Final Report

Trixie & The Four Pimps

David Bonal
Michael Davis
Jacob Dunken
Robert Niemand

5-11-99

TABLE OF CONTENTS

ABSTRACT

TRIXIE is an autonomous self-contained robot capable of being able to navigate a maze. The robot is designed to be able to compete in the Trinity College National Robot Fire Fighting contest. TRIXIE is designed as a tread based robot, which uses tank-like treads instead of wheels for locomotion. There are many components that enable TRIXIE to be able to navigate the maze. These components involve two levels, hardware and software. At the hardware level, TRIXIE is composed of external sensors such as fire sensors, white-line sensors, and a 3.5 kHz sound sensors. Also at the hardware level, there are components that allow the robot to have locomotion. These components consist of motors, H-Bridges, and Pulse Width Modulation controllers. The control of the entire robot is accomplished through the use of a Motorola 68HC11 microcontroller. Even with the use of all of the various hardware components, TRIXIE is still not complete without software to control all of the hardware. TRIXIE's control was written in the C programming language that interfaces well with the 68HC11. TRIXIE used a proportional closed-loop controller system to be able maintain a constant desired speed throughout the maze. The following document will describe in detail all of the various components of the robot, and how the components integrated to achieve a fully functioning robot.

INTRODUCTION

TRIXIE is a self-contained autonomous robot capable of navigating a maze to extinguish a fire. Being able to navigate a maze by a self-contained robot is a daunting task. TRIXIE has many on-board components, which allow it to be able to achieve the task of navigating a maze. TRIXIE's design includes all closed-loop control, no wire-wrapped boards, connectors to all boards, and all control code written in the C programming language. TRIXIE's primary goal is to be able to compete in the Trinity College National Robot Fire Fighting contest (See Appendix 1 for a diagram of the maze).

TRIXIE was designed as a track-based robot. In other words, TRIXIE uses tank-like treads instead of wheels for locomotion. The track-based design uses a cog belt and cog pulley connected to a motor. The main design consideration for using a track-based robot was ease of navigating over obstacles within the maze. Much of the control required to operate a track-based robot is derived from the same control for operating a wheel-based robot. Due to using a track-based design, TRIXIE's base was designed as a square. The biggest reason for using a square base is that it proved easier to be able to keep the tracks straight with respect to the base. Using a non-square base would have proved to be an engineering nightmare to keep the tracks as square as possible.

TRIXIE incorporated several different types of sensors in order to respond in the maze environment. TRIXIE required sensors for determining the distance from walls in the maze, sensors for recognizing the white-lines in the maze, sensors for determining intensity and location of the fire, and a sensor for recognizing a (3.5kHz) sound wave to start the robot. TRIXIE uses three SHARP GP2D12 sensors for determining the distance from the walls. One GP2D12 is placed on the front of the robot to sense distances from front walls. The other two GP2D12 sensors were placed on each side of the robot to center the robot within the halls of the maze. Being able to sense the white-lines in the maze was accomplished by using a single Panasonic PN168 phototransistor. An incandescent light was mounted on the underside of the robot near the PN168 phototransistor. Light produced from the incandescent light would reflect from a white surface thus triggering a 5-volt output from the PN168 phototransistor. The white line sensor technology was used for two of the three fire sensors. Two of TRIXIE's fire sensors used PN168 phototransistors with an ambient light filter installed in front of the transistor. The PN168 fire sensors were separated from each other to provide binocular vision. The separation of the sensors provided TRIXIE with the capability of scanning a room with a fire to determine where exactly the candle is in the room. The other fire sensor was a HAMAMATSU UVTron flame sensor obtained commercially from the HAMAMATSU Corporation. The UVTron responds to ultraviolet radiation emitted by fire. The UVTron is mounted on the front of the robot, and is TRIXIE's primary fire sensor. In order to activate TRIXIE a (3.5kHz) sound generator is used. TRIXIE has a built in microphone and tone detector for recognizing the (3.5kHz) tone. All of the sensors give TRIXIE the ability to know the status of the environment that TRIXIE is navigating in.

Motor control on the robot was accomplished through an orchestration of several devices. Each motor is controlled by a National Semiconductor LMD18200 H-Bridge. The Pulse Width Modulation (PWM) input to the LMD 18200 comes from an external Altera 7128 PWM controller. The Altera 7128 is programmed to take a duty cycle input and produce the corresponding output PWM signal for a specific motor. The complexity of operating the motors to the robot was hidden in the intricacies of the Altera PWM controller.

TRIXIE's primary controller was the Motorola 68HC11 microcontroller. All of the signal lines from the sensors were routed into the HC11. The HC11 was able to make decisions based on the sensor information to navigate the robot within the maze. The HC11 memory-mapped the Altera 7128. The 7128 is the robot's PWM controller, as well as its pulse accumulator. Each of TRIXIE's motors have a Hewlett-Packard slotted disk distance encoder connected to the motor shaft. Output from the encoders is routed into the Altera 7128 pulse accumulator. The HC11 can then make speed decisions using a proportional controller based on the desired speed of the robot, as well as the current speed of the robot. Current speed information is gathered from the pulse accumulators. Using the 68HC11 provided a very robust embedded microcontroller that is easy to program and integrate into a robot platform.

TRIXIE's goal is to be able to detect and extinguish a flame within a room inside a maze. Extinguishing the flame is accomplished through the use of a simple 12-volt motor connected to a model airplane propeller. The propeller produces enough air to eliminate the flame on a candle. The HC11 controls turning the fan on and off. In a real world environment, blowing air on a fire is highly impractical. But, since TRIXIE was designed to participate in the fire fighting competition, only candles were used for flame production. Blowing air on a candle blows out the fire with relative ease. Using a fan for fire extinguishing is a useful way to eliminate a fire in the controlled conditions that TRIXIE is designed to be in.

This document will provide detailed information on how TRIXIE incorporated all of the various components to produce a fully-functional robot that is capable of navigating a maze, detecting a fire within a maze, and extinguishing the fire.

CHASSIS

TRIXIE features tank-like quad track drive. There are two main reasons why the tank drive system was chosen. The first reason is that the tank drive offers the very beneficial differential control (exemplified by the fact that differential drive allows in place turns unlike a car-type drive system). Differential drive offers easy tight corner navigation in the maze and simple software control.

The second reason is the great amount of stability quad track offers versus two-wheel differential drive. The tank drive system can easily be implemented to have stability comparable to a four-wheel system without necessitating additional ball rollers, casters, or skids. Most differential drive systems just have two drive wheels on the ground with added caster(s), roller(s), or skid(s) for stability. With tank drive, TRIXIE

can navigate up or down ramps and small discontinuities very easily without the worry of a malfunctioning caster or catching a skid.

Designing the track drive can be very complicated, though, because of the special considerations it proposes. When turning, torque is applied to the track tread in an azimuthal direction very frequently causing the tread to jump track. Perhaps the most effective means to control track loss is tread tension and the positioning of guiding idler wheels. TRIXIE was designed with quad track drive, requiring 3 idler wheels for each track. The idler wheels utilized were modified bearings; each bearing had to have side lips to retain the track over the top and had to fit snuggly on the axle, so washers were brazed/welded to each bearing and a brass sleeve was press fit in its center.

Two Pittman GV9434H187-R4 motors (with encased gear reduction and 500 count encoders) were used to drive TRIXIE. Each was mounted inversely with L-brackets on the rear of a 10" X 10" thin aluminum plate. The motor output shafts were fitted with notched gear pulleys matched to a ribbed, nylon reinforced, rubber belt (sized to meet the circumference of idler positioning). Angle aluminum runners were bolted lengthwise on the underside of the chassis and drilled to support the bottom axles. Aluminum brackets on the top-front of the base were slotted for the last axle. This became the tension idler axle. Finally, each idler was mounted onto each side of 7/16" all thread axles situated in a quad (inverted trapezoidal) fashion. See Appendix 3, for a diagram showing this setup.

This chassis became the base for the whole robot. Over all, TRIXIE has three 10" X 10" vertically adjustable levels: one is the chassis, the second contains space for circuitry, and the third has space for more circuitry. An aluminum plate (for strength and to shield against EMI) divides the chassis and second level. The second and third tiers are divided by plexi-glass plates. TRIXIE's top is also plexi-glass. Four vertical all thread bolts provide pillar support for each layer divider and contribute the benefit of adjustable level height via a nut & lock nut setup.

All in all, TRIXIE sports a very robust aluminum tank-like chassis that supports multi levels of space for her circuitry and peripherals.

CONTROL

Pulse Width Modulation

Pulse width modulation (PWM) is a simple technique for controlling the speed of a DC motor. PWM makes use of a variable duty cycle square wave to drive the inputs of the motor. Electrically, the motor behaves like a low pass filter, so it essentially only "sees" the average value of the square wave present at its inputs. Thus, by modulating the pulse width of the input signal, the motor can be driven with different effective voltages across its input terminals. Since motor speed is directly proportional to the applied voltage, altering the duty cycle of the input square wave alters the speed of the motor. This and the fact that it is vary easy to generate a PWM signal makes PWM speed

control vary effective and simple. TRIXIE makes exclusive use of PWM to control the speed of the drive motors.

H-Bridges

Due to its nature, PWM is typically generated by a microcontroller or digital other logic sources. This causes a problem, however, because CMOS logic cannot provide enough power to drive the motors directly. Thus, a solid state device capable of switching high power based upon CMOS commands is needed. At least one such device exists and is known as an H-bridge. The H-bridge gets its name from the layout of the internal transistors that switch the high power. Four power MOSFET's are arranged in an H pattern around the motor terminals, which form the horizontal crossbar of the "H". Digital control circuitry within the H-bridge drives each MOSFET's gate. The MOSFETS, in turn, control the direction of current flow through the motor via the "H" configuration. Thus, using an H-bridge interface between a DC motor and a PWM source allows control of both motor direction and speed. TRIXIE uses the National Semiconductor LMD18200T H-bridge rated for 55 volts, 3 amps. This device offers excellent performance coupled with extreme reliability and low static sensitivity. The LMD18200T has three CMOS inputs and two power outputs. The inputs are the PWM, direction, and brake signals from the PWM controller and the outputs are the positive and negative drives for the motor. Although the LMD18200T is capable of interfacing directly between the motor and the PWM controller, Schmidt trigger inverters were used to buffer the inputs for noise reduction. The data cables between the H-bridges and the PWM controller were shielded to further reduce noise.

Implementing a PWM Controller in Altera

Two approaches exist for implementing a PWM controller: one is a software-based approach using a microcontroller such as the HC11, and the other is a hardware-based approach using a PLD such as Altera. The software/microcontroller approach is very straightforward and easier because there is no application specific hardware to develop. The major disadvantage of this approach, however, is the huge overhead PWM places on the microcontroller time budget, unless the microcontroller offers dedicated PWM output channels. Else, having to service the interrupts necessary for two channels of PWM leaves the microcontroller with little time to perform other tasks related to robot operation. The hardware approach is much more difficult due to design considerations, but its major advantage is that it removes the burden of PWM from the microcontroller, leaving it more available for other tasks. Hardware PWM is also, arguably, a cleaner solution because with a hardware PWM controller, PWM becomes transparent to the microcontroller. There is no complex PWM software cluttering the robot control code. For these reasons, a PWM controller was implemented in an Altera PLD.

Given a working knowledge of Altera Hardware Descriptor Language (AHDL), implementing a PWM controller is a fairly simple task. The heart of the system is a free-running counter that repeatedly counts from 0 to 99. This counter is clocked at a 100

times the rate of the desired PWM frequency. For example, if the desired PWM rate were 100Hz, the counter would be clocked at 10kHz. This gives each count value a resolution of 1% of the maximum duty cycle. The counter is then transformed into a PWM controller by adding logic to compare the current count value to the desired percent duty cycle. If the count is less than the desired duty cycle (DDC) the PWM output will be high. As soon as count exceeds the DDC, the PWM output goes low.

Given a maximum duty cycle of 100%, the DDC must be passed to the controller as a seven-bit number. This is no problem since most microcontrollers, including the HC11, have eight-bit ports and eight-bit data busses. Making the DDC an eight-bit number adds more functionality to the PWM controller. The MSB (most significant bit) can be used to encode the desired direction of rotation, and a DDC of 101 to 127 can be used to represent the brake command. Using this input convention along with a little more logic allows a fully functional PWM controller to be implemented in an Altera PLD. This very PWM controller described was implemented in TRIXIE, and the complete AHDL code is listed in Appendix 15.

Speed Sensing and Optical Encoders

In order for a closed-loop controller to be effective, accurate motor velocity and position data must be available. This data is provided by the optical encoders which are integrated in the Pittman motors used to drive TRIXIE. These encoders provide 500 counts per motor shaft revolution. Due to the 5.9:1 integrated gear reduction on the Pittman motors, each turn of the output shaft provides 2950 counts for the encoder. The resulting resolution is 0.122^{o} of output shaft rotation per encoder count. Thus, by counting encoder pulses over a specified period of time, accurate motor velocity and position data can be obtained.

Two methods exist for obtaining an encoder count value. The first method is utilizing a frequency to voltage converter to translate the encoder pulse train frequency to an analog signal. The analog signal may then be converted into microcontroller-usable information via an A/D converter. This method is the simplest to implement, but has several disadvantages: commercially available frequency to voltage converters do not exhibit very linear conversion characteristics and offer low conversion resolution over 5 volts. These disadvantages produce error in the digital value produced by the A/D converter and, in turn, lead to an inaccurate velocity or position reading which adversely affects closed loop control. The second method, known as pulse accumulation, solves the error problem by actually counting all the pulses that occur during a given time. Although this method is more accurate, it is harder to implement. At full motor speed, TRIXIE's encoder pulse train frequency is close to 60kHz. This means that a very wide binary ripple counter must be used and the count must be read often to prevent premature overflow (a 16 bit counter will overflow in approximately 1 second at the above pulse rate). It is possible to cascade discrete ripple counters to obtain the desired width, but a better solution is to implement the counter in an Altera PLD.

Pulse Accumulation Systems in Altera

It is very easy to develop a pulse accumulator (PAC) in Altera. The PAC's are simply N bit (in this case 16 bit) counters, which are clocked by the encoder outputs. AHDL makes the development of counters a trivial task. Adding logic to limit the maximum count and provide a counter reset yields a fully functional PAC. AHDL code for the PAC's implemented in TRIXIE is shown in Appendix 16.

Integrated PWM Controller and PACs in Altera

Using a large Altera device, it is a straightforward task to integrate two independent channels of PWM control and two PAC's into a single package. TRIXIE features this approach of creating an integrated motor control interface device with the use of an Altera EPM7128.

AHDL has provisions to create programs to implement independent functions then integrate them into a single device. This ability lends itself well to modular program design. Modular programming is very beneficial because it allows individual subsystems (functions) to be developed & tested separately and then once the individual subsystems are debugged, the integration process can proceed with very few errors. Without modular design, creating a working, bug free device like the motor control interface would be difficult.

The modular approach described was implemented in TRIXIE. The PWM core described was interfaced to a register that holds the current DDC and to reset logic, and the core's outputs were sent to I/O pins on the device with duplication to yield left and right channels of PWM. Next, a clock divider module was connected to the input clock (HC11 E-clock) and it's output were used to drive the PWM cores. Two PAC modules were added along with reset logic to serve the motor encoders. And finally, an interface module was included to allow the Altera device to communicate with the HC11 over the HC11's address/data bus. This module, along with the interfacing process, will be described shortly. The Altera code for the integrated device is shown in Appendix 18. A block diagram depicting the module connections is shown in Appendix 7.

Interfacing the Altera Controller to the HC11

There are two obvious methods for interfacing the Altera control system to the HC11. The first interfacing method involves using the parallel I/O ports. This method is simple from a hardware perspective, but it is a coding nightmare. Each individual control signal; such as read/write, reset, chip select, etc. must be manually generated by the program for communication with the Altera device. Besides being a coding hassle, this method uses the majority of the HC11's parallel I/O capability. Having a lot of free parallel I/O is absolutely essential to the robot. This is because the HC11 must process and generate many digital signals during operation. (The most effective way of handling digital I/O is through the parallel ports.)

The second interfacing method is memory mapping. Memory mapping involves placing the Altera on the HC11's address and data bus. This process is complicated from a hardware perspective, because an interface module must be created in Altera and bus-timing issues must be addressed. The main advantage of this method is simplicity of accessing the device. All of the control signal generation becomes transparent to the program communicating to the Altera device. To the program, the whole process becomes as simple as a memory read or write.

The memory mapping process begins by considering the bus timing issues. The HC11 has very stringent requirements on when activity is allowed on the bus. The device being connected to the bus must be able to meet these requirements or it will not work. The EPM7128, however, is a very fast device and has no trouble meeting the timing requirements imposed by the HC11. The next step in the memory mapping process is the development of an interface module in Altera. This module will be responsible for handling all of the bus transactions between the Altera and the HC11. It must recognize when the HC11 is accessing the Altera device and distinguish between read and write cycles; only driving the bus at the appropriate times. The interface must also recognize the destination of the read or write within the device and either direct data to or retrieve data from that destination. Like the Altera code developed so far, creating this interface in a simple task. The AHDL code to implement the interface is shown in Appendix 18.

After the interface is in place, an addressing scheme must be developed. Through the use of the address decoding logic already on the HC11 EVBU expansion, the Altera controller was assigned a base address of 0xB100. The Altera controller has eight internal registers: right DDC, left DDC, right PAC high bits, right PAC low bits, left PAC high bits, left PAC low bits, right PAC reset, and left PAC reset. These registers were assigned memory address in the interface module. They were given addresses of base +0, +1, +2, +3, +4, +5, +6, and +7 respectively. Thus, to write a new DDC to the left PWM controller, the HC11 simply rights the value to address 0xB001. The DDC registers are bi-directional (read and write), the PAC registers are read only, and the reset registers are bi-directional (always read 0xA5; write to reset).

Finally, the only step remaining is the physical connections between the Altera controller and the HC11. The Altera needs the eight data bits, the three low order bits from the expansion decoded HC11 address, the E-clock, a chip select, the read/write line, and finally, the reset line. The address, data, r/w, E, and CS lines were run together in a ribbon cable. But the reset line was, and must be, run in a separate shielded cable because it is very sensitive to noise. A block diagram of the Altera/HC11 interface is shown in Appendix 8.

SENSORS

Proximity Sensors

Proximity sensing was accomplished with a simple setup of three infrared transceiver pairs. Sharp has developed a great analog transceiver pair (GP2D12) that sells for about $5.°°. The GP2D12 is powered by 5 volts and uses IR triangulation to

output an accurate analog voltage swing of 2 ½ volts over a 10 to 40 cm range. This sensor package was used because of its low cost, clean analog distance measurement at high sampling frequencies, and because the triangulation methodology tends to yield relatively color & texture insensitive proximity sensing. One GP2D12 was used to provide data for the left side of TRIXIE, another for the right, and the third for the front. The two side GP2D12's were placed (on TRIXIE's second layer – see Functional Hardware Layout) in a way that the 10 cm near limit would not be compromised and then aimed at 45° angles from the wall normal to avoid an output voltage maximum at that normal. Then the front GP2D12 was placed (on the first level) and recessed as well to avoid the 10-cm near limit. Each sensor's output simply needed to be buffered and then fed into the first four channels of the HC11 A/D converter with no gain. This method proved to be simple and effective, yet special consideration had to be made to avoid noise coupling from the motor level & chassis onto the front GP2D12's wiring.

In the final design, instead of trying to utilize twisted pair, fat shielded wire, or other physical noise suppression for the front GP2D12, a software fix seemed to be the simplest. The A/D was set to continuously scan (every 16 μs) its first four channels when TRIXIE was in her navigation routines, so it was easy and time affordable to compare readings for the front sensor output five times to evaluate front obstacle presence. With the software fix, 99% of the noise spikes were ignored instead of false triggering obstacle avoidance routines.

All in all, this sensor setup yielded reliable proximity and obstacle sensing even when TRIXIE traveled at her maximum speed.

Line Sensor

TRIXIE was able to discern white lines at room entrances via an active illumination phototransistor arrangement. Because of its simplicity and clear voltage differentiation between reflective and non-reflective surfaces, the PN168 phototransistor was used. It needed 5 volts on the collector and a 10k resistor from the emitter to ground. The photocell is the transistor base, so photo detection triggers emitter current. To make the line sensor accurate even in low light conditions, a mini incandescent bulb was mounted to illuminate TRIXIE's undercarriage. All that was to be done was take the voltage from the PN168 emitter and feed it into an op-amp stage with variable gain. A gain range of 50 and up was used to adjust the detector to various lighting conditions. Texas Instruments makes a 5 volt single supply quad op-amp package (TLC274) that was perfect for this application, so the emitter voltage was routed from the line sensor (on the bottom level of TRIXIE – see Functional Hardware Layout) into a TLC274 (on the second layer). To yield digital-interface white line detection, the output of the op-amp stage was fed into an inverting Schmidt trigger. The Schmidt trigger output was connected to the HC11 PIA expansion port CA1 for IRQ interrupt service.

In the final design, the connection from the sensor to the TLC274 proposed a noise-coupling hazard so a shielded cable of twisted pair wire was successfully used to route it.

The net result was an accurate and adjustable line sensor that triggers an active low IRQ interrupt within the HC11 upon the crossing of a white line.

Flame Sensors

Infrared Intensity Sensing

TRIXIE features dual spectrum fire sensing: infrared "binocular fire vision" and ultraviolet photon detection. The magnetic disk of a 3-½ inch floppy was found to be the cheapest effective method of optically filtering light above the infrared portion of the spectrum. Using the same PN168 setup used in line detection and adding the disk filter results in a low cost, reliable way of sensing a flame for this application, so group consensus dictated its implementation. To enhance this setup, two 2" non-reflective (black) PVC tubes were used to house the two phototransistors (which were configured exactly like the wiring and signal amplification of the white line sensor arrangement). The tubes were threaded into two tapped holes ½ inch apart on angle aluminum (for mounting) to guarantee a flat surface where the disk filter could be adhered and seal out light leakage. The phototransistors were wrapped in electrical tape for a loose press-fit within the tube and then light sealed out with a dark-colored wax plug poured into the tube end. This binocular configuration allowed for closed loop control within TRIXIE's code to zoom in on a flame. When the flame is to the right of front center, the right sensor indicates higher fire intensity than the left and vice versa. TRIXIE can use this data to control the drive motors in an inverse fashion. As in the line sensor setup, the adjustable TLC274 gain stage of the phototransistor output allows control of near vs. far vision. The op-amp output was fed into the first two channels of the second four-channel group on the HC11 A/D converter for scanning during fire scan routines. These signals connected to the HC11 A/D were also tapped (via buffering) and fed into an inverter with Schmidt triggers to additionally provide a digital interface.

Ultraviolet Intensity Sensing

The other method utilized to sense fire was with the Hamamatsu UVtron™. The UVtron consists of an ultraviolet photon sensitive tube and its driver PC board. The entire unit costs roughly $77.°°, but is well worth the money as it is capable of detecting the flame of a lighter from the reflections off of a white wall or from a direct line of sight at over 5 meters distance. The UVtron was used to sense fire from a distance obviously. (See Appendix 3) for sensor directivity and spectrum analysis courtesy of Hamamatsu.) The UVtron setup requires a voltage supply from 11 to 30 volts DC and the easiest method to provide this was with two 9 volt alkaline batteries in series (3 mA typical consumption yields long battery life). The UVtron brags adjustable ambient ultraviolet noise cancellation circuitry. Using the first cancellation level and enclosing the PC board within an aluminum EMF cage effectively reduces applicable. Another feature of the UVtron is its convenient variety of output: it has digital outputs (O & $\bar{O}$) and a BJT open-collector stage. Each output pulses at a frequency proportional to flame intensity. Group consensus was to use the output O and the open-collector stage. Easy pulse counting resulted by connecting O to the HC11's 16-bit pulse accumulator and a red LED was

used (5 volts was wired through a current limiting resistor to the anode of the LED and the cathode to the UVtron) with the open-collector output for visual interface.

TRIXIE's net fire sensing capabilities were sensitive enough to see the flame from outside the room or at the doorway and also to execute a closed-loop approach to the candle via binocular IR sensors.

Start Buzzer Sensor

Buzzer

Radio Shack sells a 3.5 kHz piezoelectric buzzer for under $3.°° (Cat. No. 273-075) that is very simple to use. This was the method of choice because of low cost and ease. It requires a switch and 9 volt battery only to do its thing. So the result was a switch module which housed the buzzer and battery.

Tone Decoding

National Semiconductor's LM567 ease is a low cost, effective tone decoder that can be configured to lock on a 3.5 kHz signal accurately, so it was TRIXIIE's prime pickins. It was used in conjunction with another Radio Shack piezoelectric buzzer module (Cat. No. 273-092 B). The output of the module (used as a microphone) was amplified by a TLC274 with a gain of 500 and then given a 2-½ volt bias before its connection to the LM567 input. The 2-½ volt bias was necessary because the TLC274 is not rail to rail; it would only produce a half wave rectified signal without the bias. With this configuration, all that was left was to connect the digital tone decoder output to the PIA expansion parallel port on the HC11. The HC11 sits in a wait loop at the beginning of the code after initialization and scans the port pin to see if it is low for a certain amount of time. This is because the output of the configured LM567 goes low upon frequency lock-on and pulses for other frequencies around 3.5 kHz. The net result is upon start signal module activation within 12 inches of TRIXIE, she is sent into navigation algorithm code with rare false triggers even when navigating on the ramp.

Sensor Signal Conditioning: Total Requirements

Two TLC274's

These two quad single supply op-amps from Texas Instruments were used to provide the 3 buffers needed for the GP2D12 outputs and the adjustable gain stage for the line sensor. Four channels were left unused for unseen needs and/or troubleshooting, but were not used in the final design.

See Appendix 13 for the Protel schematic (sigcond.sch) representing this setup.

One TLC274 & One 74HC74

The TLC274 provides the two adjustable gain stages and the two buffers needed for the binocular fire sensors. The 74HC14 (hex inverter w/ Schmidt triggers) handled

the two digital binocular fire sensor outputs and the line sensor leaving three channels for unseen needs and/or troubleshooting. Two of those three channels were used to Schmidt trigger, double invert the CMOS HC11 output used to control TRIXIE's fire fan.

See the Appendix 12 for the Protel schematic (flamer_p.sch) representing this setup.

One TLC277 & One LM567

The TLC277 (Texas Instruments dual single supply op-amp) provided the gain for the tone receiver module output and the 2-½ volt bias for it. The LM567 (National Semiconductor configurable tone decoder) was configured for 3.5 kHz using capacitors and resistor networks.

See Appendix 9 for the Protel schematic (starter.sch) representing this setup.

This schematic and board design show two 5-volt reed relays. One was used to control the fan and the other to turn on or off the front GP2D12 during flame sensing (to avoid IR interference). The relay for the GP2D12 proved to not be needed, so in TRIXIE's final design, this was removed and replaced with an emitter-follower BJT stage to drive the fan reed relay.

How it all fits together

To get a feel for how all of this interfaces physically, see Appendix 5 for a block diagram.

FIRE EXTINGUISHING

TRIXIE features a high speed, high torque DC motor with an AirScrew turbo propeller for the purpose of extinguishing the candle. The motor driven with 12 volts from one of the two 12 volt 2.2 Ah lead acid batteries and the propeller was mounted to the motor shaft with a press fitted pinion gear. The pinion gear has a setscrew to ensure motor shaft stability and connectivity. The motor is a Radio Shack special at a cost of about $3.°° and the propeller setup cost about $5.°° at the local hobby shop. Special precautions had to be taken to prevent the propeller from interfering with the front GP2D12 or the binocular fire sensors. If the propeller were to be in the GP2D12's line of sight, TRIXIE would see a "phantom wall" and execute avoidance routines. Similarly if the propeller were in front of the binocular fire sensors, TRIXIE would take action on faulty data. To avoid these issues, the pinion set screw had to be tightened when the propeller was horizontal AND the motor was at a stable magnetic pole in its rotation; and also the propeller had to be manually positioned horizontally for each run. When the pinion gear is set properly, the motor will stop rotation upon fan turn-off in a position out of sensor sights, thereby ensuring no phantom walls before or after fire extinguishing.

Functional HC11 control of the fan was obtained mainly by use of relays. The CMOS HC11 output was Schmidt triggered, double inverted to remove any noise coupling and then fed into the BJT emitter-follower stage added to the starter PC board. That BJT stage drove the 5 volt reed relay (250 Ohm coil, 1 A 250 VAC contact) which

in turn drove the coil of a 12 volt relay (1500 Ohm coil, 3 A 30 VDC contact). The 12-volt coil relay controls the ground of the fan motor. Driving the fan motor ground avoids ground spikes and other potential noise. (If the relay is set to control the fan power, ground spikes will result [often even with a fly-back diode] causing the HC11 power supervisory circuit to reset.) TRIXIE's net fire extinguishing capabilities were impressive: she could blow an aluminum can over from more than six feet away.

POWER GRID

2 – 12 V, 2.2 Ah lead acid batteries

These two batteries were connected in series to yield 24 volts at 2.2 Amp hours of shear power. Two of these high power batteries were used because by using the tank track design: TRIXIE had to overcome much greater coefficients of friction. One of these batteries was used to drive the fan motor. For protection, this power system had its own switch and fused links to each connection. One connection was to the H-bridge board for drive motor power and another was to the fan motor.

The Pittman motors are rated for 24 volts and the H-bridges (LMD18200T) for 3 A at 55 volts, so only if TRIXIE was driven full speed after a fresh recharge of the lead acids might she exceed specifications. The lead acid batteries typically lasted an hour for continuous running and up to 4 hours for testing (the amount of run time & speed affects that).

1 – 8.4 V, 1.2 Ah Ni-Cd battery

This battery was used to provide power to all of the 5-volt logic and sensors. A 7.2-volt nickel-cadmium battery was used originally, but because of the high dropout voltage of the adjustable positive linear regulators (LM317) TRIXIE demanded the 8.4-volt battery pack. TRIXIE's sensor conditioning requirements were met with 5-volt single supply op-amps so that she would not necessitate +/- 15-volt supplies.

The connection from the 8.4-volt battery was switched and fused for safety yielding separately switchable power for the HC11. If the HC11 EPROM chip were utilized, this would not be required, but TRIXIE needed program adjustment during development and EEPROM is much better for that. The use of the EEPROM chip enabled TRIXIE to have her high power disabled, but her low power remaining on as to not lose HC11 program code.

The connection was then fed into a power distribution PC board. The distribution board had two 7805 (5-volt linear regulator) TO-220 packages and two LM317 (adjustable positive voltage linear regulator) TO-220 packages mounted thermally (electrically isolated to avoid power loops) to an aluminum-clad copper heat sink. One of the 7805's provided the 5 volts needed for the three Sharp GP2D12's and the other powered all the 5-volt logic for the binocular fire sensors (flamer_p.sch/.pcb), the tone decoder circuitry (starter.sch/.pcb), the sensor signal conditioning PC board

(sigcond.sch/.pcb), the line sensor, and for the H-bridge PC board (hbridge.sch/.pcb). One of the two LM317's was set to provide 5.17 volts for the HC11 and the Altera 7128 PLD; the other was left for unseen needs and troubleshooting, but was not implemented in the final design. The voltage level 5.17 was used to prevent the HC11 supervisory circuit from resetting from noise riding the power lines. (5.17 was also appropriate for the 3.3-volt Altera device because of the dropout voltage exhibited by the LM317 used to maintain the required 3.3 volts.) Regulator manufactures were chosen so that each regulator would be capable of supplying about one amp (to minimize heat loss).

All of the low power needs require 300 mA nominally and about 400 mA max, so the 1.2 Ah capacity of the 8.4 volt Ni-Cad resulted in about 2 ½ hours of use before recharge. Recharge wasn't typically needed because of expended batteries evidentally, but because the voltage level had dropped below the LM317 dropout voltage.

See Appendix 11 for the Protel schematic (power.sch) representing this setup.

The PC board made for the low power distribution needed modification because of an LM317 pinout error. Problem resolution was achieved by simply cutting the deviant traces and rerouting with wires underneath.

2 – 9 V, 0.9 Ah alkaline batteries

The two 9 volts were connected in series to yield 18 volts for the Hamamatsu voltage source requirement of 11 – 30 VDC. This method was simpler than using a voltage doubler and much less noisy than using 12 volts from a motor power battery. And with a typical consumption of 3 mA, long battery life was a surety.

FUNCTIONAL HARDWARE LAYOUT

TRIXIE has three tiers of space to contain all of this circuitry and accessories. On the lowest level, which can be seen in fig. 1a of Appendix 2 is where the high power and noisy systems were placed. Directly in front of the Pittman motors are the H-bridge board and the two 12 volt batteries. The batteries were placed toward the front axle to help counter the weight of the motors. The only other items on the first level are the white line sensor (mounted in holes placed at the front of the robot) and the front proximity sensor in front of the batteries.

The second layer contains the power distribution board, sensor signal conditioning board, and two side proximity detectors as is seen in fig. 1b of Appendix 2. The side sensors have milled slots for forward/back slide positioning. In the back, sits the Ni-Cad battery. And in the very front, is the fan (mounted in a plastic motor mount). On the underneath side of this level are the main power switches and fuses.

The third layer is where all of the low voltage systems are located as can be seen in fig. 1c of Appendix 2. Starting in the back there is the Altera and HC11 boards (sitting right next to each other so that the wires connecting them could be as short as possible). In the front are the starter signal and flame sensor boards. In between these two boards are the two flame sensors mounted in a small piece of angle aluminum. The only other

items on this level are the two 9V batteries with a switch for the Hamamatsu UVtron on one side and the 12 volt fan relay on the other side.

Additionally, TRIXIE's cover has two items mounted on the underneath side as seen in fig. 1d of Appendix 2. First, there is the Hamamatsu UVtron, which is mounted in the front in an aluminum EMF cage. The microphone for the sound start is mounted to the side of that with a large hole bored through the cover for acoustic exposure.

There was room for all of TRIXIE's required circuitry and components. In the final design implementation, TRIXIE was about 10" cubed, thus meeting the design specifications. She was wide enough for excellent stability, yet not too tall to compromise that stability.

CODE

All of the 68HC11 control code was written in the C language. C code had to be written in order to bridge the gap between the hardware and actually having the robot perform some function. The software was broken up into two major functions: wall following, and maze navigation.

Wall Following

Being able to follow a wall allows any robot to completely navigate a maze. Wall following is exactly what the name implies, being able to follow a wall at a certain distance. Wall following is a non-trivial task. In order to be able to follow a wall, the software needs to receive information on current wall distances, and current velocities. This information is then placed into a proportional controller which tries to maintain a motor speed that allows the robot to achieve a desired velocity as well as maintain a certain distance from a wall. Current velocity information is gathered from the pulse accumulators in the Altera 7128 device. The pulse accumulators are read every 4ms with an HC11 real-time interrupt. The pulse accumulators are then converted into a duty cycle. In order to convert the duty cycle, it was assumed that the motors had a linear response. This means that the duty cycle percentage versus RPM has a linear slope of 1/12. One revolution of the motor drives TRIXIE 8 cm. A single revolution of the shaft is generated by 5.9 motor revolutions. The shaft encoder generates 500 counts per motor revolution. This in turn gives 2950 counts per shaft revolution. Thus, 369 counts are generated for every centimeter the robot travels. At a 100% duty cycle, the motors shaft is operating at 1200 RPM. Thus, the counts from the motor encoders can be converted into a duty cycle by the following equation and the previous assumptions.

$$DutyCycle = ((counts/period)*(periods/second))\div(2950*60) \div(12)$$

This duty cycle represents the current velocity of the robot. In order to adjust the robot's speed, the following proportional controller equation is used:

$$newDutyCycle = desiredDutyCycle + Kp*(desiredDutyCycle - currentVelocityDutyCycle)$$

The equation above allows the robot to achieve a desired duty cycle based on the current velocity of the robot. Using a proportional controller with TRIXIE was more than adequate because there is so much friction in TRIXIE's drive system. The proportional controller implemented seemed to always respond well to the robot, and never seemed to over-correct. The Kp constant was set to be .5, and appeared to provide the robot with an ideal motor-control response. This equation can also be given a bias term based on input from the GP2D12 sensors. The entire proportional controller can then be used to adjust the velocity of the robot and the distance from a wall that the robot is following. The entire proportional controller that is implemented is:

$$newDutyCycle = desiredDutyCycle + Kp*(desiredDutyCycle - currentVelocityDutyCycle) \pm Kw*(followWallDistance - currentWallDistance)$$

(Note: The wall distance term is + for the left motor, and – for the right motor during a right wall follow, and opposite in sign for a left wall follow)

The newDutyCycle is then passed to a routine that prevents a duty cycle greater than 100 or less than 0 to be given to the Altera PWM controller. The newDutyCycle is then written to the Altera PWM controller and the robot continues to navigate the maze.

Maze Navigation

Since TRIXIE uses the HAMAMATSU UVTron, it was decided early on that it would be possible to utilize the UVTron in determining where in the maze the candle is located. The UVTron has a very high sensitivity and the ability to sense a burning flame at very large distances. It is possible to drive from the home position in the maze to the center of the maze (See Appendix 1) and perform a 360° scan of the maze. During the 360° scan of the maze, UVTron intensities are recorded. This scan allows TRIXIE to be able to determine where in the maze the flame is without having to do a room by room search of the entire maze. The steps in accomplishing the 360° scan occurs by turning the robot towards a room or maze, waiting for two seconds and reading the value stored in the HC11 pulse accumulator which stores how many counts the UVTron produced. This process is repeated until the robot has performed a complete 360° scan of the maze. Once the 360° scan is completed, the robot then uses a combination of dead-reckoning and closed-loop control to go into the room where the highest counts were received from

the 360° scan. Dead reckoning was only used to be able to align the robot in a position in the maze to be able to wall follow into a specific room. The maze that TRIXIE performs in has a white-line drawn across each doorway to a room. The white-line sensors that are attached to the underside of TRIXIE signal an HC11 IRQ. Once the IRQ is received TRIXIE is positioned inside the doorway. The software is able to recognize that once the white-line is crossed, the robot is within the room. Once the robot is positioned in the room, a simple fire scan is used to locate the flame within the room using input from the binocular fire vision. The robot then proceeds to the fire, and begins to extinguish the flame by turning on the fan motor, which is controlled by an I/O pin from the HC11 PIA expansion chip. Once the fire is extinguished, TRIXIE performed her job of navigating a maze, identifying a fire within the maze, and extinguishing the fire. All of the code for navigating the maze is in Appendix 20.

(Note: The current implemented code can not fully enter a room and extinguish the flame using the input from the binocular fire vision. This information is inserted into this document to allow the reader an understanding of the direction the software is headed.)

CONCLUSION

TRIXIE is a self-contained autonomous robot capable of competing at the Trinity College National Robot Fire Fighting contest. TRIXIE uses two layers of abstraction to be able to function. The first layer is a complete hardware layer, with various components that allow the robot to able to navigate and sense the environment. The second layer is the software layer, which incorporates all of the hardware into a layer of control. TRIXIE is currently not fully operational. The robot can fully navigate the maze, but is unable to zoom in on the candle within the room of the maze and extinguish the candle. All of the hardware to operate TRIXIE is fully debugged and operational. The total cost to build the robot is $100.37, see Appendix 14 for a full budget breakdown. TRIXIE is a one of a kind robot due to the track-based design, and quality workmanship that enables a high quality robust robot.

REFERENCES

Flynn, Anita M., Joseph L. Jones, and Bruce A. Seiger: Mobile Robotics: Inspiration to Implementation. 2nd ed. Mass.: A K Peters, 1999.

Sedra, Adel S., and Kenneth C. Smith. Microelectronic Circuits. 2nd ed. New York: Oxford, 1998.

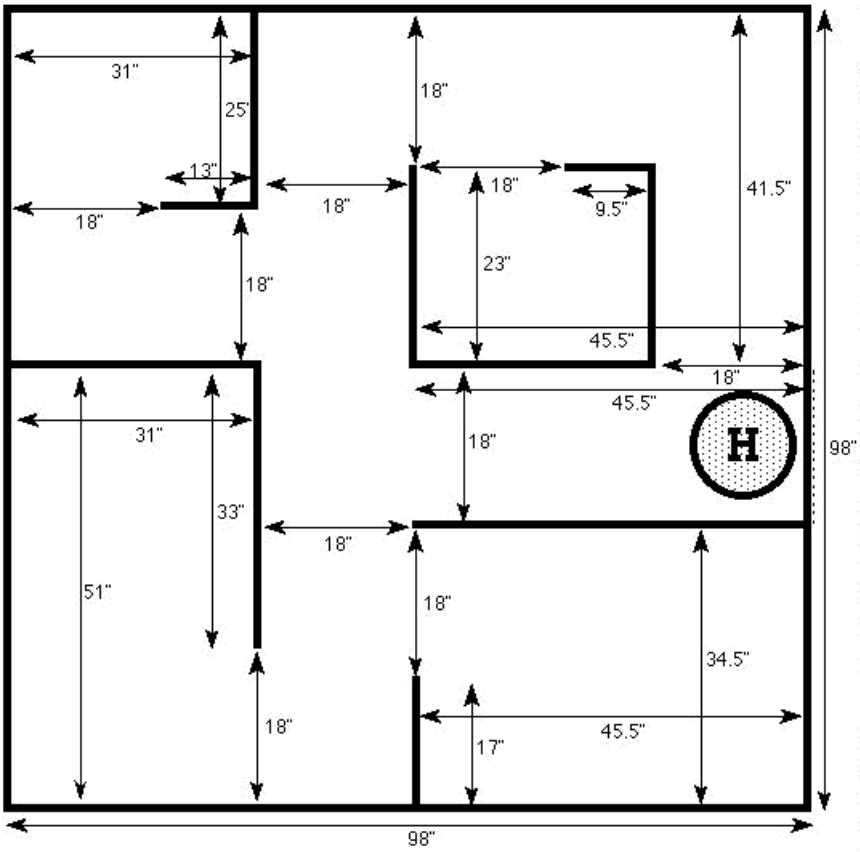
National Semiconductor Databooks.

The Trinity College Fire Fighting Home Robot Contest Web Page (www.trincoll.edu/~robot/)

Hamamatsu Web Page (www.hamamatsu.com/)

APPENDIX 1 – MAZE FLOORPAN

Trinity College FIRE FIGHTING
Home Robot Contest
1999 Arena Floor Plan
Contest Rules, Attachment A

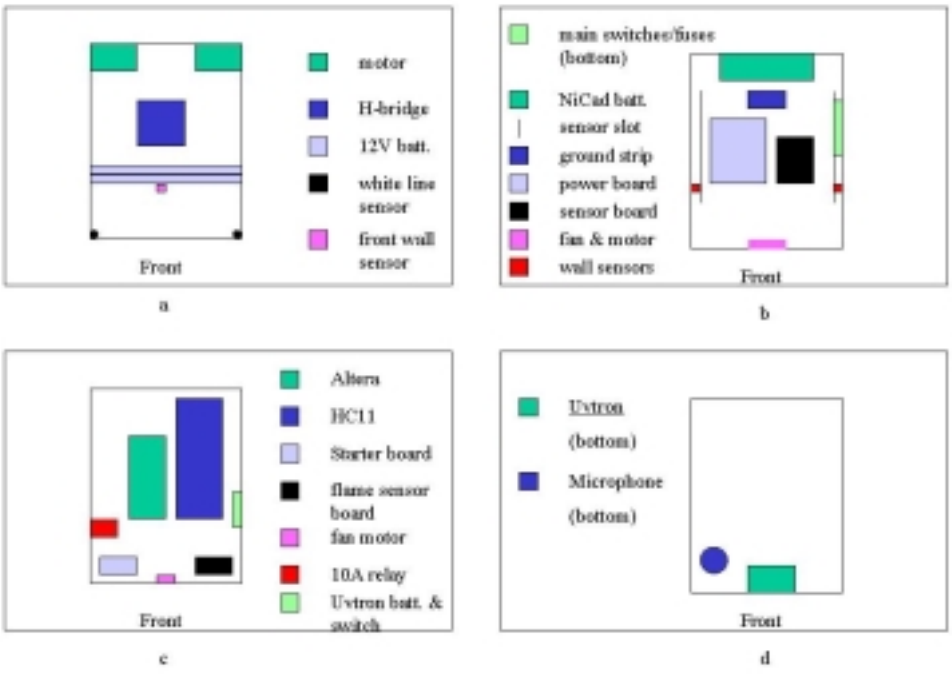


Maximum size of Robot is 12.25" x 12.25"

All walls are 13" high

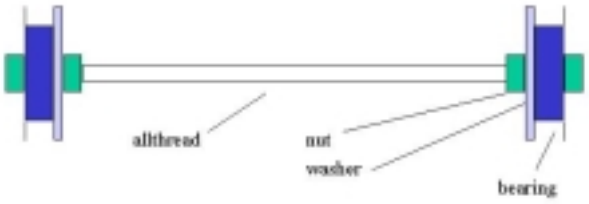
These dimensions are approximations and are NOT 100% accurate and that's why the numbers don't add up exactly. Welcome to the real world!

APPENDIX 2 – CHASSIS

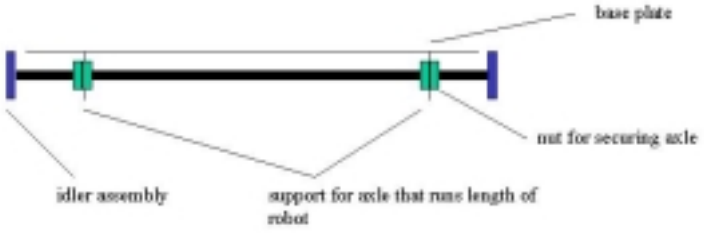


APPENDIX 3 – TRACK DESIGN

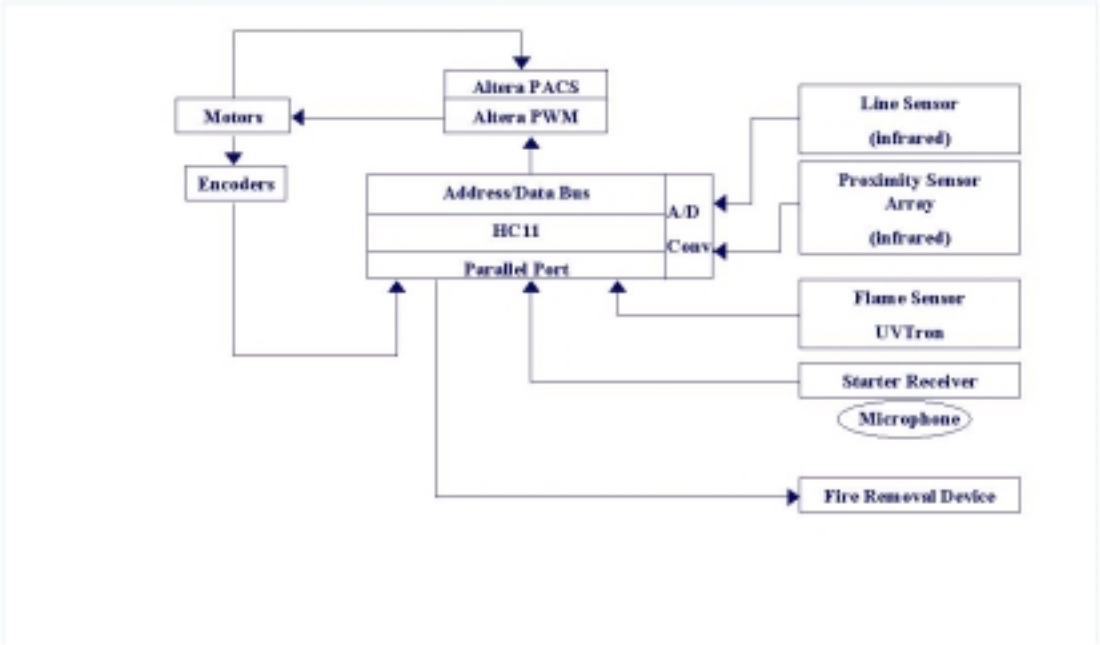
a



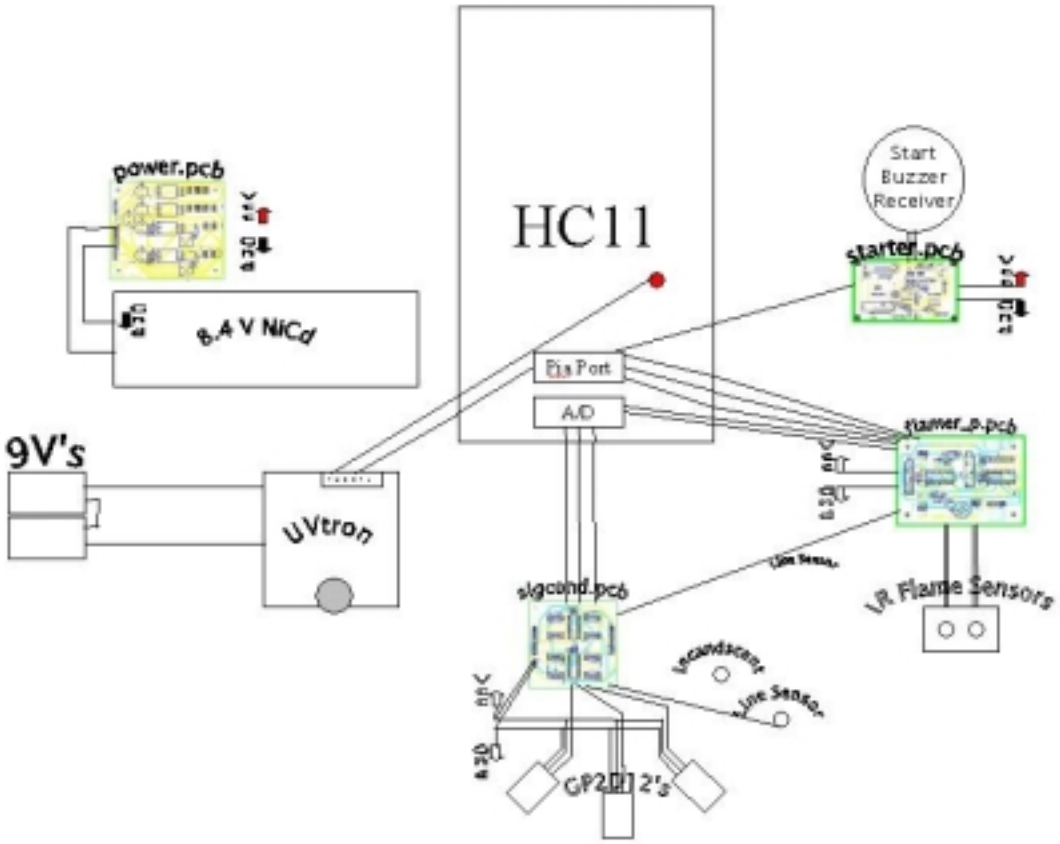
b



APPENDIX 4 – SYSTEM BLOCK DIAGRAM



APPENDIX 5 – SENSOR BLOCK DIAGRAM



APPENDIX 6 – HAMAMATSU

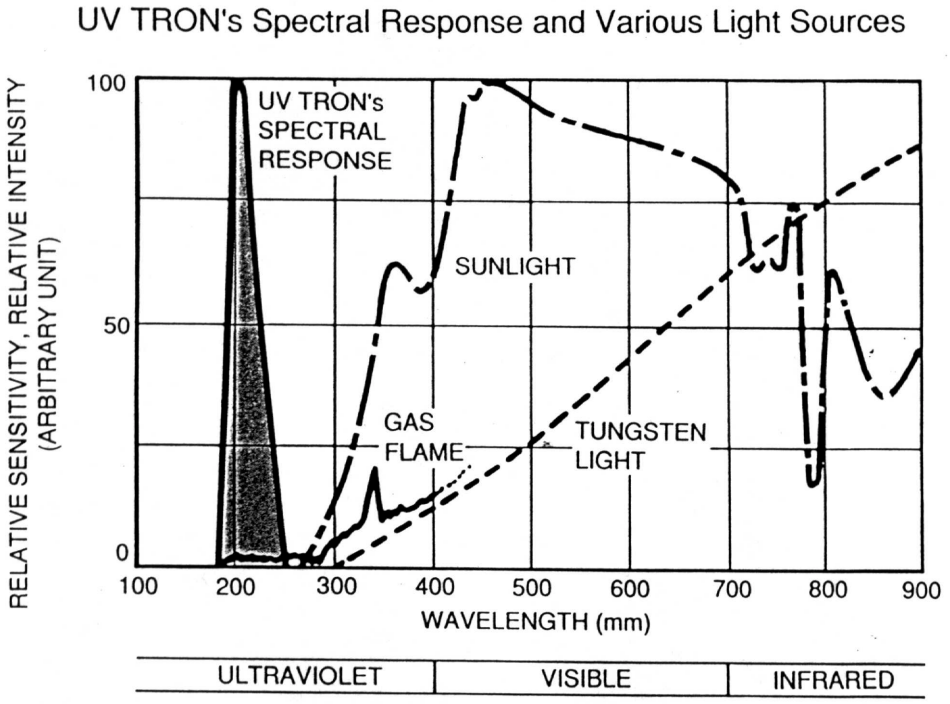


Figure 1: Hamamatsu spectral response

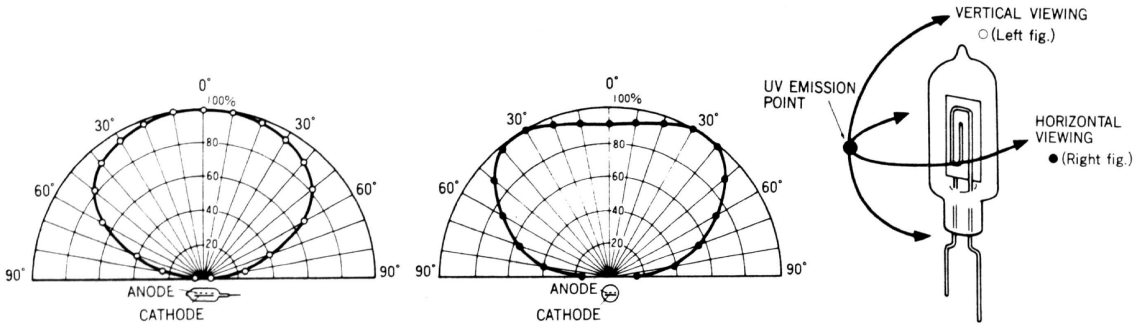
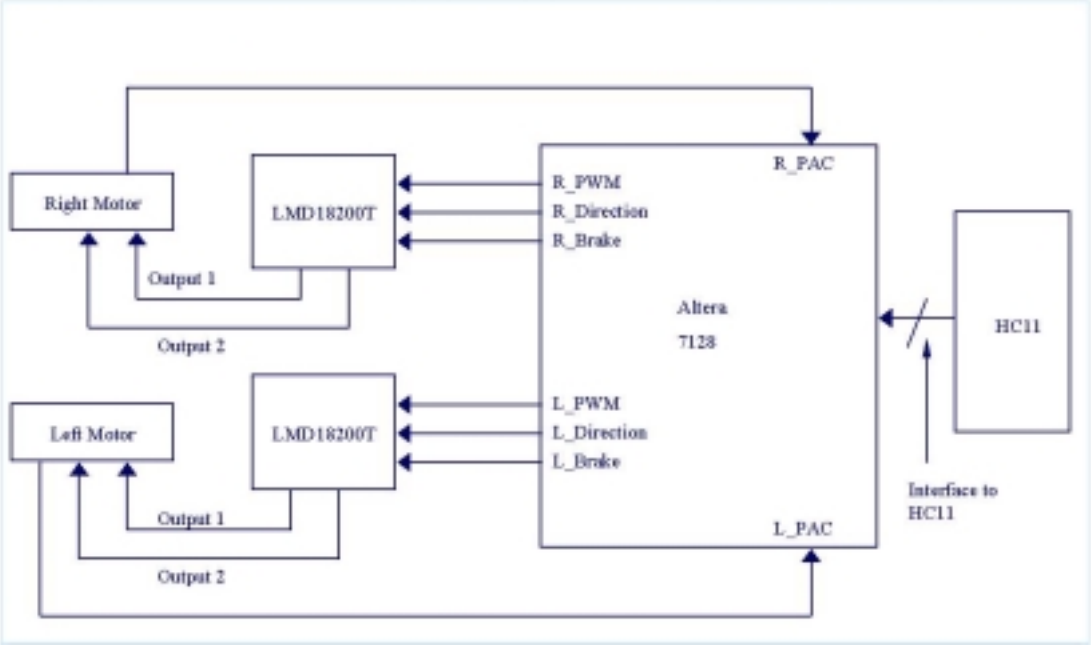
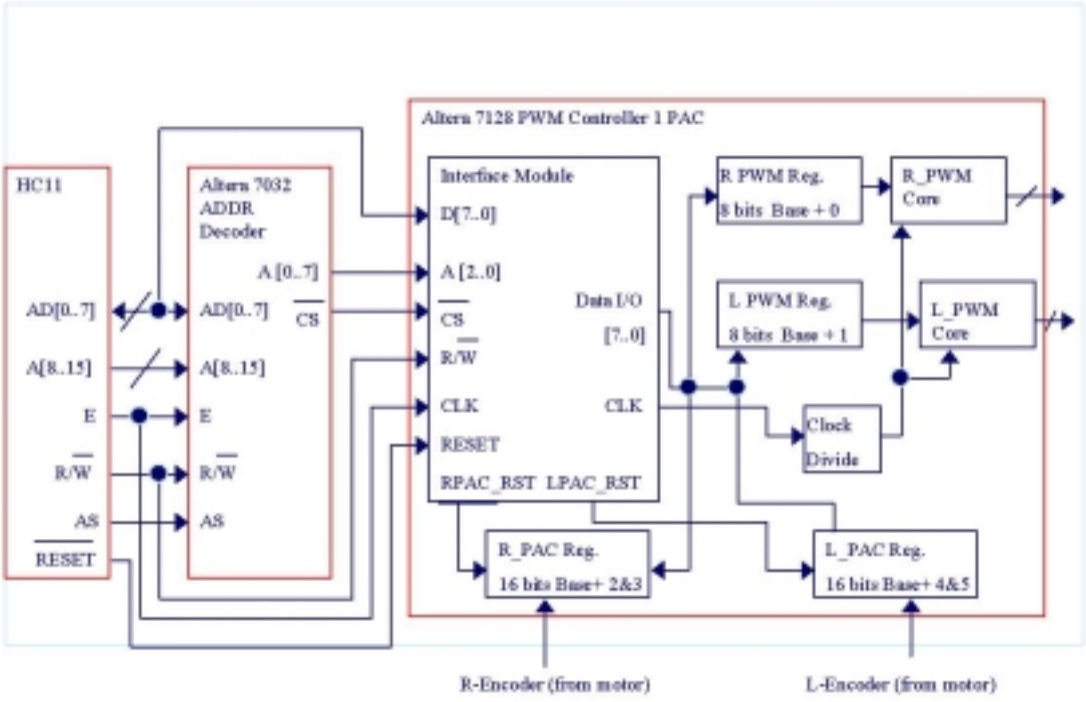


Figure 2: Hamamatsu angular sensitivity

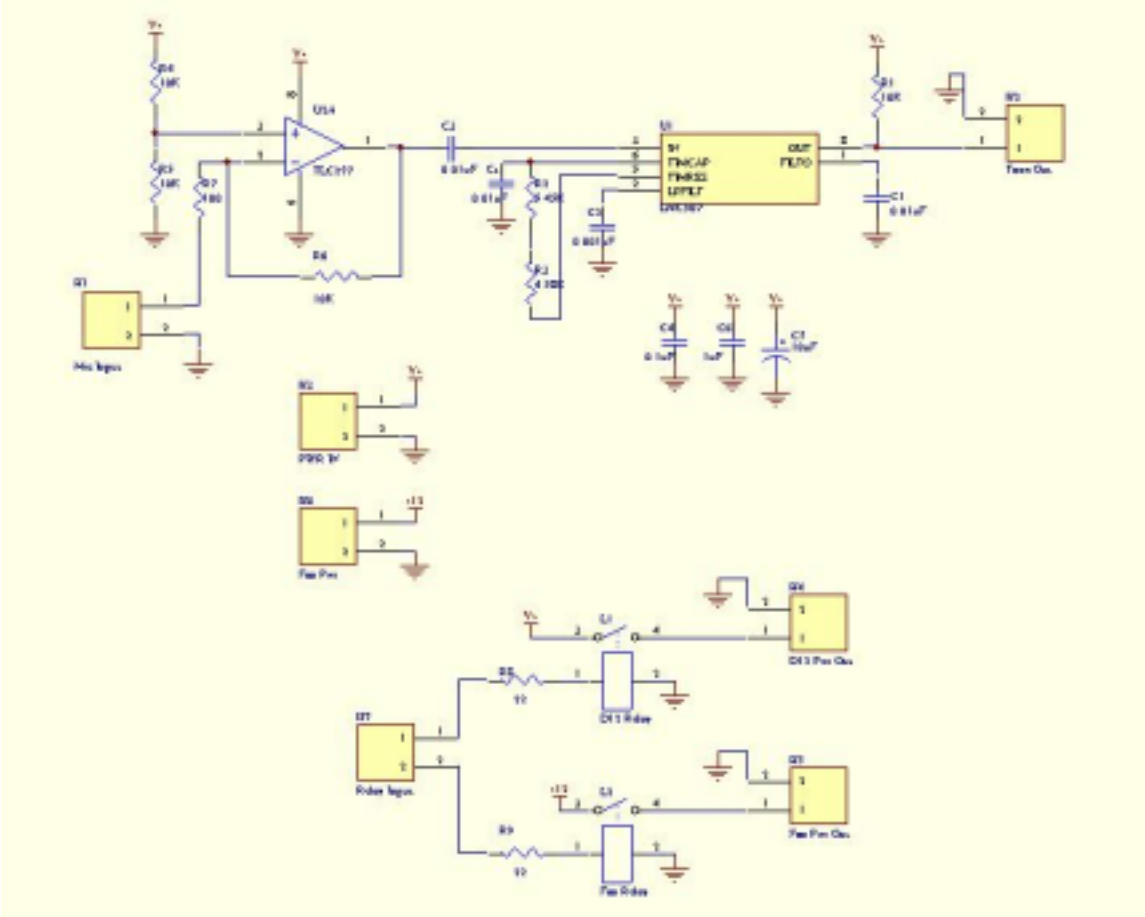
APPENDIX 7 – MOTOR CONTROL BLOCK DIAGRAM



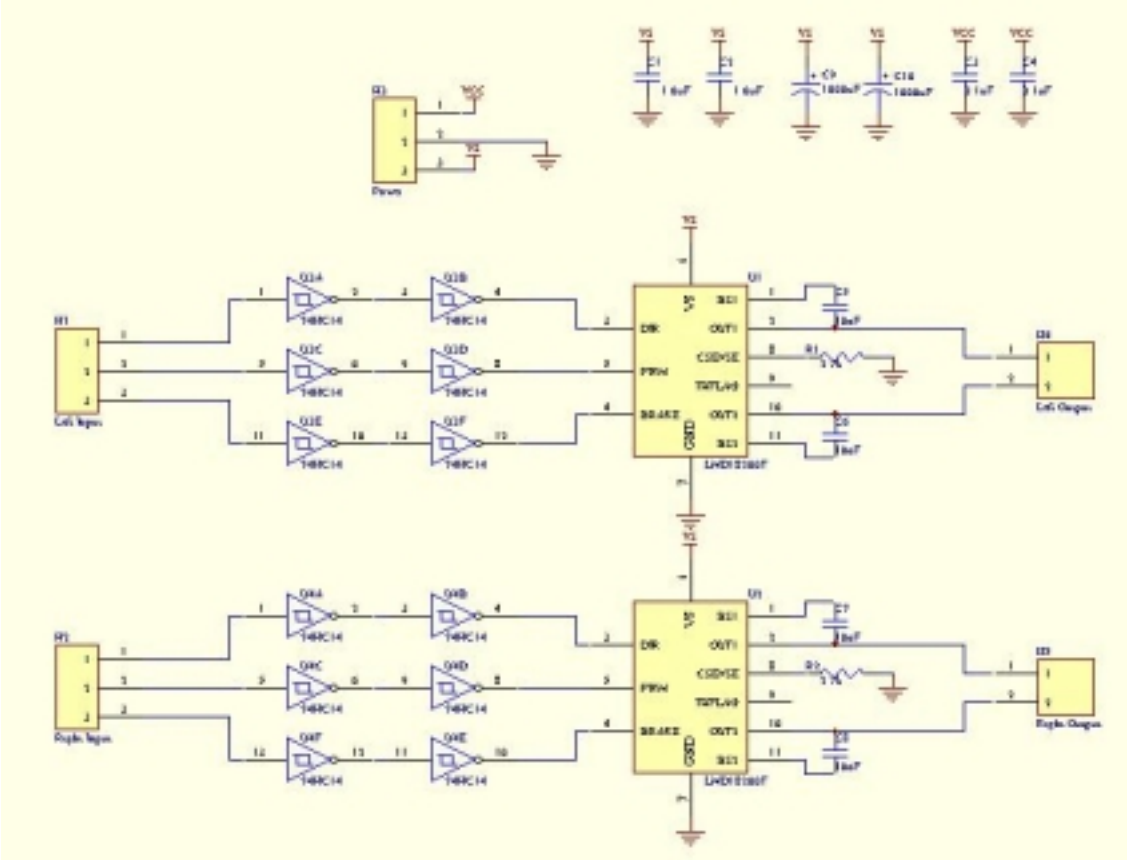
APPENDIX 8 – HC11 ALTERA INTERFACE DIAGRAM



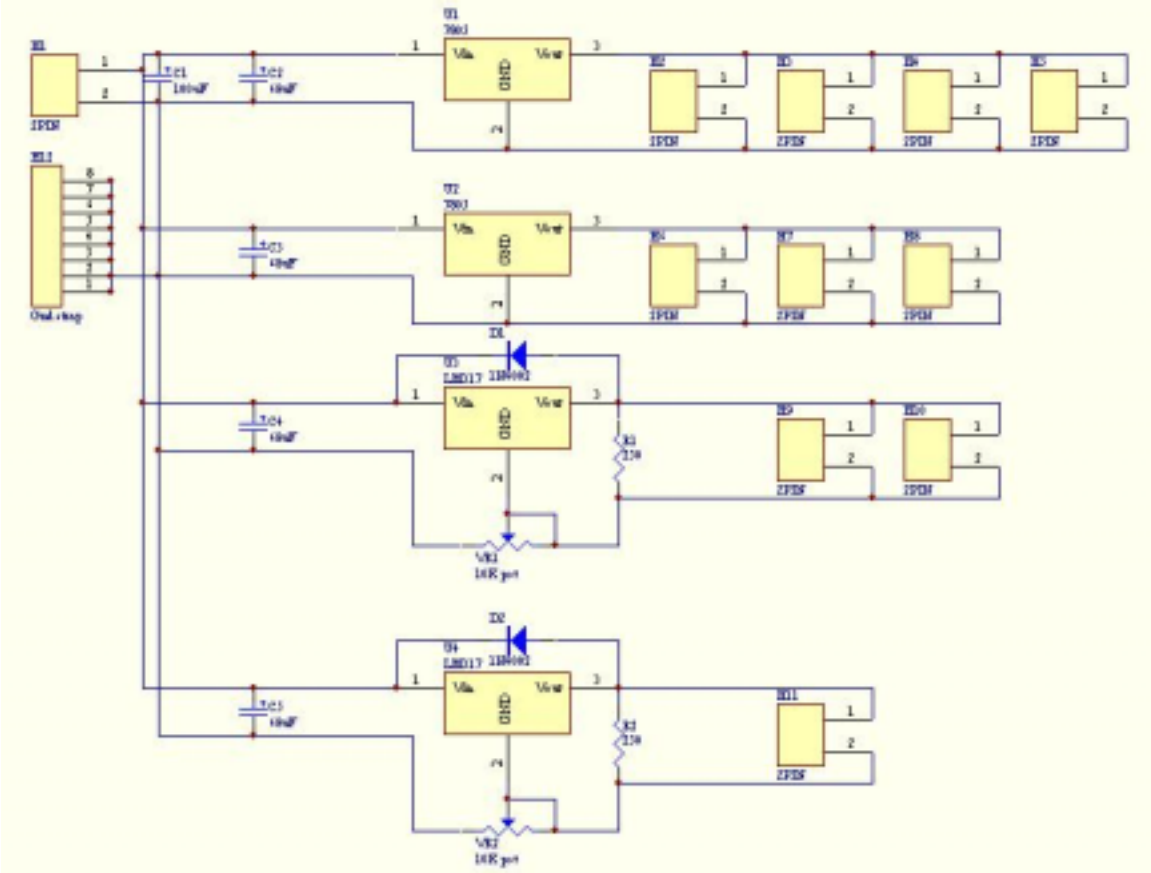
APPENDIX 9 – STARTER BOARD SCHEMATIC



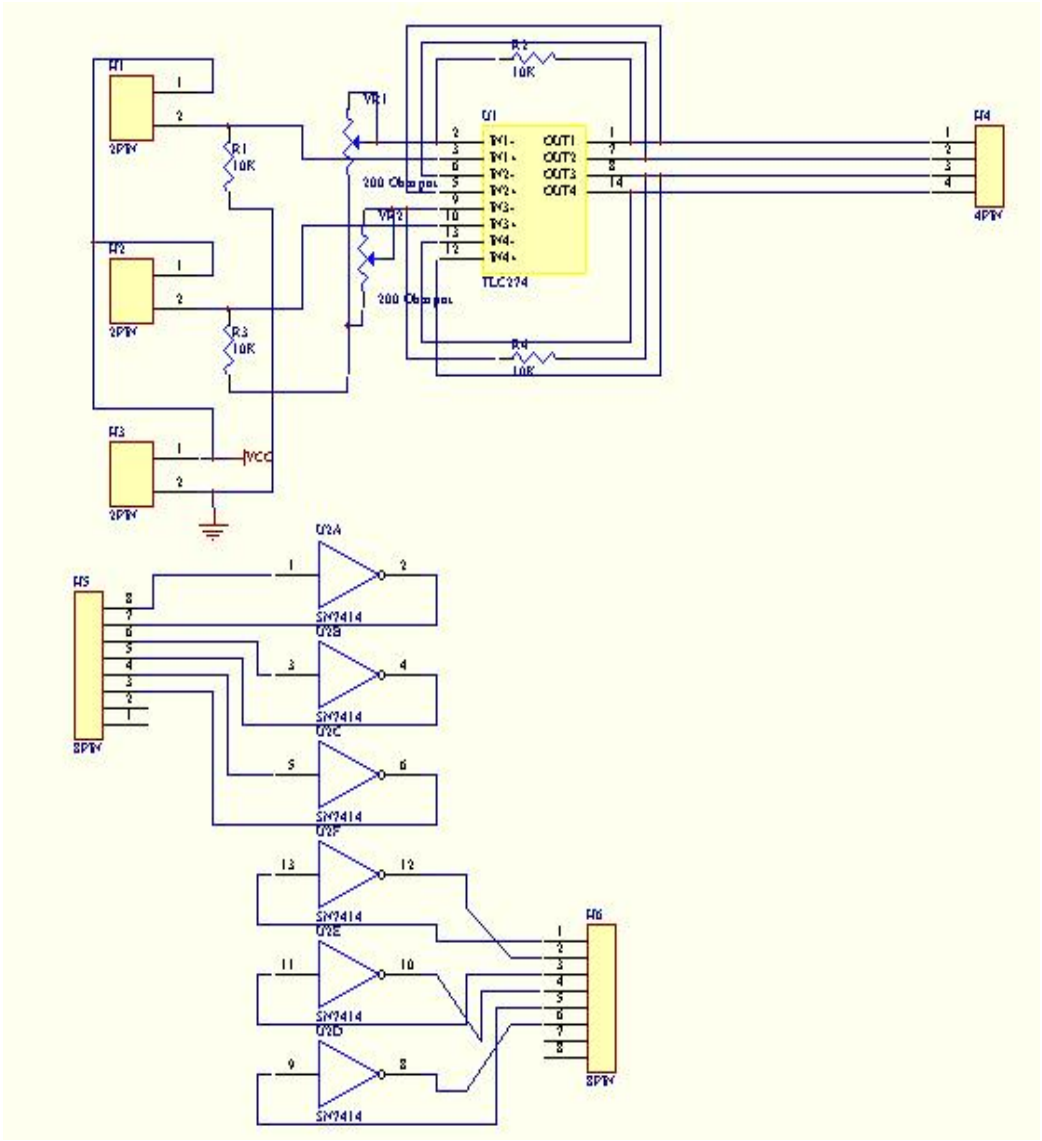
APPENDIX 10 – H-BRIDGE BOARD SCHEMATIC



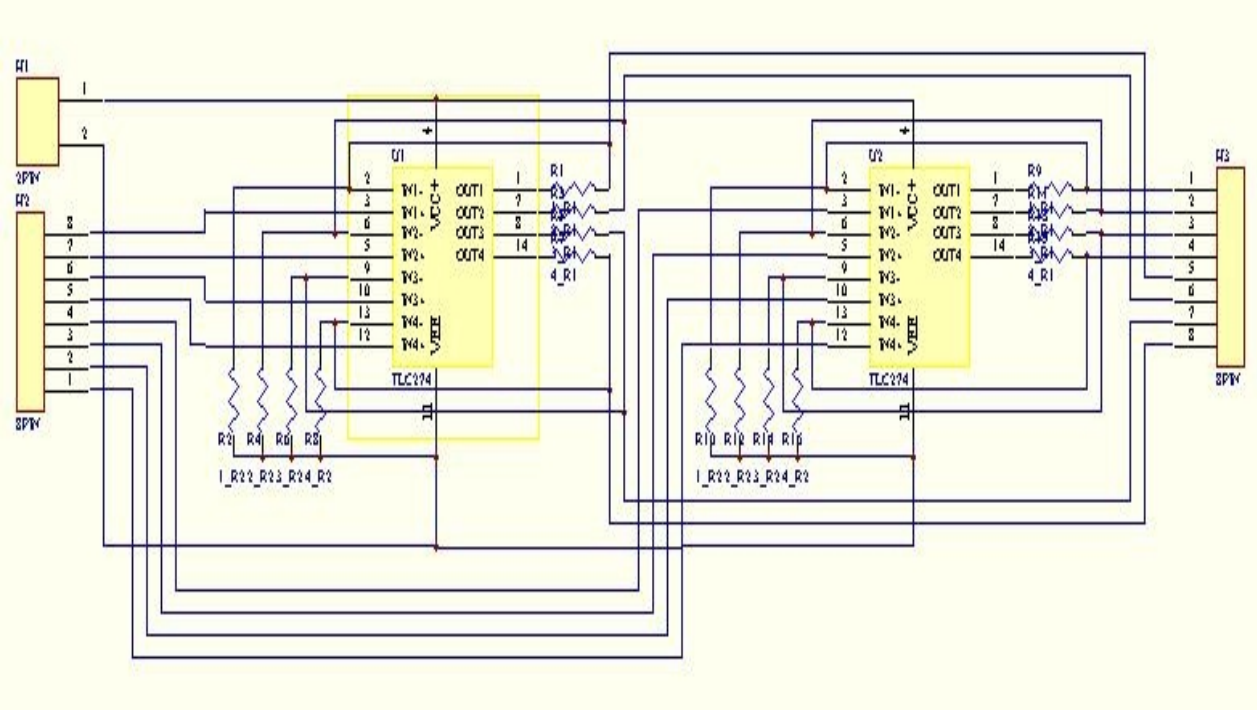
APPENDIX 11 – POWER BOARD SCHEMATIC



APPENDIX 12 – FLAME PLUS SOUND STARTER SCHEMATIC



APPENDIX 13 – SIGNAL CONDITIONING SCHEMATIC



APPENDIX 14 – FINAL BUDGET

Budget

Grainger (belts and pulleys)	$27.24
Instrument room	$18.97
Newark	$4.04
Digi-key	$2.48
Wyle	$16.00
Langley's Stock	$9.86
Lead Acid Battery (extra)	$10.00
Strerling Sensors	$11.78
Total	$100.37

APPENDIX 15 – AHDL PWM MODULE

```
%pwmcore.tdf -    PWM generator code.  Receives a 7 bit magnitude and a
sign.
                  Generates corresponding duty cycle as well as brake
and direction signals.
            GROUP 6:
            1/20/99 - Mike Davis - written
            1/27/99 - Mike Davis - modified to allow 100percent duty
cycle
%
SUBDESIGN pwmcore
(
%INPUTS%
      width[6..0]       :INPUT; %Pulse width in percent%
      sign              :INPUT; %positive => forward, negative =>
reverse%
      clock             :INPUT; %10kHz clock to achieve 100 cnts at
100Hz PWM%
      reset             :INPUT;

%OUTPUTS%
      direction         :OUTPUT; %signals to LMD18200T H-Bridge%
      brake             :OUTPUT;
      pwm                     :OUTPUT;
)

VARIABLE
      count[6..0]       :DFF;
      pwmout                  :NODE;

BEGIN
      count[6..0].clk = clock;
      count[6..0].clrn = reset;

%counter logic%
      IF (count[6..0] < 99) THEN
            count[6..0] = count[6..0] + 1;
      ELSE
            count[6..0] = 0;
      END IF;

%pwm logic%
      IF (width[6..0] > 100) THEN
            brake = VCC;
            pwm = VCC;  %brake case%
      ELSIF (width[6..0] == 100) THEN
            brake = GND;
            pwm = VCC;  %full speed - duty cycle = 100%
      ELSIF (width[6..0] == 0) THEN
            brake = GND;
```

```
            pwm = GND;  %stop%
      ELSE
            brake = GND;
            pwm = pwmout;
      END IF;

      IF (count[6..0] < width[6..0]) THEN
            pwmout = VCC;
      ELSE
            pwmout = GND;
      END IF;

      direction = sign;
END;
```

APPENDIX 16 – AHDL PAC MODULE

```
%pac.tdf -  13 bit pulse accumulator and control logic
      GROUP 6:
            2/17/99 - Mike Davis - written
%

SUBDESIGN pac
(
%Inputs%
      pulse             :INPUT;
      reset             :INPUT;

%Outputs%
      count[15..0]      :OUTPUT;
)
VARIABLE
      counter[15..0]    :DFF;

BEGIN
      counter[15..0].clk = pulse;
      counter[15..0].clrn = reset;

      IF(counter[15..0] < H"FFFF") THEN
            counter[15..0].d = counter[15..0].q + 1;
      ElSE
            counter[15..0].d = counter[15..0].q;
      END IF;

      count[15..0] = counter[15..0].q;
END;
```

APPENDIX 17 – AHDL CLOCK GENERATOR MODULE

```
%clockgen.tdf -   Clock generator.  Divides input clock by 200.

                  Generates a 10kHz clock for pwmcore from 2MHz E-
clock.
            GROUP 6:
            1/20/99 - Mike Davis - written
%
CONSTANT MAXCOUNT = 200;
CONSTANT MIDCOUNT = 100;

SUBDESIGN clockgen
(
%INPUTS%
      clockin           :INPUT;     %2MHz HC11 E-Clock%
      reset       :INPUT;

%OUTPUT%
      clockout    :OUTPUT; %10kHz clock output%
)

VARIABLE
      count[7..0] :DFF;

BEGIN
      count[7..0].clk = clockin;
      count[7..0].clrn = reset;

      IF(count[7..0] < (MAXCOUNT - 1))THEN
            count[7..0] = count[7..0] + 1;
      ELSE
            count[7..0] = 0;
      END IF;

      IF(count[7..0] < (MIDCOUNT - 1)) THEN
            clockout = VCC;
      ELSE
            clockout = GND;
      END IF;
END;
```

APPENDIX 18 – AHDL INTERFACE MODULE

```
%pwm.tdf -  PWM controller code. Recieves pulse width from HC11 and
                produces control signals for LMD18200T H-Bridges.
                Controls left and right channels independantly.

      GROUP 6:
            2/09/99 - Mike Davis - written
            2/17/99 - Mike Davis - PAC stuff added
            3/10/99 - Mike Davis & George McCone - Registers changed to
                        latches and debugged.
%
INCLUDE "pwm.inc";

SUBDESIGN pwm
(
%INPUTS%
      data[7..0]        :BIDIR; %data%
      addr[2..0]        :INPUT; %address lines%
      clockin                 :INPUT; %HC11 Eclock%
      cs                      :INPUT; %chip select%
      r/w                     :INPUT; %read/write line%
      r_encoder         :INPUT;     %right encoder input%
      l_encoder         :INPUT;     %left encoder input%
      reset             :INPUT; %reset from HC11%

%OUTPUTS%
      pwm_right         :OUTPUT;%right channel outputs%
      dir_right         :OUTPUT;
      brake_right       :OUTPUT;
      pwm_left          :OUTPUT;%left channel outputs%
      dir_left          :OUTPUT;
      brake_left        :OUTPUT;
)

VARIABLE
      left_core         :pwmcore;
      right_core        :pwmcore;
      clocker                 :clockgen;
      rpac              :pac;
      lpac              :pac;
      right_reg[7..0] :LATCH;
      left_reg[7..0]    :LATCH;
      r_latcher         :NODE;
      l_latcher         :NODE;
      dataout[7..0]   :NODE;
      read              :NODE;
      write             :NODE;
      lpacrst                 :NODE;
      rpacrst                 :NODE;

BEGIN
```

```
%define the logic for the bidirectional data bus%
      data[7] = TRI(dataout[7], read);
      data[6] = TRI(dataout[6], read);
      data[5] = TRI(dataout[5], read);
      data[4] = TRI(dataout[4], read);
      data[3] = TRI(dataout[3], read);
      data[2] = TRI(dataout[2], read);
      data[1] = TRI(dataout[1], read);
      data[0] = TRI(dataout[0], read);

%define read and write logic %
      IF((cs == GND) AND (r/w == VCC) AND (clockin == VCC)) THEN
            read = VCC;
            write = GND;
      ELSIF((cs == GND) AND (r/w == GND) AND (clockin == VCC)) THEN
            read = GND;
            write = VCC;
      ELSE
            read = GND;
            write = GND;
      END IF;

%register stuff%
      IF(reset == GND) THEN
            right_reg[7..0]=H"00";
            left_reg[7..0]=H"00";
            right_reg[7..0].ena=VCC;
            left_reg[7..0].ena=VCC;
      ELSE
            right_reg[7..0].d = data[7..0];
            left_reg[7..0].d = data[7..0];
            right_reg[7..0].ena = r_latcher;
            left_reg[7..0].ena = l_latcher;
      END IF;

%writing%
      IF((write == VCC) AND (addr[2..0] == RIGHT_ADDR)) THEN
            r_latcher = VCC;  %write to the right reg%
            l_latcher = GND;
            rpacrst = VCC;
            lpacrst = VCC;
      ELSIF((write == VCC) AND (addr[2..0] == LEFT_ADDR)) THEN
            r_latcher = GND;
            l_latcher = VCC;  %write left reg%
            rpacrst = VCC;
            lpacrst = VCC;
      ELSIF((write == VCC) AND (addr[2..0] == RPACRST_ADDR)) THEN
            r_latcher = GND;
            l_latcher = GND;
            rpacrst = GND;          %reset the right PAC%
            lpacrst = VCC;
      ELSIF((write == VCC) AND (addr[2..0] == LPACRST_ADDR)) THEN
            r_latcher = GND;
            l_latcher = GND;
            rpacrst = VCC;
            lpacrst = GND;          %reset the left PAC%
      ELSE
```

```
            r_latcher = GND;
            l_latcher = GND;
            rpacrst = VCC;
            lpacrst = VCC;
      END IF;
%reading%
      IF((read == VCC) AND (addr[2..0] == RIGHT_ADDR)) THEN
            dataout[7..0] = right_reg[7..0].q; %read the right reg%
      ELSIF((read == VCC) AND (addr[2..0] == LEFT_ADDR)) THEN
            dataout[7..0] = left_reg[7..0].q;  %read the left reg%
      ELSIF((read == VCC) AND (addr[2..0] == LPACLO_ADDR)) THEN
            dataout[7..0] = lpac.count[7..0];
      ELSIF((read == VCC) AND (addr[2..0] == LPACHI_ADDR)) THEN
            dataout[7..0] = lpac.count[15..8];
      ELSIF((read == VCC) AND (addr[2..0] == RPACLO_ADDR)) THEN
            dataout[7..0] = rpac.count[7..0];
      ELSIF((read == VCC) AND (addr[2..0] == RPACHI_ADDR)) THEN
            dataout[7..0] = rpac.count[15..8];
      ELSE
            dataout[7..0] = DEFAULT_OUT; %if no reads, use default out%
      END IF;

%define the module interconnections%
      clocker.clockin = GLOBAL(clockin);
      clocker.reset = reset;

      left_core.width[6..0] = left_reg[6..0];
      left_core.sign = left_reg[7];
      left_core.reset = reset;
      left_core.clock = clocker.clockout;

      right_core.width[6..0] = right_reg[6..0];
      right_core.sign = right_reg[7];
      right_core.reset = reset;
      right_core.clock = clocker.clockout;

      rpac.pulse = r_encoder;
      rpac.reset = reset AND rpacrst;

      lpac.pulse = l_encoder;
      lpac.reset = reset AND lpacrst;

%define the system outputs%
      pwm_right = right_core.pwm;
      dir_right = right_core.direction;
      brake_right = right_core.brake;
      pwm_left = left_core.pwm;
      dir_left = left_core.direction;
      brake_left = left_core.brake;

END;
```

APPENDIX 19 – AHDL INCLUDE FILE

```
%pwm.inc -  PWM include file.
                Defines Altera Macrofunctions
            GROUP 6:
            2/2/99 - Mike Davis - written
            2/17/99 - Mike Davis - Last Modified
%

FUNCTION pwmcore (width[6..0], sign, clock, reset)
      RETURNS (direction, brake, pwm);

FUNCTION clockgen (clockin, reset)
      RETURNS (clockout);

FUNCTION pac (pulse, reset)
      RETURNS (count[15..0]);

CONSTANT RIGHT_ADDR   = H"00";
CONSTANT LEFT_ADDR    = H"01";
CONSTANT RPACLO_ADDR  = H"02";
CONSTANT RPACHI_ADDR  = H"03";
CONSTANT LPACLO_ADDR  = H"04";
CONSTANT LPACHI_ADDR  = H"05";
CONSTANT RPACRST_ADDR = H"06";
CONSTANT LPACRST_ADDR = H"07";
CONSTANT DEFAULT_OUT  = H"A5";
```

APPENDIX 20 – CODE

```
/************************************************
 Programmer: Mike Davis, David Bonal, Rob Niemand
 Date:       5-10-99
 Purpose:    Control Code for TRIXIE to perform in
             the national fire fighting robot competition.
************************************************/
#include <stdlib.h>            /* include standards, HC11, pia, & mdef.h */
#include <stdio.h>
#include <hc11.h>
#include "pia.h"
#include "mdefs.h"
#include "rStructs.h"


/*Define Constants*/
#define FOLLOW_LEFT          1       /*Wall Following Constants */
#define FOLLOW_RIGHT         2
#define CENTERDISTANCEVALUE  2
#define STOPDISTANCE         70      /* Distance to stop from front wall*/
#define HAMATHRESH           2       /* HAMAMATSU counts */

#define OFF 0
#define ON 1

#define Kp 2
#define Kw 1
#define Ks 2


/*  Function Prototypes */

void rti_isr(void);
void getPac(pacInfo *thisPac);
void safety(unsigned char *Left, unsigned char *Right);
void turn45Degrees(unsigned  char count, unsigned char direction, unsigned char finalDC);
int fire360Scan();
void wait(int time);
void turnCount(int count, unsigned char direction, unsigned char finalDC);
void wallFollow(void);
void initializeHardware(void);
void gotoScan(unsigned char StopDistance);
void irq_isr(void);
unsigned int checkHama(unsigned int);
void enterRoom(unsigned char rtiCount);
void scantochaseFlame(unsigned char roomCount);

void gotoRoomOne(void);
void gotoRoomTwo(void);  /* not implemented */
void gotoRoomThree(void); /* not implemented */
void gotoRoomFour(void);

void getStraight(void);
void goBackwards(unsigned int threshold, unsigned int ddc);

/* Global Variables*/
unsigned int rtiDelay;
unsigned int lineDelay;
unsigned char desiredDC;

unsigned char flameIntensity[50];
unsigned char scanDivision             = 0;
unsigned char flameIntensityMaximum    = 0;
unsigned char intensityMaximumPosition = 0;
```

```
unsigned char turnDistance;  /*define D12 voltage in base10 to turn away from head-on
wall*/
unsigned char followLeftDistance; /* define D12 voltage in base10 to follow */
unsigned char followRightDistance; /* define D12 voltage in base10 to follow */
unsigned char followWall;
unsigned char leftD12;
unsigned char rightD12;
unsigned char centerD12;
unsigned char doneCalc = 0;
pacInfo pulseAcc;
pacInfo *ptrPulseAcc;
unsigned char
leftLineFlag                = 0;
unsigned char rightLineFlag = 0;
unsigned char lineFlag      = 0;

unsigned char persistentLeftLineFlag = 0;
unsigned char persistentRightLineFlag = 0;

unsigned char d12Control = ON;

main()
{

    int roomNumber;
    unsigned int starterCounter = 0;
    unsigned int temp = 0;
    OPTION  |=  0x80;         /* setup A to D conversions         */

    ptrPulseAcc = &pulseAcc;

    initializeHardware();

    turnDistance        =  50;
    followLeftDistance  = 110;
    followRightDistance = 85;
    followWall          = FOLLOW_LEFT;
    desiredDC           = 50;

    rtiDelay  = 0;      /* HC11 waits set time for user     */
    lineDelay = 0;

    while(1)
    {
        /* Wait to see 3.5 kHz signal */
       if(rtiDelay >= 1)
       {
          if(starterCounter < 50)
          {
             rtiDelay = 0;
             starterCounter = 0;
          }
          else
          {
            rtiDelay = 0;
            break;
          }
       }
       if((PIA_A & 0x04) == 0x00)
       {
           starterCounter = starterCounter + 1;
       }

    }
    printf("starter on \r\n");
    lineFlag = 0;
    /*
    followWall = FOLLOW_RIGHT;
    enterRoom(100);
    */
```

```
    gotoScan(STOPDISTANCE);     /* center robot in maze */

    roomNumber = fire360Scan(); /* scan the maze for fire */

    /* IF statements to go to the individual rooms */
    if(roomNumber == 1)
    {
       wait(150);
       gotoRoomOne();
    }
    else if(roomNumber == 2)
    {
       wait(150);
       gotoRoomTwo();
    }
    else if(roomNumber == 3)
    {
        wait(150);
        gotoRoomThree();
    }
    else
    {
        wait(150);
        gotoRoomFour();
    }
    lineFlag = 0;

    LPWM = 0;
    RPWM = 0;
    exit(0);

}

/*****************************************************************/
/*  Function prevents DC greater than 100, or less than 0      */
/*****************************************************************/
void safety(unsigned char *Left, unsigned char *Right)
{
    if(*Left > 100)
    {
        *Left = (100 | 0x80);
    }
    else if(*Left < 0)
    {
        *Left = (0 | 0x80);
    }
    else
    {
        *Left = (*Left | 0x80);
    }

    if(*Right > 100)
    {
        *Right = 100;
    }
    else if(*Right < 0)
    {
        *Right = 0;
    }
    else
    {
        *Right = *Right;
    }
}

/*****************************************************************/
/*  Function to turn 45 degrees.  Function needs a direction, count  */
/*  value, and duty cycle to return to after function exits          */
/*****************************************************************/
void turn45Degrees(unsigned char count, unsigned char direction, unsigned char finalDC)
```

```
{
    unsigned int tempLeft;
    unsigned int tempRight;

    tempLeft = 0;
    tempRight = 0;
    /* disable RTI */
    TMSK2 = TMSK2 & (~0x40);

    if(direction == FOLLOW_LEFT)
    {
        /* clear the PACS */
        PACRESETRIGHT = 0x00;
        PACRESETLEFT = 0x00;

        LPWM = (60 | 0x80);  /*FOLLOW_LEFT => turn right*/
        RPWM = (60 | 0x80);
        while(tempLeft < (5000 * count))
        {
            tempLeft = 0;
            tempRight = 0;
            tempLeft = PACHIGHLEFT << 8;
            tempRight = PACHIGHRIGHT << 8;
            tempLeft = tempLeft | PACLOWLEFT;
            tempRight = tempRight | PACLOWRIGHT;
        }

    }
    else
    {
        /* clear the PACS */
        PACRESETRIGHT = 0x00;
        PACRESETLEFT = 0x00;

        LPWM = 60;           /* FOLLOW_RIGHT => turn left */
        RPWM = 60;
        while(tempLeft < (3200 * count))
        {
            tempLeft = 0;
            tempRight = 0;
            tempLeft = PACHIGHLEFT << 8;
            tempRight = PACHIGHRIGHT << 8;
            tempLeft = tempLeft | PACLOWLEFT;
            tempRight = tempRight | PACLOWRIGHT;
        }
    }

    /* re-enable the RTI */
    TMSK2 = TMSK2 | 0x40;

    LPWM = (finalDC | 0x80);
    RPWM = finalDC;
}

/*********************************************************************/
/*  Function to turn.  Function needs a direction, count             */
/*  value, and duty cycle to return to after function exits          */
/*********************************************************************/
void turnCount(int count, unsigned char direction, unsigned char finalDC)
{
    unsigned int tempLeft;
    unsigned int tempRight;

    tempLeft = 0;
    tempRight = 0;
    /* disable RTI */
    TMSK2 = TMSK2 & (~0x40);

    if(direction == FOLLOW_LEFT)
    {
        /* clear the PACS */
```

```
        PACRESETRIGHT = 0x00;
        PACRESETLEFT = 0x00;

        LPWM = (70 | 0x80);  /*FOLLOW_LEFT => turn right*/
        RPWM = (70 | 0x80);
        while(tempLeft < count)
        {
            tempLeft = 0;
            tempRight = 0;
            tempLeft = PACHIGHLEFT << 8;
            tempRight = PACHIGHRIGHT << 8;
            tempLeft = tempLeft | PACLOWLEFT;
            tempRight = tempRight | PACLOWRIGHT;
        }
    }
    else
    {
        /* clear the PACS */
        PACRESETRIGHT = 0x00;
        PACRESETLEFT = 0x00;

        LPWM = 70;           /* FOLLOW_RIGHT => turn left */
        RPWM = 70;
        while(tempLeft < count)
        {
            tempLeft = 0;
            tempRight = 0;
            tempLeft = PACHIGHLEFT << 8;
            tempRight = PACHIGHRIGHT << 8;
            tempLeft = tempLeft | PACLOWLEFT;
            tempRight = tempRight | PACLOWRIGHT;
        }
    }
    /* re-enable the RTI */
    TMSK2 = TMSK2 | 0x40;

    LPWM = (finalDC | 0x80);
    RPWM = finalDC;
}


/*************************************************************/
/* Reads the values from the Pulse Accumulators on the Altera  */
/* 7128.  Function resets the PACS to zero after the read      */
/*************************************************************/
void getPac(pacInfo *thisPac)
{
       thisPac->pacLeft = 0;      /* read 16 bit PACS 8 bits at a time */
       thisPac->pacLeft = PACHIGHLEFT << 8;
       thisPac->pacLeft = (thisPac->pacLeft) | PACLOWLEFT;
       thisPac->pacRight = 0;
       thisPac->pacRight = PACHIGHRIGHT << 8;
       thisPac->pacRight = (thisPac->pacRight) | PACLOWRIGHT;

       PACRESETRIGHT = 0x00;       /* clear PACS */
       PACRESETLEFT  = 0x00;
}

/*************************************************************/
/* Scans the maze for the maximum UV emissions.  Function     */
/*  dead reckons the 360 scan based on the count values on the*/
/*  motor encoders.                                          */
/*************************************************************/
int fire360Scan()
{
    int temp;
    int i, roomNumber;
    int time;
    int room1, room2, room3, room3Plus, room4, home;
    int actRoom[4];
    unsigned char countValue[6];
```

```
    temp = 0;
    time = 750;

    room1 = 4200;
    room2 = 4300;
    room3 = 14400;
    room4 = 2400;
    room3Plus = 2300;
    home = 5000;

    /* first scan - turn right */
    wait(time);
    turnCount(room1, FOLLOW_LEFT, 0);
    PACNT = 0;
    wait(time);
    countValue[0] = PACNT;
    turnCount(room2, FOLLOW_LEFT, 0);
    PACNT = 0;
    wait(time);
    countValue[1] = PACNT;
    turnCount(room3, FOLLOW_LEFT, 0);
    PACNT = 0;
    wait(time);
    countValue[2] = PACNT;
    turnCount(room4, FOLLOW_LEFT, 0);
    PACNT = 0;
    wait(time);
    countValue[3] = PACNT;
    turnCount(room3Plus, FOLLOW_LEFT, 0);
    PACNT = 0;
    wait(time);
    countValue[4] = PACNT;
    turnCount(home, FOLLOW_LEFT, 0);
    PACNT = 0;
    wait(time);
    countValue[5] = PACNT;

    actRoom[0] = countValue[0];
    actRoom[1] = countValue[1];
    actRoom[2] = countValue[3] + countValue[4] + countValue[5];
    actRoom[3] = countValue[3] + countValue[4];

    for(i = 0; i < 4; i++)
    {
        if(actRoom[i] > temp)
        {
           temp = actRoom[i];
           roomNumber = i + 1;
        }
    }
    if(temp == 0)
    {
       roomNumber = 4;
    }
    return(roomNumber);
}

/*************************************************************/
/* Function will wait for desired RTI counts                 */
/*************************************************************/
void wait(int time)
{
    rtiDelay = 0;
    while(rtiDelay < time);
}

/*************************************************************/
/*  REAL TIME INTERRUPT SERVICE ROUTINE                      */
/*************************************************************/
#pragma interrupt_handler rti_isr;
```

```
void rti_isr(void)
{
    getPac(ptrPulseAcc);
    rtiDelay = rtiDelay + 1;
    TFLG2 = 0x40;
}

/*************************************************************/
/*  IRQ Interrupt service routine.  Used for line sensing     */
/*************************************************************/
#pragma interrupt_handler irq_isr
void irq_isr(void)
{
    unsigned char piaInfo, temp;

    piaInfo = PIA_CRA;
    lineFlag = (((piaInfo & 0x80) >> 7) | ((piaInfo & 0x40) >> 6));
    temp = PIA_A;
}

/*************************************************************/
/*  Main wall following routine.  Function will follow a left or*/
/*  right wall based on what the global constant is initialized */
/*  to                                                         */
/*************************************************************/
void wallFollow(void)
{

    unsigned char tempLPWM, tempRPWM;
    unsigned char centerCount;

    rtiDelay = 0;
    lineDelay = 0;
    centerCount = 0;

    while(lineFlag == 0)
    {
        while ((ADCTL & 0x80) == 0);     /* wait till conversion done */
        leftD12 = ADR2;
        rightD12 = ADR1;
        centerD12 = ADR3;
        ADCTL &= ~0x80;

        lineDelay = lineDelay + 1;

        if (followWall == FOLLOW_LEFT)
        {
            /*printf("R = %d, L = %d, C = %d\r\n", ADR1, ADR2, ADR3); */
            if(centerD12 > turnDistance)
            {
                centerCount = centerCount + 1;
                /* turn45Degrees(2, followWall, desiredDC); */
            }
            else
            {
                centerCount = 0;
            }
            if(centerCount > CENTERDISTANCEVALUE)
            {
                if(d12Control != OFF)
                {
                    turn45Degrees(2, followWall, desiredDC);
                    centerCount = 0;
                }
            }
            tempLPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacLeft >> 1)) >> Kp) -
                       ((followLeftDistance - leftD12) >> Kw);
            tempRPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacRight >> 1)) >> Kp) +
                       ((followLeftDistance - leftD12) >> Kw);

            safety(&tempLPWM, &tempRPWM);
```

```
            LPWM = tempLPWM;
            RPWM = tempRPWM;
/*
            printf("RPWM = %d  Pulse/2 = %d\r\n", tempRPWM,(pulseAcc.pacRight >> 1));
            printf("LPWM = %d  Pulse/2 = %d\r\n", (tempLPWM & ~0x80), (pulseAcc.pacLeft >>
1));
*/
        }

        if (followWall == FOLLOW_RIGHT)
        {
           /*printf("R = %d, L = %d, C = %d\r\n", ADR1, ADR2, ADR3);*/
           if(centerD12 > turnDistance)
           {
             centerCount = centerCount + 1;
             /* turn45Degrees(2, followWall, desiredDC); */
           }
           else
           {
               centerCount = 0;
           }
           if(centerCount > CENTERDISTANCEVALUE)
           {
               if(d12Control != OFF)
               {
                   turn45Degrees(2, followWall, desiredDC);
                   centerCount = 0;
               }
           }
           tempLPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacLeft >> 1)) >> Kp) +
                     ((followRightDistance - rightD12) >> Kw);
           tempRPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacRight >> 1)) >> Kp) -
                     ((followRightDistance - rightD12) >> Kw);

           safety(&tempLPWM, &tempRPWM);

           LPWM = tempLPWM;
           RPWM = tempRPWM;
        }
    }
}

/*****************************************************************/
/*  Function initializes the hardware. Sets up the Pulse accumlator*/
/*  real time interrupt, a/d converter, and the PIA expansion chip */
/*****************************************************************/
void initializeHardware(void)
{
    unsigned char temp;

    ADCTL   =   0x33;         /* Scan four channels on PE0 - PE3  */
    PACTL = PACTL | 0x40;
    PACTL = PACTL & (~0x03);
    RTI_JMP = JMP_OP_CODE;
    RTI_VEC = rti_isr;
    TFLG2 = 0x40;
    TMSK2 = TMSK2 | 0x40;

    /* enable PIA interrupts */

    PIA_CRA = 0x00;
    PIA_DDRA = 0x00;

/*
    PIA_CRA = 0x04;
*/

    PIA_CRA = 0x0d;

    temp = PIA_A;
```

```
    IRQ_JMP = JMP_OP_CODE;
    IRQ_VEC = irq_isr;

    /* set-up entire PIA_B for output */
    PIA_CRB = 0x00;
    PIA_DDRB = 0xff;
    PIA_CRB = 0x04;

    enable();                /* enable interrupts in general     */
}

/*****************************************************************/
/*  Positions the robot in the center of the maze.  Needs to know*/
/*  how far from the front wall to stop the robot                */
/*****************************************************************/
void gotoScan(unsigned char StopDistance)
{
    unsigned char tempLPWM, tempRPWM;
    unsigned char centerD12Count;
/*
    printf("In the scan\r\n");
*/
    centerD12Count = 0;
    while(1)
    {
        while ((ADCTL & 0x80) == 0);     /* wait till conversion done */

        tempLPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacLeft >>1))>> Kp) -
                  (((pulseAcc.pacLeft >> 1) - (pulseAcc.pacRight >>1)) * Ks);
        tempRPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacRight >> 1))>> Kp) +
                  (((pulseAcc.pacLeft >> 1) - (pulseAcc.pacRight >>1))
                  * Ks) + 5;

        safety(&tempLPWM, &tempRPWM);

        LPWM = tempLPWM;
        RPWM = tempRPWM;

        centerD12 = ADR3;

        if(centerD12 > StopDistance)
        {
            centerD12Count = centerD12Count + 1;
        }
        else
        {
            centerD12Count = 0;
        }
        if(centerD12Count > CENTERDISTANCEVALUE)
        {
            break;
        }
    }
    LPWM = 0;
    RPWM = 0;
}


/*********************************************************************/
/* Pseudo DR code to enter room one of the maze                      */
/*********************************************************************/
void gotoRoomOne(void)
{
    followWall = FOLLOW_LEFT;
    turn45Degrees(2, FOLLOW_LEFT, 50);
    wallFollow();
    LPWM = 0;
    RPWM = 0;
    /* Straight(); */
```

```
    enterRoom(100);
    turn45Degrees(1, FOLLOW_LEFT, 0);
    PIA_B = 0xff;
    wait(1000);
    PIA_B = 0x00;


}


/*********************************************************************/
/* Pseudo DR code to enter room two of the maze                      */
/*********************************************************************/
void gotoRoomTwo(void)
{

    d12Control = OFF;
    goBackwards(20, 40);
    wait(150);
    followWall = FOLLOW_RIGHT;
    desiredDC = 40;
    wallFollow();
    PIA_CRA = 0x04;
    LPWM = 0;
    RPWM = 0;
    enterRoom(100);
    turn45Degrees(1, FOLLOW_RIGHT, 0);
    PIA_B = 0xff;
    wait(1000);
    PIA_B = 0x00;

}


/*********************************************************************/
/* Pseudo DR code to enter room three of the maze                    */
/*********************************************************************/
void gotoRoomThree(void)
{
    goBackwards(20, 40);
    wait(150);
    followWall = FOLLOW_LEFT;
    desiredDC = 50;
    wallFollow();
    PIA_CRA = 0x04;
    LPWM = 0;
    RPWM = 0;
    enterRoom(100);
    PIA_B = 0xff;
    wait(1000);
    PIA_B = 0x00;

}


/*********************************************************************/
/* Pseudo DR code to enter room four of the maze                     */
/*********************************************************************/
void gotoRoomFour(void)
{
    followWall = FOLLOW_RIGHT;
    turn45Degrees(2, FOLLOW_RIGHT, 42);
    desiredDC = 42;
    wallFollow();
    PIA_CRA = 0x04;
    LPWM = 0;
    RPWM = 0;
    followWall =  FOLLOW_LEFT;
    turn45Degrees(1, FOLLOW_LEFT, 60);
    LPWM = 0;
    RPWM = 0;
```

```
    enterRoom(100);
    turn45Degrees(1, FOLLOW_LEFT, 0);
    PIA_B = 0xff;
    wait(1000);
    PIA_B = 0x00;




}


/* ass */
void getStraight(void)
{
    printf("IN getstraight\r\n");

    printf("persRight = %d, perLeft = %d\r\n", persistentLeftLineFlag,
persistentRightLineFlag);
    while((persistentLeftLineFlag & persistentRightLineFlag) == 0)
    {
        printf(" in loop => right = %d, left = %d \r\n",persistentLeftLineFlag,
persistentRightLineFlag);
        LPWM = 25*persistentRightLineFlag;
        RPWM = 25*persistentLeftLineFlag;
    }

}

/*****************************************************************/
/*  Waits for a desired number or RTI counts, and then grabs the  */
/*  value out of the HC11 pulse accumulator which is the number of*/
/*  Hamamatsu counts the HC11 received.                          */
/*****************************************************************/
unsigned int checkHama(unsigned int rtiTime)
{

    PACNT = 0;
    wait(rtiTime);
    return(PACNT);
}


/*****************************************************************/
/*  Function will wall follow for a desired number or RTI counts  */
/*****************************************************************/
void enterRoom(unsigned char rtiCount)
{

    unsigned char tempLPWM, tempRPWM, temp;
    unsigned char centerCount;

    rtiDelay = 0;
    centerCount = 0;
        lineDelay = 0;
        lineFlag = 0;

    while ((rtiDelay < rtiCount))
    {
        while ((ADCTL & 0x80) == 0);     /* wait till conversion done */
        leftD12 = ADR2;
        rightD12 = ADR1;
        centerD12 = ADR3;
        ADCTL &= ~0x80;

                lineDelay = lineDelay + 1;


        if (followWall == FOLLOW_LEFT)
                {
```

```
            /*printf("R = %d, L = %d, C = %d\r\n", ADR1, ADR2, ADR3); */
            if(centerD12 > turnDistance)
            {
               centerCount = centerCount + 1;
               /* turn45Degrees(2, followWall, desiredDC); */
            }
            else
            {
               centerCount = 0;
            }
            if(centerCount > CENTERDISTANCEVALUE)
            {
               turn45Degrees(2, followWall, desiredDC);
               centerCount = 0;
            }
            tempLPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacLeft >> 1)) >> Kp) -
                       ((followLeftDistance - leftD12) >> Kw);
            tempRPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacRight >> 1)) >> Kp) +
                       ((followLeftDistance - leftD12) >> Kw);

            safety(&tempLPWM, &tempRPWM);

            LPWM = tempLPWM;
            RPWM = tempRPWM;
/*
            printf("RPWM = %d  Pulse/2 = %d\r\n", tempRPWM,(pulseAcc.pacRight >> 1));
            printf("LPWM = %d  Pulse/2 = %d\r\n", (tempLPWM & ~0x80), (pulseAcc.pacLeft >>
1));
*/
               }

        if (followWall == FOLLOW_RIGHT)
        {
           /*printf("R = %d, L = %d, C = %d\r\n", ADR1, ADR2, ADR3);*/
           if(centerD12 > turnDistance)
           {
             centerCount = centerCount + 1;
             /* turn45Degrees(2, followWall, desiredDC); */
           }
           else
           {
               centerCount = 0;
           }
           if(centerCount > CENTERDISTANCEVALUE)
           {
               turn45Degrees(2, followWall, desiredDC);
               centerCount = 0;
           }
           tempLPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacLeft >> 1)) >> Kp) +
                      ((followRightDistance - rightD12) >> Kw);
           tempRPWM = (desiredDC) + ((desiredDC -(pulseAcc.pacRight >> 1)) >> Kp) -
                      ((followRightDistance - rightD12) >> Kw);

           safety(&tempLPWM, &tempRPWM);

           LPWM = tempLPWM;
           RPWM = tempRPWM;
               }
        if (lineDelay > 5)
               {
                      temp = PIA_A;
                   PIA_CRA = 0x0d;
               }
        }



     /*   turn45Degrees(2,followWall,0); */

        LPWM = 0;
    RPWM = 0;
```

```
}

/*****************************************************************/
/*  Function will have the robot move backwards until front sensor*/
/*  is less than or equal to the distance that is passed into the */
/*  fucntion.  Function also needs to be passed a desired duty    */
/*  to run backwards at.                                         */
/*****************************************************************/
void goBackwards(unsigned int threshold, unsigned int ddc)
{
    unsigned char tempLPWM, tempRPWM;
    unsigned char centerD12Count;
/*
    printf("In the scan\r\n");
*/
    while(1)
    {
        while ((ADCTL & 0x80) == 0);     /* wait till conversion done */

        tempLPWM = (ddc) + ((ddc -(pulseAcc.pacLeft >>1)) >> Kp) -
                  (((pulseAcc.pacLeft >> 1) - (pulseAcc.pacRight >>1)) * Ks);
        tempRPWM = (ddc) + ((ddc -(pulseAcc.pacRight >> 1))>> Kp) +
                  (((pulseAcc.pacLeft >> 1) - (pulseAcc.pacRight >>1))
                  * Ks) + 5;

        safety(&tempLPWM, &tempRPWM);

        LPWM = (tempLPWM & ~0x80);
        RPWM = (tempRPWM | 0x80);

        centerD12 = ADR3;

        if(centerD12 <= threshold)
        {
            break;
        }
    }
    LPWM = 0;
    RPWM = 0;
}

/*****************************************************************/
/*  Untested function to position the robot in front of the flame  */
/*****************************************************************/
void scantochaseFlame(unsigned char roomCount)
{
    unsigned char tempMaximum;
        unsigned char tempxMax;
        unsigned char midMaximum;
        unsigned char x;
        unsigned char maxCount;
        unsigned int tempLeft;
    unsigned int tempRight;

        ADCTL = ADCTL | 0x37;
        scanDivision = 0;
    tempLeft = 0;
    tempRight = 0;
    /* disable RTI */
    TMSK2 = TMSK2 & (~0x40);

    if (followWall == FOLLOW_LEFT)
        {
          for(scanDivision=0; scanDivision < roomCount; scanDivision++)
          {
        /* clear the PACS */
        PACRESETRIGHT = 0x00;
        PACRESETLEFT = 0x00;

               while ((ADCTL & 0x80) == 0);     /* wait till conversion done */
               flameIntensity[scanDivision] = (ADR1 + ADR2) >> 1;
```

```
        ADCTL &= ~0x80;
    LPWM = (20 | 0x80);  /*FOLLOW_LEFT => turn right*/
    RPWM = (20 | 0x80);
    while(tempLeft < 2000)
    {
        tempLeft = 0;
        tempRight = 0;
        tempLeft = PACHIGHLEFT << 8;
        tempRight = PACHIGHRIGHT << 8;
        tempLeft = tempLeft | PACLOWLEFT;
        tempRight = tempRight | PACLOWRIGHT;
    }
        LPWM = 0;
        RPWM = 0;
      }
    }
else
    {
      for(scanDivision=0; scanDivision < roomCount; scanDivision++)
      {
    /* clear the PACS */
    PACRESETRIGHT = 0x00;
    PACRESETLEFT = 0x00;

        while ((ADCTL & 0x80) == 0);     /* wait till conversion done */
        flameIntensity[scanDivision] = (ADR1 + ADR2) >> 1;
        ADCTL &= ~0x80;
    LPWM = 20;  /*FOLLOW_RIGHT => turn left*/
    RPWM = 20;
    while(tempLeft < 2000)
    {
        tempLeft = 0;
        tempRight = 0;
        tempLeft = PACHIGHLEFT << 8;
        tempRight = PACHIGHRIGHT << 8;
        tempLeft = tempLeft | PACLOWLEFT;
        tempRight = tempRight | PACLOWRIGHT;
    }
        LPWM = 0;
        RPWM = 0;
      }
    }


    tempMaximum = flameIntensity[0];
    tempxMax = 1;
    midMaximum = 1;
    for (x = 1; x < roomCount; x++)
{
        if (flameIntensity[x] == tempMaximum)
        {
            maxCount = maxCount + 1;
        }
        else
        {
            if (midMaximum < maxCount)
            {
                midMaximum = maxCount;
            }
            maxCount = 0;
        }

        if (flameIntensity[x] >= tempMaximum)
        {
            tempMaximum = flameIntensity[x];
            tempxMax = x;
        }

        midMaximum = midMaximum >> 1;
        tempxMax = tempxMax + midMaximum;
}
```

```
        flameIntensityMaximum = tempMaximum;
        intensityMaximumPosition = (roomCount - midMaximum);

        if (followWall == FOLLOW_LEFT)
        {
                        /* clear the PACS */
        PACRESETRIGHT = 0x00;
        PACRESETLEFT = 0x00;

        LPWM = 20;  /*FOLLOW_LEFT => turn left back to maximum flame intensity*/
        RPWM = 20;
        while(tempLeft < intensityMaximumPosition)
        {
            tempLeft = 0;
            tempRight = 0;
            tempLeft = PACHIGHLEFT << 8;
            tempRight = PACHIGHRIGHT << 8;
            tempLeft = tempLeft | PACLOWLEFT;
            tempRight = tempRight | PACLOWRIGHT;
        }
                LPWM = 0;
                RPWM = 0;
        }
        else
        {
                        /* clear the PACS */
        PACRESETRIGHT = 0x00;
        PACRESETLEFT = 0x00;

        LPWM = (20 | 0x80);  /*FOLLOW_RIGHT => turn right back to maximum flame
intensity*/
        RPWM = (20 | 0x80);
        while(tempLeft < intensityMaximumPosition)
        {
            tempLeft = 0;
            tempRight = 0;
            tempLeft = PACHIGHLEFT << 8;
            tempRight = PACHIGHRIGHT << 8;
            tempLeft = tempLeft | PACLOWLEFT;
            tempRight = tempRight | PACLOWRIGHT;
        }
                LPWM = 0;
                RPWM = 0;
        }
}
```