPsend(txfifo_ad);
    n=0;
    while(n<=length-2)
    //for( n=0; n<length; n++)
    {
        null = SSPsend(payload[n]);
        n++;
    }

    CSN_l = 1;

    return;

}

void write_txfifo_r(char* payload)
{
    unsigned short length, txfifo_ad, null;
    int n;

    txfifo_ad = 0x3E;
    length = payload[0];

    CSN_r = 0;
    null = SSPsend(txfifo_ad);
    n=0;
    while(n<=length-2)
    //for( n=0; n<length; n++)
    {
        null = SSPsend(payload[n]);
        n++;
    }

    CSN_r = 1;

    return;

}

/**********************************
 *  Reads message from TXFIFO     *
 **********************************/
char* read_txfifo_l(void)
{
    unsigned short length, txfifo_ad, null;
    int n;
    char msg[19];
```

```
    txfifo_ad = 0x3E | 01000000b;

    CSN_l = 0;
    null = SSPsend(txfifo_ad);
    length = SSPsend(SNOP);
    msg[0] = length;
    n=0;

    for( n=1; n<=17; n++)
        {
        msg[n] = SSPsend(0);
        n++;
        }

    CSN_l = 1;

    return msg;

}

char* read_txfifo_r(void)
{
    unsigned short length, txfifo_ad, null;
    int n;
    char msg[19];

    txfifo_ad = 0x3E | 01000000b;

    CSN_r = 0;
    null = SSPsend(txfifo_ad);
    length = SSPsend(SNOP);
    msg[0] = length;
    n=0;

    for( n=1; n<=17; n++)
        {
        msg[n] = SSPsend(0);
        n++;
        }

    CSN_r = 1;

    return msg;

}


/**********************************
 *  Sends the repel packets       *
 **********************************/
void send_packet_l(char* packet)
{

    failure = 0;
    counter = 0;
    BE = BEinit;
    NB = NBinit;

    send_side = 0; // Sending on the left

    write_txfifo_l(packet);
    CSMA_CA_l();

    return;
}

void send_packet_r(char* packet)
```

```
{

    failure = 0;
    counter = 0;
    BE = BEinit;
    NB = NBinit;

    send_side = 1; // Sending on the right

    write_txfifo_r(packet);
    CSMA_CA_r();

    return;
}

/*********************************
 * Algorithm that assures        *
 * compliance with IEEE 802.15.4 *
 * (CSMA)                        *
 *********************************/
void CSMA_CA_l(void)
{
    unsigned short state;
    bit sent;
    int delay;

    if (counter == 0)
    {
        delay = rand() % ((1 << BE) - 1);
        timer_320us(delay);
    }

    else
    {
        CSN_1 = 0;
        state = SSPsend(STXONCCA);
        state = SSPsend(SNOP);
        CSN_1 = 1;
        sent = (state >> 3);
        putstring("\r\n State: ");
        puthex(state);

        if (!sent)
        {
            delay = rand() % ((1 << BE) - 1);
            NB = NB + 1;

            if (BE == BEMax);
            else BE++;

            if (NB > NBMax)
            {
                failure = 1;
                putstring(" FAILURE\r\n"); //START CHECKING HERE
            }
            else timer_320us(delay);
        }
    }
    return;
}


void CSMA_CA_r(void)
{
    unsigned short state;
    bit sent;
    int delay;
```

```
    if (counter == 0)
    {
        delay = rand() & ((1 << BE) - 1);
        timer_320us(delay);
    }

    else
    {
        CSN_r = 0;
        state = SSPsend(STXONCCA);
        state = SSPsend(SNOP);
        CSN_r = 1;
        sent = (state >> 3);
        putstring("\r\n State: ");
        puthex(state);

        if (!sent)
        {
            delay = rand() & ((1 << BE) - 1);
            NB = NB + 1;

            if (BE == BEMax);
            else BE++;

            if (NB > NBMax)
            {
                failure = 1;
                putstring(" FAILURE\r\n"); //START CHECKING HERE
            }
            else timer_320us(delay);
        }
    }
    return;
}

void timer_320us(int delay)
{
    tmr0l = -3*delay;

    putstring(" TMR0L: ");
    puthex(tmr0l);
    putc(' ');
    putint(delay);

    CSMATMR  = 1; // Timer0 interrupt ENABLED

    return;

}
```

(4) Data Sheets

Microcontroller:
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1335&dDocName=en010304

Chipcon:
http://www.chipcon.com/files/CC2420_Data_Sheet_1_4.pdf

H-bridge driver:
http://www.datasheetcatalog.com/datasheets_pdf/A/3/9/4/A3940.shtml

Initial Tecel H-bridge board:
http://www.tecel.com/d200/

Linear Regulators:
http://www.national.com/pf/LM/LM1117.html