Project WEAVER
Wireless Enabled Active Video Experimental Rover
Manual

EE/CS 3992

Tyler Lloyd
Amber Blake
Janos Opra

December 16, 2004

Table of Contents

INTRODUCTION

This paper will detail our design of a wirelessly controlled vehicle equipped with a camera and a collision avoidance system that we developed as a senior design project. This project contains both hardware and software design aspects. Hardware was designed to replace the original motor driving electronics on the vehicle, drive the collison avoidance modules, and interface to an 802.11 PCMCIA card. Central to the hardware design was the Digital Signal Processor, DSP56F807. Software was developed to run on this DSP to control the systems on the vehicle, and additional application software was developed for the PC to receive commands from the user and send them to the vehicle over a wireless connection. The following documentation will detail both the hardware and the software components. The hardware component will be discussed first, followed by the software component.

DESIGN OVERVIEW

The entire system can be summarized by referring to the block diagram in Appendix A. The bill of materials associated with this high level overview is included in Appendix B.

From the block diagram, it can be seen that our project entails two major blocks one block contains the work required on the PC and the other is the work for the vehicle. The PC block really does not contain a hardware aspect; it contains the application level software to communicate commands from the user to the vehicle. The second block is the vehicle which has a lot of hardware associated with it. The vehicle itself is a modified RC vehicle. We are using a RC truck, because it provides more space, but the actual chassis of the vehicle is unimportant. At the heart of the implementation is a DSP. The Camera that will be used is a commercial 802.11b enabled camera. We chose to use this camera due to the difficulty we experienced in obtaining a video encoder integrated circuit. Motion control will be handled by using H-bridge motor drivers and Pulse Width Modulators (PWMs) available on the DSP. The IR LEDs and Sensors shown in the block diagram are for motion control. They are driven using a PWM and are read back by the DSP. Two battery packs will be included on the vehicle, one for the motors and another for everything else. The voltages for all the parts excepting the motors will need to be regulated.

Connecting these two blocks is a wireless interface. The Wireless standard that we chose to use is IEEE 802.11b. We chose this protocol to accommodate the interface of the camera.

ROVER HARDWARE COMPONENTS

The following sections detail the hardware components. Each section describes a particular group of components that perform a function. The sections include our reasoning behind our component choices. A complete schematic of the hardware can be found in appendix C.

POWER REGULATION CIRCUITRY

Regulators:

The Texas Instruments PT6213P and PT6302B integrated switching voltage regulators were chosen for their high current capacity and high efficiency. They were also chosen because they are packaged as a 12 pin single in-line IC. This makes them much easier to use and install because they only need one external capacitor to operate. The PT6213P has 84% efficiency for 3.3V output with a 2A max current and a 9V minimum input voltage. The 3.3V is needed to power the wireless card, Motorola DSP, collision avoidance, and other circuitry. The high current capacity of this regulator will be needed mostly for the wireless card which requires 430mA to run. The PT9302B has 89% efficiency for 5.0V output with a 3A max current and a 9V minimum input voltage. The 5.0V is only needed to power the Dlink wireless camera and the infrared sensors. The 3A high current capacity was chosen because we thought the camera needed 2.5A at 5V; however, we later discovered that only 900mA was needed. This additional current capacity will allow us to add additional features later without needing to change parts.

Batteries:

NiMH batteries were chosen for their high current capability, big capacity, and reusability. Alkaline batteries might have supplied the high current but they had less than half the capacity of NiMH. Alkaline batteries would also have cost more in the long run since they are not rechargeable and we will need to test and run our equipment more than

one time. We bought ten Accupower C-cells and eight energizer AA-cells. Ten C-cells will provide 12V and will be used to power the two switching regulators. These cells have a 4500mAh capacity that will give about 4-5 hours of running time for our circuitry and camera. The eight AA-cells have a 2300mAh capacity and will be used to power only the motors on the vehicle. The motors were kept on a different power supply because of possible noise interference with the sensitive circuitry.

Battery Charger:
The Accupower Accumanager 20 was chosen because it can quickly charge batteries and it is able to charge high capacity batteries. It is a microprocessor-controlled charger capable of charging any size cell with any size capacity. It can sense the capacity of the cell so it can increase the charging rate for larger batteries and will stop charger only when the cell has reached its full capacity. This decreases the charging time and will prevent the overcharging of batteries, which most chargers cannot do. Other battery chargers only have a timer controlling the amount of charge a cell will receive so they are not able to charge the high capacity cells or change the rate of charge.

DSP CIRCUITRY

We chose to use a DSP56F807 DSP by Motorola. It has a lot of microprocessor functionality built in that will make it easier to use, since no one in our group has programmed DSPs before. We chose this DSP for many reasons. First, it has twelve pulse width modulator (PWM) pins. We ended up using all of them. Second, it is programmed using a JTAG interface that will also aid in circuit debugging. Third, we already had access to the software development kit because of a seminar Tyler attended. The final reason we chose this processor is because a reference design was available.

The DSP circuitry includes an 8 MHz crystal that is multiplied by ten inside the chip by a PLL to provide an 80 MHz system clock. Using an 8 MHz crystal and a PLL improves power efficiency according to Motorola. Since the vehicle is battery operated power consumption is a concern to us. The crystal driving circuitry on the DSP is driven from the analog power pins. The analog voltage is supposed to be really clean, and slightly lower than VCC. Since we are not performing any analog to digital conversions we do not care to keep the power ripple free. We used the diode D6 to drop the voltage a little. We actually used a schottky diode instead of the silicon junction diode specified in the schematic to keep the voltage drop to a minimum.

The JTAG programming connector P3 on the schematic is connected into the appropriate processor pins. The switch J4 and the diode D7 provide a way to reset the processor without resetting the JTAG port. These parts were called out in the reference design.

Power supply bypassing is very important to the operation of the DSP. We used a 0.1 µF capacitor for every VCC and AVCC pin. The two 2.2 µF capacitors are required according to the data sheet, and reference design, and must have an ESR of less than 150 milliohms.

Other components that can be loosely grouped with the DSP include a DB9 connector for serial RS-232 communication. However, we forgot to include a level shifter to get true RS-232 levels on the schematic, and had to rework the board to add one. We chose a Maxim MAX3221 RS-232 level shifter for this purpose. The J6, J7 and J2 components provide places to probe signal levels, and access unused I/O pins that can still be used. J2 even includes a 3.3 V and a Ground pin with a bypass capacitor.

There are two LEDs that will be used for debugging. We can set the LEDs to flash, or toggle in software. They can be used to signal when a message was received over the wireless, or when a certain command decoded in the motor driver. The processor pins do not have enough current driving capacity to drive the LEDs directly, so an inverter was used to act as a buffer and increase the current capacity. R4 and R5 provide current limits for the LEDs.

WIRELESS TRANSMISSION CIRCUITRY

The wireless transmission circuitry consists of a low-voltage PCMCIA connector, 802.11b PCMCIA card, pull-up resistors, and bypass capacitors to regulate the input voltage. We chose to use a 3Com 802.11b card (3CRWE62092B) because it was readily available. Since the other components were standard, we used the ones that

we had readily available through work and other resources.

The pin connections of the PCMCIA interface were researched, and all of the necessary connections were drawn out to the processor. Most of these connections are simply wires; however, some have pull-up resistors as required by the PCMCIA standard. There were also signals that were not drawn to the processor, but required external connections such as the signals VS1 and VS2. VS1 and VS2 are voltage sensing outputs used to determine whether the card is low-voltage or not. Since we are only using one type of card, these signals did not provide us with useful information. Therefore, the required pull-up resistors for these signals were attached, but the signals are not connected to the processor. A test point was placed on the signal INPACK as it is not required to run the card, but provides useful information for debugging purposes. Specifically, this signal is asserted low by the card when it recognizes the memory address. It is expected that this will be useful in debugging the software on the DSP to control the wireless card. Standard bypass capacitors were used in parallel on the Vcc and Vpp voltage inputs to the card to provide additional current to the card, and to reduce noise.

The external address and data bus on the DSP were used as the address and data lines to the wireless card. The address bits A16 up to A25 were pulled to ground since the address bus on the DSP is not that wide, and there is not enough memory on the wireless card to require all of these bits.

A table of all of these connections is included below. Together these connections will allow for detection of the PC card, reading the card's information structure (CIS), and performing the I/O transactions between the laptop and the vehicle. In order to perform these operations the card must be started up in memory mode to read the CIS. The card must then be switched into I/O mode to download the firmware, and read and write to the registers containing the data received and sent by the card. This is done by simply changing the purpose of some of the pins on the card, and did not require a significant hardware change. A circuit diagram is attached in Appendix C.

Signal	***Pin #***	***Pull-up Resistor***	***Description***
IORD#	44	N/A	I/O read command signal
IOWR#	45	N/A	I/O write command signal
OE#	9	N/A	Output Enable
WE#	15	N/A	Write Enable
REG#	61	N/A	Register Select
CE2#	42	N/A	Card Enable 2
CE1#	7	N/A	Card Enable 1
ISIO16#	33	N/A	16 bit I/O register access signal
WAIT#	59	10 Kohm	Wait signal for long transactions
CD1#	36	10 Kohm	Card Detect 1
CD2#	67	10 Kohm	Card Detect 2
READY	16	10 Kohm	Ready signal
IREQ#	16	10 Kohm	Interrupt Request
RESET	58	N/A	Reset signal
INPACK#	60	10 Kohm	Input Acknowledge – Test Point
VS1#	43	100 Kohm	Voltage Sense 1 – Not connected to processor
VS2#	57	100 Kohm	Voltage Sense 2 – Not connected to processor
A25:A0	----------	N/A	Address lines
D15:D0	----------	N/A	Data lines
Vcc	51 & 17	N/A	Voltage supply

Signal	*Pin #*	*Pull-up Resistor*	*Description*
Vpp1	18	N/A	Voltage supply
Vpp2	52	N/A	Voltage supply
Ground	35, 1, 68, 34	N/A	Ground

TABLE 1: PCMCIA CONNECTIONS

COLLISION AVOIDANCE CIRCUITRY

There were two main parts to the collision avoidance hardware, IR sensors (part number: PNA4602M) and IR LEDs (part number: QEC113). We chose to use these parts because we had previous experience with them and they are easy to use. The circuit will notify the processor when an object is detected within its sight. The IR sensors assert a low signal when they see an infrared light beam with a 38 kHz frequency. In order to utilize this feature, a 38 kHz signal will be sent from the processor to the IR LEDs. When the emitted light beam strikes a wall or other solid surface, it will be reflected back. The IR sensors will detect the IR light and assert a low signal. This signal was fed back into the processor through a Schmitt Trigger inverter to translate the low assertion into a high signal of 3.3 V. In order to minimize the number of PWM pins necessary to drive the LEDs, Schmitt Trigger inverters were also used to amplify the 38 kHz signal and send it through potentiometers to each of the three IR LEDs. This decision allowed us to use only one PWM pin instead of three to drive the three IR LEDs. Potentiometers were included in series with each IR LED so that the current passing through the IR LEDs can be modified to adjust the likelihood of the signal being detected by the IR sensors. These potentiometers will be adjusted initially to provide the most reliable detection possible with the actual current resulting from the circuitry on the vehicle. Real time sensitivity control will be performed using software residing on the DSP. A circuit diagram is included in Appendix C.

MOTION CONTROL

The RC truck that we acquired to use has three motors. One motor is used to control left and right steering. One motor is used to move forward and backward, and the last motor controls the gear ratio for the forward and backward motor.

Most high end RC vehicles use a servo to turn left and right. We did not buy a high end vehicle. Our turning motor, B3 on the schematic, is connected to an H-bridge. The MOSFETs, AND gates, and resistors form a circuit that will provide bidirectional control of the motor and brake capability.

The turning motor is composed of a simple motor that turns a gear that turns the steering arm. The steering arm has contacts that ride on a PCB. There are six wires that connect to this motor. Two are for the motor to rotate left or right. The other 4 connect to the PCB and provide the mapping to where the arm is located. We will call LR1 and LR2 the outputs and because of the pull down resistors and the contacts on the PCB to the two 3.3 V wires, we can create a truth table to explain how to move the motor.

LR1	LR2	Behavior
0	0	Straight
0	1	Not maximum left
1	0	Not maximum right
1	1	Maximum left or right

TABLE 2: LEFT, RIGHT TRUTH TABLE

The steering arm has a spring that will return the wheels to a straight direction. In order to keep the vehicle turning the motor must be moved to the desired position and then the brake must be applied to keep it in the correct position. If the motor does not provide enough torque to stay still when the break is applied, the H-bridge must be used to keep the arm in the same position.

The motor that controls high and low gears works in a similar manner. The H-bridge driver is identical to that used for the turning motor. There are only five wires on the gear motor, B2 on the schematic. Again, two wires connect directly to a motor that can rotate left or right. The other three wires are connected to contacts on a PCB. There is

one wire that can connect to either the low gear wire or the high gear wire, but not both. The truth table for this motor output is shown in table 2.

Low_Gear	High_Gear	Behavior
0	0	Undefined
0	1	High gear selected
1	0	Low gear selected
1	1	Not Possible

TABLE 3: HIGH, LOW GEAR TRUTH TABLE

The Main drive motor, B1 on the schematic, is set up differently. The motor has a no load current of 0.5A, and is rated to at 3.29A for maximum efficiency. To supply as much current as possible, we found a DMOS full-bridge motor driver IC. This part, manufactured by Allegro, is capable of supplying 2.8A of repetitive current. We should get plenty of speed from our vehicle through this part. The part has its own charge pump to turn on the internal FETS completely. It requires 4 processor pins. One is a PWM that determines speed, one is a sleep pin, and the other two determine direction and braking. The part has a heat sink built in, that is connected to a large surface ground plane on the PCB.

WIRELESS CAMERA

Our original plan was to purchase a video compression IC and CCD camera separately and design our own video hardware. However, since the frame rate was one of our primary concerns and we were unable to purchase any video compression hardware as previously intended, we chose to purchase an 802.11b enabled camera. We chose the Dlink DCS900W Internet camera because it has an integrated 802.11b wireless connection and video compression for higher frame rates. It has a 640x480 resolution at a frame rate of 20, which will provide a high quality video stream. We require the high frame rate to ensure we can see where the vehicle is traveling in real time. The camera is also ActiveX and Java compatible which made it easier to write the software needed for viewing the video stream. This camera was mounted to the top of our vehicle to give us the best view for driving the vehicle remotely.

PCB DESIGN AND VERIFICATION

We used Protel DXP 2004 by Altium for our schematic and PCB design software. Tyler has used this family of software extensively. We made all of the components for the schematic view and for the PCB footprints that were not already available. We measured the available space in the vehicle and made the new board outline so that it would be big enough for our needs. We are using the mounting holes that were used on the factory board. After the board was designed, and the schematics were reviewed as a group, gerbers were created. The gerbers were sent to a PCB fabricator and complete boards were returned one week latter. Finally the boards were reviewed for correctness. From the five boards that we received, we failed 3 because of manufacturing defects that created obvious shorts between traces.

BASIC ASSEMBLY

We populated the PCB and performed some basic continuity checks on the power pins. Initially, we intended to put the entire vehicle together, and then work on software. As we got to the point of putting things together, we decided that we needed the board to be exposed while we did some more testing. We didn't know exactly what current the motor drivers are capable of delivering, nor did we know what current the turning and gear motors require. We didn't want to destroy the motors because we turned them too far or to fast. It made more sense to keep the board exposed while we worked on the basics, then we finished the assembly.

DEBUG STRATEGY

We rejected three boards out of our lot of five because of visual defects. We populated one of the boards that remained and performed a continuity check on the ground pins. Next we supplied power to the regulators and verified that the voltage outputs are within tolerance. We checked voltage levels around the card, and total current consumption for the card. The next step was software on the DSP to disable the motor drivers. That software was not necessary to this point, but once the motors drivers could be turned on and off, their current capacities could be

measured, and appropriate series resistors could be ordered. Hardware testing and modification continued throughout the software development process until the hardware was at an acceptable level.

CONCLUDING REMARKS \ REQUIRED MODIFICATIONS

We started the hardware design process with a functional description of what we wanted the hardware to do. As we designed the schematic we thought of more parts that we should add, including the serial connector, and the expansion port on the available I/O pins. We completed the schematic and started working on the board. As we designed the board outline, we had to make some calculated guesses at dimensions, printout some scaled outlines, compare to the mechanical body, and make adjustments. Eventually we got the board ready to build, and sent out the files. As soon as it was too late to change anything, more changes were found; most notable is the missing RS-232 driver. We have to make modifications to our existing hardware because the PCB is too expensive to revise. The board does fit in the space required, and the mounting holes line up with the mounts in the vehicle.

ROVER SOFTWARE COMPONENTS

WIRELESS COMMUNICATION

This task combined two separate components, one on the laptop and the other on the DSP. The laptop software sent commands to the vehicle over the 802.11 link, to the DSP where they were supposed to be received through the hardware interface designed above. The DSP software will be discussed first, followed by the PC software.

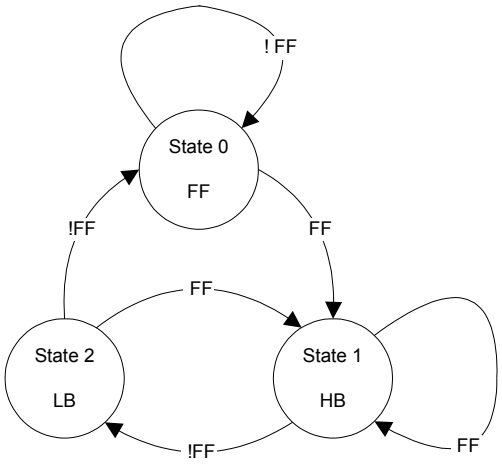
The intent was to develop a somewhat simplified driver for the DSP to allow it to drive the 802.11 card through the hardware interface designed above. However, we were unable to communicate appropriately with the card due to the limitations of the processor that we were unaware of during the hardware design process. Specifically, the address bus on the DSP would not acknowledge a wait signal from the 802.11 card. The wait signal informs the host machine that the 802.11 card requires more time to complete the operation. The inability to recognize this signal results in unreliable communication.

The faulty hardware design resulted from our incomplete understanding of the functionality of this specific DSP. We believed that the address bus on the processor was a peripheral to GPIO ports, so that we could generate our own bus logic if necessary by directly driving the GPIO ports. This was not the case. Only a few of the address and data pins could actually be driven as GPIO ports, and the external memory interface on the processor was very limited. The only control over the timing of the signals was through wait states; at this point we also discovered that the external memory interface was not timed using the system clock but the PLL clock output. This meant that the external bus speed was twice that which was expected. The processor was slowed down to try to make the wait signal irrelevant, but the necessary speed would not allow the processor to drive the 40kHz signal necessary to drive the IR LEDs for the collision avoidance module. This fix also did not guarantee that the wait signal would never be a problem.

There was one other problem that we were unable to resolve due to the hardware design, but it requires a little more explanation about the internal workings of the 802.11 card. This type of a PCMCIA card can be accessed in two separate ways. The card is initially brought up by the system using the PCMCIA memory only interface, and the Card Information Structure is read from the attribute memory space on the card by a card services driver. This structure contains information about the capabilities of the card, from which the card services driver determines what firmware is required by the card. The interface used to access the card is then switched from the PCMCIA memory interface to the PCMCIA IO interface and the firmware is downloaded on to the card. In the PCMCIA IO interface, address and data registers can be written and read to access another memory bank, the common memory bank, on the card. In order to read and write these registers, different read and write strobes than in the memory only interface must be used. Due to the limitations of the DSP's external memory interface, we were unable to quit driving the memory read and write strobes. This resulted in an inability to download the firmware onto the card even when the processor was slowed. An attempt was made to write to the memory space using the address and data registers and then read the data back; however, the operation was not successful even when the wait signal was not asserted during either operation. This led us to conclude that the read and write strobes were the problem. This could be fixed by

multiplexing the memory read and write strobes to the two separate pins on the PCMCIA card, and only driving one or the other. Another possible solution would be to not access the card in the memory only interface since we do not need the information in the Card Information Structure if we limit ourselves to only using one type of card. However, neither of these solutions would allow us to recognize the wait signal, so any communication would be unreliable at best and most likely non functional. The only real solution to this problem is to add a host bus adapter between the processor and the PCMCIA connector, possibly an FPGA. The FPGA could then latch the address, data, a flag indicating a read or write operation, and a flag indicating IO or memory read write with the timing of the host bus. A flag could then be raised to the processor indicating that the data from the PCMCIA card is ready and waiting in an internal register on the FPGA. A second memory access could then be used with a specific address to retrieve the data. If the operation was a write operation, then the FPGA will not need to notify the processor when the write operation is complete, but it would need an internal buffer to ensure that one write operation is completed before another one is started. This solution was considered during the hardware design, but was decided against due to cost and hardware complexity considerations. By the time the necessity was apparent, there was neither time nor money to perform the hardware modification.

To allow the rest of the project to be completed, a serial interface was designed so that the vehicle could accept commands over the serial port. The command instruction that was agreed upon required sixteen bits to communicate a command. Since, the serial interface on the DSP only allows one byte to be received at a time; a state machine was defined to receive the entire two byte command. Each command that was sent was proceeded by the byte 0xFF to synchronize the commands. The state machine diagram is given below.



In State 0 the FSM waits for a byte containing all ones, 0xFF, before moving to State 1 where it receives the high byte of the command. Once the high byte is received, it moves to State 2 where it receives the low byte of the command. Then the FSM transitions back to State 0 to wait for the next synchronizing byte.

The development of the wireless communications software on the PC was more successful than that on the DSP. The code to send the commands over the 802.11 link was successfully written and tested. This code was structured as a class where the opening of the socket was performed in the constructor, and member routines were written to send and receive data over the 802.11 link using the Ws2_32 library. A console application was created as a test program to either send or receive data over a network connection on a computer. Once the operation of the base class was verified using this program, it was integrated into the main GUI. Final testing was performed by using the main GUI to receive the commands from the joystick code, and send the command over the 802.11 link. The console application placed on another machine in a receive-only configuration so that the data being transmitted could be viewed.

When the unfortunate problems with the DSP wireless communication were discovered, this code had to be replaced by serial communications code. The serial communications code is also structured as a class that creates the COM port in the constructor and uses member functions to receive and send data. The Platform SDK functions for creating and reading files were used to perform these operations as outlined in the MSDN documentation. One difficulty that was encountered was that the name of the COM port had to be prepended with \\.\ to open COM ports

with certain numbers. This is not easily found in the MSDN documentation, and was discovered by researching programming discussion boards on the internet. This code was tested using another test console application that sent a single command to the car and waited for the DSP to echo the command back. This test application and the debugging software for the car were used to verify that the commands sent from the PC were actually received by the car. Then, the test application was modified to send commands in a loop similar to the one in the main GUI, and that code was executed to verify that the DSP could receive commands in rapid succession. Finally, the code to control the serial port was integrated into the GUI and tested again.

MOTION CONTROL

We decided to use a joystick as our input source. The joystick provided a variable source that offers ease of use, and maps well to the motion of the car. The joystick has two controls, one for forward/reverse, and one for left/right. We also mapped a couple of buttons to enable, and disable the collision avoidance, and to switch from high to low gear. The joystick provides an output for each axis that varies positive and negative. We scaled the output to provide ten steps in each direction, using two nibbles of data. We mapped the forward/reverse, left/right command bits to another nibble. We used two bits from a fourth nibble for obstacle avoidance, and high/low gear selection. These four nibbles were aggregated to form a two byte word that was then set to the vehicle. The high byte determines the motion commands, six possible combinations, and the low byte contains the magnitude, with 21 combinations possible. The command byte includes settings for forward, reverse, right, left, collision avoidance, and stop. The magnitude byte consists of the two magnitude nibbles; the most significant nibble represents the forward/reverse magnitude and the least significant represents the left/right magnitude. A command was never issued with a magnitude of one because the joystick didn't always return to the center. Unless the first position was ignored, the joystick would send out movement commands when the vehicle was supposed to be stopped. The table below shows the different possible values for the command and magnitude bytes. These may be combined using a logical OR operation to form commands such as forward and right or reverse and left. For example, to move forward and right, OR the commands for forward and right for the high byte, and OR the magnitude of forward with the magnitude of right to get the low byte.

Command		Magnitude			
Right	00000001	Center	00000000	00000001	00010000
Left	00000010	Right/Left 2	00000010	Forw/Rev 2	00100000
Reverse	00000100	Right/Left 3	00000011	Forw/Rev 3	00110000
Forward	00001000	Right/Left 4	00000100	Forw/Rev 4	01000000
Collision Avoidance Off	10000000	Right/Left 5	00000101	Forw/Rev 5	01010000
		Right/Left 6	00000110	Forw/Rev 6	01100000
		Right/Left 7	00000111	Forw/Rev 7	01110000
		Right/Left 8	00001000	Forw/Rev 8	10000000
		Right/Left 9	00001001	Forw/Rev 9	10010000
		Right/Left 10	00001010	Forw/Rev 10	10100000

The motion control on the vehicle was developed in stages. The first stage was to decode the incoming message into the nibbles used to form the word. This process was done using a series of AND and shift operations. Each nibble is compared to what the car is currently doing, and any differences are implemented immediately. This allows each of the motors to be controlled independently from each other, and prevents decreasing the duty cycles significantly. We wrote a set of unit tests that verified that the command was being decoded correctly.

The next phase of development actually ran the motors. We experimented with some PWM frequencies before measuring the values from a functional RC vehicle. The forward/reverse motor uses a 64Hz frequency that has the following available duty cycles, corresponding to magnitudes issued from the laptop: 0%, 55%, 60%, 65%, 70%, 75%, 80%, 85%, 90%, 95%, and 100%. We burned out one two PWM pins on the processor and two of the motor driver ICs before we learned that the frequency should be this slow. We lifted the dead PWM pin, and routed a spare

I/O pin to the trace so that we did not need to replace the processor.

Initially, we could not turn on the h-bridges to drive the left/right, and high/low gear motors on the vehicle. We had to modify our circuit to make them work. We initially had P type MOSFETs in the H-bridge. In order to drive the MOSFETs on, we would need a voltage that is higher than the supply, and unavailable. We changed the MOSFETs to N type, and then discovered that we needed to reverse the source and drain pins. The discovery was made because the circuit would over-current every time the supply was connected to the motor power. The MOSFETs we used have built in flyback diodes, that are always forward biased when the source and drain are reversed. Our two h-bridges provided four low impedance paths to ground, which caused the high current draw. After making these modifications, we still could not supply a lot of current to the turning motors. This resulted in limited turning ability with the additional friction when the car was driven on the ground. Our solution was to replace the motor that turns the wheels with an airplane servo. The servo moves to a position defined by an input frequency and duty cycle. The carrier frequency is 68 KHz, and the duty cycle is 7% for center, ±0.7% in 0.1% increments for left and right.

SENSOR PROCESSING AND COLLISION AVOIDANCE

Most of the sensor processing and collision avoidance software was located on the DSP. The only component on the PC was the code in the GUI to alter the command code and the bitmap on the GUI depending on whether or not the collision avoidance feature was active. The sensor processing and collision avoidance software on the DSP consisted of two separate functions. The first function read the input pins that the sensors were connected to and set global variables indicating which sensors were detecting objects. The second function took the encoded command word, called the sensor polling function, and modified the command word to avoid any obstacles that the sensors detected. The algorithm used to modify the command input takes into account both what is seen by the sensors and what direction the given command is telling the car to go. After the command was modified, it was copied into the global variable that was used by the motor control software so that the modified command would be executed instead of the one received from the user.

The algorithm used actively avoided the obstacles that were detected in the sensors; the logic is as follows. If all of the sensors detected obstacles, the command would be altered to stop the vehicle. If only the left and right sensors detected objects and the command received from the user was to move forward, the command would be modified to move forward at half of the received speed. This was done to alert the user, but not to divert the course since there were no sensors in the front of the car. If only one of the sensors detects an object, and the given command is to move in that direction, then the command is modified to move in the opposite direction with half of the magnitude received in the original command. On the other hand, the left and right sensors are ignored when the command received is to move the car backwards unless something is also detected in the back sensor. In this case, the algorithm modifies both the forward/reverse and left/right portions of the command as if the both detecting sensors were the only ones detecting.

The command decoding and modification function was written and tested on a PC, using a console application, before being transferred to the DSP. The console application has a test mode where it reads commands out of a buffer in order, generates random values for the sensors, modifies the command according to these sensor readings, and finally checks the result against a truth table of values for that particular command. If the result was not what it should have been, an error message was printed to the screen. A set of commands was generated that included all different possible combinations of forward, reverse, left, and right. These commands were run and checked with 1000 different sets of random sensor readings to ensure that the function was operating correctly. The console application was also capable of operating in a simulation mode that allowed the user to input the directions and magnitudes for the vehicle. The program would output the changes it made to the original command, if any, based on a random set of sensor readings. After this testing procedure the code was moved onto the DSP with very little trouble.

The code to monitor the sensors could not be tested on the PC; consequently, it was written and tested on the DSP. The code checks the value on each of the PWM fault pins that the sensor outputs are connected to and sets the global flags used by the decoding function accordingly. This function was tested by driving an LED from the DSP each time one of the global flags was set. The LED was also used to tune the distance that the sensor could see by adjusting the potentiometers and sensors.

Once this testing was completed, the code was merged into the main loop of the code on the DSP such that if the collision avoidance feature was active, the command modifying function was called before the motor control function. Subsequent testing of the integrated system was successful.

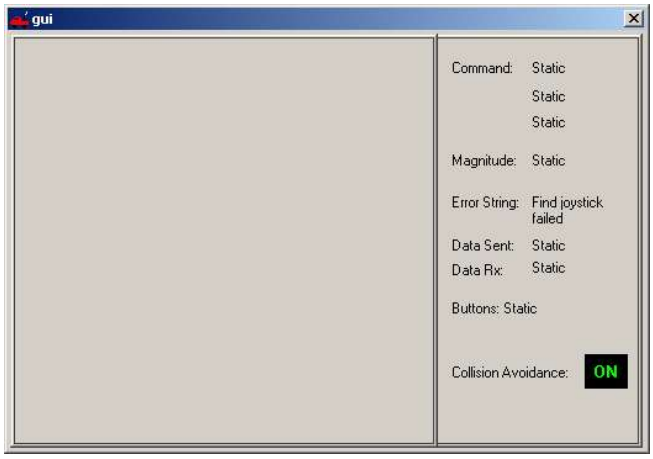
VIDEO INTERFACE

The video interface software component was necessary to capture the video coming across the 802.11 link from the camera on board the vehicle and display it in the GUI. This code actually ended up being fairly simple since the camera came with an ActiveX control. The hardest part of this task was determining what an ActiveX control is, how to insert it into the program, and how to access its features. After digging through the MSDN documentation, it was discovered that it was fairly simple to insert an ActiveX control into a dialog. The control is simply drawn into the graphical editor, the desired ActiveX control selected, and the compiler auto-generates the code. However, until this point the GUI was a non-dialog based MFC application; nothing could be found about how to insert the ActiveX control into this type of an application. In order to use the ActiveX control, the GUI was rewritten as a dialog based application and the ActiveX control was inserted. This ActiveX control was not documented since it was not meant to be used by the general pubic, so in order to use it (or even find which one to use) we had to be a little sneaky. It was fairly easy to determine which ActiveX control to use. Since the ActiveX control had to be installed on the computer, we only had to observe which .ocx file was being installed. To figure out how to use the control to view the video from the camera, we analyzed the JAVA code that was used to display the video using the ActiveX control from the standard webpage interface that is normally used for the camera. This code was essentially a bunch of function calls to the ActiveX control that acquired the camera and displayed the video. These calls were translated into the framework given by our application, but it didn't quite work. Analyzing the other available functions in the ActiveX control revealed a function called GetRealTimeData. An educated guess was made that this would acquire the data from the camera, and this function call was added. This time the video was displayed in the window as expected. We had some difficulty with the video freezing, but this occurred even with the standard webpage access. This led us to the conclusion that it was a problem with the ActiveX control. Another interesting thing is that accessing the camera this way does not require the security login required by the standard interface, kind of scary considering it was relatively easily to implement.

FRAMEWORK SOFTWARE

The framework software served as a base from which we were able to add each of the separate modules. For example, the motion control was added to one instance of the framework, communication was developed into another, and everything was integrated into another. The Motorola software was developed in the Codewarrior SDK provided by Metrowerks. The basic framework provided functions to control the I/O pins, PWMs, timers, and set up the clock phased locked loop (PLL).

GUI WRAPPER



The GUI for this project was implemented using a dialog-based MFC application and contained four main portions: the joystick control code, the serial port control code, the video interface code, and code for a supporting dialog box. The joystick control code is the code described by the PC motion control section above. A function within this code was called from a timer interrupt within the GUI to read the updated command from the joystick. Then, the command was sent, using the serial port control code, over the serial link to the car. The video interface code was only called in the constructor of the GUI to establish a link with the camera and display the video. The supporting dialog box was displayed by selecting "Transmission Settings" from the system menu. This dialog box allows the user to change the COM port over which

the program is transmitting commands.

CONCLUSIONS AND LESSONS LEARNED

Overall this project was successful. The only notable failure was the inability to implement the embedded wireless interface. This component was successfully replaced with a serial interface, and the rest of the project functioned as planned. Given additional time and resources we would have been able to incorporate the necessary modifications for a wireless connection.

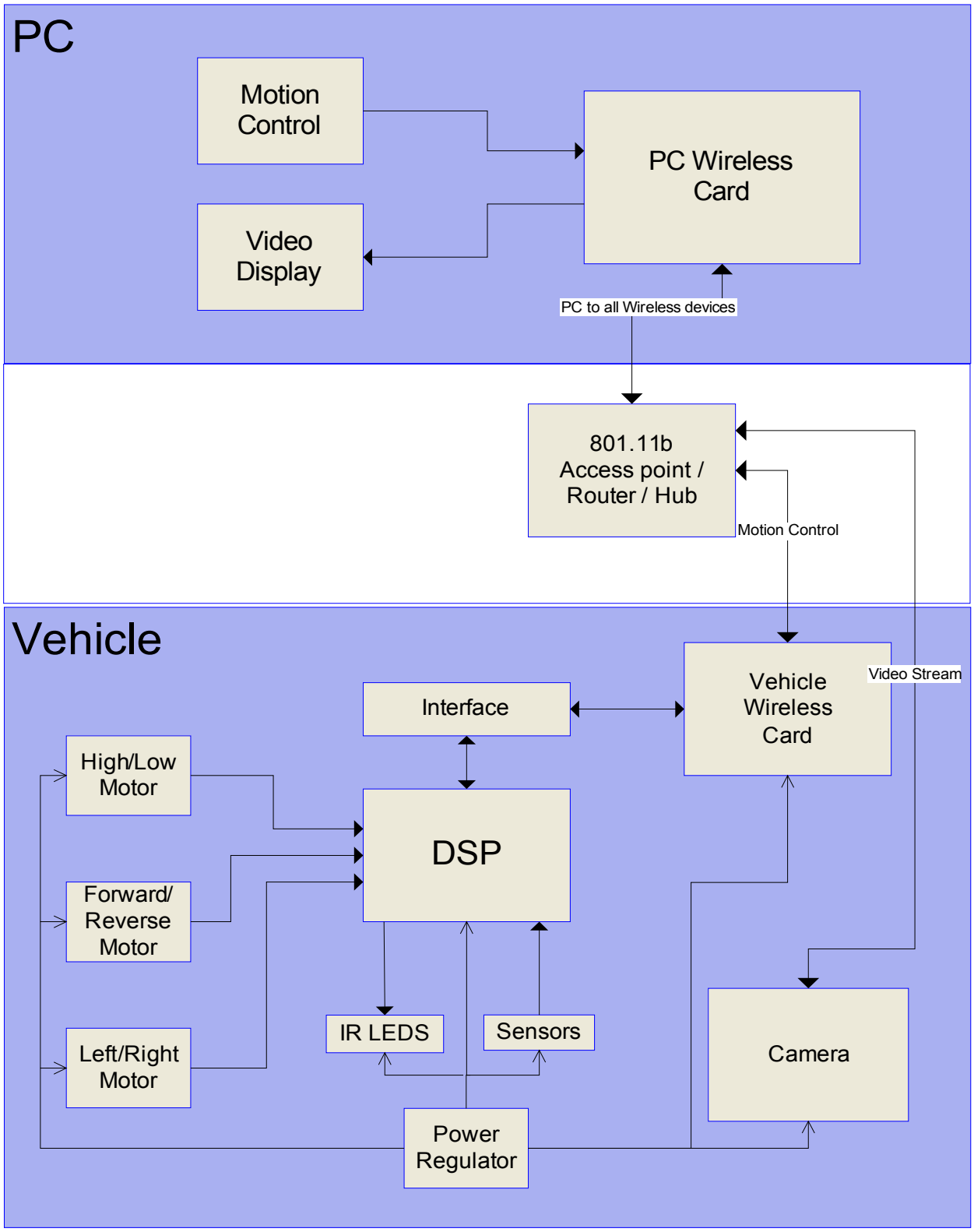
Looking back other modifications become obvious. For example, since we no longer need the DSP to perform MPEG compression, we could have used a microcontroller with better external memory support. With this capability, we may have been able to implement the 802.11 wireless interface. We should have performed more rigorous circuit analysis to find the logic error with the H-bridge. We also should have done more analysis of the existing electronics in the car before gutting the electronics. Doing this analysis would have saved us some time trouble shooting the frequencies used to drive the motors.

ACKNOWLEGEMENTS

We acknowledge Magpie systems, L-3 communications and Circuit Graphics for there help and support. We also used typical circuits shown in data sheets from Allegro and Motorola and the sample code provided on MSDN and included with the DirectX SDK.

APPENDIX A

BLOCK_DIAGRAM



APPENDIX B

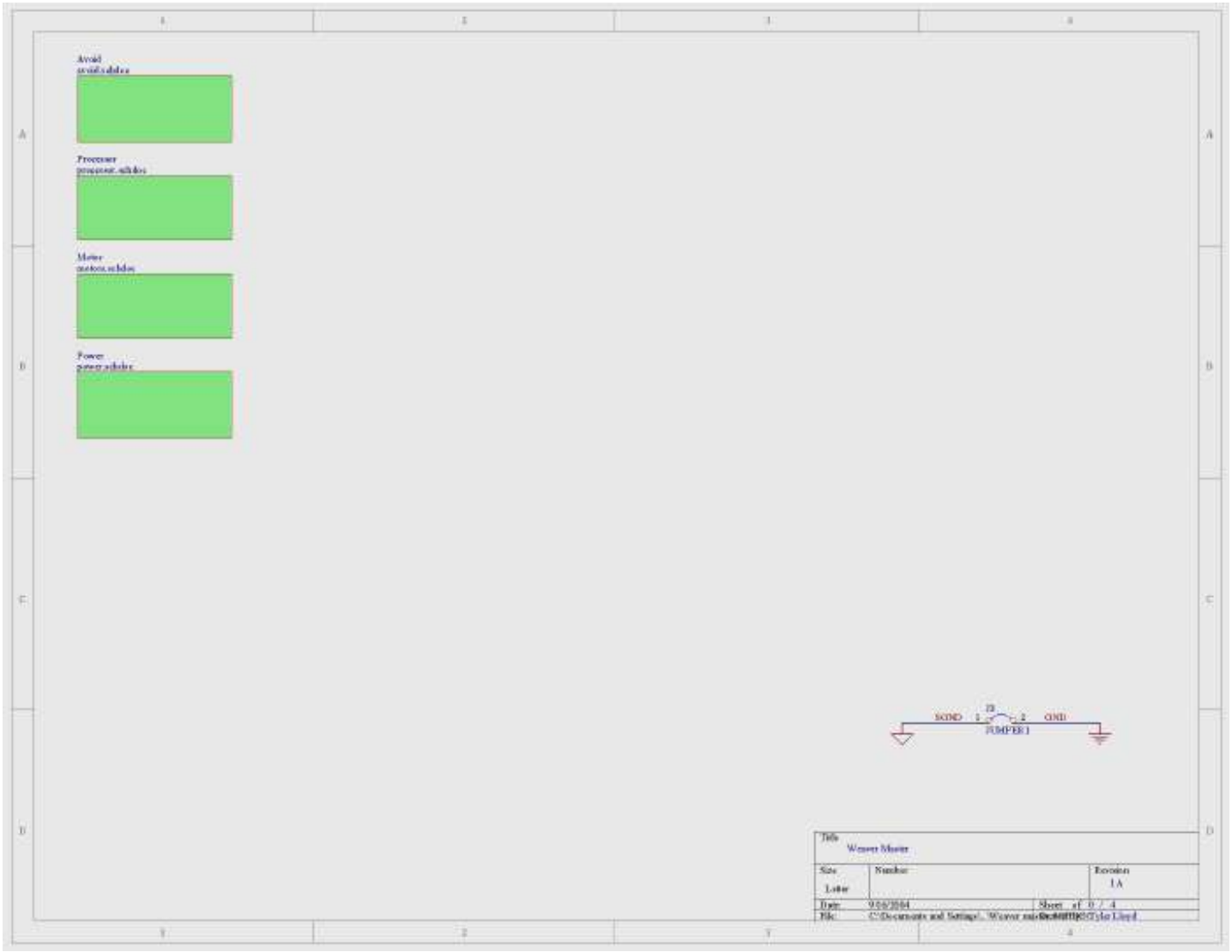
BILL OF MATERIALS

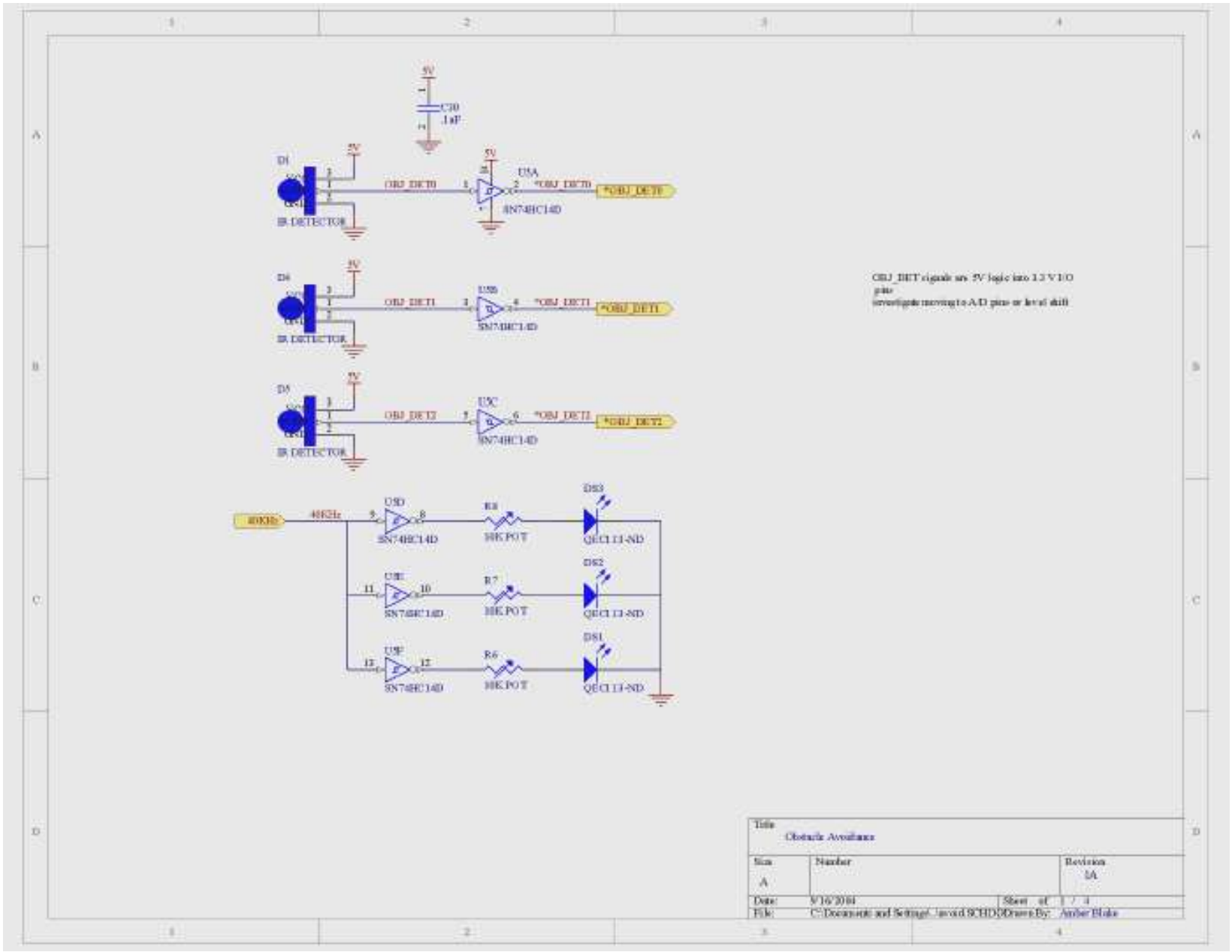
Comment	Description	Designator	Footprint	Manufacturer	Source
Motor	RS-380SH-4045	B1	RB5-10.5	Mabuchi Motor	Radio Shack RC Truck
Motor	Motor, Main Drive	B2	SIP5	unknown	Radio Shack RC Truck
Motor	Motor, Main Drive	B3	SIP6	unknown	Radio Shack RC Truck
Battery	10 C-cells high current	BT1	BAT-2		
Battery	6 AA high current	BT2	BAT-2		
0.1uF	Capacitor, non-polarized	C1	M1206	unspecified	Magpie Systems
10uF	Capacitor, non-polarized	C2	M1210	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C3	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C4	M1206	unspecified	Magpie Systems
10uF	Capacitor, non-polarized	C5	M1210	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C6	M1206	unspecified	Magpie Systems
.22uF 25V	Capacitor, non-polarized	C7	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C8	M1206	unspecified	Magpie Systems
1uF	Capacitor, non-polarized	C9	M1206	unspecified	Magpie Systems
100uF	TPS Series Low ESR Surface Mount Capacitor	C10	M6032	AVX	Magpie Systems
0.1uF	Capacitor, non-polarized	C11	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C12	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C13	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C14	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C15	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C16	M1206	unspecified	Magpie Systems
2.2uF	sm capacitor 805 footprint	C17	M805	unspecified	Magpie Systems
2.2uF	sm capacitor 805 footprint	C18	M805	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C19	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C20	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C21	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C22	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C23	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C24	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C25	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C26	M1206	unspecified	Magpie Systems
*0.1uF	Capacitor, non-polarized	C27	M1206	open	
1uF	Capacitor, non-polarized	C28	M1206	unspecified	Magpie Systems
100uF	TPS Series Low ESR Surface Mount Capacitor	C29	M6032	AVX	Magpie Systems
.1uF	Capacitor, non-polarized	C30	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C31	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C32	M1206	unspecified	Magpie Systems
0.1uF	Capacitor, non-polarized	C33	M1206	unspecified	Magpie Systems
IR DETECTOR	Bipolar Photodetector	D1	SIP3	Panasonic	Digi-Key
Red LED	Surface mount LED	D2	DIO1206	unspecified	Magpie Systems
Red LED	Surface mount LED	D3	DIO1206	unspecified	Magpie Systems
IR DETECTOR	Bipolar Photodetector	D4	SIP3	Panasonic	Digi-Key
IR DETECTOR	Bipolar Photodetector	D5	SIP3	Panasonic	Digi-Key
RLS4148	sm 1n4148 signal diode	D6	DIO1206	unspecified	Magpie Systems
SS13	1 amp Schottky Diode sm	D7	DIO1206	unspecified	Magpie Systems
QEC113-ND	Typical INFRARED GaAs LED	DS1	LED-0	Fairchild	Digi-Key
QEC113-ND	Typical INFRARED GaAs LED	DS2	LED-0	Fairchild	Digi-Key
QEC113-ND	Typical INFRARED GaAs LED	DS3	LED-0	Fairchild	Digi-Key
DB9	Male DB-9	J1	SIP9	unspecified	Magpie Systems
GPIOD+	Connector	J2	SIP10	open	
JUMPER1	Jumper	J3	SIP2	wire	

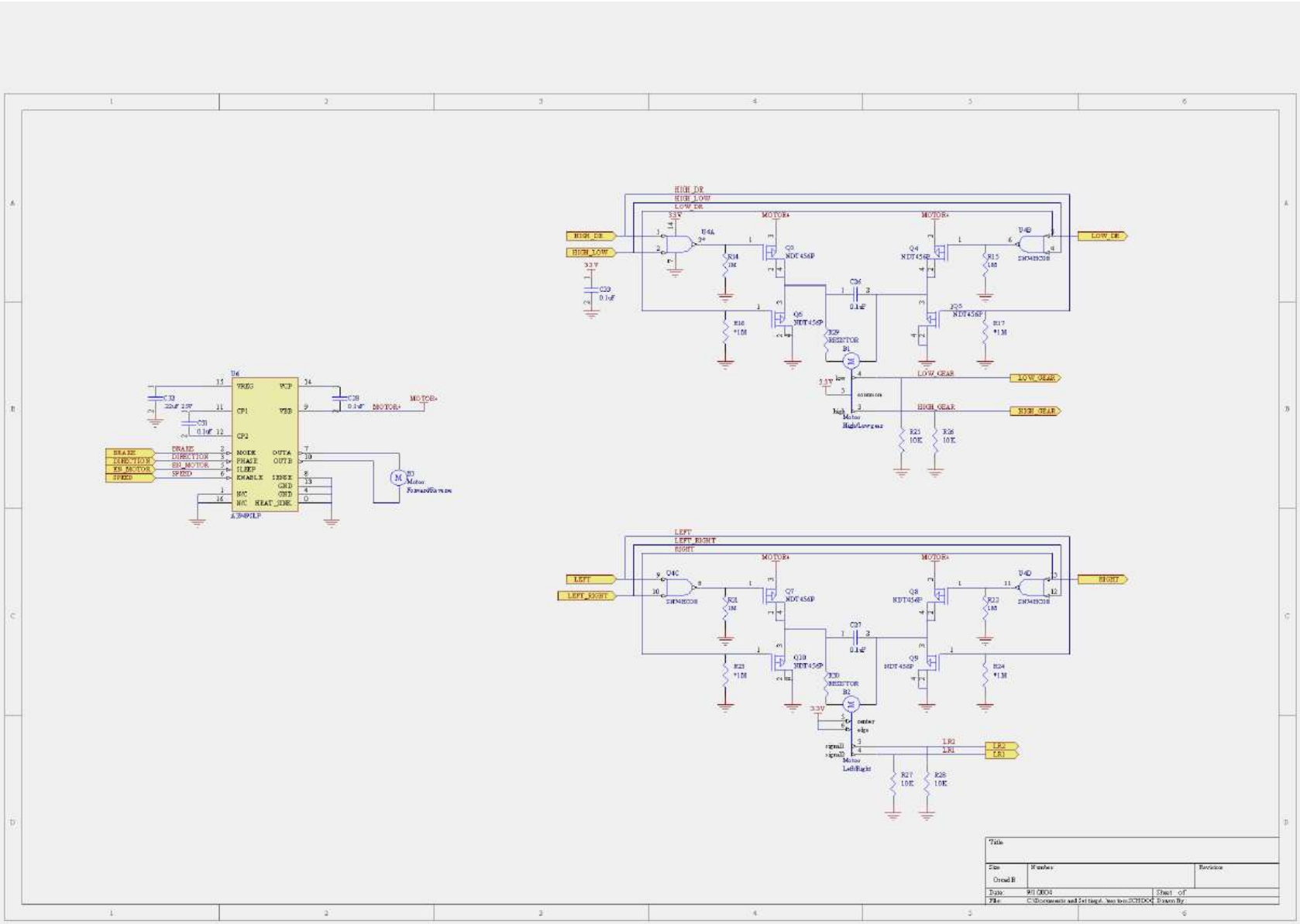
*Stand Alone	Programming Switch	J4	SIP2	unknown	Radio Shack RC Truck
*INPACK	Test-point	J5	SM1	open	
CLK_OUT	Test-point	J6	SM1	open	
GPIOE4	Test-point	J7	SM1	open	
PCMCIA connector	Single thru-hole PCMCIA socket	P1	AMP535659-1	unspecified	Magpie Systems
Camera Power	Connector for the Dlink Camera	P2	SIP2		
DSP807	Sherlock 2.00 mm 10 pin	P3	MOL 35362	Molex	Magpie Systems
NDT456P	P channel MOSFET	Q1	SOT-223	unspecified	Magpie Systems
NDT456P	P channel MOSFET	Q2	SOT-223	unspecified	Magpie Systems
NDT456P	P channel MOSFET	Q3	SOT-223	unspecified	Magpie Systems
NDT456P	P channel MOSFET	Q4	SOT-223	unspecified	Magpie Systems
NDT455N	N channel MOSFET	Q5	SOT-223	unspecified	Magpie Systems
NDS356P	P channel MOSFET 1.6A, 30 V	Q6	SOT-23	unspecified	Magpie Systems
NDT456P	P channel MOSFET	Q7	SOT-223	unspecified	Magpie Systems
NDT456P	P channel MOSFET	Q8	SOT-223	unspecified	Magpie Systems
NDT456P	P channel MOSFET	Q9	SOT-223	unspecified	Magpie Systems
NDT456P	P channel MOSFET	Q10	SOT-223	unspecified	Magpie Systems
RESISTOR	RESISTOR	R1	M2512	open	
10K	RESISTOR	R2	M1206	unspecified	Magpie Systems
10K	RESISTOR	R3	M1206	unspecified	Magpie Systems
620 Ohm	RESISTOR	R4	M1206	unspecified	Magpie Systems
620 Ohm	RESISTOR	R5	M1206	unspecified	Magpie Systems
10K POT	VARIABLE RESISTOR	R6	VARRES	unspecified	Magpie Systems
10K POT	VARIABLE RESISTOR	R7	VARRES	unspecified	Magpie Systems
10K POT	VARIABLE RESISTOR	R8	VARRES	unspecified	Magpie Systems
10 K	RESISTOR	R9	M1206	unspecified	Magpie Systems
*1M	RESISTOR	R10	M1206	open	
RESISTOR	RESISTOR	R11	M2512	open	
10K	RESISTOR	R12	M1206	unspecified	Magpie Systems
10K	RESISTOR	R13	M1206	unspecified	Magpie Systems
*1M	RESISTOR	R14	M1206	open	
1M	RESISTOR	R15	M1206	unspecified	Magpie Systems
*1M	RESISTOR	R16	M1206	open	
*1M	RESISTOR	R17	M1206	open	
1M	RESISTOR	R18	M1206	unspecified	Magpie Systems
10K	RESISTOR	R19	M1206	unspecified	Magpie Systems
100K	RESISTOR	R20	M1206	unspecified	Magpie Systems
10K	RESISTOR	R21	M1206	unspecified	Magpie Systems
100K	RESISTOR	R22	M1206	unspecified	Magpie Systems
10K	RESISTOR	R23	M1206	unspecified	Magpie Systems
10K	RESISTOR	R24	M1206	unspecified	Magpie Systems
10K	RESISTOR	R25	M1206	unspecified	Magpie Systems
1M	RESISTOR	R26	M1206	unspecified	Magpie Systems
1M	RESISTOR	R27	M1206	unspecified	Magpie Systems
9.1 M	RESISTOR	R28	M1206	unspecified	Magpie Systems
1M	RESISTOR	R29	M1206	unspecified	Magpie Systems
1M	RESISTOR	R30	M1206	unspecified	Magpie Systems
ON_SWITCH	Switch	S1	ON_SWITCH	unknown	Radio Shack RC Truck
A3949SLP	Allegro, DMOS Full-Bridge Motor Driver	U1	TSSOP16	Allegro	Allegro
DSP56F807	16 bit digital signal processor	U2	DSP56F807	Motorola	Magpie Systems
SN74HC08	Quad 2-input Positive AND Gates	U3	MSO14	TI	Magpie Systems
SN74HC14D	Hex inverter	U4	MSO14	TI	Magpie Systems
SN74HC14D	Hex inverter	U5	MSO14	TI	Magpie Systems
5V 3A ISR	5V 3A Integrated Switching Regulator	V1	PSIP-T12	TI	Magpie Systems
3.3V 2A ISR	3.3 V 2 A Integrated Switching Regulator	V2	PSIP-T12	TI	TI
8.0MHZ connectors MAX3221	Crystal Cgrid locking wire to wire connector RS-232 level shifter	X1	MC505	Epson Molex Maxim	Magpie Systems Magpie Systems Maxim

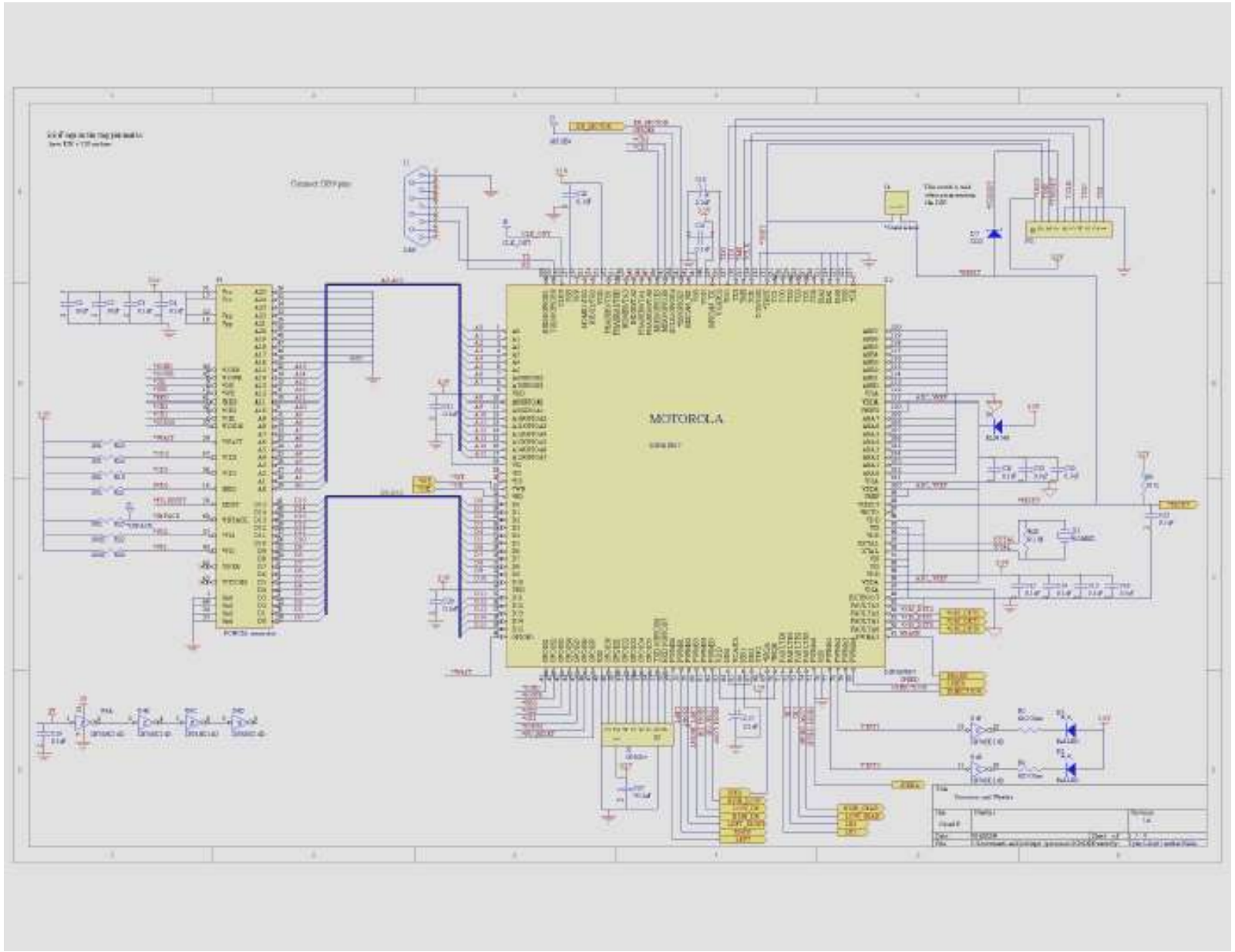
APPENDIX C

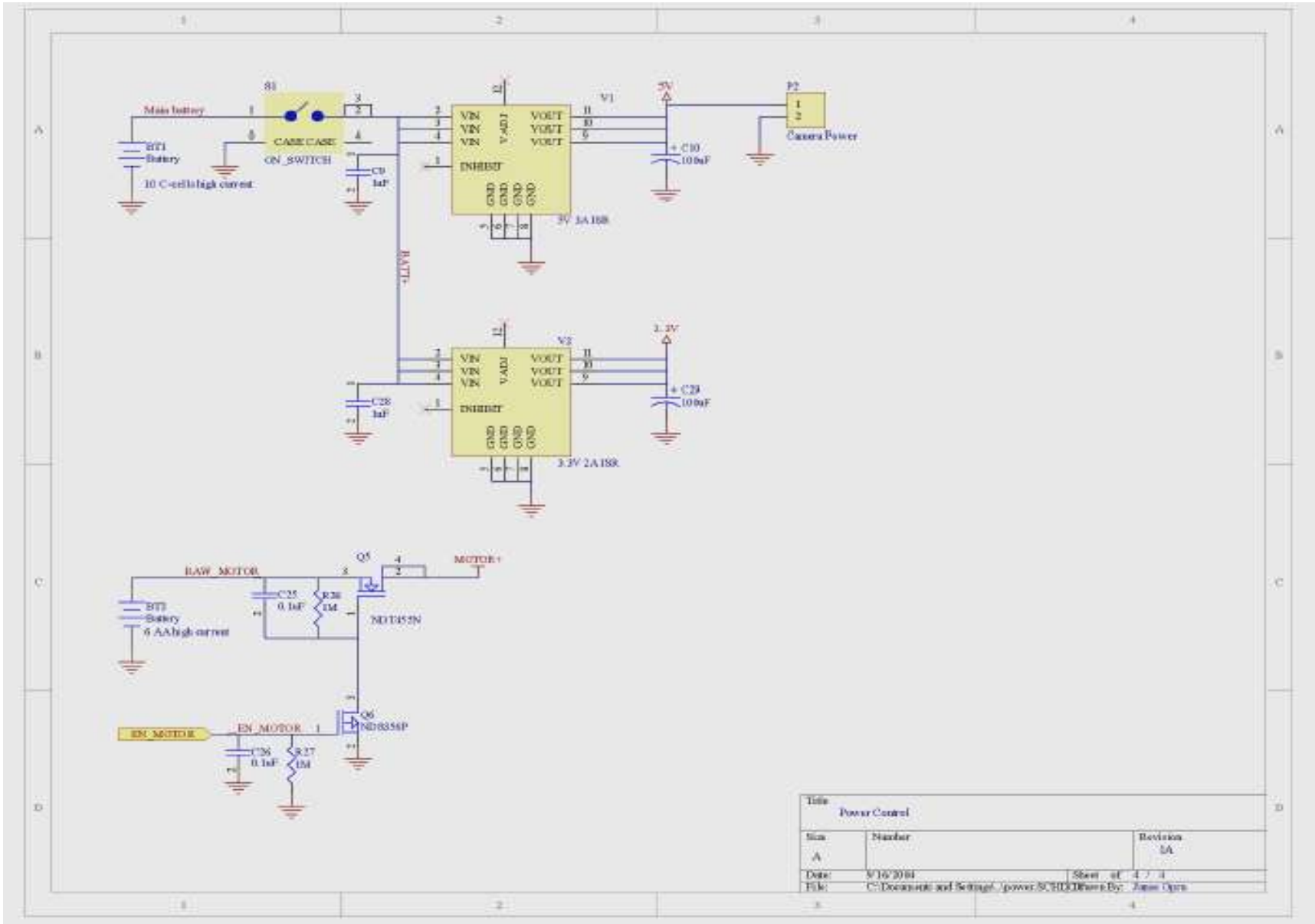
WEAVER REVISION 1A SCHEMATIC




5V
C30
.1uF
D4
IR DETECTOR
OBJ_DET0
U5A
*OBJ_DET0
SN74HC14D
D6
IR DETECTOR
OBJ_DET1
U5B
*OBJ_DET1
SN74HC14D
D5
IR DETECTOR
OBJ_DET2
U5C
*OBJ_DET2
SN74HC14D
40KHz
40KHz
U5D
SN74HC14D
R8
10K POT
DS3
QEC113-ND
U5E
SN74HC14D
R7
10K POT
DS2
QEC113-ND
U5F
SN74HC14D
R6
10K POT
DS1
QEC113-ND
OBJ_DET signals are 5V logic into 3.3 V I/O
pins
investigate moving to A/D pins or level shift
Title
Obstacle Avoidance
Size
A
Number
Revision
1A
Date:
9/16/2004
Sheet of
3 / 4
File:
C:\Documents and Settings\Jarod\SCHDOCDrawn By:
Amber Blake


HIGH_DR
HIGH_LOW
LOW_DR
3.3V
MOTOR+
HIGH_DR
HIGH_LOW
LOW_DR
LOW_GEAR
HIGH_GEAR
LEFT
LEFT_RIGHT
RIGHT
BRAKE
DIRECTION
EN_MOTOR
SPEED
MOTOR+
Title
Size
Number
Revision
Date:
Sheet of
File:
Drawn By:


MOTOROLA


Main battery
BT1
Battery
10 C-cells high current
S1
CASE CASE
ON_SWITCH
C9
1uF
V1
VIN
VIN
VIN
V-ADJ
VOUT
VOUT
VOUT
INHIBIT
GND
GND
GND
GND
5V 3A HR
5V
+ C10
100uF
P2
Camera Power
BATT+
V2
VIN
VIN
VIN
V-ADJ
VOUT
VOUT
VOUT
INHIBIT
GND
GND
GND
GND
3.3V 2A HR
3.3V
+ C29
100uF
C28
1uF
RAW_MOTOR
Q5
MOTOR+
C25
0.1uF
R26
1M
BT2
Battery
6 AA high current
NDT455N
Q6
NDS8356P
EN_MOTOR
EN_MOTOR
C26
0.1uF
R27
1M
Title
Power Control
Size
A
Number
Revision
1A
Date:
9/16/2008
Sheet of 4 / 4
File:
C:\Documents and Settings\...\power.SCHDOC
Drawn By: