Autonomously Controlled Front Loader

Senior Project Report

by

Steven Koopman and Jerred Peterson

Submitted to:

Dr. Donald Schertz

EE 451/452 Senior Capstone Project

May 13, 2008

Abstract

The goal of this project was to implement a low cost autonomous vehicle to do a specific task, which in this case was to load a truck with material from a bin. This project could be used as an initial point in an entire chain of autonomous vehicles that could control the operations of a gravel yard or construction site. This report details the hardware and software design that was done to modify a vehicle to accomplish the previously mentioned task.

Table of Contents:

Abstract ... 2
Introduction ... 4
System Description ... 4
  Vehicle ... 4
  Sensors ... 5
  Microcontroller ... 6
  System Operation ... 6
System Requirements ... 7
Hardware Overview ... 8
  Drive Motors ... 9
  Arm and Bucket Motors ... 9
  Vehicle Sensors ... 10
  Infrared Transistor ... 10
  Infrared Encoders ... 11
  Compass ... 12
  Ultrasonic ... 12
  3.3 To 5 V Unidirectional Buffer Chip ... 13
  5V regulator ... 13
Software Overview ... 13
  Software Block Diagram ... 14
  Software Function Description ... 15
    Low Level Foreground Functions ... 16
    High Level Foreground Functions ... 21
    Background Operations ... 24
Final Results ... 24
Future Work ... 26
  Large Projects ... 26
  Small Projects ... 27
Index of Appendices ... Appendix Index-1
Appendix A: Detailed hardware circuitry and connections ... A-1
Appendix B: Final project code ... B-1
  testinit.c ... B-1
  testmain.c ... B-2
Appendix C: Vehicle Modification Information ... C-1
Appendix D: Vehicle Operation Information ... D-1

Introduction

This project intends to modify a previously existing vehicle to perform a specific task without human intervention. This will require specialized sensors allow the microcontroller to make intelligent decisions about how to achieve the task. A simple task like repeatedly loading a truck from a bin of materials with a front loader is a good starting point for realizing what the system will need. This project is a small step towards making full scale autonomous vehicles, which would have several benefits, such as more consistent work quality and lower operation cost.

The following sections will go into a more detailed description of the system, give a general overview of all of the system requirements, and then go into more details about the hardware and software on the vehicle. After this, there will be a summary of final results, and suggestions for future work.

System Description

This paper has mentioned that a starting vehicle will be modified to include sensors to provide information to complete the task at hand. This section will go into more detail about the vehicle, sensors on the vehicle, and then information about the microcontroller and microcontroller interface. The final section will go through the basic operation of the vehicle through one iteration of operation.

Vehicle

The vehicle that was chosen as a base for this project was the RC controlled Bobcat T190. This vehicle was chosen because it had independent track motor control, motorized arm lift, and bucket tilt. Most other vehicles did not have bucket tilt control, or were too expensive. This vehicle costs approximately $100.00, and had the basic functionality that was needed to complete the task. The vehicle also has basic contact switches to provide information when the arm or bucket reaches the extent of motion allowed. These contact switches are enough to handle basic arm and bucket positioning, so that external sensors would not be needed. Since this was the only vehicle that met the project needs, it was the clear choice.

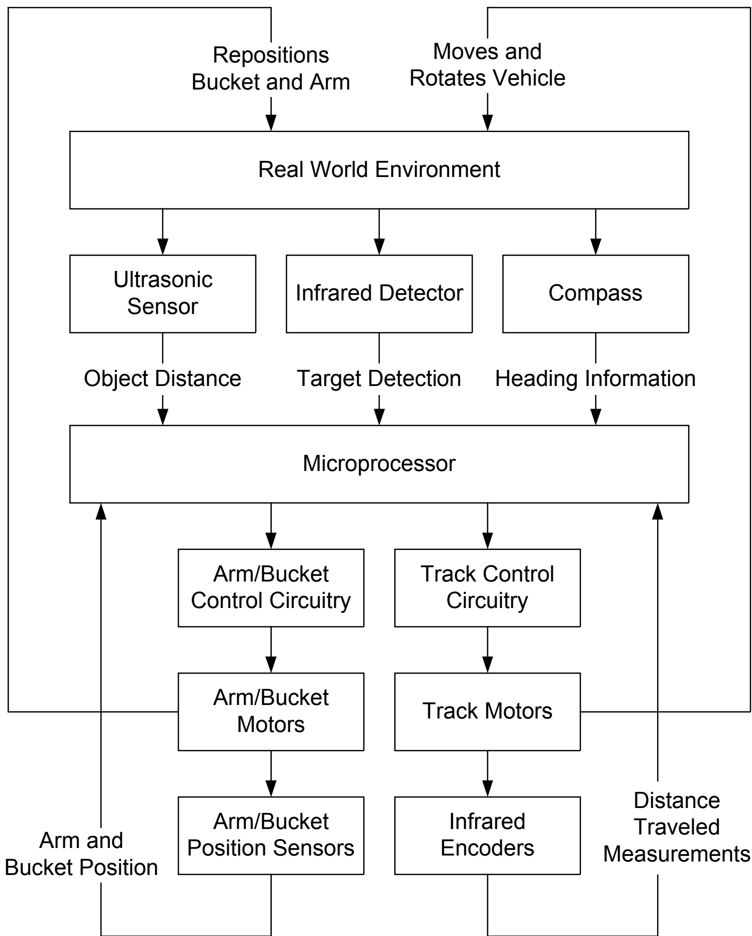


Figure 1: System Block Diagram

Sensors

One important piece of information is how far an obstacle is in front of the vehicle. For this, an ultrasonic sensor is used. It releases an ultrasonic burst in front of the vehicle, and records the time for the first reflection to arrive back at the sensor. The time that it takes for the ultrasonic pulse to return is proportional to the distance of the object in front of the front loader.

Another sensor that is critical to operation of the vehicle is a sensor that can distinguish between the truck and load. For this task, an infrared transistor was added to the vehicle which would pick up infrared LEDs mounted on the truck and load. The load and truck are modified to have infrared LEDs that flash at different rates. This allows the microcontroller to differentiate between the truck and the load, and allows the vehicle to ignore constant infrared light that is in the environment.

In order to better line up with the truck and material bin, a high resolution compass is also mounted on the vehicle. When the vehicle is started, the operator will point the vehicle in the direction to load the bucket and then the direction to dump the material into the truck. This will allow the vehicle to get very close to perpendicular to the truck and load, and make it operate more effectively.

The final sensor for this project is a pair of infrared encoders. Due to variances in track motors, gear trains, and track flexibility, telling the vehicle to drive in a straight line may not make the vehicle drive in a straight line. Adding rotary encoders to the track would provide feedback for a control system to maintain the relative velocities of the two tracks. For example, if the left track is rotating more slowly than the right track, the left motor signal can be increased, while the right motor signal can be decreased.

This could also be used as a measurement of distance traveled by the robot, or if the motors are slowing down for some reason.

Microcontroller

The microcontroller used for this project is the Silicon Labs C8051F340 development kit. This microcontroller can operate up to 48 million instructions per second, has 4 independent timers, 5 programmable counter arrays, 40 input/output pins, serial peripheral interface, as well as many other useful features. The microcontroller is also able to be controlled off of a 7.2 V battery without an external regulator.

One small drawback about this microcontroller is that it operates off of 3.3 V input and output signals, while the rest of the vehicle operates off of 5 V input and output signals. This means that for every input or output from the microcontroller, it needs to be routed through a level converting buffer gate. This is probably a good thing, as it provides an extra stage of protection for the microcontroller. The level shifting buffers used are only unidirectional, which means no pin can be used as an input/output pin.

Also, the microcontroller has an internal debugging interface that does not interfere with the functionality of the microcontroller. The F340 has a potentiometer, two push buttons for inputs, and two LEDs as output devices. A slightly improved output interface would be nice, but would become nearly obsolete after debugging is completed.

System Operation

An example of how the sensors are used throughout one cycle of the vehicle should bring the entire project together, and show how all of the components interact. In the following example, it is assumed the microcontroller has been configured and is operating properly, and has had the directions for truck and load initialized.

The first step in the process is to locate the infrared beacon above the load material. The vehicle will tell the drive motors to rotate the vehicle in place until the infrared beacon is detected by the IR sensor. Once the IR sensor detects the light and determines that the vehicle is pointing in the direction of the load, the vehicle can use the ultrasonic sensor to drive close to the bin. When the vehicle is close to the bin, the vehicle can read the compass, and correct its orientation to be nearly perfectly lined up with the bin. At this point, the vehicle will lower the bucket and arm into a position to load the bucket.

After that, the vehicle drives forward to load the bucket, lifts the bucket up off the ground to secure the load, and turns to look for the truck.

Now the vehicle locates the infrared beacon for the truck by rotating in place until the truck beacon is found by the IR sensor. Once it is found, it uses the ultrasonic sensor and track motors to drive to the truck. The vehicle then raises the arm all the way up, and positions the bucket over the truck. The bucket then tilts down to empty into the truck. After this, the vehicle can back up and put the arm back down to a more stable position. This would be when the cycle repeats until the truck is filled.

As can be seen, the task that the vehicle has to do takes approximately half a page to describe in modest detail. Now, breaking the process down into code that interfaces to sensors and tries to achieve the desired results would certainly take much longer than this to do. The next section will go over some of the important requirements that apply to the vehicle, mainly dealing with operation of the vehicle.

System Requirements

There are a lot of requirements for this project, but most of them are common sense requirements. One example of this is that the drive motors on the vehicle have to be powerful enough to drive the vehicle under load. Without this requirement, the vehicle could not operate, but this is an obvious requirement for the project. This section will cover more about the high level requirements for the vehicle, rather than individual sensor and vehicle requirements. It will also discuss how the requirements are accomplished with the sensors that are on the vehicle.

The largest requirement for the vehicle is that it needs to be able to detect where the truck and load are, tell the difference between them, and align with them properly. This requires nearly every sensor that was added to the vehicle. First, it requires the infrared beacon to detect the truck and load from any place in the sample test area. It also needs the infrared beacon to tell the difference between the truck and load. In order to properly align with the target, the compass will have to tell the vehicle how far from the desired heading it is, and guide the vehicle to the correct heading. After that, the vehicle needs to drive a certain distance from the target, which uses the ultrasonic sensor. If the requirement to drive in nearly a straight line was added, this would involve the infrared encoders and would be all of the sensors added to the vehicle. This is one of the largest routines inside the microcontroller, since it integrates nearly all of the sensors with the drive motor controls in order to achieve the final desired state.

The next large requirement is the vehicle needs to be able to load material into the bucket. There are a few different variations of how this could be accomplished, and they require slightly different sets of sensors. One of the best ways that can be thought of how to do this, besides adding additional sensors in the bucket to tell how full it is, would use the ultrasonic sensor as well as the track encoders. The vehicle would look for the period of the rotary encoders increasing a certain amount without a change in control signals for the motors. This would imply that the vehicle is meeting a significant amount of resistance, and is slowing down. At a certain threshold, the vehicle would recognize that the bucket is full, and begin to secure the load and back up. The ultrasonic sensor could be used as a backup system, and record the distance from the object in front of the front loader. If the distance approaches and stops at a threshold, like the distance from the ultrasonic sensor to the bucket, the vehicle could recognize that it is full and exit the loading routine. Other ways to do the bucket loading routine would be using just one of the processes mentioned above, or just driving in a set distance into the material and backing up.

The remaining requirement would be dumping the material into the truck. After the front loader is aligned with the truck, the arm needs to be raised all the way up to dump material into the truck. This requires driving the arm raise motor and monitoring the switch coming into the vehicle. Once the arm is all the way up, the vehicle needs to drive right next to the truck, probably using the ultrasonic sensor. Once there, the bucket needs to be tipped down until the bucket sensor is contacted. Then the vehicle will be backed up away from the truck, and the arm and bucket returned to a more natural driving position. This function requires the use of all of the motors on the vehicle, along with some of the sensors included on the vehicle.

This concludes the overview of the high level requirements of the project. This should clarify how the sensors are going to be utilized in this project. The next section will go into the circuitry required for the sensors to operate properly.

Hardware Overview

This section will discuss the circuitry implemented on the vehicle for proper operation. The first thing to discuss is the circuitry required to operate the motors and sensors embedded inside the vehicle, followed by the circuitry used to make the additional sensors work properly. For more detailed information about hardware construction, from full circuit connection diagrams to header and buffer chip connections, consult the appendix.

Drive Motors

For all the motors on the vehicle, a quadruple half H-Bridge was used to create a full H-bridge for each motor. The L293 was chosen, because it met the required current limits, had decent heat dissipation capabilities, and has been previously used in a number of Junior and Senior Projects.

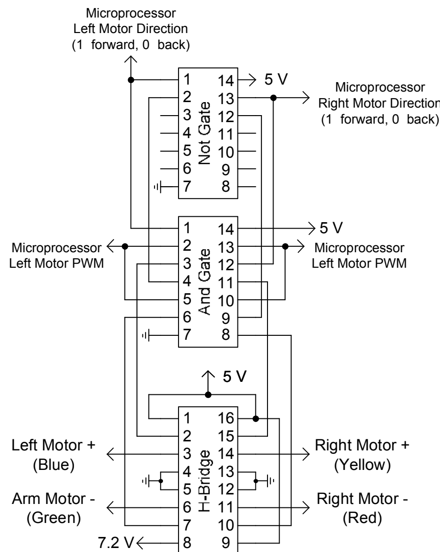


Figure 2: Drive Motor Circuitry

One drawback about this chip is that it does not have much logic circuitry integrated inside it. The circuit acts like two switches that depend on two input signals. If an input signal is low, the output is connected to low voltage, and if an input signal is high, the output is connected to high voltage. Because the microcontroller will use special components to generate a Pulse-Width Modulated (PWM) signal, it is desirable for one control signal to be the PWM, and the other to be the direction for the motor. This is implemented by the logic circuit shown in Figure 2. This figure also shows the necessary bias connections for the circuit to work.

This circuitry provides an inherent variable speed control for the motors. Simply by varying the duty cycle of the PWM signal, the motors will effectively receive a different voltage signal across their terminals. In the original vehicle, the motor control was either off, or full speed one direction or another. This made motor and track differences negligible and allowed the vehicle to drive in a straight line, but it was very difficult to control via remote control.

Arm and Bucket Motors

The arm and bucket motor circuitry is simpler than the drive motor circuitry, but is also important to the project success. The remote control vehicle used the internal sensors to determine when the motors had reached their limits of operation. If the motors were driven past this point, damage may result to the sensors and/or motors. This is especially true about the bucket tilt motor as it works using a linear actuator to move the bucket up and down. By driving this too far in either direction, the linear actuator would jam and the bucket would be inoperable until the actuator is repaired. In the new circuit implementation, special logic protection circuitry is provided to guard against the majority of problems that might be encountered.

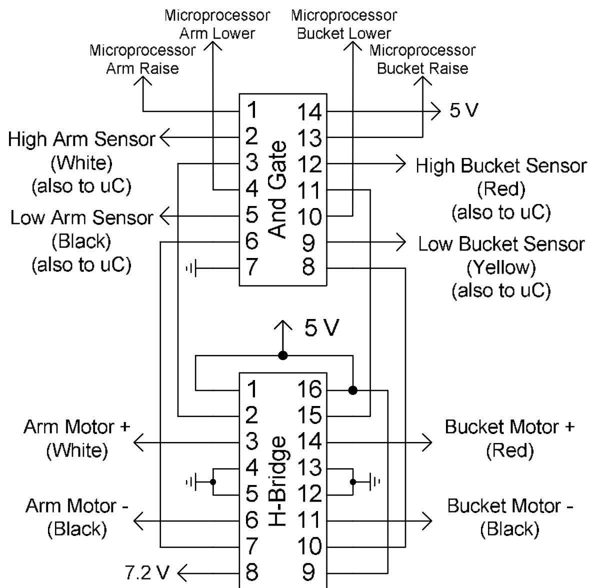


Figure 3: Arm and Bucket Motor Circuitry

The protective circuitry uses simple AND logic to shut off certain motor control signals when the switch is pressed. Aside from the AND logic, this H-bridge does not have any extra external logic circuitry. It is easy to provide motor control signals from the microcontroller to control the direction, and variable speed control is not needed for this component. The arm motor is already very slow, and the rate of bucket tilting has virtually no impact on the project.

Vehicle Sensors

The remaining hardware left from the vehicle is the limit switches for the arm and bucket motor. These signals work fine during testing without an external pull-up resistor. Unfortunately after vehicle construction was complete, the output signal from these sensors was somewhat noisy. Either software can handle this by looking for the signal to be at a certain level for the majority of the time, or a pull-up resistor may be added to the circuit board to provide a better signal. Initial investigation with attaching a pull-up resistor to these sensors appeared to have minimal effect on noise, so this project uses simple software correction to detect if the state is on or off.

This system has difficulty when the vehicle is operating the track motors at the same time as the arm and bucket motors, so the arm and bucket sensors are only utilized while the vehicle is stopped. To avoid extra noise, only one motor is allowed to operate at a time in the intended bucket and arm maneuvering routine. This happens to provide a safer order of operations for the vehicle. If the arm is all the way down, and the bucket is tipped down, the bucket will press against the tracks, and it may become difficult to correct. The software can check for which state the arm and bucket are in, and prevent this order of operations from occurring, so the vehicle will not damage itself.

Infrared Transistor

In order to get the infrared transistor to work over a large range, the current signal coming from the transistor had to be amplified by a significant amount in order to have the necessary detection range. In order to achieve this, there are two op-amp stages the signal passes through. The first one amplifies

the signal while inverting it and the second one inverts it again and filters out noise. After the op-amp stages is a Schmitt-Trigger, which is used to provide a clean digital signal for the microcontroller, as well as remove abrupt changes in the signal output from very small amounts of noise. This can be seen in Figure 4, with the transistor shown on the left, with the op-amp stages in the middle, and the Schmitt-Trigger on the right.

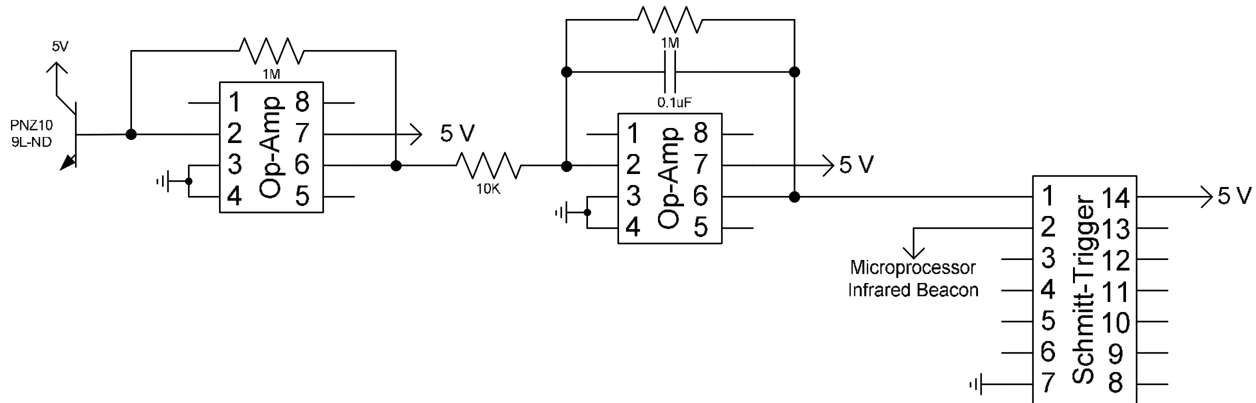


Figure 4: Infrared Transistor Circuitry

Infrared Encoders

Similar to the infrared transistor, the infrared emitter/reflector or infrared encoder has a Schmitt-Trigger on the output to reduce noise. Other than this, the encoder's diode and transistor need to be biased correctly so they will operate. The necessary circuit connections and resistor values are shown in Figure 5. Since the encoder works over a smaller range, the reflected signal is stronger, and does not need any op-amp stages to amplify it.

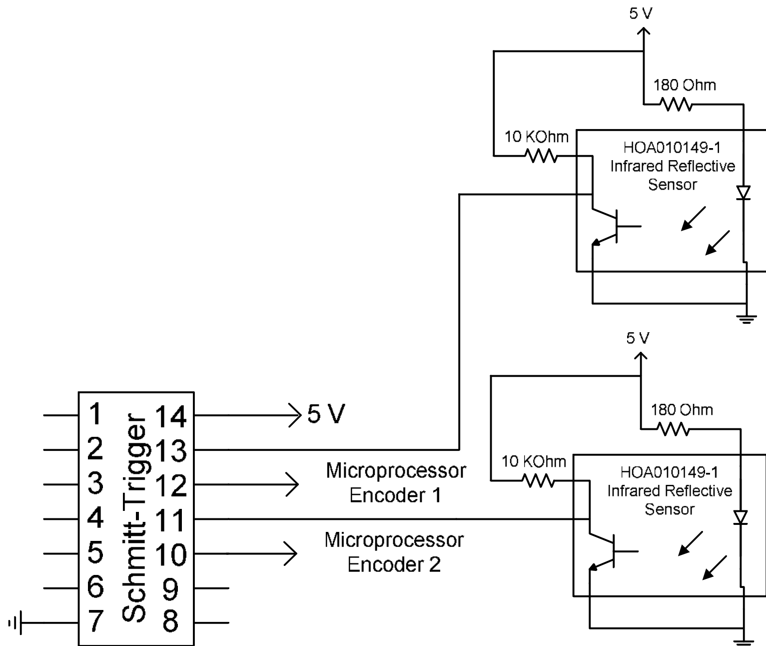


Figure 5: Infrared Encoder Circuitry

Compass

The compass is a fairly simple sensor to connect properly. Since Vector V2X compass is a self contained module, it just needs to be connected and biased properly, as shown in Figure 6. There are two outputs and one input to the compass circuitry. The input is the enable signal on the P/C line, while the outputs are the SDO signal and SCLK signal. The compass provides a serial output signal through the SDO, and the SCLK signal allows proper reading of this signal.

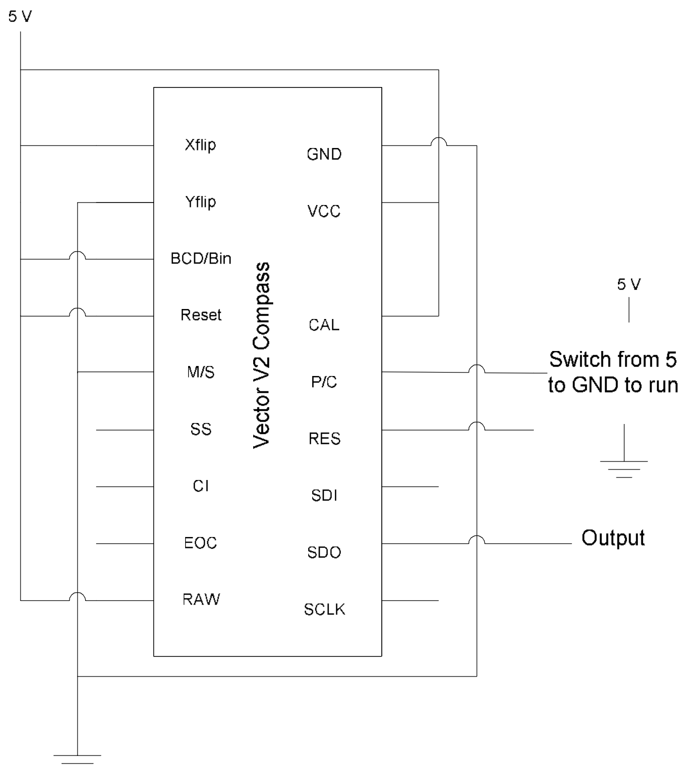


Figure 6: Compass Circuitry

The compass requires a special serial port function in order to avoid writing code for a serial port routine. Fortunately, the F340 has a serial peripheral interface that can handle this. Unfortunately the microcontroller only reads eight bits at a time, so software will have to keep track which byte is received, and then reconstruct the output signal from the received bytes. This will have to be done on the microcontroller, and is not a hardware problem.

Ultrasonic

The ultrasonic sensor chosen for this project is the SRF05 ultrasonic sensor. This model can detect objects out to approximately 5 meters, and down as close as about 1 inch. Fortunately, this sensor is also a self-contained module, so it only requires biases and connection to the microcontroller to work.

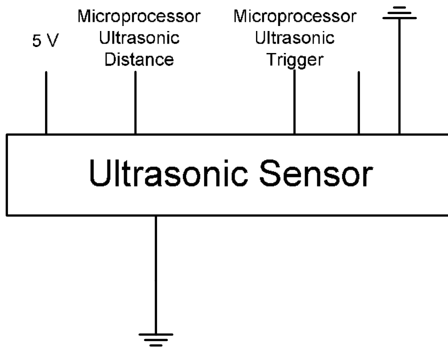


Figure 7: Ultrasonic Circuitry

As shown in Figure 7, the ultrasonic is really simple and has minimal connections. It has 1 input, and 1 output; the input is a short pulse to denote a new ultrasonic reading is desired, and the output is a signal whose positive edge is proportional to the distance of the object.

3.3 To 5 V Unidirectional Buffer Chip

Now that all of the sensor circuitry has been mentioned, the remaining major hardware element to discuss is the 3.3 to 5 V, or 5V to 3.3V converter. This chip corrects the improper voltage levels that would be seen between the microcontroller and the rest of the circuits. The buffer can only convert from one voltage level to another at any instance, and since there are more than 8 inputs and outputs in this system, there are a total of four buffer chips on the main circuit board. Two of these convert from 3.3 to 5V, and the other two convert from 5 to 3.3V. For more information about what signals go through which chip, consult the Appendix.

5V regulator

Since this system uses a battery as a power source, the voltage level may not be very constant. In order to smooth out the voltage used to power logic chips and other circuitry, a 5V regulator was attached to the main circuit board. The only circuit for the regulator is a large capacitor across the output terminal of the device. With the capacitor, the voltage regulator functions as well as could be expected.

This concludes the discussion on the hardware overview of this project. The next section will discuss the software overview, starting with the software block diagram.

Software Overview

Even though hardware implementation took a majority of the lab work, the software running on the microcontroller is the most important part of the project. This section will begin with the software block diagram, software function descriptions, and a description of how microcontroller special functions are used. For the final code used on the vehicle, refer to the appendix.

Software Block Diagram

Figure 8 contains the software block diagram that is used for this project. From the block diagram, it is clear that there are multiple loops for the code to go through. The software would have 2 embedded loops, one to record how many cycles of operation the vehicle has performed, and another to run the vehicle between the material and truck. Inside the location and navigation loop, the majority of the code is being used. Unfortunately the loop to record how many cycles of operation was not completed in final software, but is a negligible portion of the project code. The software block diagram depicted in Figure 8 is essentially the process described earlier about how the front loader must operate.

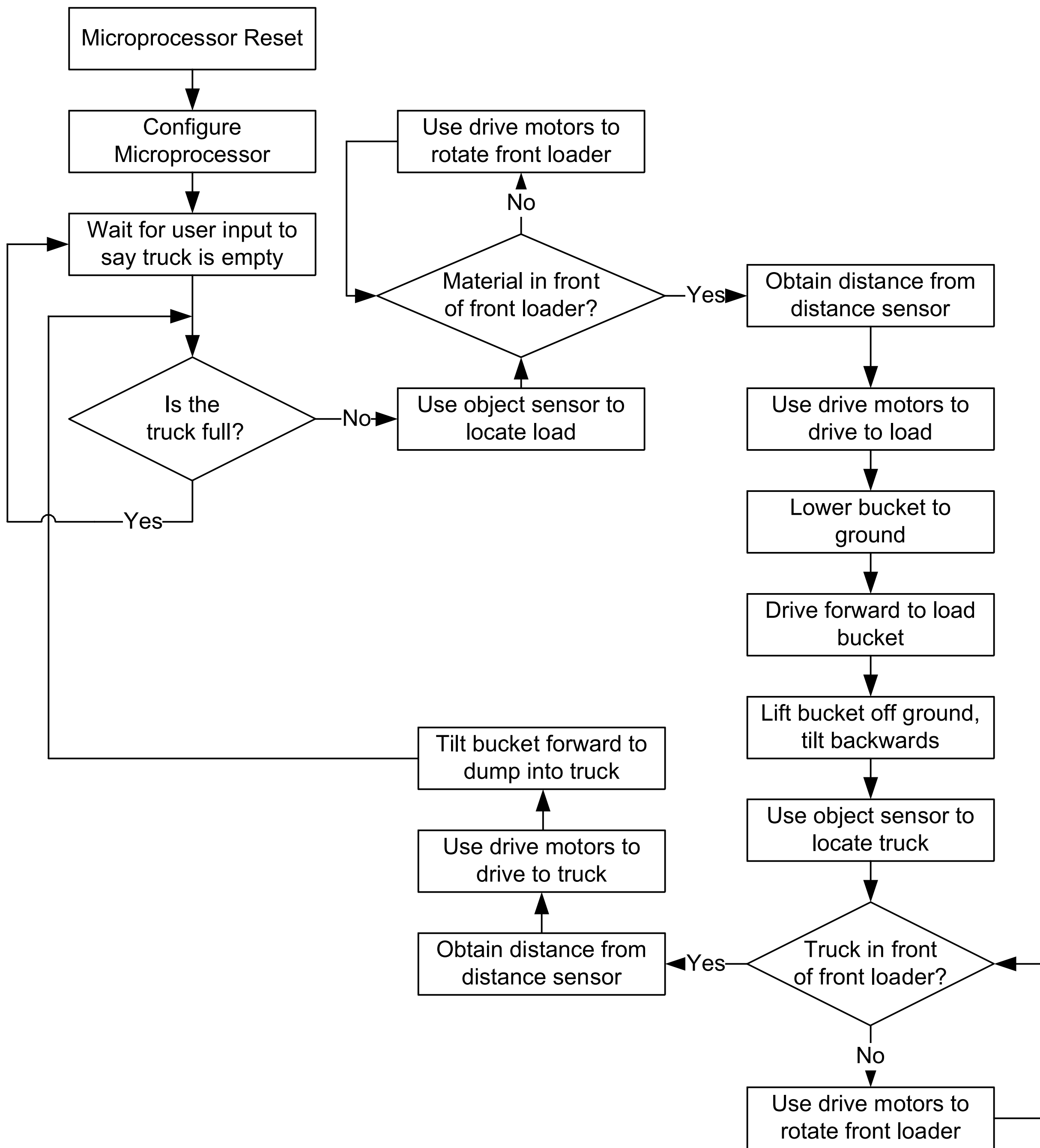


Figure 8: Software Flowchart

Software Function Description

There are several functions that are used to achieve this project's goals. The order that these functions are discussed in this section is arranged as follows: foreground low level functions, foreground high level functions, and background operations.

Low Level Foreground Functions

The definition in this paper for a low level function is a function that does not rely much on other functions, and accomplishes some small task, such as updating the drive motor PWM signal, or controlling the arm and bucket motor. The functions near the beginning of this section discuss things that are directly relevant to motor control, and the later sections discuss compass and timing functions.

Motor Control

There are four functions that are used to control the motors: newlinearpwm, newrotationalpwm, vehiclestop, and bucketraise_armraise. Due to varying loads, different levels of battery charge and somewhat low resolution encoders, these are not velocity commands. This could be changed to a velocity command function with some additional work with the vehicle, but setting PWM percentages was enough for this project. Another reason why this is not a velocity routine is because each subsystem was being tested and coded independent of other sections. To ease debugging issues, only drive motors were used at first, and then encoders were integrated into the velocity controller routine. More information about the velocity controller routine is provided later.

The function newlinearpwm updates the internal variables corresponding to the desired direction for the vehicle to go, and then calls a function to reset the velocity controller to handle this change. This function adds the same percentage to both the left and right motor, in order to allow them to drive in a straight line. For initial simplification for operation in a linear region, this function can only generate new PWM signals between +/- 50 to 80% duty cycle. Positive PWM implies moving forward, and negative PWM implies moving backward. The PWM range is limited to allow the newrotationalpwm and internal controller to have a small range of control to correct for track differences. This function sets the direction control bits for the motor to move in the desired direction, and leaves the initial PWM configuration to another function known as encoder_reset.

Encoder_reset is used to update the PCA registers with the correct values to generate the duty cycle desired, as well as configuring new controller parameters for the rotary encoders. The motor control functions newrotationalpwm, newlinearpwm and vehiclestop call this function, so the code will be consistent.

The next function that is discussed is newrotationalpwm. Newrotationalpwm takes the desired percent PWM change, and adds it to one motor side, and subtracts it from the other motor side. For example, if a rotational PWM of 40 percent is given when there is no linear velocity, +40 percent duty

cycle is generated for the left track, and -40 percent is generated for the right track. There are a few different conditions for the rotational PWM function. If there is no linear velocity, the truck may rotate between +/- 40 to 90 %. This allows the vehicle to spin in place to locate beacons. If there is a linear velocity, the rotational PWM is limited between +/- 10 %. This mode is intended to allow the vehicle turn slightly left and right while driving, in case a small turn is desired.

Now that the newlinearpwm and newrotationalpwm functions have been discussed, more detail about the velocity controller can be provided. Whenever encoder_reset is called, it calculates a new increment amount for the left track, new decrement amount for the right track, and sets up PCA0 and PCA1 with the correct increments to do the first few PWM pulses for the motors to move. This allows the controller to work for all possible newlinearpwm and newrotationalpwm states.

Unfortunately this routine uses floating point calculations, which is not as efficient as fixed point calculations. The track with the lowest absolute PWM percent was set to an increment/decrement of 1, while the other track was set equal to the ratio of the two PWM percentages. In order to correct this to a fixed point calculation, some minor changes would need to be done. First, some arbitrary minimum increment could be decided, such as 100. Then calculating the respective track increments/decrements would be set to fixed point calculations, where the faster track increment would be set to 100*high PWM / low PWM. Finally when the velocity controller is operating, an attenuation of 1/100 would be added to the control system, to compensate for the arbitrary minimum increment. Unfortunately, this problem was discovered with less than one lab period left, so this is left for future work.

Another potential flaw in the encoder_reset function is that every time the routine is called, the variables for the controller are reset. This would cause a problem if some function was continuously setting new motor speeds, which would cause the controller to essentially be in permanent reset mode. Adding code only to reset internal variables only when a velocity command is starting a vehicle from a rest position may be desirable.

Inside the interrupts for the left and right encoder, the period for the current encoder pulse is updated. After this, assuming the vehicle is not stopped, there is a global control count variable which is incremented in the left encoder interrupt, and decremented in the right encoder interrupt, and then the encoder_pwmupdate function is called. Some basic signal limiting is done, and then the output PWM percentages are updated. At the beginning of this function, there are two variables of interest; pwmlimit and gainfactor. The pwmlimit variable is the extent +/- the output PWM is allowed to move from the desired PWM signal, and gainfactor is the gain of the simple proportional controller. If the left motor is

slower than the right motor, the left motor PWM is increased until it reaches desiredleftpwm + pwmlimit. At this point, the right motor PWM is decreased until it reaches desiredrightpwm-pwmlimit. If the right motor PWM is decreased to desiredrightpwm-pwmlimit, the system can be considered saturated, and the error signal will probably grow until it reaches the limit for the variable. If the left motor is faster than the right motor, the similar process is executed, except the right motor PWM is increased to the limit, and then the left motor PWM is decreased to its limit.

A simple PI controller was implemented initially, but severe oscillations were noticed in the vehicle with an integrator gain of one. Due to a lack of time to create a model for the gear trains and tracks, the PI controller was abandoned for a simpler proportional controller. With an additional month or two, this would have been investigated, but in order to get final results, a proportional controller was used.

The next function discussed is the vehiclestop function. This function is intended to be a fast stop routine to stop all of the motors operating on the vehicle. This function stops the arm/bucket motor and reverses the track directions for a small amount of time, approximately 10 milliseconds. After 10 milliseconds elapse, the motors are stopped, and linear and rotational PWM is set to zero. Initially when the vehicle was operating, there was a large amount of overshoot between telling the vehicle to stop, and where the vehicle would actually stop. This function combines all of the motor stopping routines into one, and appears to stop the vehicle faster than using newlinearpwm(0) and newrotationalpwm(0).

The last motor control function is the bucketraise_armraise function. This function stops all the vehicle motors, and then checks to see if it is desired to have the arm and/or bucket raised or lowered. The first thing to be controlled is the bucket, which will drive in the desired direction until it presses the contact switch for long enough to guarantee a good contact. The next step then moves the arm motor until it hits the desired switch for long enough to guarantee a good contact. This function does not allow the arm and bucket to be lowered in the same function call. This situation has caused difficulties when the bucket and arm are both lowered, since the arm cannot lower far enough to press the contact switch, and leads to an infinite loop. Upon reset or reprogramming, the tracks rotate at full speed, and grind against the bucket, which may cause damage to the track motors and bucket. This is the reason for excluding both of the arm and bucket to be lowered in one function call.

One note about the contact switches: there is a high amount of noise coming back on the sensors when the motor is operating, which causes a lot of false zeros. In order to avoid adding additional pull-up resistors with more power consumption, the sensor is just watched to stay stopped for approximately

1 second before it is read as a true zero. In the future, pull-up resistors may be added to reduce noise, but this was the easiest fix for the problem.

Compass functions

There are two functions that are callable that directly interact with the compass, and one function that tries to orient the vehicle to a specific compass heading. The two functions that directly interact with the compass are compass_enable and compass_disable, compass data is handled through the SPI0 interrupt and the function to rotate to a heading is called degreeheading. The information coming from the compass is passed in through the SPI0 interrupt, where the two bytes of information are stored, and reconstructed as soon as both are received. Once the information is reconstructed, a new compass reading flag is set, and the interrupt waits for another byte of information.

Compass_enable and compass_disable are very short function calls that mostly reset states inside the microcontroller, and control the enable/disable pin for the compass. In compass_enable p0_5 is set while the SPI interrupts are cleared, and the read data is set to zero. After this, p0_5 is set back to zero to start the compass. In compass_disable, p0_5 is set high to disable the compass, and interrupts are cleared and data is cleared. Current code only enables the compass and never disables it. It is unsure what would happen if the compass is disabled in mid-transmit of data, so the compass is never intentionally shut off.

The function degreeheading takes two parameters, degreeface and tolerance. The desired ability is to rotate the vehicle to the compass heading degreeface within +/- tolerance. The function is supposed to rotate the vehicle the fastest direction to orient to the heading desired, with a fast rotation for a large change, and smaller rotations for getting within tolerance. Unfortunately there appears to be some problem in the hardware and/or software when this is used.

First, the function does not rotate the shortest angle to the desired heading; it always seems to rotate counterclockwise. This appears to be a software problem, with the source of the problem currently being unknown. The other problem is the vehicle occasionally does not orient to the correct heading after completing one cycle of operation. The heading that the vehicle tries to face is sometimes off by 20 to 45 degrees, which causes the vehicle to be unable to complete another cycle. The source of this problem is unknown, whether it is residual magnetic interference from the bucket tilt motor, if one of the magnetic coils has slightly changed into a different orientation, if there is some corruption of internal variables or a bug in this function that cannot be found. It is unlikely that the internal variable is

getting corrupted, as the variable is in external data and only written to when the bin and truck direction are being read in. It is believed this is some odd hardware problem with the compass, as the vehicle will sometimes operate normally for more than one complete cycle without this problem arising.

Timing functions

The remaining low level functions that the vehicle has are the two timing functions, waitsec and waitmsec. In the function waitsec, the function counts increments of 20 sets of the 50 ms flag. Technically the timing is slightly off for the 50 ms flag, since it is based on Timer 2's period. Timer 2 is set up to operate from a 4 MHz clock, and it's a 16 bit timer. The overflow time for this would be 2^16/4 million, or approximately 16.384 milliseconds. Three counts sets of this overflow sets the 50 ms flag (and also controls the ultrasonic sensor start flag). Technically this is a period of 49.152 milliseconds, which is not counting time to vector and return from interrupts. Timer 2 also operates as a continuous timer to provide measurements for remaining tasks on the microcontroller. The larger source of error for short times would be the fact that the 50 ms flag may be much less than 50 ms from being set. This could mean the time for waiting one second would vary from about 1 second down to about .934 seconds. The waitsec routine is not a critical timing function, and works well enough for this vehicle. The function should also work for a few minutes, although it has not been tested for anything much longer than 20 seconds.

The other timing delay, waitmsec, uses the 1 millisecond flag set by the overflow of the PCA clock. This period is slightly over 1 ms, but is still not very critical to vehicle operation. The PCA timer does not have an auto-reload state, so the value is overwritten with the time that would make a 1 millisecond period. This makes sure that the PWM code will not miss a 100% duty cycle transition, but may cause slight variation in the period of the output signal (expected not to be much longer than +/- 1 microsecond for every millisecond period). The PCA timer should be set to the highest priority interrupt.

These are all of the low level functions implemented in code up to this point. The next section will describe the code that achieves higher level tasks, such as finding the truck and load, driving to them, and everything else.

High Level Foreground Functions

Currently there are two high level functions in this project, one to locate the desired infrared beacon called locateIRbeacon, and another to drive a certain distance from an object called drivetobeacon. These two functions accomplish the majority of the goals of this project, with the help of the low level tasks mentioned earlier. Since these functions are so crucial to the project, the next few paragraphs go through a detailed description of the function inputs, outputs and behavior.

LocateIRbeacon

The locateIRbeacon function takes a variable called targetloc, and proceeds based on that. If targetloc is zero, the function tries to locate any IR source. If targetloc is greater than zero, the function tries to locate the load material which has a beacon that flashes at 10 Hz, which is set from the internal variable loadperiod. If targetloc is less than zero, the function tries to locate the truck beacon that flashes at 5 Hz, which is set from the internal variable truckperiod. There are four variables of interest in the beginning of this function: loadperiod, truckperiod, remainingtargets and waitcount. Loadperiod is the period of the load beacon, which has a tolerance of +/- 40 ms, and truckperiod is the period for the truck beacon, with a tolerance of +/- 40 ms. Remainingtargets is the number of infrared targets that the function is allowed to see before returning an error, and waitcount is the variable that controls how long the vehicle will rotate in place until returning a timeout error. Remainingtargets was implemented for such instances as the vehicle is looking for the bin of material, but can only see the truck or other infrared sources. Rather than having an endless loop here, the vehicle will break out of the loop after seeing the number of targets held within the remainingtargets variable. Currently the number of infrared sources allowed to be seen before an error is generated is three. This may need to be increased at a later date, as constant IR sources (like from windows) have a tendency of being counted multiple times.

The function returns one of two values at this point, 0 or -1. If the function accurately finds the desired target, the returned value is 0. If the function cannot find the desired target, it returns -1. The vehicle starts by rotating in place clockwise, monitoring the IR transistor signal waiting for it to go low. If the IR transistor does not go low within approximately 6 seconds, the vehicle stops and returns -1. Another way for this function to return a -1 value is if multiple IR sources are located, while the desired IR source was not found. Currently if a -1 value is found in the main loop, the vehicle pauses 4 seconds and repeats the main loop. Future expansion could replace this code with a more elaborate location

routine, but it currently seemed unnecessary when the testing area was a small size and set up in a simple configuration.

Now that the inputs and outputs of the locateIRbeacon function have been discussed, the process the function goes through to locate an IR beacon will be explained. First, the function initializes variables and compass limits for the current target, and then enters an infinite loop. This loop stops the vehicle, checks to make sure it should still be looking for targets, and then begins rotating clockwise at the slowest speed allowed. The vehicle enters a timing loop, which checks for an infrared signal and records the elapsed time. If either the vehicle times out, or an infrared source is found, the vehicle leaves the timing loop and stops. The vehicle checks on which condition it exited, and continues operating if it found an infrared beacon. Next, the software drives to less than half a meter from the beacon in order to get proper frequency measurements from the beacon.

Once the vehicle is close to the beacon, the heading from the compass is checked to verify that the vehicle is facing the right general direction. If the vehicle is within +/- 20 degrees of the desired target heading, the period of the IR light is measured. The vehicle waits up to 1 second for the light to shut off, in order to avoid problems from DC infrared light. Once the light shuts off, a timer variable is increased through one whole cycle of the signal, or until 1 second elapses. Once the period is measured, it is compared against the desired period within +/- 40 ms. If the period is not a match, or the IR light source is a DC source, the vehicle turns clockwise for a quarter second before repeating the infinite loop to find an infrared beacon. If the period is close enough to a match, the function returns zero and proceeds.

One small problem with this function is that the function was originally written to use the IR beacon, but not the compass. Once the function was near completion, the compass interface was finished. The resulting code for the compass to verify if a target was within the acceptable degree heading was added, but more could be done to integrate compass readings with the infrared signal results.

Drivetobeacon

The drivetobeacon function is intended to do one or two tasks. First the function is meant to drive a specific distance from the current target the ultrasonic is seeing. The other task it may be told to do is to follow an infrared source. This function drives with a velocity roughly proportional to distance,

in order to reduce the force acting on the material that may be carried in the bucket, and to make certain there is less chance of accidentally running into objects.

The drivetobeacon function takes three variables: the desired distance from the object detected by the ultrasonic sensors, the period the infrared light would be off, and if the function should attempt to follow the infrared source. The drivetobeacon function was originally a drive to a specific distance and stop, but was modified to try and track an infrared target. The function has two different return values, 0 and -1. The only way to get a -1 return value at this point is when the vehicle loses the infrared beacon and cannot find it with small rotational changes.

The function handles both cases of needing to back the vehicle up, and driving the vehicle forward. The code is primarily 2 if statements inside a large while loop. The first if statement differentiates between if the vehicle needs to drive forward, drive backward or if it is at the desired position. The second if statement keeps track of the infrared beacon, to make sure the vehicle can continue to see it. The first if statement calculates the distance that is needed to travel, and sets the forward PWM to twice the necessary distance. The PWM percentage should be limited naturally in code between 50 and 80 percent, but the function limits the forward PWM percent between 50 and 80 percent as well.

The concept behind trying to track the infrared beacon is that the vehicle would turn slightly clockwise and counterclockwise to look for a lost signal. Unfortunately with only one IR transistor, the software does not know which direction to turn to try and reorient to the beacon, so the code turns slightly one direction for a set amount of time, and then turns slightly back for twice the time as the other turn. If the infrared light is lost through the entire attempt, an error is flagged, and returned to the main function. Unfortunately, not much testing was done on how well the routine would track the infrared light. More improvement could probably be done with the function, but it works well enough for this project's purpose.

The only suggested improvement to the drivetobeacon function is to add code to prevent the software from rapidly changing directions of the motors. If something got in front of the ultrasonic sensor while driving to a target, such as a hand trying to correct something on the vehicle, the vehicle could possibly rock back and forth. This occasionally caused material to spill from the bucket, back into the vehicle. Some filtering for the ultrasonic sensor, or to provide a waiting period to change vehicle velocity would probably be good.

Other Tasks

The other high level tasks for this project would be the bucket loading and bucket dumping routines. These routines are fairly simple, just reposition the bucket/arm and then drive forward until some condition is met, and then reposition the bucket/arm. Currently these tasks are coded directly into the main function, so there is no actual function call for them. The programming for these functions is fairly simple, so the discussion of them will be skipped.

Background Operations

The software operating on the vehicle has nearly all of the sensor interfacing done through interrupts. The ultrasonic sensor is set up for constant operation using the timer 2 interrupt routine. This routine generates a short, 10 us high pulse every 50 ms or so. The returned signal from the ultrasonic sensor is brought in through the PCA3 interrupt, which is set to trigger on both edges. On a rising edge, the timer 2 time is captured, and the falling edge computes the difference in time. This difference is turned into a distance, and sets a new distance reading flag in the flag register.

The infrared encoders are operated on EX0 and EX1, which calculate the corresponding track encoder period, and then call the velocity controller for an update. The left and right track encoder periods may be read at any time in the main code.

The compass serial interface is handled through the SPI0, and data is reconstructed inside the SPI0 interrupt. When the data is reconstructed, the new compass reading flag is set, which may be read at any time.

The only sensor which is directly read in code is the infrared sensor, since it is just a single port. Additional interrupts and flags for this sensor seemed unnecessary, since it was easy enough to program directly into code.

This concludes the software section of this report. The next sections will deal with final progress, and future work for this project.

Final Results

This section covers the final results of this project, mainly dealing with hardware and software. Starting with the hardware for this project, all of the hardware successfully communicates with the microcontroller. The microcontroller can successfully keep track of every sensor when tested individually, and can usually keep track of sensors when operated in parallel.

There are a few exceptions to the sensors working as intended. The first instance is that the ultrasonic and infrared sensors do not work when blocked by the bucket. This is nearly an unavoidable problem, but the sensors do work when the arm is in the fully raised or lowered position. Another exception to this is the compass. The compass does not work when the bucket is fully raised, which is probably caused by the bucket tilt motor near the bucket of the vehicle. The compass readings may also be distorted while any of the motors on the vehicle are operating, because of the electromagnetic fields generated by the motors. This has led to usually only taking readings from the compass while the vehicle is stationary, and waiting for a few readings after moving the vehicle, in order to avoid false readings. Other than this, the sensors have appeared to work as desired with the current hardware setup.

The software operating on the vehicle is able to navigate and differentiate between the bin of material and the truck fairly well. There is a small problem where the vehicle will line up slightly off from the desired heading. Part of this is because the infrared beacons do not provide a good reference to point directly at the bin or truck, since the light spreads out somewhere between 10 and 15 degrees. The other part of the problem is the compass appears to give false readings at times. It is not known if this is a software issue that the serial interface misses a bit of information, or if the hardware for the compass is really giving a messed up direction. When the vehicle is driving around, the balanced spindles on the compass may rotate a little bit. This may cause the compass to provide different directions than were provided earlier.

Software does appear to provide a decent way of loading the bucket without any additional sensors. Using the period generated by the rotary encoders, a threshold can be set to tell the vehicle that it has slowed down enough, and probably has a full bucket. If the rotary encoders fail though, the ultrasonic sensor can also be used to tell the vehicle to stop as well. If the same distance reading is received for approximately a second, the microcontroller can assume the bucket is full and stop loading it. This combination of sensors has been seen to be very effective, as long as there is no human intervention during loading. If the vehicle is bumped or touched during loading, the vehicle will sometimes stop loading too early. Fortunately this would not be too much of a problem on a full sized vehicle, as it would take a large amount of force to have the same effect as seen on this small vehicle.

The final combination of software and hardware does accomplish the goals for this project. The vehicle is fairly cheap in cost at around only $300 to 400 plus labor, all the sensors to accomplish a set of tasks have been modified and added to the vehicle, and the vehicle can operate without human

intervention. Occasionally human intervention was needed to compensate for low resolution infrared beacons, but for the most part the vehicle could complete one or more cycles of loading a truck.

Future Work

This project has a lot of possibilities for future work. The first section will talk about potential expansions that would require a large amount of work to complete, while the second section will talk about more minor modifications.

Large Projects

The first large project that could be done is to add a camera and computer interface to the vehicle. With advances in wireless USB technology, a wireless webcam could be mounted to the top of the vehicle, and could transmit information to an external computer. This computer could then process the images from the camera and send instructions to the microcontroller on the vehicle. This would be a large project, as image processing routines would need to be developed, and the vehicle software would need to be modified for this new setup. Navigating the vehicle by image processing would be a necessary step towards large scale implementation of the autonomous front loader.

Another large project would be to create another autonomous vehicle to do some task that would interact with the autonomous front loader. An example would be to operate the truck that is being loaded to go off and dump someplace else, and then request to be filled. The cooperative software between vehicles could then be developed, which could be evolved into the operation of a whole facility though time. This vehicle and project should have some good information for hardware to be used, and can prove that a difficult task is possible to be done.

One more task that could be completed is to model the track motors and gear trains on the vehicle, in the hope to develop a better velocity controller routine. This project would probably start with replacing or upgrading the infrared encoders to a higher resolution. Once that is done, various controllers for good velocity control could be investigated. One of the difficult tasks for this is the fact that the track wheel only goes up to approximately 2 revolutions per second. If a fixed period controller was to be used, the highest period that would be desired would probably be 10 milliseconds. This would operate at approximately 100 Hz, which should provide decent control over the velocity of the vehicle. If the highest number of pulses per period was desired to be 10, that would mean the rotary encoder would need a resolution of approximately 500 pulses per revolution. This rotary encoder would not be

easily made in the same way the rotary encoder for this project was, which would almost assure that a varying period controller is needed. Developing a good varying period controller would be difficult, and may make a good project.

One final task that might take a long period of time is to redo most/all of the software programming on the vehicle in order to make it more robust and more user friendly. In the autonomous mobile agent course, there is a standard library (an ARIA library) for the specific robot that is being used. Writing some similar user interface so that this vehicle may be used more readily by more people may be a difficult task. This task would most likely require more modeling of the system, and perhaps addition of more sensors.

Small Projects

The first thing that would probably be done for this vehicle if there was more time would probably be to do minor hardware changes. Making the drive motors active low would be the first minor hardware change. Whenever the microcontroller is being programmed, all output pins go to high, which causes the track motors to go full speed forward. Making these active low would prevent potential damage to the vehicle while programming it. A simple rewiring of the direction pins through one of the available inverting gates would fix this. A subsequent software change would be needed in order to compensate for this change. Another hardware change would be to replace the battery with a higher charge capacity battery with better charging equipment. The current batteries that came with this vehicle appear to fully discharge within one week even if they are not connected to any load. The batteries also have a tendency to not hold a charge for longer than 20 minutes after being charged for a few hours. This caused much frustration near the end of the project, when testing was delayed due to lack of battery power.

Another hardware change that would be suggested is to modify/replace the infrared transistor. The sensor does not receive high duty cycle signals higher than 10 Hz or so, which was a disappointment in this project. Getting rid of this infrared circuitry would be desired if the system is moving towards an image processing approach. If the vehicle is not moving towards image processing, adding another upgraded infrared transistor may be beneficial. This would allow better navigation to the truck and load. The compass also needs some work to figure out why the readings are sometimes off. Replacing the compass with a newer model may be desired. The current model, the Vector V2X, is no longer manufactured, which may not help if more vehicles are desired. Other hardware changes would

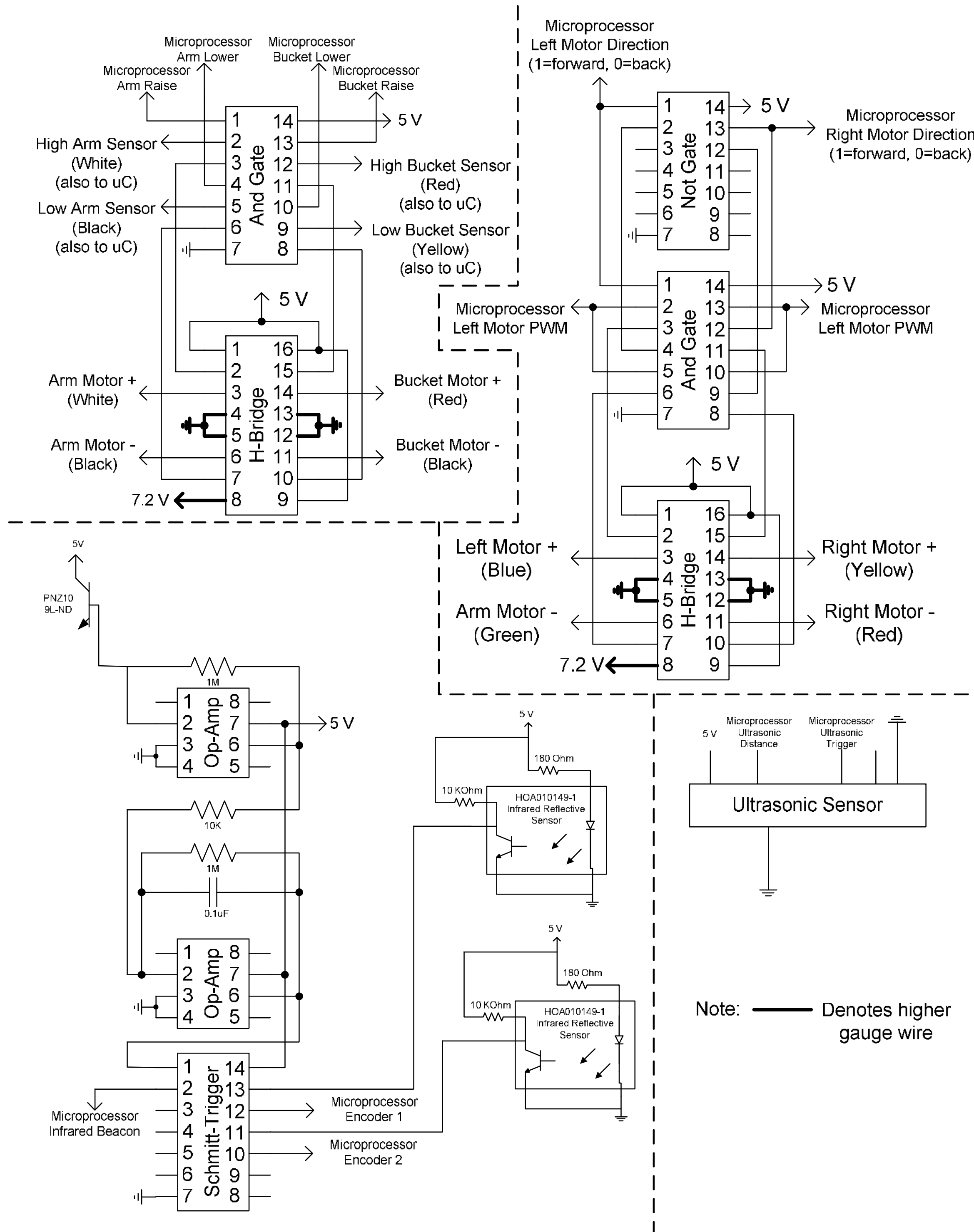
involve swapping the position of the ultrasonic and infrared beacon, and making a more permanent mounting station inside the vehicle for a circuit board and microcontroller.

Index of Appendices

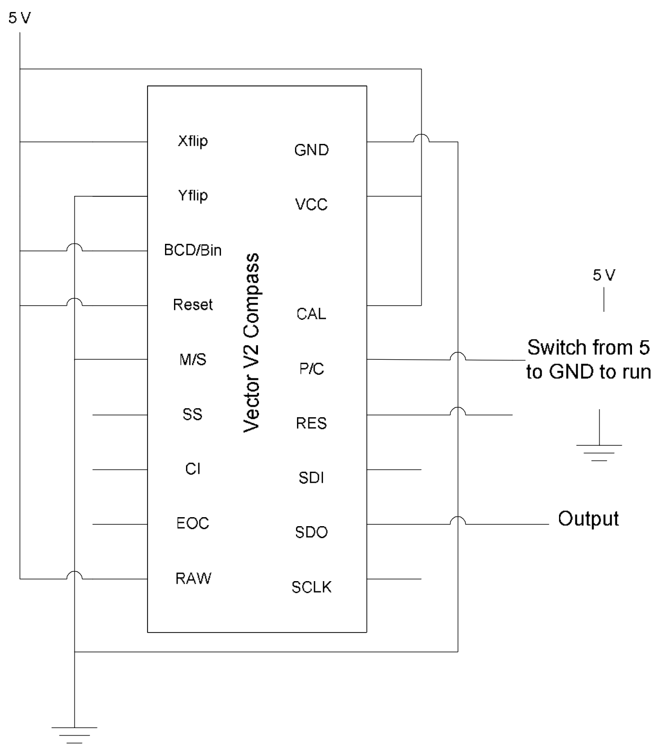
Appendix A: Detailed hardware circuitry and connections
Appendix B: Final project code
Appendix C: Vehicle Modification Information
Appendix D: Vehicle Operation Information

Appendix A: Detailed hardware circuitry and connections

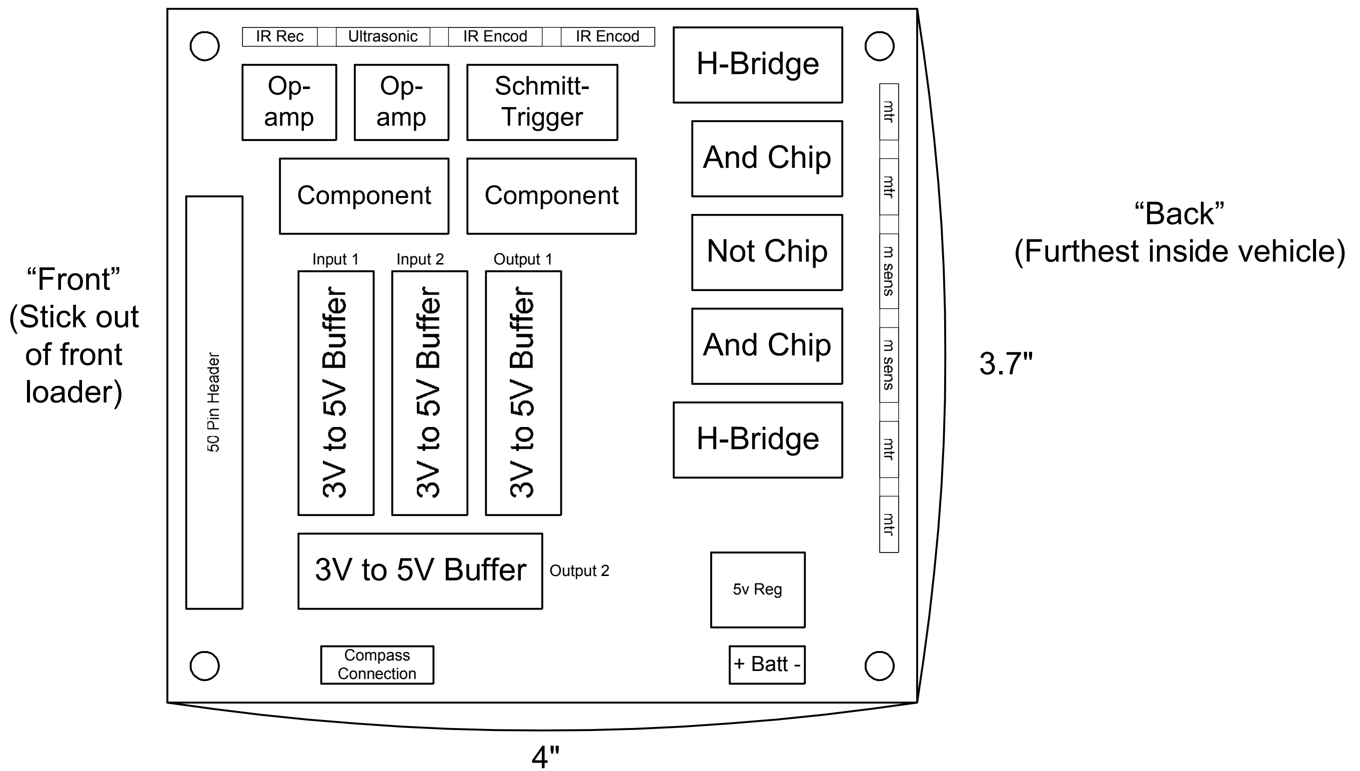
Overall Circuit Diagram:



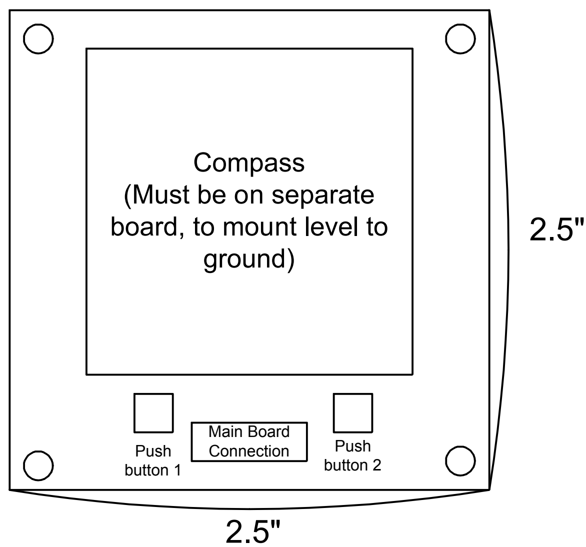
Compass connections:



Main circuit board layouts:



Compass board layout:



Buffer Chip Connections:

5V side

3.3V side

Input 1 chip

5V	1	VCCA	VCCB	24	3.3V
0V	2	DIR	VCCB	23	3.3V

Left Mtr Dir	3	A1	OE BAR	22	0V
Right Mtr Dir	4	A2	B1	21	P3.2
Left Mtr PWM	5	A3	B2	20	P3.3
Right Mtr PWM	6	A4	B3	19	P2.2
Arm Raise	7	A5	B4	18	P2.3
Arm Lower	8	A6	B5	17	P4.0
Bucket Raise	9	A7	B6	16	P4.1
Bucket Lower	10	A8	B7	15	P4.2
0v	11	GND	B8	14	P4.3
0v	12	GND	GND	13	0v

Input 2 chip

5V	1	VCCA	VCCB	24	3.3V
0V	2	DIR	VCCB	23	3.3V
ultrasonic trigger	3	A1	OE BAR	22	0V
Compass start	4	A2	B1	21	P3.1
	5	A3	B2	20	P0.5
	6	A4	B3	19	
	7	A5	B4	18	
	8	A6	B5	17	
	9	A7	B6	16	
	10	A8	B7	15	
0v	11	GND	B8	14	
0v	12	GND	GND	13	0v

Output 1 chip

5V	1	VCCA	VCCB	24	3.3V
5V	2	DIR	VCCB	23	3.3V
arm sensor high	3	A1	OE BAR	22	0V
arm sensor low	4	A2	B1	21	P3.4
bucket sensor high	5	A3	B2	20	P3.5
bucket sensor low	6	A4	B3	19	P3.6
infrared beacon	7	A5	B4	18	P3.7
infrared encoder 1	8	A6	B5	17	P3.0
infrared encoder 2	9	A7	B6	16	P0.7
ultrasonic distance	10	A8	B7	15	P0.6
0v	11	GND	B8	14	P2.4
0v	12	GND	GND	13	0v

Output 2 chip

5V	1	VCCA	VCCB	24	3.3V
5V	2	DIR	VCCB	23	3.3V
compass output	3	A1	OE BAR	22	0V
compass serial clock	4	A2	B1	21	P0.4
external push button 1	5	A3	B2	20	P0.2
external push button 2	6	A4	B3	19	P2.0
	7	A5	B4	18	P2.6
	8	A6	B5	17	
	9	A7	B6	16	
	10	A8	B7	15	
0v	11	GND	B8	14	
0v	12	GND	GND	13	0v

Main circuit board to microprocessor connection:

connected to	Microprocessor port			microprocessor port	connected to
3.3v pin on buffer chips	3.3V	2	1	GND	ground pin on buffer chip
available for future use**	P4.6	4	3	P4.7	available for future use**
available for future use**	P4.4	6	5	P4.5	available for future use**
Bucket Raise input 1 pin 15	P4.2	8	7	P4.3	Bucket Lower input 1 pin 14
Arm raise input 1 pin 17	P4.0	10	9	P4.1	Arm Lower input 1 pin 16
n/c	3.3V	12	11	GND	n/c
bucket sensor high output 1 pin 19	P3.6	14	13	P3.7	bucket sensor low output 1 pin 18
arm sensor high output 1 pin 21	P3.4	16	15	P3.5	arm sensor low output 1 pin 20
Left Mtr Dir inpur 1 pin 21	P3.2	18	17	P3.3	Right Mtr Dir input 1 pin 20
IR Beacon pin output 1 pin 17	P3.0	20	19	P3.1	ultrasnd trig input 2 pin 21
n/c	3.3V	22	21	GND	n/c
External Push Button 2 output 2 pin 18	P2.6	24	23	P2.7	available for future use
Ultrasonic distance output 1 pin 14	P2.4	26	25	P2.5	available for future use*
Left mtr PWM input 1 pin 19	P2.2	28	27	P2.3	Right Mtr PWM input 1 pin 18
External Push Button 1 output 2 pin 19	P2.0	30	29	P2.1	available for future use*
n/c	3.3V	32	31	GND	n/c
available for future use	P1.6	34	33	P1.7	available for future use
available for future use	P1.4	36	35	P1.5	available for future use
available for future use	P1.2	38	37	P1.3	available for future use
available for future use	P1.0	40	39	P1.1	available for future use
n/c	3.3V	42	41	GND	n/c
Left Motor Encoder output 1 pin 15	P0.6	44	43	P0.7	Right Motor Encoder output 1 pin 16
Compass output, output 2 pin 21	P0.4	46	45	P0.5	Compass start input 2 pin 20
Compass Serial Clock output 2 pin 20	P0.2	48	47	P0.3	N/C, held for 4 pin SPI interface
available for future use	P0.0	50	49	P0.1	available for future use

* = may have push button or potentiometer connected to port

** = Port 4 is byte addressable only

Motor header connection to main circuit board: (Note: not all XXX pins are keyed, but enough are keyed on each header to provide only one way to connect the header)

pin		Connected To
1	XXX	
2		Available for future use
3		Available for future use
4		Available for future use
5		Low Arm Sensor
6		Bucket Sensor Ground
7		High Bucket Sensor
8	XXX	
9		Low Arm Sensor
10		Arm Sensor Ground
11		High Arm Sensor
12	XXX	
13		Arm Motor +
14	XXX	
15		Arm Motor -
16	XXX	
17		Bucket Motor +
18	XXX	
19		Bucket Motor -
20	XXX	
21		Right Motor +
22		Right Motor -
23	XXX	
24		Left Motor +
25		Left Motor -

Sensor header connection to main circuit board:

pin		Connected To
1	XXX	
2		Available for future use
3		Available for future use
4		Available for future use
5		5V
6		Microprocessor Ultrasonic Distance
7		Microprocessor Ultrasonic Trigger
8		Ground
9	XXX	
10		Transistor Collector of IR Reflector
11		Transistor Emitter of IR Reflector
12		Diode Cathode of IR Reflector
13		Diode Anode of IR Reflector
14	XXX	
15		Transistor Collector of IR Reflector
16		Transistor Emitter of IR Reflector
17		Diode Cathode of IR Reflector
18		Diode Anode of IR Reflector
19	XXX	
20		Emitter of IR transmitter
21		Collector of IR Transmitter
22		Base of IR Transistor

Notes: pin 5 through 8 connects to ultrasonic sensor, pin 10 through 13 connects to encoder 1, and pin 15 through 18 connects to encoder 2.

Appendix B: Final project code

testinit.c

This code is the initialization code generated by the config2 tool.

Code:

```
/////////////////////////////////////
//  Generated Initialization File  //
/////////////////////////////////////

#include "C8051F340.h"

// Peripheral specific initialization functions,
// Called from the Init_Device() function
void Timer_Init()
{
	TMR2CN    = 0x00;

    TMR2RL   = 0x00;

	ET2           = 1;
	TR2           = 1;
}

//new
void PCA_Init()
{
    PCA0CN    = 0x40;
    PCA0MD    &= ~0x40;
    PCA0MD    = 0x03;
    PCA0CPM0  = 0xC2;
    PCA0CPM1  = 0xC2;
    PCA0CPM2  = 0x31;

    PCA0CPM3  = 0x10;
}

void SPI_Init()
{
    SPI0CFG   = 0x30;
    SPI0CN    = 0x01;
}

void Port_IO_Init()
{
    P0MDOUT   = 0x3F;
    P1MDOUT   = 0xFF;
    P2MDOUT   = 0xDE;
    P3MDOUT   = 0xFF;
    P4MDOUT   = 0xFF;
    P0SKIP    = 0xE3;
    P1SKIP    = 0xFF;
    P2SKIP    = 0x21;
    P3SKIP    = 0xFF;
    XBR0      = 0x0A;
    XBR1      = 0x45;
    XBR0      = 0x0A;
    XBR1      = 0x45;
}
```

```
void ADC_Init()
{
    AMX0P     = 0x04;
    AMX0N     = 0x1F;
    ADC0CN    = 0x82;
}

void Oscillator_Init()
{
    int i = 0;
    OSCICN    = 0x83;
    CLKMUL    = 0x80;
    for (i = 0; i < 20; i++);    // Wait 5us for initialization
    CLKMUL    |= 0xC0;
    while ((CLKMUL & 0x20) == 0);
    CLKSEL    = 0x03;
}

void Interrupts_Init()
{
    IP        = 0x05;
    EIE1      = 0x10;
    EIP1      = 0x10;
    IT01CF    = 0x76;
    IE        = 0xED;
}

// Initialization function for device,
// Call Init_Device() from your main program
void Init_Device(void)
{
    PCA_Init();
    Timer_Init();
    SPI_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}
```

testmain.c

This is the main code for the project.

```
#include "C8051F340.h"

#include "testinit.c"
#include <math.h>

unsigned int timedelay=60;

//unsigned int bob;                   //random variable for testing
//drive wheel diameter = 1.7 inches
unsigned short int t2of = 0x00;       //number of times timer 2 overflowed
               //incremented in timer 2 interrupt, used for timing ultrasonic
               //input signal.

unsigned int ledperiod=0;
unsigned long int ultrastart=0; //time offset for timer 2 when ultrasonic timing starts
unsigned long int timespan=0;   //duration of ultrasonic pulse from whatever speed timer 2 is

xdata float pi = 3.1416;
xdata float degtorad= 3.1416/180;

xdata unsigned int dist=0;                    //last ultrasonic distance reading

xdata short int desired_left_pwm=0;   //desired pwm percentage for left motor, can be + or -
xdata short int desired_right_pwm=0;  //desired pwm percentage for right motor, can be + or -
```

```
//desired pwm's to be used by encoder interrupts to maintain proper pwm speeds

xdata short int linear_pwm = 0;                          //"linear" pwm sets both left and right pwm
to the same value
xdata short int rotational_pwm=0;                   //"rotational" pwm is added to left pwm, and
subtracted from right pwm
//if rotational_pwm is positive, vehicle should spin somewhat clockwise
//if rotational_pwm is negative, vehicle should spin somewhat counterclockwise

unsigned short int flags=0x00; //initialize flag register
//bit 0 = distance update flag
//flags&=0xFE; resets distance flag
//flags|=0x01; sets distance flag
//flags&0x01 masks off all other flags

//bit 1 = ~50 ms flag
//flags&=0xFD; resets 50 ms flag
//flags|=0x02; sets 50 ms flag
//flags&0x02 masks off all other flags

//bit 2 = 1 ms flag
//flags&=0xFB; resets 1 ms flag
//flags|=0x04; sets 1 ms flag
//flags&0x04 masks off all other flags

//bit 3 = new compass reading flag
//flags&=0xF7; resets compass reading flag
//flags|=0x08; sets compass reading flag
//flags&0x08 masks off all other flags

xdata unsigned int t2count = ~0x0000; //number of increments of timer 2 minus 1
unsigned short int t2overflowcount=0; //number of times timer 2 overflowed, intend to use in
//velocity control routine
xdata unsigned int pcaoffset=0xD11F;  //starting value of PCA register to genetate 1khz PWM
xdata unsigned int pcacount=~0xD11F;
xdata unsigned long int tempvariable=0;

// < port pin used and description of signal>                       <c-code for pin>
//ASSUMING PORT 3 PIN 0 IS INFRARED TRANSISTOR SIGNAL              P3_0
__sbit __at (0xB0) IR_beacon;
//assuming port 3 pin 1 is ultrasonic trigger                       P3_1

//assuming port 3 pin 2 is left motor DIRECTION                     P3_2
//1 = forward, 0 = reverse, check
//assuming port 3 pin 3 is right motor DIRECTION                   P3_3
//1 = forward, 0 = reverse, check

__sbit __at (0xB2) left_mtr_forward;
__sbit __at (0xB3) right_mtr_forward;

//assuming port 2 pin 2 is left motor pwm                           P2_2
//assuming port 2 pin 3 is right motor pwm                          P2_3
//assuming port 2 pin 4 is ultrasonic timing pin                   P2_4

//-------------------------------------------------------------------
//                         Steve's variables
//-------------------------------------------------------------------

//unsigned int pwm_left_count = 0;
//unsigned int pwm_right_count = 0;
short int pwm_left_out = 0;
short int pwm_right_out = 0;
int t2difcount_left = 0;
int t2difcount_right = 0;
unsigned int t2overflowcount_left = 0;
unsigned int t2overflowcount_right = 0;

xdata unsigned int truckdirection = 0;
xdata unsigned int bindirection = 0;

//-------------------------------------------------------------------
```

```
//
//----------------------------------------------------------------------

//simple control variables
float contrlvar=0.0;
//implememnt simple controller to try to balance the left encoder pulses with right
//in desired ratio
//positive implies more left encoder pulses than right (in a desired ratio)
//negative implies more right encoder pulses than left (in a desired ratio)

//Ex: left wheel to drive at 60% speed, right at 80% speed
//left increment is smallest, set it to 1.0.  Right increment = right speed/left speed = 1.333...
float leftinc=0.0;
float rightinc=0.0;

unsigned long int leftencoderperiod =0;             //time between interrupts, in microseconds
unsigned long int rightencoderperiod = 0;    //time between interrupts, in microseconds
unsigned long int leftencoderoffset=0;              //time from beginning to past interrupt, in
microseconds
unsigned long int rightencoderoffset=0;             //time from beginning to past interrupt, in
microseconds

//compass variables
unsigned short int bytenumber=0;                    //record which byte is currently being recieved
unsigned short int firstbyte=0;                             //first recieved byte
unsigned short int secondbyte=0;                    //second recieved byte
unsigned int vehicledirection=0;                    //recieved orientation from compass

//functions

void compass_enable(void)
{
        P0_5=1;         //prepare compass to operate
        SPIF =0;        //clear all interrupt flags
        WCOL =0;
        MODF =0;
        RXOVRN =0;
        flags&=0xF7; //resets compass reading flag

        bytenumber=0;  //reset internal variables
        firstbyte=0;
        secondbyte=0;
        vehicledirection=0;

        ESPI0=1;        //enable SPI0 interrupt

        P0_5=0;         //activate the compass
}

//*
void compass_disable(void)
{
        P0_5=1;         //deactivate compass

        ESPI0=0;        //disable SPI0 interrupt

        SPIF =0;        //clear all interrupt flags
        WCOL =0;
        MODF =0;
        RXOVRN =0;

        bytenumber=0;  //reset internal variables
        firstbyte=0;
        secondbyte=0;
        vehicledirection=0;
}
//*/

void encoder_reset (void){
        int tempcal;
```

```
	//disable rotary encoder interrupts during update
	EX0=0;
	EX1=0;

	//reset timer 2 overflow count for new controller
	t2overflowcount=0;
	leftencoderoffset=(unsigned long int) (TMR2/4);
	rightencoderoffset=leftencoderoffset;

	//correct to unsigned magnitudes, and update pwm generation
	//direction is controlled in the desired speed routines
	//(do not want controller change drive direction to prevent
	//oscillations)
	if(desired_left_pwm<0)
		pwm_left_out = -desired_left_pwm;
	else
		pwm_left_out = desired_left_pwm;

	if(desired_right_pwm<0)
		pwm_right_out=-desired_right_pwm;
	else
		pwm_right_out=desired_right_pwm;

	tempcal=100-pwm_left_out;
	tempcal=(int)(tempcal*(pcacount/100));

	tempcal = tempcal + pcaoffset + 1;
	if(tempcal<pcaoffset)
		tempcal=0xffff;

	PCA0CP0=tempcal;

	tempcal=100-pwm_right_out;
	tempcal=(int)(tempcal*(pcacount/100));
	tempcal = tempcal + pcaoffset + 1;
	if(tempcal<pcaoffset)
		tempcal=0xffff;
	PCA0CP1 = tempcal;

	//possibly old controller variable reset
	t2difcount_left = t2count;
	t2overflowcount_left = t2overflowcount;

	//proportional controller variable initialization
	contrlvar=0.0;

	if(pwm_left_out==0||pwm_right_out==0)
	{
		//controller has no effect, just drive at desired pwm
		//cannot maintain ratio of periods for the encoders when one or more motors is stopped
		//for motors
		leftinc=0.0;
		rightinc=0.0;
	}
	else if(pwm_left_out<=pwm_right_out)
	{
		leftinc=1.0;
		rightinc=((float)pwm_right_out)/((float)pwm_left_out);
	}
	else
	{
		rightinc=1.0;
		leftinc=((float)pwm_left_out)/((float)pwm_right_out);
	}
	leftencoderperiod =0;
	rightencoderperiod = 0;

	//re-enable encoders
	EX0=1;
	EX1=1;
}
```

```
//updates linear pwm velocity.  Intend to bound between 50% and 80%
//as this range is decently linear.  Will allow rotational pwm to be +/- 10%,
//and controller will allow to adjust it another +/- 20% to correct for desired
//linear and rotational speed
void newlinearpwm(int newpwmpercent)
{
	//if pwm percent is zero, set it to zero
	//if it is higher or lower, bound it between 50 and 80 % pwm
	int rot_corr=1;
	if(newpwmpercent<0)
		rot_corr=-1;

	if(newpwmpercent==0)
	{
		linear_pwm=0;
		desired_left_pwm=rotational_pwm;
		desired_right_pwm=-rotational_pwm;
		if(rotational_pwm>=0)
		{
			left_mtr_forward=1;
			right_mtr_forward=0;
		}
		else
		{
			left_mtr_forward=0;
			right_mtr_forward=1;
		}
	}
	else
	{
		if(rotational_pwm>10)  //correct for if vehicle was rotating in place
			rotational_pwm=10;
		if(rotational_pwm<-10)
			rotational_pwm=-10;
		//resume linear velocity decision loop
		if(newpwmpercent>0)
		{
			//bound between +50 and +80%
			if(newpwmpercent<50)
				newpwmpercent=50;
			if(newpwmpercent>80)
				newpwmpercent=80;
			linear_pwm=newpwmpercent;
			desired_left_pwm = newpwmpercent+rot_corr*rotational_pwm;   //assume this is 
within bounds of +40 to +90%
			desired_right_pwm = newpwmpercent-rot_corr*rotational_pwm;
			left_mtr_forward=1;		//set motors to drive forward
			right_mtr_forward=1;
		}
		else
		{
			if(newpwmpercent>-50)
				newpwmpercent=-50;
			if(newpwmpercent<-80)
				newpwmpercent=-80;

			linear_pwm=newpwmpercent;
			desired_left_pwm = newpwmpercent+rot_corr*rotational_pwm;   //assume this is 
within bounds of +40 to +90%
			desired_right_pwm = newpwmpercent-rot_corr*rotational_pwm;
			left_mtr_forward=0;		//set motors to drive backward
			right_mtr_forward=0;
		}
	}
	encoder_reset();
}

void newrotationalpwm(int newpwmpercent)
{
	//correction flag for if motors are going backward
```

```
	//when the motors are going backward, the vehicle will rotate
	//counter-clockwise if pwm is just directly added to it
	int rot_corr=1;
	if(linear_pwm<0)
		rot_corr=-1;

	if(newpwmpercent==0)
	{
		rotational_pwm=0;
		desired_left_pwm=linear_pwm;
		desired_right_pwm=linear_pwm;
	}
	else if(linear_pwm==0)//if vehicle is not driving forward
	{
		if(newpwmpercent>0)
		{//rotate clockwise between +40 and +90 % pwm
			if(newpwmpercent>90)
				newpwmpercent=90;
			if(newpwmpercent<40)
				newpwmpercent=40;
			rotational_pwm=newpwmpercent;
			desired_left_pwm=newpwmpercent;
			desired_right_pwm=-newpwmpercent;
			left_mtr_forward=1;
			right_mtr_forward=0;
		}
		else
		{//rotate counter-clockwise between -40 and -90% pwm
			if(newpwmpercent<-90)
				newpwmpercent=-90;
			if(newpwmpercent>-40)
				newpwmpercent=-40;
			rotational_pwm=newpwmpercent;
			desired_left_pwm=newpwmpercent;
			desired_right_pwm=-newpwmpercent;
			left_mtr_forward=0;
			right_mtr_forward=1;
		}
	}
	else
	{
		//bound between +/- 10% pwm
		if(newpwmpercent>10)
			newpwmpercent=10;
		if(newpwmpercent<-10)
			newpwmpercent=-10;
		rotational_pwm=newpwmpercent;

		desired_left_pwm=linear_pwm+rot_corr*newpwmpercent;
		desired_right_pwm=linear_pwm-rot_corr*newpwmpercent;
		//motor directions should already have been set in linear
		//pwm routine, will not change them here.
	}
	encoder_reset();
}

//REMEMBER: This is occurring inside an interrupt, may need to modify if it gets too long.
void distupdate(int newdist)
{
	//removed distance offset for bucket distance and whatnot, since some of the times it is there,
	//and some of the times it is not (arm up => no threshold, arm down => threshold)
	//assume programmer will compensate on his or her own for this at this time.
	//note: distance = centimeters

	//set flag for new distance update
	flags|=0x01;
	//filtering

	//no filtering at this time... sensor does not appear to have much noise

	//update output
```

```
        dist=newdist;
}

//wait time function, not tested for times longer than 12-13 seconds
//uses 50 ms flag
void waitsec(unsigned int seconds)
{
        unsigned int exitcount=seconds*20;
        unsigned int timeout=0;
        while(1)
        {
                if(flags&0x02>0)
                {
                        flags&=0xFD;            //reset 50 ms flag
                        timeout++;
                        if(timeout>=exitcount) //20 pulses per second * 10 seconds = 200 pulses
                        {
                                break;
                        }
                }
        }
}

void waitmsec(unsigned long int milliseconds)
{

        while(milliseconds>0)
        {
                if(flags&0x04>0)
                {
                        flags&=0xFB; //resets 1 ms flag
                        milliseconds--;
                }
        }
}

//try to use a fast stopping routine to reduce rotational and linear overshoot
//reverse direction on drive motors, wait for a few subsequent pwm signals to help
//remove vehicle inertia, then stop and reset all variables
//
//also, reset arm and bucket motor signal to 0000 to stop both.
void vehiclestop(void)
{
//plan: reverse directions on drive motors for a few pwm updates, then stop them and return
        unsigned int delaytime=10;

        left_mtr_forward=!left_mtr_forward;             //reverse direction
        right_mtr_forward=!right_mtr_forward;

        P4=P4&0xF0;                     //stop arm and bucket motors
        flags&=0xFB;            //reset 1 ms flag
        //note: should be replaced with call to waitmsec(delaytime);
        while(1)
        {       //wait for a few cycles of PWM signal before stopping
                //leave magnitude the same, since it makes the most sense
                //(if going 100% forward, go 100% backward for a period of time
                //versus going backward 0% for the same period of time
                if(flags&0x04)          //check 1 ms flag
                {
                        flags&=0xFB;    //reset 1 ms flag
                        delaytime--;
                        if(delaytime==0)
                        {
                                break;
                        }
                }
        }

        PCA0CP0=0xFFFF;         //stop motors
        PCA0CP1=0xFFFF;
```

```
        desired_left_pwm=0;    //set all internal velocity states to zero
        desired_right_pwm=0;
        linear_pwm=0;
        rotational_pwm=0;
        encoder_reset();
}

//desire function to control arm and bucket motors
//process:
//stop drive motors => drive motors produce enough noise to cause false triggers of
//the sensor.  There may also be issues of noise if arm and bucket motors running
//simultaneously as well.  Hardware change: Add 4 pullup resistors to vehicle sensors
//current software correction: wait for signal to be low for ~1 second
//possible software correction: check that the majority of the past second has been high,
//or that the pulses caused by noise are no longer than a certain time
//inputs: 2 integers, bucket_raise and arm_raise
//for bucket_raise: if it is 0, nothing happens, if it is < 0, the bucket is lowered
//and if it is > 0, the bucket is raised.
//for arm_raise: if it is 0, nothing happens, if it is < 0, the arm is lowered
//and if it is > 0, the arm is raised
//
//NOTE: P4 is only byte addressable, and motor control signals use port 4.
//need to write the whole byte to the port in order for this to work properly.
//will take port values, and use logic operations to get desired signals
//(to preserve port 4 high byte information)
//ADDITIONAL NOTE: Bucket grinds against drive tracks if the bucket AND arm are
//down at the same time.  Code prevents this operation in one function call
void bucketraise_armraise(short int bucket_raise, short int arm_raise)
{
        unsigned int specialsensorcount=0;
        unsigned short int tempp4=0;
        //could save linear and rotational pwm and restore after process complete...


        if((bucket_raise<0)&&(arm_raise<0))
        {
                arm_raise=0;   //disable arm motor reaction
        }
        vehiclestop();

        if(bucket_raise>0)
        {//raise bucket until sensor is hit
                tempp4=P4;
                tempp4|=0x04;  //set bucket raise
                tempp4&=0xf4;  //clear bucket lower and arm signals
                P4=tempp4;             //reset all at once
                while(1)
                {
                        while(P3_6==1)
                        {;}
                        specialsensorcount=1;
                        while(P3_6==0)
                        {
                                if(flags&0x02>0)
                                {
                                        flags&=0xFD;
                                        specialsensorcount++;
                                        if(specialsensorcount>=20)
                                        {
                                                break;
                                        }
                                }

                        }
                        if(specialsensorcount>=20)
                                break;
                }

        }

        if(bucket_raise<0)
```

```
	{//lower bucket until sensor is hit
		tempp4=P4;
		tempp4|=0x08;  //set bucket lower
		tempp4&=0xf8;  //clear bucket raise and arm signals
		P4=tempp4;		//reset all at once
		while(1)
		{
			while(P3_7==1) //wait for sensor to go low
			{;}
			specialsensorcount=1;  //reset timing variable
			while(P3_7==0) //time how long sensor is low
			{
				if(flags&0x02>0)	//use 50 ms flag for longer timing
				{
					flags&=0xFD;
					specialsensorcount++;
					if(specialsensorcount>=20)	//check for longer than 1 second of
0's
					{
						break;
					}
				}

			}
			if(specialsensorcount>=20)
				break;
		}

	}

	if(arm_raise>0)
	{//raise arm until sensor is hit
		tempp4=P4;
		tempp4|=0x01;  //set arm raise
		tempp4&=0xf1;  //clear arm lower and bucket signals
		P4=tempp4;		//reset all at once
		while(1)
		{
			while(P3_4==1)
			{;}
			specialsensorcount=1;
			while(P3_4==0)		//noisy signal, wait for approx 2 milliseconds of zero
readings to stop
			{
				if(flags&0x02>0)
				{
					flags&=0xFD;

					specialsensorcount++;
					if(specialsensorcount>=20)
					{
						break;
					}
				}

			}
			if(specialsensorcount>=20)
				break;
		}
	}

	if(arm_raise<0)
	{//lower arm until sensor is hit
		tempp4=P4;
		tempp4|=0x02;  //set arm lower
		tempp4&=0xf2;  //clear arm raise and bucket signals
		P4=tempp4;		//reset all at once
		while(1)
		{
			while(P3_5==1)
			{;}
```

```
                        specialsensorcount=1;
                        while(P3_5==0)
                        {
                                if(flags&0x02>0)
                                {
                                        flags&=0xFD;
                                        specialsensorcount++;
                                        if(specialsensorcount>=20)
                                        {
                                                break;
                                        }
                                }

                        }
                        if(specialsensorcount>=20)
                                break;
                }
        }

}

short int drivetobeacon(unsigned short int desired_dist, unsigned short int beacon_off_time, unsigned
short int ir_track)
{
        unsigned int curdist=0;
        unsigned int distdiff=0;
        unsigned short int pwmpercent=0;
        unsigned int irbeacontimeout=0;
        flags&=0xFE;            //reset distance flag

        while(1)//drive to wall
        {
                while(flags&0x01==0)            //wait for new distance reading
                {;}
                flags&=0xFE;                            //reset flag
                curdist=dist;                           //read dist

                if(curdist>=desired_dist) //vehicle is far from object
                {
                        //start of vehicle too far away

                        distdiff=curdist-desired_dist;
                        if(distdiff>3)          //allow tolerance of 3 cm from target
                        {
                                //try to have velocity proportional to distance
                                //with pwm between 50 and 80% pwm with distdiff between 4 and 50
                                //limit to 50% to hopefully minimize non-linearities...

                                if(distdiff>50)
                                        distdiff=50;
                                pwmpercent=distdiff*2;
                                if(pwmpercent<50)
                                        pwmpercent=50;
                                if(pwmpercent>80)
                                        pwmpercent=80;

                                newlinearpwm(pwmpercent);
                        }
                        else
                        {
                                break;
                        }
                        //end of vehicle too far
                }

                else
                {
                        //start of vehicle too close

                        distdiff=desired_dist-curdist;
                        if(distdiff>3)          //allow tolerance of 3 cm from target
```

```
			{
				//try to have velocity proportional to distance
				//with pwm between 50 and 100% pwm with distdiff between 2 and 25
				//limit to 50% to hopefully minimize non-linearities...

				if(distdiff>50)
					distdiff=50;
				pwmpercent=distdiff*2;
				if(pwmpercent<50)
					pwmpercent=50;
				if(pwmpercent>80)
					pwmpercent=80;
				newlinearpwm(-1*(int)pwmpercent); //tell vehicle to drive backwards
			}
			else
			{
				break;
			}
			//end of vehicle too close

		}

		//verify IR transistor is still visible
		if(ir_track==1)
		{
			if(IR_beacon==0)
			{
				//reset parameters
				irbeacontimeout=0;
				newrotationalpwm(0);
			}
			else
			{
				//turn slightly to look for IR beacon, counterclockwise first, then
clockwise
				//process idea: turn counterclockwise (opposite of the direction to seek
IR led) first
				//for some time (50 ms maybe?), then turn clockwise for twice the time of
turning counterclockwise
				//(half to get back to zero, half to look to the right of the vehicle).
If IR beacon is not found,
				//stop and call the IR location routine again...
				irbeacontimeout++;
				if(irbeacontimeout==beacon_off_time)
				{
					flags&=0xFB;			//reset 1 ms flag
					newrotationalpwm(-10);
				}
				else if(flags&0x04>0)	//check for 1 ms flag
				{
					flags&=0xFB;		//reset flag
					irbeacontimeout++;
					if(irbeacontimeout>(70+beacon_off_time))	//end time for turning
counterclockwise
						newrotationalpwm(10);
					if(irbeacontimeout>(270+beacon_off_time))	//end time for turning
clockwise
					{	//note: time is at least 2*counterclockwise time +
clockwise time
						//allowing more than 2*counterclockwise time due to
momentum
						//give up "small" looking, and start process again
						vehiclestop();
						return(-1);
					}
				}
			}
		}

	}
```

```
        return(0);

}

void degreeheading(int degreeface, short int tolerance)
{
        xdata int absoluteheading=0;
        xdata int deglimit=0;
        xdata int deltaangle=0;
        xdata short int directioncorrection=1;

        vehiclestop();
        waitmsec(200); //let magnetic interference die down
        flags&=0xF7; //resets compass reading flag
        //look for two compass readings within +/- 1 degree (unchecked for wrap around)
        while(1)
        {
                while(flags&0x08==0)
                {;}     //wait for new compass reading
                absoluteheading=vehicledirection;
                flags&=0xF7; //resets compass reading flag
                while(flags&0x08==0)
                {;}     //wait for new compass reading
                if(absoluteheading>=(int)(vehicledirection)-1 & absoluteheading<=vehicledirection+1)
                        break;
        }
        flags&=0xF7; //resets compass reading flag
        //current compass value is in absoluteheading

        //check cases:
        //degreeface = 271, heading = 90, rotate counterclockwise
        //      deltaangle = 181, subtract 360 degrees, delta angle=-179 degrees.  rotate counterclockwise
        //degreeface=269, heading = 90, rotate clockwise
        //      deltaangle = 179, leave as is, rotate clockwise
        //degreeface=90, heading=271, rotate clockwise
        //      deltaangle = -181, add 360, delta angle = 179, rotate clockwise
        //degreeface=90, heading = 269; rotate counterclockwise
        //      deltaangle = -179, leave as is, rotate counterclockwise
        //degreeface=359, heading = 1, rotate counterclockwise
        //      deltaangle = 358, subtract 360, deltaangle=-2, rotate counterclockwise
        //degreeface = 1, heading = 359, rotate clockwise
        //      deltaangle = -357, add 360, deltaangle=3, rotate clockwise
        //test cases rotate correct direction

        deltaangle=degreeface-absoluteheading;
        if(deltaangle>180)
                deltaangle-=360;
        if(deltaangle<-180)
                deltaangle+=360;
        if(deltaangle>tolerance)
        {
                directioncorrection=1;
        }
        else if(-deltaangle>tolerance)
        {
                directioncorrection=-1;
                deltaangle=-deltaangle;
        }
        else {directioncorrection=0;}

        while(1)
        {
                newrotationalpwm(30*directioncorrection);
                if(deltaangle<=15)
                {
                        waitmsec(50);
                        vehiclestop();
                        waitmsec(200);
                        flags&=0xF7; //resets compass reading flag
                        while(flags&0x08==0)
                        {;}     //wait for new compass reading
```

```
				flags&=0xF7; //resets compass reading flag
				while(flags&0x08==0)
				{;}	//wait for new compass reading
				flags&=0xF7; //resets compass reading flag
				absoluteheading=vehicledirection;

				deltaangle=degreeface-absoluteheading;
				if(deltaangle>180)
					deltaangle-=360;
				if(deltaangle<-180)
					deltaangle+=360;
				if(deltaangle>tolerance)
				{
					directioncorrection=1;
				}
				else if(deltaangle<-tolerance)
				{
					directioncorrection=-1;
					deltaangle=-deltaangle;
				}
				else
				{
					break;
				}
			}
			else
			{
				while(1)
				{
					if(rotational_pwm==30*directioncorrection)
					{}
					else
					{
						newrotationalpwm(30*directioncorrection);
					}

					absoluteheading=vehicledirection;

					deltaangle=degreeface-absoluteheading;
					if(deltaangle>180)
						deltaangle-=360;
					if(deltaangle<-180)
						deltaangle+=360;
					if(deltaangle>15)
					{
						directioncorrection=1;
					}
					else if(-deltaangle>15)
					{
						directioncorrection=-1;
						deltaangle=-deltaangle;
					}
					else //leave
					{
						if(deltaangle<0)
							deltaangle=-deltaangle;
						vehiclestop();
						break;
					}
				}
			}
		}
}

//function to look for infrared beacon.  Stop the vehicle, rotate
//in a circle, and try to locate the desired IR source
//targetloc ==0, find next clockwise infrared beacon
//targetloc > 0, find load material
//		load material beacon flashes at XX Hz
//targetloc < 0, find truck
//		truck beacon flashes at YY Hz
```

```
//note: infrared circuitry does not appear to work much faster than ~10 Hz
//may try to come up with better detection circuitry
//
//returns 0 if desired target found
// ""     -1 if target could not be found
//
//assuming load blinks at 10 Hz, and truck blinks at 5 Hz initially
//and they have high enough duty cycles to locate
//allow period to vary +/- 40 ms from desired period, to handle inconsistencies in timing, sensors,
//IR led timing signals, etc.
short int locateIRbeacon(short int targetloc)
{
	xdata unsigned int compasstolerance=20;
	xdata int highthresh=0;
	xdata int lowthresh=0;

	unsigned int timeout=0;			//time out variable used for looking for load, and for non-
changing IR light
	//set timeout for the non-changing IR light before going through the target cases if needed to
change
	unsigned short int remainingtargets=3;		//number of infrared targets to check
	unsigned short int waitcount=120;	//look for ~6 seconds (number of 50 ms flags to clear)
	unsigned int loadperiod=100;	//period of load IR beacon, in ms
	unsigned int truckperiod=200;	//period of truck IR beacon, in ms
	unsigned int countvar=0;	//counter variable for target identification

	if(targetloc>0)
	{
		highthresh=bindirection+compasstolerance;
		lowthresh=bindirection-compasstolerance;
		if(highthresh>360)
			highthresh=highthresh-360;
		if(lowthresh<0)
			lowthresh+=360;
	}
	else if(targetloc<0)
	{
		highthresh=truckdirection+compasstolerance;
		lowthresh=truckdirection-compasstolerance;
		if(highthresh>360)
			highthresh=highthresh-360;
		if(lowthresh<0)
			lowthresh+=360;
	}

	while(1)
	{//desired target location loop
		vehiclestop();			//stop vehicle
		//check if the number of IR signals found has exceeded allowed limit
		if(remainingtargets==0)
		{	//too many IR targets found while desired target not found, return error
			return(-1);
		}
		newrotationalpwm(30);	//rotate clockwise slowly
		timeout=0;				//reset timeout variable
		countvar=0;				//reset count variable
		ledperiod=0;

		while(1)
		{//IR beacon locate loop
			if(IR_beacon==0)
			{
				break;
			}
			else
			{
				if(flags&0x02>0)
				{
					flags&=0xFD;		//reset 50 ms flag
					timeout++;
					if(timeout>=waitcount)
```

```
                        {
                                break;
                        }
                }
        }
}//end of IR beacon locate loop

vehiclestop(); //stop vehicle

//check for timing out of previous loop
if((timeout>=waitcount))
{
        //could not locate any target, return error
        //could replace in future with semi-intelligent drive routine
        return(-1);
}

remainingtargets--;             //past section where code can time out
                                        //decrease number of false IR targets
allowed

if(dist>45)
        drivetobeacon(40,30,0);

flags&=0xF7; //resets compass reading flag
waitmsec(250);
if(vehicledirection>=lowthresh|vehicledirection<=highthresh)
{       //vehicle may be within bounds
        if(highthresh>lowthresh)
        {
                if(vehicledirection>=lowthresh&vehicledirection<=highthresh)
                {
                        //vehicle is in the right location, do nothing
                }
                else    //vehicle out of bounds, restart loop
                {
                        newrotationalpwm(30);
                        waitmsec(250);
                        vehiclestop();
                        continue;
                }
        }
}
else
{
        newrotationalpwm(30);
        waitmsec(250);
        vehiclestop();
        continue;
}

countvar=1000; //highest period IR signal + tolerance
timeout=1000;  //highest period IR signal + tolerance

//IR_beacon should be 0 at this point,
//check to see which case function is looking for
if(targetloc==0)
{       //looking for any IR target, and one was found, so exit loop
        return(0);
}
else
{       //wait for the start of a new cycle, and eliminate DC IR light
        while(IR_beacon==0)
        {
                if(flags&0x04>0)                //check 1 ms flag
                {
                        flags&=0xFB;    //reset 1 ms flag
                        timeout--;
                        if(timeout==0)
                                break;
                }
```

```
                        }
                        //IR_beacon should be 1 now, or timed out from dc source
                        if(timeout==0)
                        {       //no change in sensor in desired time, must be (virtually) constant IR
source
                                //like the sun, or lights, or etc... rotate a little bit then redo loop
                                newrotationalpwm(30);
                                waitmsec(250);
                                vehiclestop();
                                continue;                               //restart loop
                        }

                        //measure period of one transition (consists of high and low cycle)
                        ledperiod=0;
                        while(IR_beacon==1)
                        {
                                if(flags&0x04)          //check 1 ms flag
                                {
                                        flags&=0xFB;    //reset 1 ms flag
                                        ledperiod++;            //increase ms count time
                                }
                                if(ledperiod>countvar) //assume no frequency below .5 Hz...
                                        break;
                        }
                        while(IR_beacon==0)
                        {
                                if(flags&0x04)          //check 1 ms flag
                                {
                                        flags&=0xFB;    //reset 1 ms flag
                                        ledperiod++;            //increase ms count time
                                }
                                if(ledperiod>countvar) //assume no frequency below .5 Hz...
                                        break;
                        }

                        if(targetloc>0)
                        {//looking for load material
                                if((ledperiod>(loadperiod+40))|(ledperiod<(loadperiod-40)))
                                {//target is not desired target, repeat loop again
                                        newrotationalpwm(30);
                                        waitmsec(250);
                                        vehiclestop();
                                        continue;
                                }
                                break;
                        }
                        else
                        {
                                //looking for truck
                                if((ledperiod>(truckperiod+40))|(ledperiod<(truckperiod-40)))
                                {//target is not desired target, repeat loop again
                                        newrotationalpwm(30);
                                        waitmsec(250);
                                        vehiclestop();
                                        continue;
                                }
                                break;
                        }
                }
        }
        return(0);
}

void main (void) {
        unsigned short int distancecount=0;
        unsigned short int prevdistance=0;
        unsigned short int pwmpercent=0;
        unsigned long int pcavalues=0;

        short int returnval=0;
        short int target=1;             //start looking for load
```

```
	unsigned int loadpwmtest=0;

	P4=0x01;	//set vehicle to raise arm
	PCA0CP1=0xFFFF;	//stop vehicle pwm
	PCA0CP0=0xFFFF;

	Init_Device();
	TMR2RL   = ~ t2count;  //overwrite timer 2 reload so config cant screw it up
	PCA0MD &= ~0x40;		//overwrite pca mode
	IT0=1;  //configure external interrupts to edge triggered
	IT1=1;  //configure external interrupts to edge triggered
	EX0=1;
	EX1=1;
	vehiclestop();

	bucketraise_armraise(1,-1); //raise bucket then lower arm
	P4&=0xF1;		//set arm raise manually for short duration
	P4|=0x01;
	waitsec(2);		//wait a short time, so arm is off ground
	P4&=0xF0;		//stop arm motor

	compass_enable();

	while(P2_0==1)
	{;}
	waitsec(1);
	while(1)
	{
		while(flags&0x08==0)
		{;}	//wait for new compass reading
		bindirection=vehicledirection;
		flags&=0xF7; //resets compass reading flag
		while(flags&0x08==0)
		{;}	//wait for new compass reading
		if(bindirection>=(int)(vehicledirection)-1 & bindirection<=vehicledirection+1)
			break;
	}

	flags&=0xF7; //resets compass reading flag
	waitsec(2);
	while(P2_0==1)
	{;}
	waitsec(1);
	while(1)
	{
		while(flags&0x08==0)
		{;}	//wait for new compass reading
		truckdirection=vehicledirection;
		flags&=0xF7; //resets compass reading flag
		while(flags&0x08==0)
		{;}	//wait for new compass reading
		if(truckdirection>=(int)(vehicledirection)-1 & truckdirection<=vehicledirection+1)
			break;
	}
	truckdirection = vehicledirection;
	flags&=0xF7; //resets compass reading flag
	waitsec(1);

	//main loop
	while(1)
	{
//*
		//begin IR location routine
		if(target==1)
			degreeheading(bindirection,4);

		if(target==-1)
			degreeheading(truckdirection,4);

		returnval=locateIRbeacon(target);
```

```
		if(returnval==-1)
		{
			waitsec(4);	//wait and repeat main loop
			continue;
		}

		if(target==1)
			degreeheading(bindirection,4);

		if(target==-1)
			degreeheading(truckdirection,4);

		//end of IR location
//*/
		//begin drive to object routine
		//drive to object

		returnval=drivetobeacon(40,50,1);
		if(returnval==-1)
		{
			waitsec(4);
			continue;
		}

		if(target==1)
		{//load material routine

			bucketraise_armraise(1,-1); //raise bucket then lower arm

//-------------------------------------------------
//          Test load function
//-------------------------------------------------
			distancecount=0;
			prevdistance=0;
			newlinearpwm(50);
			waitmsec(400);
			prevdistance=dist;
//bit 0 = distance update flag
//flags&=0xFE; resets distance flag
//flags|=0x01; sets distance flag
//flags&0x01 masks off all other flags
			flags&=0xFE; //resets distance flag
			loadpwmtest = rightencoderperiod;
			loadpwmtest += leftencoderperiod;

			while (1){
				if (1.05 * loadpwmtest < rightencoderperiod + leftencoderperiod)
					break;
				if(flags&0x01==1) //new distance reading
				{
					flags&=0xFE;
					if(dist<prevdistance)
					{
						prevdistance=dist;
						distancecount=0;
					}
					else if (dist==prevdistance)
					{
						distancecount++;
						if(distancecount>10)
							break;
					}
				}
			}
			P4&=0xF1;
			P4|=0x01;
			newlinearpwm(-50);
			waitmsec(200);
			newlinearpwm(0);
			waitsec(1);
			waitmsec(500);
```

```
                        newlinearpwm(-50);
                        waitsec(1);
//                      waitmsec(500);
                        P4&=0xF0;
                        vehiclestop();
                        newrotationalpwm(40);
                        waitsec(1);
                        vehiclestop();
                        target = -1;

//-----------------------------------------------
                }
                else if (target==-1)
                {
                        bucketraise_armraise(1,1);          //raise bucket and arm all the way up for
dumping
                        //drivetoobject(5);                         //drive close to object, since bucket
is not in way
                        returnval=drivetobeacon(6,20,0);
                        bucketraise_armraise(-1,0);         //dump material
                        //drivetoobject(30);                //back up from object to look for bin
                        returnval=drivetobeacon(40,20,0);
                        bucketraise_armraise(1,-1);         //tip up bucket and lower arm

                        P4&=0xF1;               //set arm raise manually for short duration
                        P4|=0x01;
                        waitsec(3);             //wait a short time, so arm is off ground
                        P4&=0xF0;               //stop arm motor

                        newrotationalpwm(30);
                        waitsec(2);
                        vehiclestop();

                        target=1;               //target material
                }

        }//end of main while loop

}//end of main function

void Timer2_ISR (void) interrupt 5
{
   unsigned short int i = 0;
   t2overflowcount++;
   t2of++;
   TF2H = 0;
   if (t2of > 0x02 ) {
   t2of=0x00;
   flags|=0x02;         //set ~50 ms flag
   P3_1 = 1;
   for (i = 0; i < 45; i++); //manually wait time for ultrasonic trigger generation
   P3_1 = 0;}

}

void PCA_ISR (void) interrupt 11
{
        unsigned short int resetvar;
        if(CCF0==1)
        {
                CCF0=0;
                //module 0 interrupt
        }
        else if(CCF1==1)
        {
                CCF1=0;
                //module 1 interrupt
        }
        else if(CCF2==1)
        {
                CCF2=0;
```

```
                //module 2 interrupt
                if( P2_4==1)
                {
                        ultrastart = (long int)TMR2;
                        ultrastart = ultrastart - ~t2count;
                        ultrastart=(long int) ultrastart/4;
                        tempvariable=(long int)t2count/4;
                        tempvariable=(long int)tempvariable*t2of;
                        ultrastart = ultrastart + tempvariable; //correct for delays
                }
                else
                {
                        timespan = TMR2;
                        timespan = timespan - ~t2count; //subtract off offset from reload time
                        timespan = (long int)timespan/4;
                        tempvariable=(long int)t2count/4;
                        tempvariable=(long int)tempvariable*t2of;

                        timespan = timespan + tempvariable;  //add on how many overflow counts occurred
                        timespan=timespan-ultrastart;         //correct for delay
//move into function, for potential filtering
                        flags|=0x01;
                        dist=(int)(timespan/58);
                }

        }
        else if(CCF3==1)
        {
                CCF3=0;
                //module 3 interrupt
//              resetvar=SWRSF;    // software reset
//              resetvar|=0x01;
                RSTSRC|=0x10;
        }
        else if(CCF4==1)
        {
                CCF4=0;
                //module 4 interrupt
        }
        else
        {
                PCA0 = pcaoffset; // + PCA0; //removed adding on offset of PCA to hopefully make pwm more
reliable
                //may result in some "jitter" in the 1 khz frequency, but only by probably about 40-50
instructions max, or ~1 us
                //will assume negligible impact
                flags|=0x04;   //set 1 ms flag
                CF = 0;
        }
}

//simple velocity controller

void encoder_pwmupdate(void)
{
        unsigned int templeftpwm, temprightpwm;
        short int pwmlimit=20;        //how far +/- from desired pwm vehicle can go
        unsigned short int gainfactor=5;
        short int deltachange=0;

        EX0=0;
        EX1=0;

        if(desired_left_pwm<0)
                desired_left_pwm=-desired_left_pwm;
        if(desired_right_pwm<0)
                desired_right_pwm=-desired_right_pwm;

        if(desired_left_pwm<40)
                desired_left_pwm=40;
        if(desired_right_pwm<40)
```

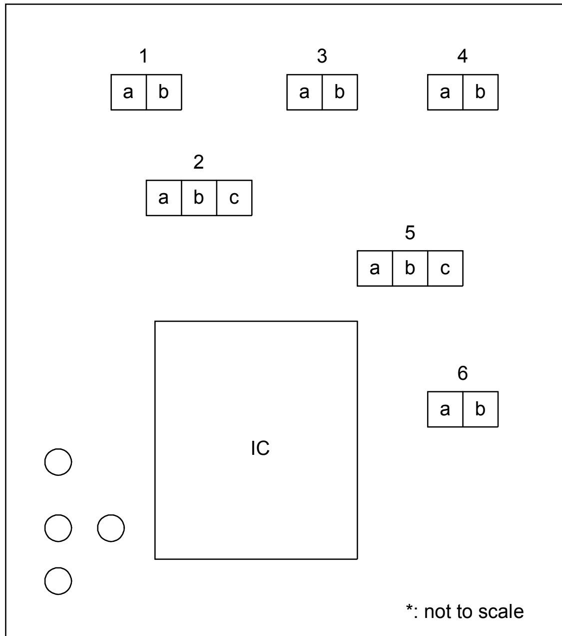
```
        desired_right_pwm=40;

if(contrlvar>50.0)
        contrlvar=50.0;
if(contrlvar<-50.0)
        contrlvar=-50.0;

if(contrlvar<1.5 && contrlvar>-1.5)
        deltachange=0;
else
        deltachange=(short int)(gainfactor*contrlvar);

if(deltachange>2*pwmlimit)
        deltachange=2*pwmlimit;
if(deltachange<-2*pwmlimit)
        deltachange=-2*pwmlimit;

//update corresponding motor pwm percentage (bound between 30 and 100%
if(deltachange>=0)
{
        //left motor too fast and/or right motor too slow
        //try to increase right motor pwm, if unable use all change, decrease left motor pwm
        pwm_right_out= desired_right_pwm + deltachange;
        if(pwm_right_out>desired_right_pwm+pwmlimit)
        {
                deltachange=pwm_right_out-desired_right_pwm - pwmlimit; //remaining change
                pwm_right_out=desired_right_pwm + pwmlimit;
                pwm_left_out=desired_left_pwm-deltachange;
                if(pwm_left_out<desired_left_pwm-pwmlimit)
                {
                        pwm_left_out=desired_left_pwm-pwmlimit;
                }
        }
}
else
{
        pwm_left_out=desired_left_pwm - deltachange;//(remember, deltachange is negative)
        if(pwm_left_out>desired_left_pwm+pwmlimit)
        {
                deltachange=pwm_left_out-desired_left_pwm - pwmlimit; //remaining change
                pwm_left_out=desired_left_pwm+pwmlimit;
                pwm_right_out=desired_right_pwm-deltachange;
                if(pwm_right_out<desired_right_pwm-pwmlimit)
                {
                        pwm_right_out=desired_right_pwm-pwmlimit;
                }
        }
}

if(pwm_left_out<40)
        pwm_left_out=40;
if(pwm_left_out>100)
        pwm_left_out=100;
if(pwm_right_out<40)
        pwm_right_out=40;
if(pwm_right_out>100)
        pwm_right_out=100;

//to calculate pwm signal, invert percentage, multiply it by the
//total number of increments per pwm cycle, divide by 100 to cancel percentage
//add to offset, then add 1 for the case of 100% pwm
//(without the +0, the PCA counter never increments to the PCA value, and the
//signal sometimes transitions, and sometimes doesnt)
templeftpwm=(100-pwm_left_out)*(int)(pcacount/100)+pcaoffset+1;
temprightpwm=(100-pwm_right_out)*(int)(pcacount/100)+pcaoffset+1;
if(templeftpwm<pcaoffset)//check for overflow from 0% case
        templeftpwm=0xffff;
if(temprightpwm<pcaoffset)
        temprightpwm=0xffff;
//update pca's near simultaneously
PCA0CP0=templeftpwm;
```

```
	PCA0CP1=temprightpwm;

	EX0=1;
	EX1=1;
}

void EXT0_ISR (void) interrupt 0
{
	unsigned short int tempt2ofcount=t2overflowcount;
	unsigned int timer2val=TMR2;//grab timer 2 value


	IE0 = 0;
	//update interrupt period

	leftencoderperiod=(unsigned long int) ((timer2val/400) + (t2count/400)*tempt2ofcount);	//total
time in .1 ms units
	if(leftencoderperiod<leftencoderoffset)	//timer 2 overflowed, adjust for it
	{
		leftencoderperiod+=~ ((unsigned long int)(0)) - leftencoderoffset;
	}
	else
	{//subtract off offset for the time between last interrupt
		leftencoderperiod-=leftencoderoffset;
	}

	leftencoderoffset=(unsigned long int)((timer2val/400) + (t2count/400)*tempt2ofcount);

	if(desired_left_pwm==0)
	{	//left motor shut off, make sure pwm is 0%
		PCA0CP0=0xffff;
	}
	else
	{
		contrlvar+=leftinc;
		encoder_pwmupdate();
	}
}

void EXT1_ISR (void) interrupt 2
{
	unsigned short int tempt2ofcount=t2overflowcount;
	unsigned int timer2val=TMR2;//grab timer 2 value


	IE1 = 0;
	//update interrupt period

	rightencoderperiod=(unsigned long int) ((timer2val/400) + (t2count/400)*tempt2ofcount);	//total
time in .1 ms units
	if(rightencoderperiod<rightencoderoffset)	//timer 2 overflowed, adjust for it
	{
		rightencoderperiod+= ~((unsigned long int)(0)) - rightencoderoffset;
	}
	else
	{//subtract off offset for the time between last interrupt
		rightencoderperiod-=rightencoderoffset;
	}

	rightencoderoffset=(unsigned long int)((timer2val/400) + (t2count/400)*tempt2ofcount);


	if(desired_right_pwm==0)
	{
		PCA0CP1=0xffff;
	}
	else
	{
		contrlvar-=rightinc;
		encoder_pwmupdate();
	}
```

```
}

void SPI0_ISR (void) interrupt 6
{//compass reading program.
	//note: compass operating in binary mode, first 7 bits are useless, and the remaining 9 bits are
data.
	unsigned short int readbyte=0;
	readbyte=SPI0DAT;	//read current byte

	SPIF =0;	//clear all interrupt flags
	WCOL =0;	//does not check for errors, since unsure how to correct them in this code
	MODF =0;
	RXOVRN =0;

	if(bytenumber==0)
	{//first recieved byte, save it and return
		firstbyte=readbyte;
		bytenumber=1;
	}
	else
	{
		secondbyte=readbyte;
		bytenumber=0;
		vehicledirection = (0x01&firstbyte)*0x0100 + secondbyte;  //combine together the two
bytes of information
		vehicledirection&=0x01FF;		//mask off unused bits
		flags|=0x08; //sets compass reading flag
	}
}
```

Appendix C: Vehicle Modification Information

Original RC vehicle drive circuit connectors:



Description of original test results:

Controls referenced from facing behind the vehicle towards the front of it (normal driver position)

1) Right Motor Power
   a. (red wire) 0v for forward, 7.9v for reverse
   b. (yellow wire) 7.9v for forward, 0v for reverse
2) Arm Lift Sensor
   a. (black) Low arm sensor, shorted to ground when arm is as low as possible
   b. (white) High arm sensor, shorted to ground when arm is as high as possible
   c. (purple) Ground
3) Arm Lift Motor Power
   a. (black) 0v for raising arm, 5v for lowering arm
   b. (white) 5v for raising arm, 0v for lowering arm
4) Bucket Tilt Motor
   a. (black) 0v for raising bucket, 5v for lowering bucket
   b. (red) 5v for raising bucket, 0v for lowering bucket
5) Bucket Tilt Sensor
   a. (yellow) Bucket tilt low, shorted to ground when bucket is as low as possible
   b. (red) Bucket tilt high, shorted to ground when bucket is as high as possible
   c. (black) Ground
6) Left Motor Power
   a. (green) 0v for forward, 7.9v for reverse
   b. (blue) 7.9v for forward, 0v for reverse

In order to add/change rotary encoder wheel:
Step 1: Pry center cap from the middle of one of the drive wheels, remove support rod inside vehicle.
Step 2: Remove existing encoder wheel
Step 3: Place new encoder wheel
Step 4: Reassemble drive motors, replace support rid in vehicle (make sure gear train gear is lined up), replace center cap.

Repairing bucket linear actuator: (VERY annoying to do, try not to apply voltage to bucket motor without protective circuitry in place)

Step 1: Raise arm all the way up, remove screws from vehicle arms.
Step 2: Separate arm sleeves, remove screws holding bucket motor in place.
Step 3: Remove screws from plate underneath bucket motor. Be careful when opening plate, there is a gear held in place by pressing against the plate.
Step 4: Locate linear actuator (it is in one of the black sleeves that connects to the bucket, the other sleeve contains the contact switch for the bucket). Rotate linear actuator sleeve until it is approximately the same length as the sensor sleeve. This should allow the motor to control the bucket actuator again.
Step 5: Replace gear into linear actuator, verify the contact switch is still operating correctly, replace plastic plate.
Step 6: Replace some screws into system, test bucket functionality. If bucket is fixed, put all screws back in. If bucket does not work, make sure step 4 and 5 are completed (mainly make sure the gear is in the linear actuator the correct way).

Appendix D: Vehicle Operation Information

Setup Procedure for Vehicle:

Step 1: Make sure infrared LED drive circuitry is operational, and correctly connected. If no circuitry is available, previous circuitry was made from an L293 chip, with function generators controlling 2 of the switches. Function generators do not provide enough current for LED's to operate, so higher current source to drive them is required.
Step 2: Set load beacon frequency to 10 Hz, 0 to 5v square wave, 60 % duty cycle.
Step 3: Set truck beacon frequency to 5 Hz, 0 to 5v square wave, 80% duty cycle
Step 4: Turn on power supply, verify LED operation. (Potentially use a cheap camera to see infrared light, such as cell phone camera)
Step 5: Power up microprocessor and vehicle
Step 6: Orient vehicle in direction to load the bucket, press the pushbutton for P2_0, MOVE HAND AWAY FROM COMPASS. Let vehicle sit stationary for 2-3 seconds.
Step 7: Orient vehicle in direction to dump into the truck, push the pushbutton for P2_0, MOVE HAND AWAY FROM COMPASS. Vehicle will get reading for truck, and then begin operating.

If vehicle does not start, look for a flashing red light on ultrasonic sensor. If no light is flashing there, the ultrasonic sensor is not operating, and something is not connected properly, or the battery is dead. If the light is flashing, try resetting vehicle and repeating step 6 and 7, as the push button might not have been recognized as being pushed one time.