Chapter 1: INTRODUCTION

1.1 AIM OF THE PROJECT

To create an RC Car that can identify a parking space and parallel park by itself. The RC Car drives down a street searching for a parking space to its side using a distance sensor. When the car has identified a space,the car checks to see whether that space is large enough to park in it. If it determines that there is sufficient space, the car will begin parallel parking into that space. It uses information from sensors placed on the front, side, and rear of the car to direct the car into the parking space. Once the car gets parked, it will remain in that position until it is reset.

Objectives

The following is the list of objectives for the Autonomous Vehicle:

1. Easy to Use

1.1. Can be turned on and off with the touch of a button.

1.2. Vehicle will be able to control its autonomous behaviour

1.3. Battery can be changed easily and charged quickly.

2. Marketable

2.1. Can be used indoors and outdoors.

2.2. Can be operated on various surfaces, including wood floors, tile, and carpet.

2.3. Vehicle can be used during day or night (light or dark and shade).

3. Functionality

3.1. Vehicle will be fast and lightweight.

3.2. Vehicle will be able to control its speed depending on its distance from objects.

3.3. Vehicle will be able to make sharp turns.

Constraints

The following is a list of constraints for the Autonomous Vehicle:

1. Functionality

1.1. Vehicle must be able to travel at a good speed.

1.2. Vehicle must not weigh much (including battery weight).

2. Easy to Use

2.1. Vehicle must initiate autonomous behaviour automatically.

2.2. Must pause for few seconds after continuous operation.

1.2 Assumptions and Limitations

As was described in the previous section, the main goal of this vehicle design is to implement a small RC sized vehicle that can parallel park itself, while manoeuvring around objects located in its path. The design entailed assumptions and limitations that had to be compiled by our design group. Below you will find a list of the assumptions and limitations that were made by our group.

Assumptions:

The following is the list of assumptions for the Autonomous Vehicle design:

- Vehicle must have a nice and rugged body with an attractive design.
- Sensors should not be too obvious on exterior of the car (blend in with design).
- Vehicle must have four wheels.

Limitations:

The following is the list of limitations for the Autonomous Vehicle design:

- Vehicle must be able to operate with a single battery.
- Wheels must function properly on wet and dry floors, as well as carpet.
- Sensors should be of enough range.
- Vehicle must be light weighted.

1.3 Environmental Consideration:

The autonomous vehicle is designed to operate in an indoor and outdoor environment; however, it will not be operable when it is raining due to the susceptibility of the vehicles electronics equipment to water damage. Also, due to the sensitive nature of the vehicles sensors located on the exterior of the car, the vehicle will allow it to tolerate minor hits to the body of the car and falls from short distances as long as the sensors are not directly impacted. This mean that the car cannot be operated in a rocky or bumpy terrain where the sensors would be exposed to damage by direct or indirect impact that would cause them to become loose.

The operating temperature range of the autonomous vehicle is ideally between 40 degrees Fahrenheit and 120 degrees Fahrenheit. This is due to the temperature requirements of the

various electric components of the car, including the infrared sensors, the microcontroller, and the motor driver circuits. Although the vehicle will work when operated slightly outside this range, the components function best when operated at the specified range. However in extreme case when temperature becomes too high, components have the risk of breaking down despite of all considerations taken care of. So temperature is an important issue that is taken care of.

Chapter 2: REQUIREMENTS

The requirements of the project can be broadly divided in two categories:

- **Hardware.**
- **Software**

2.1 Hardware:

The hardware used consists of five basic components:

- RC Car.
- H- Bridge.
- Distance Sensors
- Voltage Regulator
- USB AVR Programmer

We have used the Atmega32 Microcontroller. It takes the data from the IR Sensors placed on the front, back and right of car and then in coordination with the H-Bridge powers the motors. Control of voltage was an important issue, taken care by using an appropriate regulator.

2.2 Software:

- AVR Studio 4

In order to achieve proper functioning of the car, accurate codes and programming was the important requirement. The programming was divided in various logical modules and done in C language. Programming in small modules proved to be more efficient and accurate as the task was simplified and debugging was easier.

Detailed description of hardware and software is done in the following pages.

- Extreme Burner AVR

A GUI Software for programming AVR Microcontrollers. It can drive a USBasp compatible hardware. USBasp is a USB programmer for AVR Microcontrollers.

Chapter 3: HARDWARE

3.1 RC CAR:

The first step of our hardware design involved fully understanding the mechanics our RC car, what every part in the car is used for, and how those parts contribute to the control of the car. All radio controlled cars have four main parts:

- Transmitter:
  It is held in hands to control the car. It sends Radio waves to the receiver.
- Receiver:
  It comprises of an antenna and a circuit board inside the car. It receives signals from the transmitter and activates motors inside the car as commanded by the transmitter.
- Motor(s) :
  Motors are important requirement for turning the wheels, steering the vehicle, operating the propellers, etc.
- Power source:
  A battery source is provided that fulfils the need of power supply, by the RC Car.

3.1.1 Working of RC Car:

The receiver changes the radio signal broadcast from the transmitter into suitable electrical control signals for the other components of the control system. Most radio systems utilize amplitude modulation for the radio signal and encode the control positions with pulse width modulation.

Electronic speed controls are commanded by the receiver through pulse width modulation; pulse duration sets either the amount of current that an electronic speed control allows to flow into the electric motor.

Most radio systems utilize amplitude modulation for the radio signal and encode the control positions with pulse width modulation. Upgraded radio systems are available that use the more robust frequency modulation and pulse code modulation. The radio is wired up to either electronic speed controls or servomechanisms (shortened to "servo" in common usage) which perform actions such as throttle control, braking, steering, and on some cars, engaging either

forward or reverse gears. Electronic speed controls and servos are commanded by the receiver through pulse width modulation; pulse duration sets either the amount of current that an electronic speed control allows to flow into the electric motor or sets the angle of the servo. On the models the servo is attached to at least the steering mechanism; rotation of the servo is mechanically changed into a force which steers the wheels on the model, generally through adjustable turnbuckle linkages. Servo savers are integrated into all steering linkages and some nitro throttle linkages. A servo saver is a flexible link between the servo and its linkage that protects the servo's internal gears from damage during impacts or stress.

A variety of RC Cars are available, like, electric models, nitro powered models, gas powered models, fuel models. For our design we have used the electric model.

Electric models

Electrically powered models utilize mechanical or electronic speed control units to adjust the amount of power delivered to the electric motor. The power delivered is proportional to the amount of throttle called for by the transmitter - the more you pull the trigger, the faster it goes. The voltage is "pulsed" using transistors to produce varying output with smooth transitions and greater efficiency. Electronic speed controllers use solid state components to regulate duty cycle, adjusting the power delivered to the electrical motor. In addition, most electronic speed controllers can use the electric motor as a magnetic brake, offering better control of the model than is possible with a mechanical speed control. Mechanical speed controllers use a network of resistors and switch between them by rotating a head with an electrode around a plate that has electrical contacts. Mechanical speed controllers are prone to being slow to react because they are actuated by servos, waste energy in the form of heat from the resistor, commonly become dirty and perform intermittently, and lack a dedicated braking ability. They are less expensive than high performance electronic speed controls and usually ship in older hobby-grade models. They are gradually being phased out. Most electric cars up to recently used brushed motors but now many people are turning to brushless motors for their much higher power and because they require much less maintenance. They are rated either in relative turns or Kv. The Kv number tells how many RPM the motor will turn per volt, assuming no load and maximum efficiency. However, the ability of the system to put out power is dependent on the quality of the batteries used, wires and connectors supplying power.

After understanding the mechanics of the car, the easiest way to control our car was found to be to directly control the inputs to the DC brush motors controlling the front and rear wheels, bypassing all of the car’s internal circuitry. To do this, we scoped the control signals of the

car. The control signals were very simple. There is one motor for the reverse and forward movement of the rear wheels and one motor to turn the front wheels left and right. These motors are controlled by a simple 5V DC input. A +5V turns the rear wheels forward and the front wheel to the left. A -5V input turns the rear wheels backwards and turns the front wheels to the right. To more easily control the motors we soldered wires to their plus and minus terminals. This allows us to easily apply a +/- 5V without opening up the car again.

3.2 H- BRIDGE:

An **H-bridge** is an electronic circuit which enables a voltage to be applied across a load in either direction. These circuits are often used in robotics and other applications to allow DC motors to run forwards and backwards.

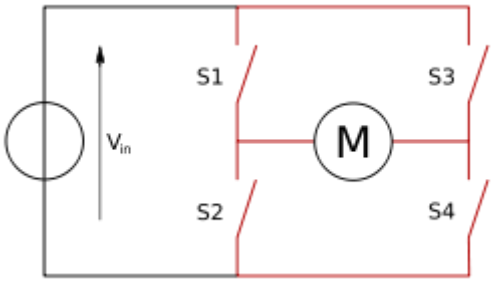


Fig.1. Structure of an H-bridge

The term "H-bridge" is derived from the typical graphical representation of such a circuit. An H-bridge is built with four switches (solid-state or mechanical). When the switches S1 and S4 (according to the first figure) are closed (and S2 and S3 are open) a positive voltage will be applied across the motor. By opening S1 and S4 switches and closing S2 and S3 switches, this voltage is reversed, allowing reverse operation of the motor.

The term "H-bridge" is derived from the typical graphical representation of such a circuit. An H-bridge is built with four switches (solid-state or mechanical). When the switches S1 and S4 (according to the first figure) are closed (and S2 and S3 are open) a positive voltage will be applied across the motor. By opening S1 and S4 switches and closing S2 and S3 switches, this voltage is reversed, allowing reverse operation of the motor.

Using the nomenclature above, the switches S1 and S2 should never be closed at the same time, as this would cause a short circuit on the input voltage source. The same applies to the switches S3 and S4. This condition is known as shoot-through.

3.2.1 OPERATION OF H-BRIDGE :

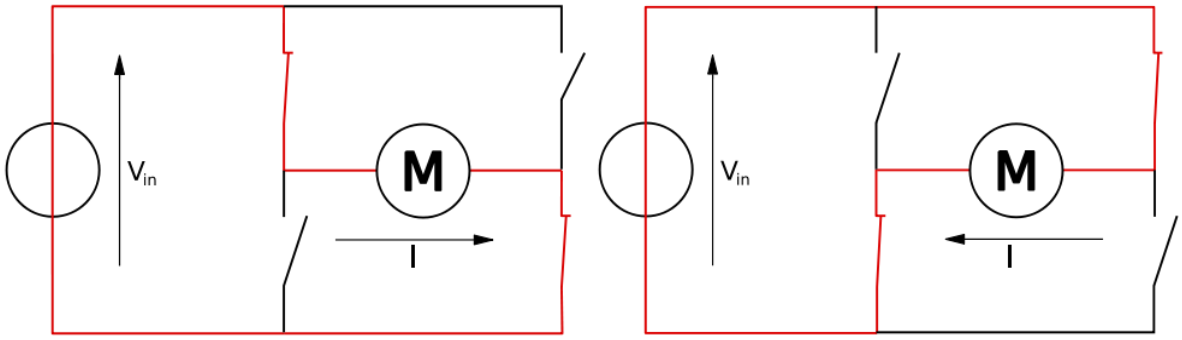


Fig.2 The two basic states of an H-bridge.

The H-Bridge arrangement is generally used to reverse the polarity of the motor, but can also be used to 'brake' the motor, where the motor comes to a sudden stop, as the motor's terminals are shorted, or to let the motor 'free run' to a stop, as the motor is effectively disconnected from the circuit. The following table summarises operation.

S1	S2	S3	S4	Result
1	0	0	1	Motor moves right
0	1	1	0	Motor moves left
0	0	0	0	Motor free runs
0	1	0	1	Motor brakes
1	0	1	0	Motor brakes

Table 1. Operation table of H-Bridge

A solid-state H-bridge is typically constructed using reverse polarity devices (i.e., PNP BJTs or P-channel MOSFETs connected to the high voltage bus and NPN BJTs or N-channel MOSFETs connected to the low voltage bus).

The most efficient MOSFET designs use N-channel MOSFETs on both the high side and low side because they typically have a third of the ON resistance of P-channel MOSFETs. This

requires a more complex design since the gates of the high side MOSFETs must be driven positive with respect to the DC supply rail. However, many integrated circuit MOSFET drivers include a charge pump within the device to achieve this.

3.2.2 ST Micro L298HN H-Bridge :

In the project we use an ST Micro L298HN H-Bridge to control the motors of the RC Car. It allows us to switch between +/-5V across the motor. It also allows us to source the power from the batteries while using the processor to control the transistors in the H-Bridge. The control algorithm turns the appropriate transistors on/off, applying the proper voltage across the brush motor.

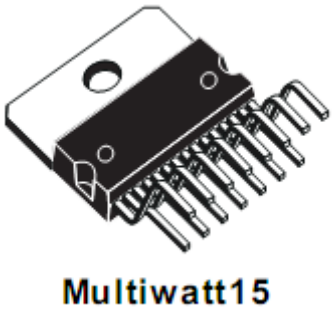


Fig. 3 A typical L298HN H-Bridge.

The L298 is an integrated monolithic circuit in a 15-lead Multiwatt and PowerSO20 packages. We have used 15-lead Multiwatt package.

L298HN H-Bridge is a high voltage, high current dual full-bridge driver designed to accept standard TTL logic levels and drive inductive loads such as relays, solenoids, DC and stepping motors. Two enable inputs are provided to enable or disable the device independently of the input signals.

3.2.2a Properties of L298HN H-Bridge

- Operating supply voltage upto 46 volts.
- Total DC current upto 4A.
- Low saturation voltage.
- Overtemperature protection.
- Logical 0 input voltage up to 1.5A.
  (high noise immunity)

3.2.2b Pin diagram and functions of pins :

Below is shown a pin diagram of an ST Micro L298HN H-Bridge:

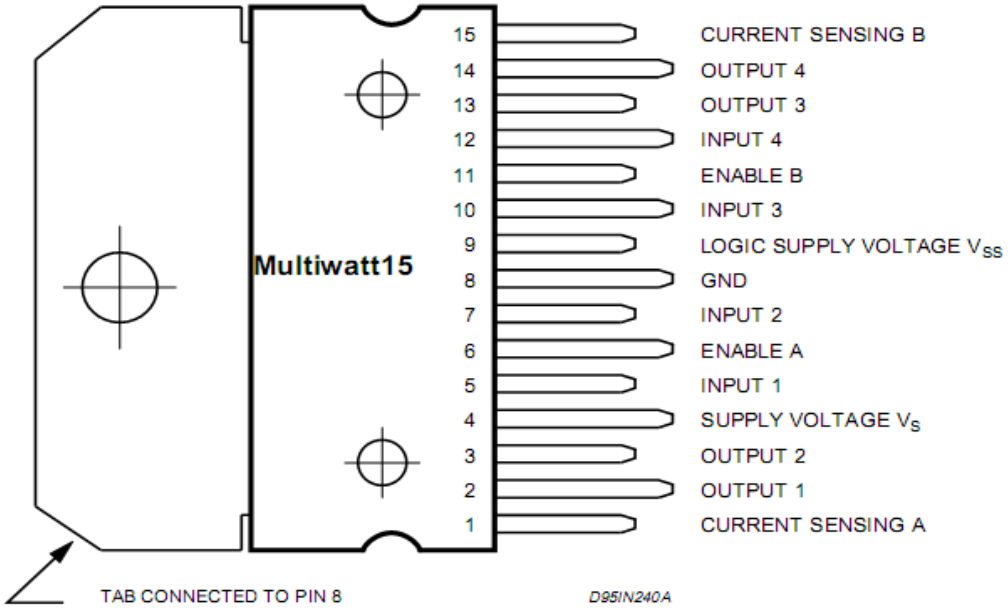


Fig.4. Pin diagram of an ST Micro L298HN H-Bridge.

The various pin functions are explained as:

PIN FUNCTIONS (refer to the block diagram)

MW.15	PowerSO	Name	Function
1;15	2;19	Sense A; Sense B	Between this pin and ground is connected the sense resistor to control the current of the load.
2;3	4;5	Out 1; Out 2	Outputs of the Bridge A; the current that flows through the load connected between these two pins is monitored at pin 1.
4	6	V_S	Supply Voltage for the Power Output Stages. A non-inductive 100nF capacitor must be connected between this pin and ground.
5;7	7;9	Input 1; Input 2	TTL Compatible Inputs of the Bridge A.
6;11	8;14	Enable A; Enable B	TTL Compatible Enable Input: the L state disables the bridge A (enable A) and/or the bridge B (enable B).
8	1,10,11,20	GND	Ground.
9	12	VSS	Supply Voltage for the Logic Blocks. A100nF capacitor must be connected between this pin and ground.
10; 12	13;15	Input 3; Input 4	TTL Compatible Inputs of the Bridge B.
13; 14	16;17	Out 3; Out 4	Outputs of the Bridge B. The current that flows through the load connected between these two pins is monitored at pin 15.
–	3;18	N.C.	Not Connected

Table 2. Pin functions table.

3.2.2c H-Bridge schematic:

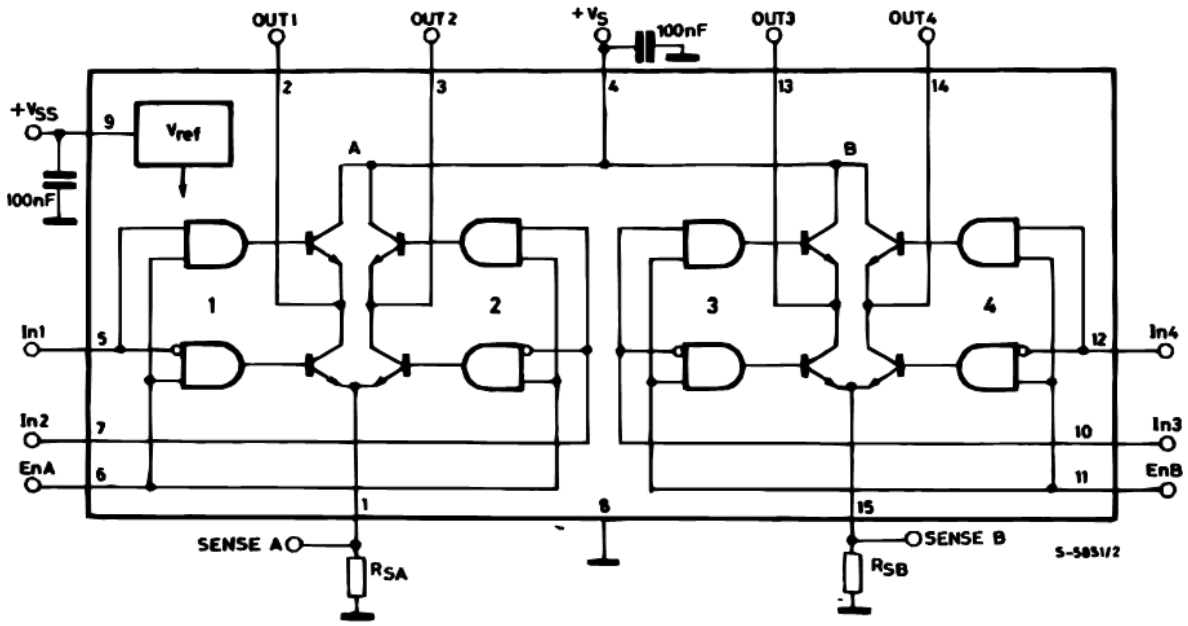


Fig. 5 Block diagram of L298HN H-Bridge

The various pin configurations of H-Bridge when used with motor are explained as below:

Front Motor (Left/Right)	Rear Motor (Forward/Reverse)

Pin	Connected To	Pin	Connected To
In 1	Port B7	In 3	Port B3
In 2	Port B6	In 4	Port B2
En A	Port B5	En B	Port B1
Out 1	+ Motor Terminal	Out 3	+ Motor Terminal
Out 2	- Motor Terminal	Out 4	- Motor Terminal

Table 3: H-Bridge Pin Configuration with motor.

In addition configuring the H-Bridge to control the motors, the H-Bridge needs to be protected from inductive spikes caused by turning the DC brush motors on and off. We used diodes on the output to protect from these spikes.

The H-Bridge is wired as follows:

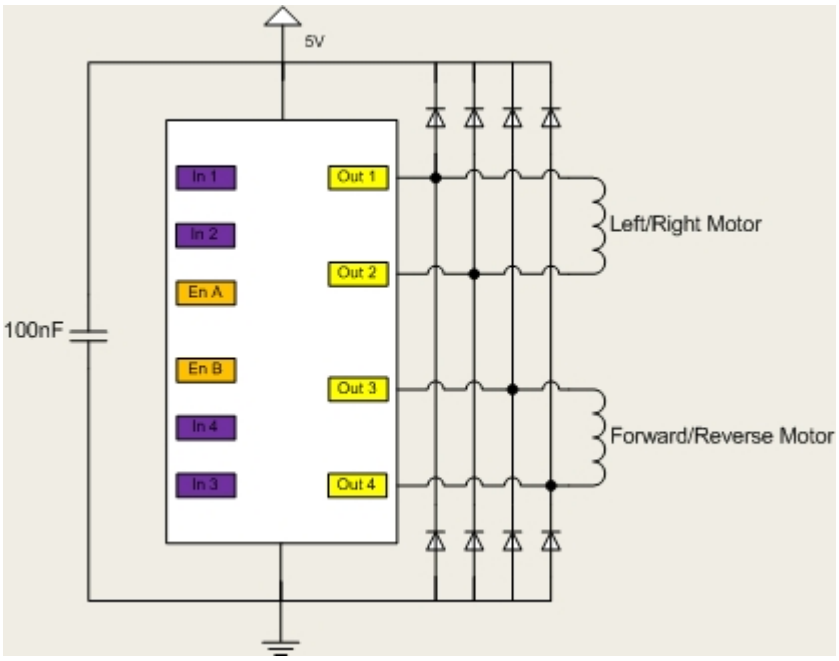


Fig. 6: Inductive Current Protection on H-Bridge Outputs

The diodes are preferred to be shottky diodes.

3.3 DISTANCE SENSORS :

In the project infra-red distance sensors are used to determine the distance between our car and nearby objects.

3.3.1 Basic working of an IR Sensor :

These sensors use triangulation to compute the distance and/or presence of objects in the field of view. The basic idea is this: a pulse of IR light is emitted by the emitter. This light travels out in the field of view and either hits an object or just keeps on going. In the case of no object, the light is never reflected and the reading shows no object. If the light reflects off an object, it returns to the detector and creates a triangle between the point of reflection, the emitter, and the detector.

The angles in this triangle vary based on the distance to the object. The receiver portion of these new detectors is actually a precision lens that transmits the reflected light onto various portions of the enclosed linear CCD array based on the angle of the triangle described above. The CCD array can then determine what angle the reflected light came back at and therefore, it can calculate the distance to the object.

Below is shown a figure which clears the concept.

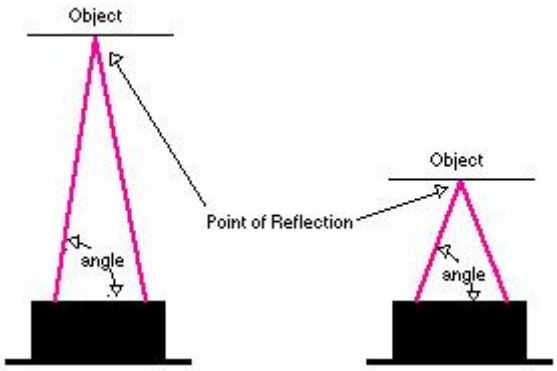


Fig.7 Figure explaining angle formation in a typical IR Sensor.

This new method of ranging is almost immune to interference from ambient light and offers amazing indifference to the colour of object being detected. Detecting a black wall in full sunlight is now possible

In our project we have used three infrared distance sensors to determine the distance between our car and nearby objects. We placed a sensor on the front, the right side, and the rear of the car. For the front and rear, we used 4-30cm sensors. For the right side, we used we used a 10-80cm sensor. We decided to use a sensor with a larger range for the side so that we could more easily detect a parking space. However, this made aligning the parking the car more difficult, so the relies more heavily on the front and rear sensors to park the car. To slightly improve the short distance range of the sensors, the sensors were placed as far back on the car as possible.

3.3.2 TSOP Based obstacle detection sensor module(Single TSOP) :

We have used three TSOP based IR sensors in our project. They are placed at the front, rear and at the right of our RC car.

3.3.2a General Description:

The TSOP-OBSD–Single is a general purpose proximity sensor. Here we use it for collision detection. The module consist of a IR emitter and TSOP receiver pair. The high precision TSO receiver always detects a signal of fixed frequency. Due to this, errors due to false

detection of ambient light are significantly reduced. The module consists of 555 IC, working in astable multivibrator configuration. The output of TSOP is high whenever it receives a fixed frequency and low otherwise. The on-board LED indicator helps user to check status of the sensor without using any additional hardware.

The power consumption of this module is low. It gives a digital output and false detection due ambient light is low.

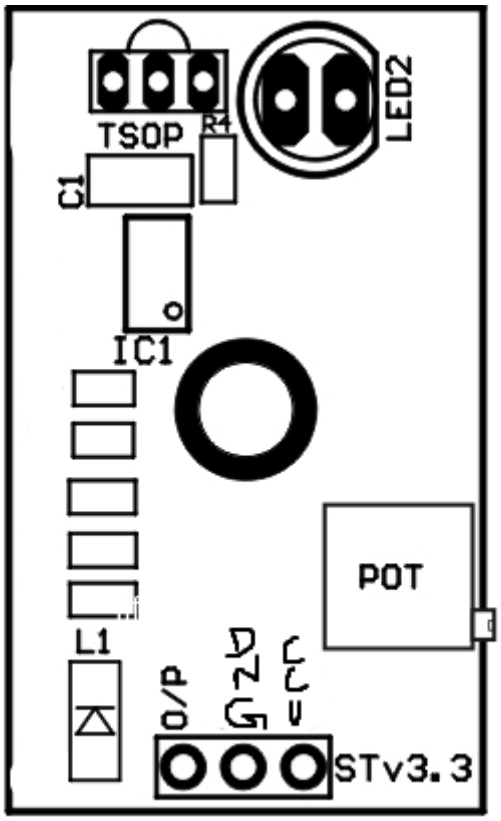


Fig.8. TSOP IR Sensor Circuit Layout

3.3.2b Pin Configuration:

The above figure is a top view of the TSOP module. The following is its pin description-

Pin No.1- Connection: Output | Description: Digital Output (High or Low)

Pin No.3- Connection: VCC | Description: Connected to circuit supply

Pin No.2- Connection: GND | Description: Connected to circuit ground

3.3.2c Functional Block Diagram /Schematic Diagram:

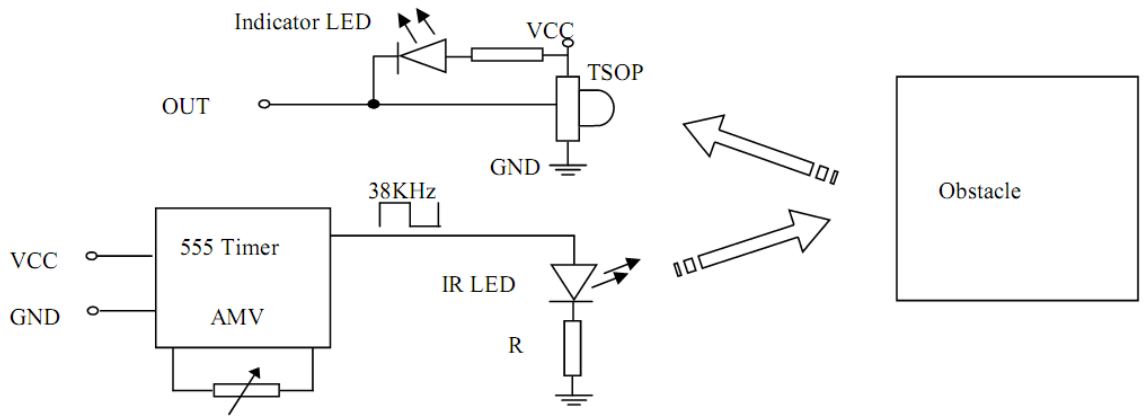


Fig.9. Functional block diagram of TSOP Sensor

3.3.2d Overview of Schematic:

The 555 is used as a astable multivibrator. The frequency of the 555 is tuned using the potentiometer. The output of 555 is given to the IR transmitter. TSOP detects a frequency of 38 KHz. The output of TSOP goes low when it receives this frequency. Hence the output pin is normally high because, though the IR LED is continuously transmitting, due to no obstacle, nothing is reflected back to the TSOP. The indication LED is off. When an obstacle is encountered, the output of TSOP goes low, as the required frequency is reflected from the obstacle surface. This output is connected to the cathode of the LED, which then turns ON.

3.3.2e Maximum Ratings :

Symbol	Quantity	Minimum	Typical	Maximum	Unit
o/p	Output Voltage	0	-	5	V
V_{CC}	Operating Voltage	4.5	5	5.5	V
GND	Ground Reference voltage	-	0	-	V

Table.4. Maximum and Minimum rating of various pins

3.4 The Microcontroller :

A **microcontroller** (also microcomputer, MCU or µC) is a small computer on a single integrated circuit consisting internally of a relatively simple CPU, clock, timers, I/O ports, and memory. Program memory in the form of NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for small or dedicated applications. Thus, in contrast to the microprocessors used in personal computers and other high-performance or general purpose applications, simplicity is emphasized. Some microcontrollers may use four-bit words and operate at clock rate frequencies as low as 4 kHz, as this is adequate for many typical applications, enabling low power consumption (milliwatts or microwatts). They will generally have the ability to retain functionality while waiting for an event such as a button press or other interrupt; power consumption while sleeping (CPU clock and most peripherals off) may be just nanowatts, making many of them well suited for long lasting battery applications. Other microcontrollers may serve performance-critical roles, where they may need to act more like a digital signal processor (DSP), with higher clock speeds and power consumption.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, and toys. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems.

We have used Atmega32 Microcontroller form ATMEL in our project.

3.4.1 ATMEGA-32 :

We have used ATMEGA-32 as the microcontroller. A detailed description of it is given in following pages.

The ATmega32 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega32 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

3.4.2 Brief introduction to ATMEGA32 :

As explained above, ATMEGA32 is based on the AVR enhanced RISC Architecture. The AVR core combines a rich instruction set with 32 general purpose working registers. All the 32 registers in it are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

The ATmega32 provides the following features: 32K bytes of In-System Programmable Flash Program memory with Read-While-Write capabilities, 1024 bytes EEPROM, 2K byte SRAM, 32 general purpose I/O lines, 32 general purpose working registers, a JTAG interface for Boundaryscan, On-chip Debugging support and programming, three flexible Timer/Counters with compare modes, Internal and External Interrupts, a serial programmable USART, a byte oriented Two-wire Serial Interface, an 8-channel, 10-bit ADC with optional differential input stage with programmable gain (TQFP package only), a programmable Watchdog Timer with Internal Oscillator, an SPI serial port, and six software selectable power saving modes. The Idle mode stops the CPU while allowing the USART, Two-wire interface, A/D Converter, SRAM, Timer/Counters, SPI port, and interrupt system to continue functioning. The Power-down mode saves the register contents but freezes the Oscillator, disabling all other chip functions until the next External Interrupt or Hardware Reset. In Power-save mode, the Asynchronous Timer continues to run, allowing the user to maintain a timer base while the rest of the device is sleeping. The ADC Noise Reduction mode stops the CPU and all I/O modules except Asynchronous Timer and ADC, to minimize switching noise during ADC conversions. In Standby mode, the crystal/resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low-power consumption. In Extended Standby mode, both the main Oscillator and the Asynchronous

Timer continue to run. The device is manufactured using Atmel's high density nonvolatile memory technology. The Onchip ISP Flash allows the program memory to be reprogrammed in-system through an SPI serial interface, by a conventional nonvolatile memory programmer, or by an On-chip Boot program running on the AVR core. The boot program can use any interface to download the application program in the Application Flash memory. Software in the Boot Flash section will continue to run while the Application Flash section is updated, providing true Read-While-Write operation. By combining an 8-bit RISC CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATmega32 is a powerful microcontroller that provides a highly-flexible and cost-effective solution to many embedded control applications.

The ATmega32 AVR is supported with a full suite of program and system development tools including: C compilers, macro assemblers, program debugger/simulators, in-circuit emulators, and evaluation kits.

3.4.3 Features :

- High-performance, Low-power AVR® 8-bit Microcontroller.
- Advanced RISC Architecture.
    - 131 Powerful Instructions – Most Single-clock Cycle Execution.
    - 32 x 8 General Purpose Working Registers.
    - Fully Static Operation.
    - Up to 16 MIPS Throughput at 16 MHz.
    - On-chip 2-cycle Multiplier.
- High Endurance Non-volatile Memory segments.
    - 32K Bytes of In-System Self-programmable Flash program memory.
    - 1024 Bytes EEPROM.
    - 2K Byte Internal SRAM.
    - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM.
    - Data retention: 20 years at 85°C/100 years at 25°C(1).
    - Optional Boot Code Section with Independent Lock Bits.
- In-System Programming by On-chip Boot Program.
- True Read-While-Write Operation.
    - Programming Lock for Software Security.
- JTAG (IEEE std. 1149.1 Compliant) Interface.

- Boundary-scan Capabilities According to the JTAG Standard.
- Extensive On-chip Debug Support.
- Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface.

- Peripheral Features.
  - Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes.
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture mode.
  - Real Time Counter with Separate Oscillator.
  - Four PWM Channels.
  - 8-channel, 10-bit ADC.
  - Byte-oriented Two-wire Serial Interface.
  - Programmable Serial USART.
  - Master/Slave SPI Serial Interface.
  - Programmable Watchdog Timer with Separate On-chip Oscillator.
- Special Microcontroller Features.
  - Power-on Reset and Programmable Brown-out Detection.
  - Internal Calibrated RC Oscillator.
  - External and Internal Interrupt Sources.
  - Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby and Extended Standby.
- I/O and Packages.
  - 32 Programmable I/O Lines.
  - 40-pin PDIP, 44-lead TQFP, and 44-pad QFN/MLF.
- Operating Voltages.
  - 2.7 - 5.5V for ATmega32L.
  - 4.5 - 5.5V for ATmega32.
- Power Consumption at 1 MHz, 3V, 25°C for ATmega32L.
  - Active: 1.1 mA.
  - Idle Mode: 0.35 mA.
  - Power-down Mode: < 1 μA.

3.5 USB AVR Programmer v2.0 :

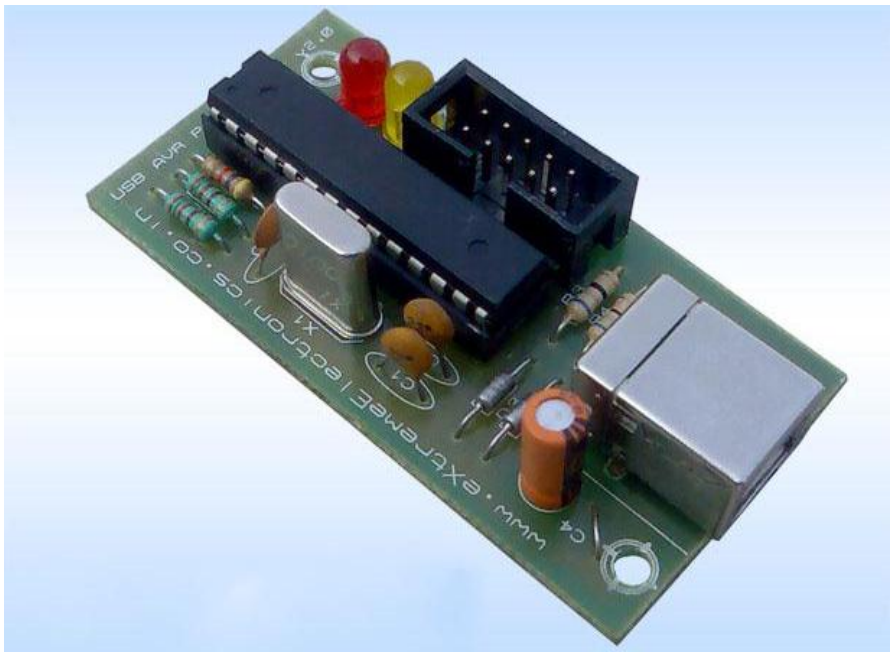


Fig.10. USB AVR Programmer

The USB AVR programmer developed by Extreme Electronics is a complete solution for programming popular AVR series of micro controllers with ease. This is a USB based programmer and is much more faster than the Serial or Parallel port programmers.

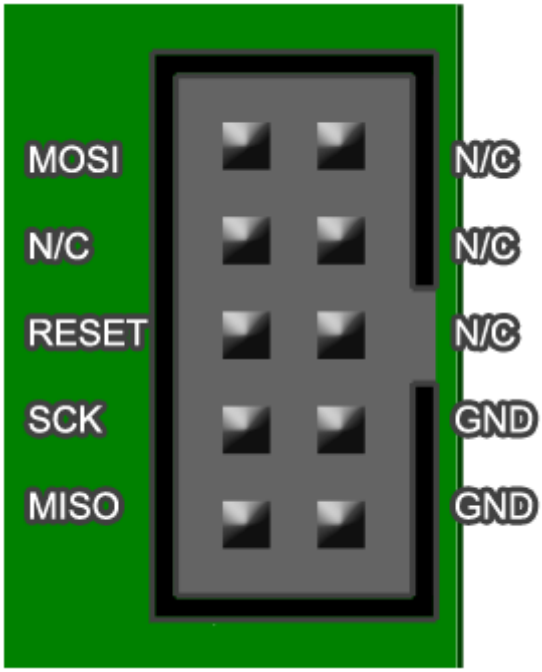
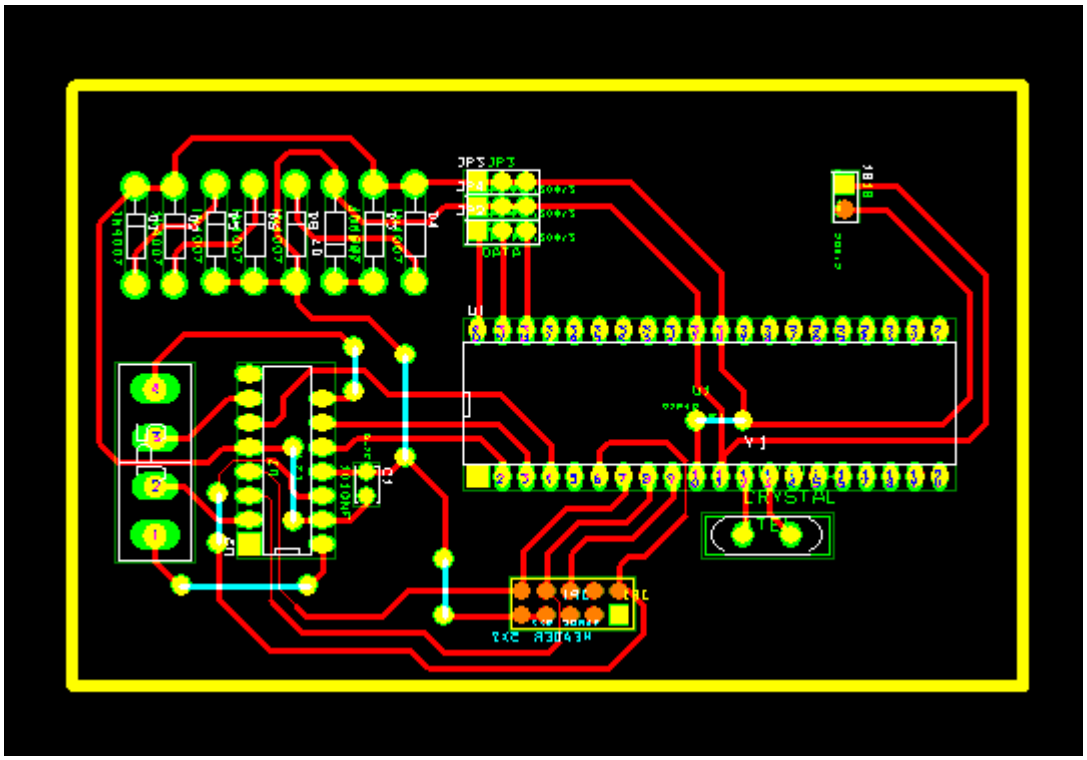


Fig.11.USB Programmer FRC Cable connections

Final Circuit Layout



Chapter 4: SOFTWARE AND PROGRAMMING

4.1 SOFTWARE:

4.1.1AVR Studio 4:

Fig.12. AVR Studio 4 Logo

AVR Studio 4 is the Integrated Development Environment (IDE) for developing 8-bit AVR applications in Windows NT/2000/XP/Vista/7 environments.

AVR Studio 4 provides a complete set of features including debugger supporting run control including source and instruction-level stepping and breakpoints; registers, memory and I/O views; and target configuration and management as well as full programming support for standalone programmers.

4.1.1a AVR Studio 4 features : [Version Used: 4.17 (Build 666)]

- Integrated Assembler
- Integrated Simulator
- Integrates with GCC compiler plug-in
- Support for all Atmel tools that support the 8-bit AVR architecture, including the AVR ONE!, JTAGICE mkI, JTAGICE mkII, AVR Dragon, AVRISP, AVR ISPmkII, AVR Butterfly, STK500 and STK600
- AVR RTOS plug-in support
- support for AT90PWM1 and ATtiny40
- Command Line Interface tools updated with TPI support
- Online help

4.1.2 Extreme Burner AVR:

A GUI Software for programming AVR Microcontrollers. It can drive a USBasp compatible hardware. USBasp is a USB programmer for AVR Microcontrollers.

4.1.2a Extreme Burner AVR Features:

- Graphical User Interface (GUI) for ease of operation.
- Reading of On Chip Flash Memory, EEPROM Memory and Fuse Bits.
- Writing of On Chip Flash Memory, EEPROM Memory and Fuse Bits.
- Open/Save Hex file.
- Direct EEPROM area editing.
- Reliable Error Detection and Correction.
- Easy Installation.

4.2 PROGRAMMING:

Before programming, the logical structure for the design was prepared. This provided a vision of what was required to be done and how to achieve the requirements. Thus it simplified the task of designing the programming concept.

The logical breakdown is explained as below.

4.2.1 Logical Structure

Our project is broken down into two major components: the control system and the move car algorithm. The move car algorithm directs the car and the control system implements the directions of the move car algorithm.

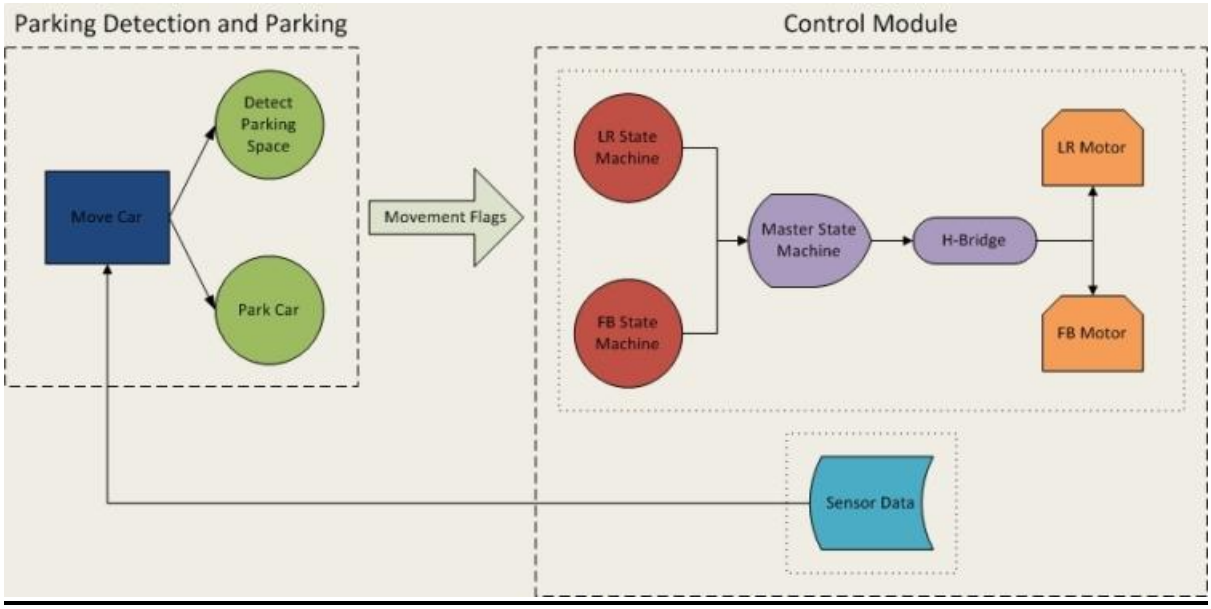


Fig.13. Logical Structure of Design

Control System

The control system contains all the hardware and its associated software. It allows the parking and parking detection algorithms to interface with the car. The software in this module is broken up into three major sections: the Left-Right/Front-Back (LR/FB) state machines, master state machine, and distance calculations. The LR/FB state machines determines which direction to move the car based on flags set by the detect parking space and park car algorithms. Once the LR/FB state machines decides which direction to move the car, the master state machine implements this movement by sending the correct input and enable signals to the H-Bridge. The distance calculations implemented independently every millisecond.

Movement System

Move car contains the detect parking space and parallel parking algorithms. All functions in move car interface with the control module by setting movement flags. The parking space detection and parking algorithms use information from the distance sensors to set these movement flags and guide the car.

Move car works by initializing the movement flags of the car. It sets the car on a default trajectory and then calls detect parking space. Once a parking space has been detected, the parking algorithm is called. After the car has successfully parked, it idles until it is reset.

4.2.2 Software Module Breakdown And The State Machines :

The software for this project has been partitioned into 2 files based on functionality. There are 2 files, ControlModule.c and AlgorithmModule.c.

The state machines in ControlModule control the motors, and are:

- fbStateMachine
- lrStateMachine
- masterStateMachine

The state machines in the AlgorithmModule use the sensor data and various algorithms to determine what should be the next movement the car must make. They assert flags which tell the ControlModule state machines to actually move the motors.

The state machines in AlgorithmModule are:

- moveCar
- detectParking
- parkCar

The various state machines we have used in the project are explained as follows:

Control Module

fbStateMachine()

Function:

The fbStateMachine controls the motor for Forward-Backward operations. It is controlled by the isForward and isReverse flags. These flags serve as indicators as to whether the car should be traveling forward or reverse. In order to control the velocity of the forward-backward motion we anded the enable bit with a a PWM signal.

Working:

In State 0, the motor is at rest. The corresponding FB control bits are 00. When the algorithm requires the car to go forward or reverse, the corresponding flags (isForward and isReverse) are set, and the FB state machine switches states to 1 or 3 respectively.

In State 1, the motor rotates to drive the car forward. The state machine remains in this state while isForward is equal to 1. Once isForward is de-asserted, the state machine moves to a buffer state to stop the car from moving forward due to inertia.

After isForward is set to 0, leaving state 1 and stopping the motor isn't enough. The wheels might continue to rotate due to inertia, and so a buffer state, State 2, is required. It makes the motor go in Reverse for 1 cycle (50ms) of the FB State Machine, before going back to the rest state, State 0.

If isReverse is asserted, the state machine jumps to State 3. The state machine remains in this state while isReverse is equal to 1. Once isReverse is de-asserted, the state machine moves to a buffer state to stop the car from moving in reverse due to inertia.

After State 3, a buffer state, State 4, is needed to stop the wheels from continuing to rotate in reverse due to inertia. This is a 1 cycle Forward motion, similar in function to State 2's reverse functionality. Once done, the FB State Machine goes back to its rest state, State 0.

Timing:

The fbStateMachine is called upon every 50ms. This is enough time to evaluate the flags set in the AlgorithmModule, but at the same time fast enough to make the motor motion very accurate.

lrStateMachine()

The lrStateMachine() works the same way are the fbStatemachine. A forward corresponds to a left turn and a right corresponds to a reverse turn.

The diagram for both is shown in the following page. It demonstrates the working of both the state machines.

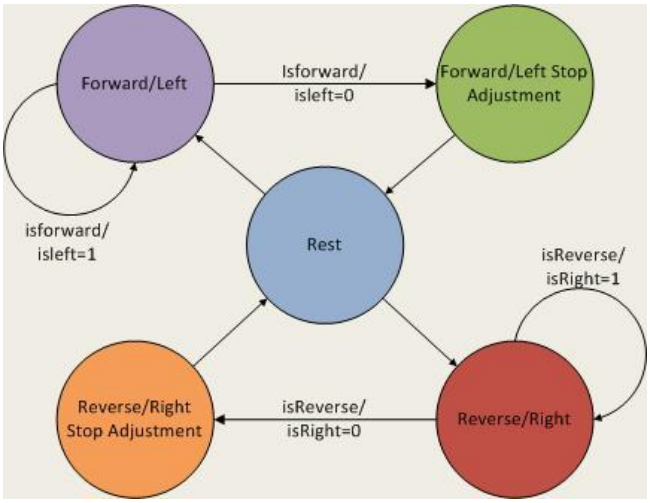


Fig.14. FB/LR Motor State Machine

masterStateMachine()

Function:

This uses the FB and LR control bits to call the required functions in order to send the appropriate input signals to the H-Bridge and make the motors rotate in the appropriate direction.

Working:

In this function, the 2 FB and LR control bits are combined to create 4 master control bits by left shifting the FW bits by 2 and adding it to the LR bits.

Therefore,

fbBits = fb.controlBits; // (FB FB)

lrBits = lr.controlBits; // (LR LR)

masterBits = (fbBits<<2) + (lrBits); // (FB FB)(LR LR)

As a result, each of the 7 car movements (stop, forward, forward-left, forward-right, reverse, reverse-left, reverse-right) have a unique masterBits combination associated with them. The master control bits are then used in the function to decide which motor control function is to be called.

Timing

This state machine is invoked in each iteration of the infinite while loop in main. In other words, it can be considered to be executing continuously and not at intervals. This is essential because parking requires a great deal of accuracy when controlling the motors. Therefore, it is neede to update the motors as often as possible, which would require to call masterStateMachine as often as possible.

Algorithm Module

moveCar()

Function:

This is the master state machine of the algorithm module. It decides which mode the car is in, i.e., whether the car is moving forward to detect a parking spot, aligning itself once a parking spot has been detected, or actually going through the motion of parking.

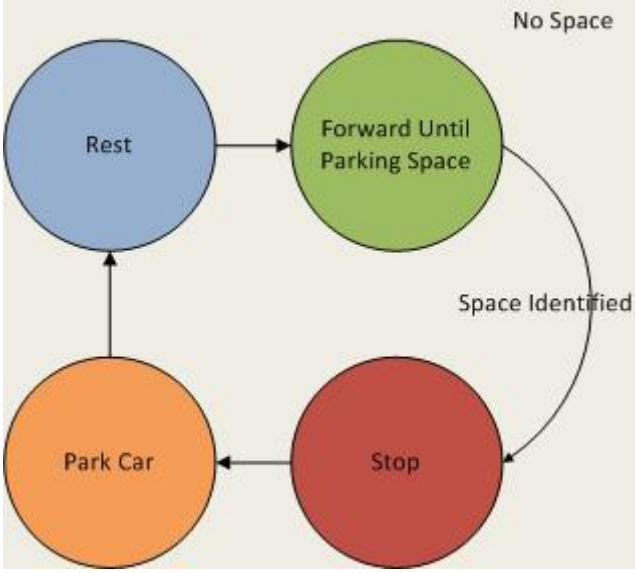


Fig.15. Move Car Motor State Machine

Working:

This is a 5 state linear state machine, as shown in the diagram above.

It starts off in State 0. In this state, the car is at rest. It gives enough time for all transients in the car to stabilize. Once everything is stable, it moves to State 1.

In State 1, car moves forward till it detects a parking spot. While in this state, the car invokes the detectParking state machine each time the moveCar state machine is called in the Control Module.

Once a parking lot has been detected, the state machine moves into State 2. It remains in State 2 until the car has parked itself. The parkCar state machine is invoked for each cycle that the moveCar state machine is in State 2. Once the car has been parked by parkCar state machine, the isParked flag is asserted, and moveCar moves onto state 3.

When we reach State 3, the car parked itself. The car will eternally remain in this state hereafter, since the car has parked itself and is at rest.

Timing:

The moveCar state machine is invoked every 100ms. The moveCar state machine also serves as a clock for the detectParking and parkCar state machines. When in State 1, each clock tick

of the moveCar state machine serves as a clock tick for the detectParking machine. When in State 3, each clock tick of the moveCar state machine serves as a clock tick for the parkCar machine.

detectParking

Function:

The function of detectParking state machine is, as its name suggests, to detect a parking space to park in. It accomplishes this by continuously polling the distance values from the side distance sensor.

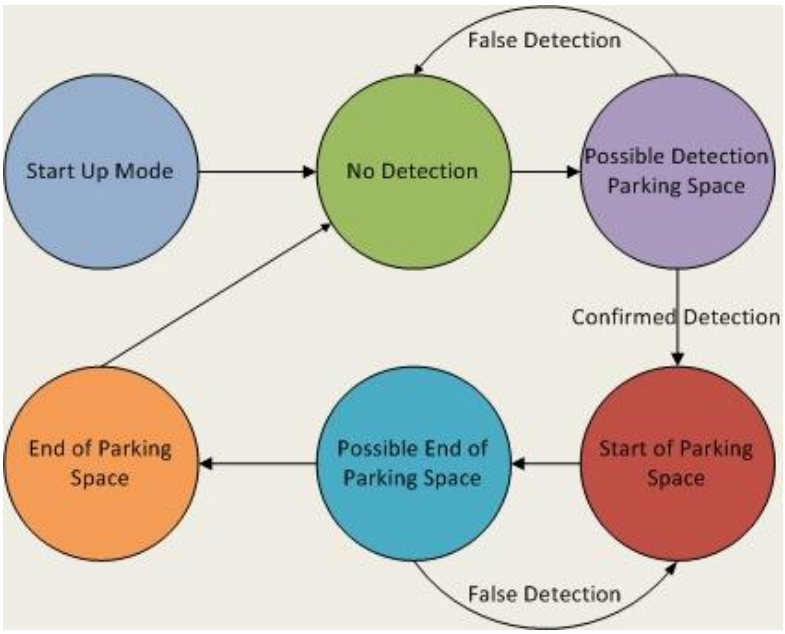


Fig.16. Detect Parking Space State Machine

Working:

detectParking is a 6 state state machine, as can be seen in the diagram above.

State 0 serves as a start-up. This is essential because the first few cycles of the detectParking take place while the side distance sensor is still calibrating itself. Once the wait state is done, the state machine enters state 1.

State 1, essentially, searches for a sudden increase in the side distance value. A sudden increase corresponds to the beginning of a parking space. It does this by checking the (sDistance – rsDist) value. If there is a sudden depression, sDistance will increase and so it's difference from its own previous value (rsDist) will be a large number. When this does occur, the state machine goes onto State 2.

In State 2 it attempts to confirm that it indeed is detecting a valid depression, by calculating (sDistance – rrsDist). Since State 2 is invoked 1 clock tick after the depression was last detected in State 1, rrsDist will store the value of the side distance before the depression

began, i.e., from 2 clock cycles earlier. If (side distance – rrsDist) is still a large number, we can confirm that a depression has been detected, and we move to State 3.

In State 3, we keep track of how long the depression is. This is done by incrementing the detect.controlBits for each state machine clock tick that we are still in the depression. When there is a sudden decrease in the value of the side distance, we move to state 4, since it signals a probable end of the parking lot.

State 4 confirms that the possible end of the parking space, as detected in State 3, is indeed the end of the space. This is done in a manner similar to the confirmation done in State 2 using the rrsDist variable.

Once a parking space has been detected by the above states, the state machine moves into State 5 wherein it checks the control Bits (which kept track of how long the parking space was by incrementing for each cock tick while in the depression) to make sure the parking space is large enough. If large enough, then the isParkingLot flag is asserted which would direct moveCar to stop and start the parking sequence.

Timing:

Each tick of the detectParking state machine corresponds to a tick of the moveCar function. When moveCar is in State 1, it calls detectParking on each of its ticks. Therefore, detectParking is called every 100ms until a parking space has been located.

parkCar()

Function:

The function of the parkCar state machine is to park the car once a parking spot has been identified. The algorithm to park the car continuously interacts with its surroundings through the forward, side and rear sensors.

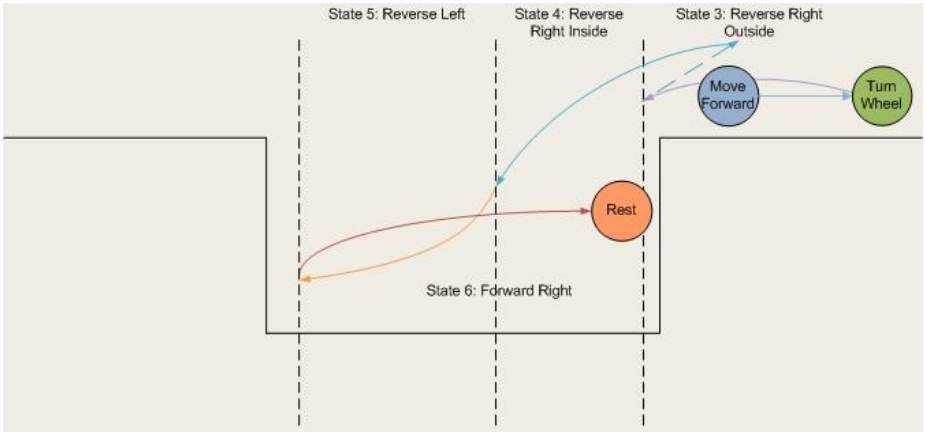


Fig.17. Parking Motion of Car.

Working:

The parkCar function tries to simulate how a human would parallel park. It is, essentially, just the following 4 motions:

1. Reverse Right until car is inside the parking lot.
2. Forward motion and redo 1 if the car is not aligned.
3. Reverse Left until the car is fairly straight and close to the back wall.
4. Forward Right until the car is straight and close to the front wall.

The above routine is accomplished using a 7 state machine.

State 0 makes the car move forward by a certain amount. The idea is to give the car enough space to move and rotate into the parking space.

State 1 simply turns the front wheels to the right. We turn the wheel before reversing the car so as to not lose turning radius by turning as the car reverses. Once the wheel is turned, the state machine moves onto state 2.

State 2 commands the car to go reverse right for a specified amount of time until the car has passed the edge of the parking space. Once past the edge of the space, it moves to state 3.

In State 3, the car continues in reverse right until it is either a certain distance from inside of the parking space, or the rear distance is close to the edge. These conditions, as can be seen from the figure above, are checks to verify that the car is deep enough inside the parking lot to be able execute the reverse left maneuver. Once the conditions are met, the car stops and the state machine moves to state 4.

If at any point in states 1, 2 or 3 the car's AI decides it is not in a position to go through with the parking, it will go back to State 0, and redo the whole procedure.

In State 4, the car moves reverse left. It does this until the rear of the car is close to the side wall of the parking space, which can be judged by the rear distance sensor value. Once close enough to the rear value, it stops and moves to state 5.

State 5 commands the car to go forward right. This attempts to straighten out the car completely and to align it nicely inside the spot. It goes forward right until it is close to the side wall of the parking space, as judged by the forward distance sensor. Once aligned, the car is parked and it moves to state 6.

State 6 is a 1 cycle stop before progressing back to state 0. Also, here the isParked variable is set so that the moveCar state machine can move out of parking mode to rest mode.

Timing:

Each tick of the parkCar state machine corresponds to a tick of the moveCar function. When moveCar is in State 3, it calls parkCar on each of its ticks. Therefore, parkCar is called very 100ms while the car is being parked.

Chapter 5: EMBEDDED SYSTEMS & CODING

5.1 EMBEDDED SYSTEM:

An **embedded system** is a computer system designed to perform one or a few dedicated function often with real-time computing constraints. It is *embedded* as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a personal computer (PC), is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today.

Embedded systems are controlled by one or more main processing cores that are typically either microcontrollers or digital signal processors (DSP).The key characteristic, however, is being dedicated to handle a particular task, which may require very powerful processors. For example, air traffic control systems may usefully be viewed as embedded, even though they involve mainframe computers and dedicated regional and national networks between airports and radar sites. (Each radar probably includes one or more embedded systems of its own.)

Since the embedded system is dedicated to specific tasks, design engineers can optimize it to reduce the size and cost of the product and increase the reliability and performance. Some embedded systems are mass-produced, benefiting from economies of scale.

Physically, embedded systems range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights, factory controllers, or the systems controlling nuclear power plants. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

In general, "embedded system" is not a strictly definable term, as most systems have some element of extensibility or programmability. For example, handheld computers share some elements with embedded systems such as the operating systems and microprocessors which power them, but they allow different applications to be loaded and peripherals to be connected. Moreover, even systems which don't expose programmability as a primary feature generally need to support software updates. On a continuum from "general purpose" to "embedded", large application systems will have subcomponents at most points even if the

system as a whole is "designed to perform one or a few dedicated functions", and is thus appropriate to call "embedded".

5.2 EMBEDDED SOFTWARE:

Embedded software is computer software which plays an integral role in the electronics it is supplied with.

Embedded software's principal role is not Information technology but rather the interaction with the physical world. It's written for machines that are not, first and foremost, computers. Embedded software is 'built in' to the electronics in cars, telephones, audio equipment, robots, appliances, toys, security systems, pacemakers, televisions and digital watches, for example. This software can become very sophisticated in applications like airplanes, missiles, process control systems, and so on.

5.3 ATMEGA32 :

5.3.1 PIN DIAGRAM OF ATMEGA32 :

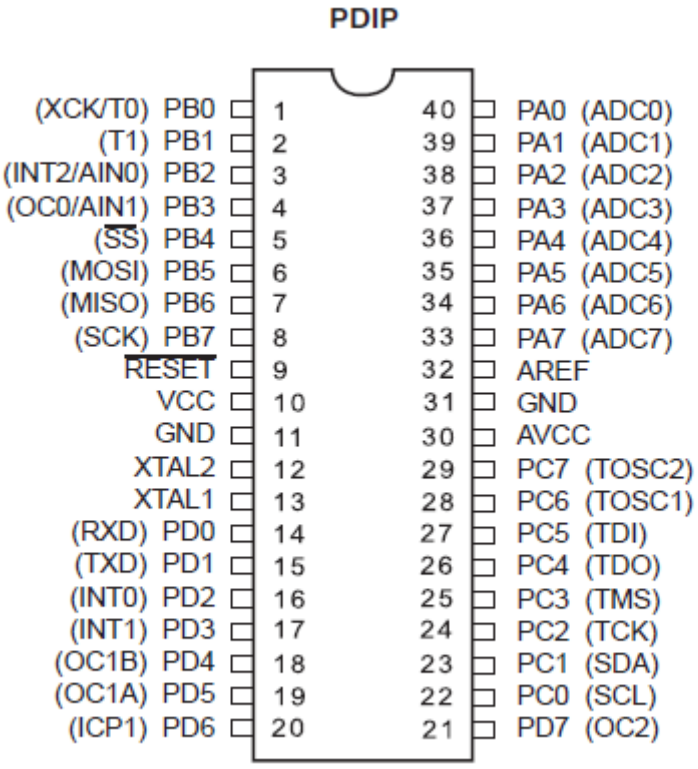


Fig.18. Pin diagram of ATMEGA32.

5.3.2 Pin description OF Atmega32:

- **VCC** Digital supply voltage.
- **GND** Ground.
- **Port A (PA7 to PA0)** Port A serves as the analog inputs to the A/D Converter. Port A also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. Port pins can provide internal pull-up resistors (selected for each bit). The Port A output buffers have symmetrical drive characteristics with both high sink and source capability. When pins PA0 to PA7 are used as inputs and are externally pulled low, they will source current if the internal pull-up resistors are activated. The Port A pins are tri-stated when a reset condition becomes active, even if the clock is not running.
- **Port B (PB7 to PB0)** Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.

  Port B also serves the functions of various special features of the ATmega32
- **Port C (PC7 to PC0)** Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running. If the JTAG interface is enabled, the pull-up resistors on pins PC5(TDI), PC3(TMS) and PC2(TCK) will be activated even if a reset occurs. The TD0 pin is tri-stated unless TAP states that shift out data are entered. Port C also serves the functions of the JTAG interface and other special features of the ATmega32.
- **Port D (PD7 to PD0)** Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running.

  Port D also serves the functions of various special features of the ATmega32.

- **RESET** Reset Input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running. Shorter pulses are not guaranteed to generate a reset.
- **XTAL1** Input to the inverting Oscillator amplifier and input to the internal clock operating circuit.
- **XTAL2** Output from the inverting Oscillator amplifier.
- **AVCC** AVCC is the supply voltage pin for Port A and the A/D Converter. It should be externally connected to VCC, even if the ADC is not used. If the ADC is used, it should be connected to VCC through a low-pass filter.
- **AREF** AREF is the analog reference pin for the A/D Converter.

Chapter 6: STEPS OF OPERATION

In order to successfully complete the project, our group developed a management strategy, in which the steps for the execution were laid out. This was done in order to successfully achieve the target of building "Autonomous parallel parking car", within constraints.
A detailed look into the strategy followed is explained below.

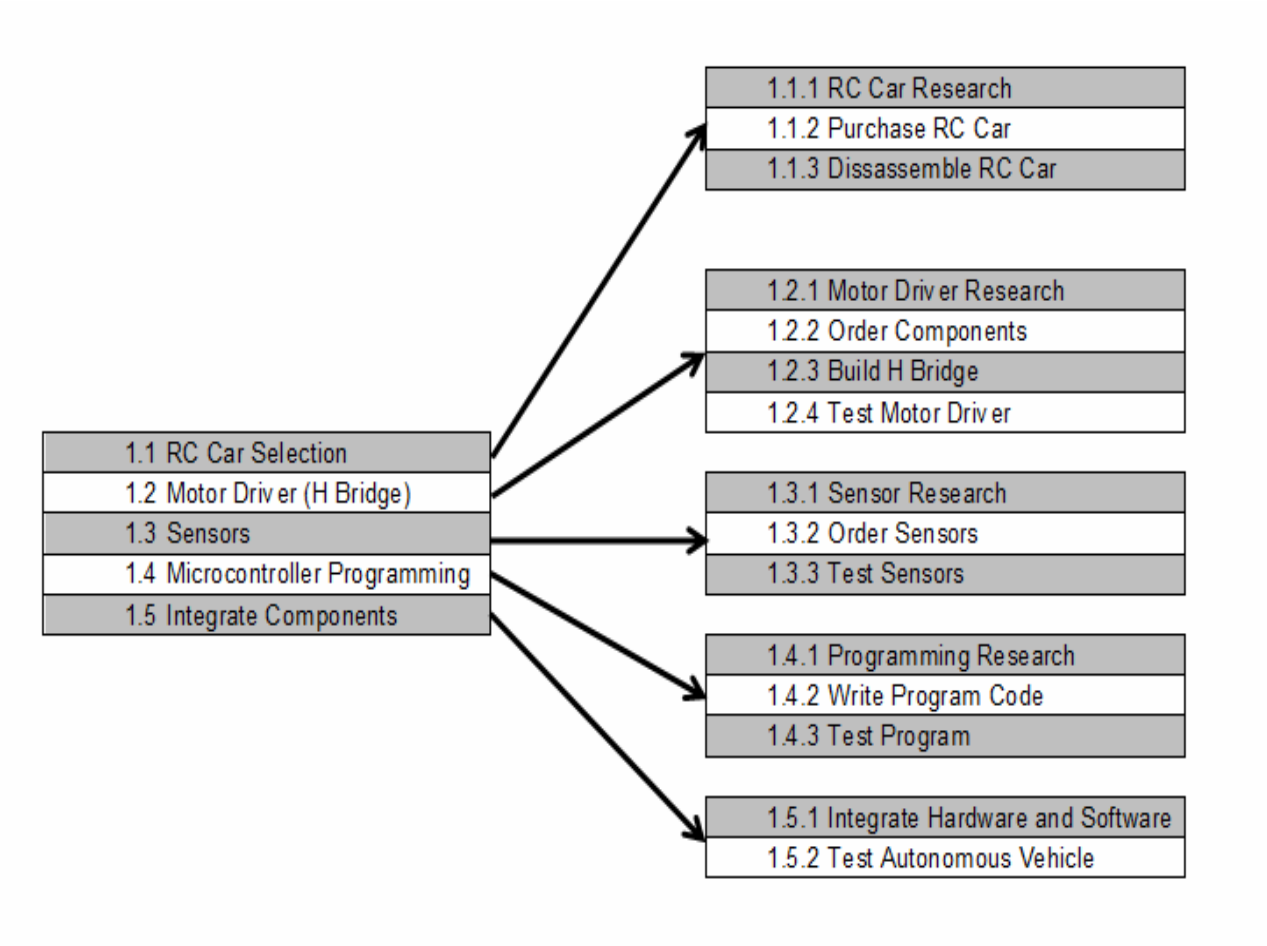


Table.6. Strategy table.

6.1 Task Breakdown Phases:

The individual phases of the Task Breakdown Structure are described below.

Phase 1.1- RC Car Selection

Objective: To chose a suitable RC car for the design, and its applications.

List of Tasks that were required:

• RC Car Research

• Order the Car

• Dissembling

Approach Followed: There are two types of RC cars that could be considered for our design. First type is a TOY RC CAR that offers affordable solution, but has major drawbacks in the hardware, steering and spacing. The second type is Hobbyist RC Car, this car delivers a full house of features, with a design very similar to original car, featuring great servo and torque.
Decision: A toy RC Car was decided to be suitable for design RC Car and further proceeded.

Phase 1.2- Motor Driver (H-Bridge):

Objective: Design a DC motor driver (H-Bridge).

List of Tasks That Were Required:

- Theoretical Research
- Order Components
- Build the Circuits
- Test the Circuits

Approach: The major concern of the design is current handling ability of the circuit. Depending on the motor a suitable H bridge must be implemented. High current motors found in Hobbyist RC car require a design using power MOSFET for high current handling. Less powerful dc motor found in Toy RC car could be design using ICs or regular MOSFET drivers.

Result: Appropriate Motor Driver supporting the specification of the motor was found and used in the design.

Phase 1.3- Sensor Selection:

Objective: Select application appropriate sensor for the autonomous vehicle

List of Tasks That Were Required:

- Theoretical Research
- Order the sensors
- Test Sensors

Approach: Sensors are chosen upon the behavioral applications of the design. A detailed study of IR Sensors and their applications was made.

Results: Proper selection of sensors was made and their applications were understood.

Phase 1.4- Microcontroller Programming

Objective: To program a microcontroller to operate the autonomous vehicle

List of Tasks that were Required:

- Theoretical Research
- Programming the Microcontroller
- Test the Program

Approach: Programming skills were needed to be elevated and learn the hardware of the microcontroller. There are several ways to complete the programming using the assembler or a higher level language like C or C++.

Results: A working program that would operate the car was developed, in modules. The modules were then run in parallel with each other.

Phase 1.5- Complete the Design

Objective: To combine hardware design with software design into a final prototype

List of Tasks That Were Required:

- Put together all Hardware and Software
- Test the Autonomous Vehicle

Approach: All hardware components were properly mounted inside the frame of the car. Sensors were correctly spaced out to avoid blind spots.

Results: Had a complete functional autonomous car.

6.2 Project Milestones :

The first project milestone that we encountered was the programming of the microcontroller since we could not test any circuits that we built until the microcontroller could interact with the components. The second milestone that we encountered was the designing and building of the H bridge interfacing circuit along with regulator since we could not operate the motors until we powered the DC motors. Finally, the third major milestone that we encountered was the integration of the various hardware components with the microcontroller since a program cannot be finalized until the hardware is built and the hardware cannot be finished completely until it is tested with the microcontroller. This last milestone was surpassed by constant efforts by continuously testing the integration and making the appropriate changes to the hardware and the program.

Chapter 7: Results of Design

7.1 Speed of Execution :

All components of the software were done as state machines.

The Motor Control state machines update at ticks every 50ms. This was ample time for the state machines to compute the necessary controlBits and assert the required inputs to the H-bridge. As a result, we were able to obtain highly accurate and sensitive responses from the motors to the control code.

The Algorithm Control state machines update at ticks every 100ms. This was enough time for the state machines to compute the necessary parameters, and to assert the necessary flags for the Control Module to interpret them and translate it into motor motion.

The response of the car to its surroundings is also very fast. The sensors have a response time of 20ms, which is quick enough for them to be processed in real time.

7.2 Accuracy :

Distance Sensors

The sensors were very accurate within their specified range. Even with integer calculations, we were able to calculate distances with a +/- 1cm accuracy. Because we could not control the movement of the car with this degree of accuracy, the accuracy of our distance sensors are sufficient.

Parking Space Detection

The sequence to detect a parking space works very accurately. In the many trials that we performed, it always detected the parking space and stopped on detection.

Parking Algorithm

The parking algorithm we have written works very well when the car is close to a set distance from the side of the parking lot. It, however, becomes less accurate when the car is placed at larger distances from the parking space.

The parking algorithm we have written works very well when the car is close to a set distance from the side of the parking lot. It, however, becomes less accurate when the car is placed at larger distances from the parking space.

7.3 Safety and Interference :

There were not many safety concerns with our project. In order to minimize disturbance to other project groups, and avoid the car colliding into students, we made a separate test area in the hallway. We used this test area for all testing purposes.

Also, since the car is completely autonomous, there was no human contact required (except for turning on the car). Therefore, there wasn't an issue of interference with the systems in the car.

7.4 Usability :

In our opinion, this project has tremendous potential. With some more work on the parking algorithm, we feel that a system for the RC car to park irrespective of its orientation and distance from the parking lot can be developed. With enough research, this can be developed for real cars too.

It can also be used as a learning tool for people who want to learn driving. By observing the motion of this car, students can learn how to parallel park better.

Lastly, this project could serve as a useful reference point for future projects dealing with R/C cars. The Control Module we have implemented to control the R/C car can be used universally for any 2-wheel drive R/C car.

Chapter 8: Intended Users and Uses

In this section the intended uses of our project, along with the intended users of our design is mentioned. This will provide information regarding who and how our project should be used, as well as providing a foundation for possible future uses of our design,

8.1 Intended Users :

The intended users of our autonomous vehicle design project are people interested in adding autonomous functionality to electronics, including, but not limited to, toy car manufacturers and automobile manufacturers. Our design could also be used by instructors and students to enhance their knowledge of electronics, in particular the interaction of microcontrollers with peripheral devices such as sensors and motor driver circuits.

8.2 Intended Uses :

The principal intended use of our design is a method for RC toy vehicles to Parallel Park between two objects and avoiding collisions into walls and objects. Furthermore, our design has the potential for other various different uses, not only on vehicles for autonomous behaviour, but also as a collision detection system for moving objects. This collision detection system could be used for machines to detect safe boundaries for operation, or for vehicles operating within confined spaces where signalling the distance available for manoeuvring or when an object is approaching would prevent a collision.

Chapter 9: Projection of the Project into the Future

The "Autonomous Vehicle" project that we implemented will incur a fairly simple production process since it will only entail modifying certain aspects of existing products. All that is needed is the modification of the electronics board to account for the microcontroller and the sensor connections and the body of the car to mount the sensors. The vehicle design could also be utilized in the field of automobile manufacturing since autonomous features are constantly being added to consumer automobiles. This can be seen in vehicles that have the ability to alert the driver of an imminent collision, as well as the feature to Parallel Park itself. With further research and more advanced designs, our basic autonomous vehicle design could also be used in other types of vehicles with safety features that prevent certain types of collisions.

In order to further our knowledge on the autonomous behaviour of electronic vehicle and objects, we will need to keep researching various sensor types and different types of microcontrollers to find an effective combination or sensor proximity effectiveness with microcontroller features and speed.

Moreover with further research & development we can use separate sensors on the left and right hand sides to make the car even more independent.

Chapter 10: CONCLUSION

Overall, we feel the project met most of our expectations, as we were able to build an autonomous car which could detect a parking space, and park in it. When we started out, we intended the car to be able to locate a parking spot, and park irrespective of its distance from the parking space and its orientation. We were, however, unable to make it robust enough to accommodate parking from different orientations and distances. However, we feel the basic algorithm would remain the same, and this algorithm can be built upon to accommodate these features.

This was also a tremendous learning experience for us, especially with the hardware. We learn a tremendous amount about motor control systems, efficient circuit design, and hardware debugging. We also learned a lot about software. Through this project, we got valuable experience in developing efficient software using memory and run-time optimizations, something that cannot be gained through routine assignments.

ANNEXURE

Annexure 1: Project Codes

1.Algorithm Module-

```
#include <inttypes.h>

#include <avr/io.h>

#include <avr/interrupt.h>

#include <stdio.h>

#include <util/delay.h>

#include <math.h>


#include "funcdef.h"

#include "structs.h"


//external character variables

extern char isForward, isReverse, isLeft, isRight, isParkingLot, isLot = 0, isEntered = 0;;

//external unsigned int variables

extern unsigned int delayCounter, startupCounter, parkingLotWidth, fDistance, sDistance,
rDistance;

//extern stateControl move, park;


char counter = 0, isParked = 0;

int additionFactor = 0, startWidth = 0, endWidth = 0, rstartWidth = 0, rendWidth = 0,
angularLotWidth1 = 0, rsDist = 0, rrsDist = 0;
```

```
float angularLotWidth = 0;
//************************************************************
//testCar
//************************************************************
void testCar(void)
{
	setMovement (1,0,1,0);
}
//************************************************************
//moveCar
//************************************************************
void moveCar (void)
{
	//fprintf(stdout,"sDistance: %d \n\r", sDistance);
	fprintf(stdout,"rDistance: %d \n\r", rDistance);
	//fprintf(stdout,"move.state: %d \n\r", move.state);
	switch (move.state)
	{
	case 0: // rest
			isEntered = 0;
			move.nextState = 1;
			move.controlBits = 0;
			break;
```

```
case 1: // go forward till you detect a parking spot
                if ((isParkingLot)&&(!isParked))
                        {
                        move.nextState = 2;             //if parking spot found, goto
state 2
                        setMovement(0,1,0,0);
                        move.controlBits = 0;
                    //increase speed of car slightly since reverse is weaker
                        findParkingParameters();  // find the angularLotWidth
                        OCR1A = 36000;
                        }
                else
                        {
                        move.nextState = 1;             //else keep going forward
                        move.controlBits = 0;
                        detectParking ();       //keep running detectParking till a
parking space is found
                        isParkingLot = isLot;     //isLot is set in detectParking()
                        setMovement(1,0,0,0);
                        OCR1A = 42000;
                        }
                break;
case 2: //park the car
```

```
                    if (!isParked)
                        {
                        move.nextState = 2;
                        move.controlBits = 0;
                        parkCar();
                        }
                    else
                        {
                        move.nextState = 3;
                        move.controlBits = 0;
                        }
                    break;
        case 3: //back to rest
                    move.nextState = 3;
                    move.controlBits = 0;
                    break;
        }
        move.state = move.nextState;
        rrsDist = rsDist;
        rsDist = sDistance;
}
//*************************************************************
//parkCar
```

```
//************************************************************

void parkCar (void)

{

        // not aligned while doing the reverse right, move forward again and restart

        if ((park.state <= 3) && ((sDistance - rrsDist) > 5) && (rrsDist < sDistance))

                {

                park.state = 0;

                delayCounter = 0;

                }


   // PARKING STATE MACHINE

        switch (park.state)

        {

        case 0: //move the car forward to give it enough space to reverse

                        isParked = 0;


                        //Calculating the emount to go forward by

                        if (isEntered == 0)

                                        additionFactor = 4;

                        else

                                        additionFactor = 10;


                        //Go forward to provide enough turning radius
```

```
                if (delayCounter < additionFactor)
                        {
                        setMovement(1,0,0,0);
                        delayCounter = delayCounter + 1;
                        park.nextState = 0;
                        }
                else
                        {
                        setMovement(0,0,0,0);
                        delayCounter = 0;
                        park.nextState = 1;
                        isEntered = 1;
                        }
                break;

case 1: //turn wheel right and HALT!!!
                setMovement (0, 0, 0, 1);
                park.nextState = 2;
                delayCounter = 0;
                break;

                // continue till you reach back the edge of the Lot
case 2: if (delayCounter < ((endWidth-10)*2))
```

```
                {
                park.nextState = 2;
                setMovement (0, 1, 0, 1);
                delayCounter = delayCounter + 1;
                }
        else
                {
                park.nextState = 3;
                setMovement (0, 0, 0, 1);
                delayCounter  = 0;
                }
        break;

case 3: //reverse right inside the Lot
        if ((sDistance > angularLotWidth) || (rDistance < 15))
        {
                setMovement (0, 0, 0, 0);       //pause
                park.nextState = 4;                     //start the reverse left
maneuver
                delayCounter = 0;                       //reset delayCounter
        }
        else  //reverse right till break point
        {
```

```
                setMovement (0, 1, 0, 1);                    //reverse right
                delayCounter = delayCounter + 1;     //update counter
                park.nextState = 3;                                  //continue
reverse right
            }
            break;

        case 4: //reverse left for 4 cycles or until rDistance lesser than 6cm
            if ((rDistance > 6))
            {
                setMovement (0, 1, 1, 0);                    //reverse left
                delayCounter = delayCounter + 1;     //update counter
                park.nextState = 4;                                  //continue
reverse left
            }
            else  //stop and move to next state
            {
                setMovement (1, 0, 0, 0);                    //pause
                park.nextState = 5;                                  //start
forward maneuver
                delayCounter = 0;                                    //reset
delay counter
            }
            break;
```

```
case 5: //forward right for 6 cycles or until fDistance lesser than 10cm
                if ((fDistance > 10))
                {
                        setMovement (1, 0, 0, 1);
                        delayCounter = delayCounter + 1;
                        park.nextState = 5;
                }
                else
                {
                        setMovement (0, 0, 0, 0);
                        delayCounter = 0;
                        park.nextState = 6;
                }
                break;
case 6: //Parking is complete!!!
                        setMovement (0, 0, 0, 0);
                        park.nextState = 6;
                        isParked = 1;
                        delayCounter = 0;
                        break;
}
park.state = park.nextState;
```

```
}
//************************************************************
//detectParking
//************************************************************
void detectParking (void)
{
        //fprintf(stdout,"detect.state: %d \n\r", detect.state) ;
        //fprintf(stdout,"Rear Distance: %d \n\r", sDistance) ;
        //fprintf(stdout,"rsDist: %d \n\r", rsDist) ;
        //fprintf(stdout,"startWidth: %d \n\n\r", startWidth);

        switch(detect.state)
        {
        case 0: //startup mode
                        if (startupCounter < 4)
                                {
                                startupCounter = startupCounter + 1;
                                detect.nextState = 0;
                                }
                        else
                                {
                                startupCounter = 0;
                                detect.nextState = 1;
```

```
                }
        break;
case 1: //normal mode
        if (abs(sDistance - rsDist) < 7)
                {
                detect.nextState = 1;
                detect.controlBits = 0;
                }
        else
                {
                detect.nextState = 2;
                detect.controlBits = 0;
                }
        break;
case 2: //possible start of parking lot
        if (abs(sDistance - rrsDist) < 7)
                {
                detect.nextState = 1;
                detect.controlBits = 0;
                }
        else
                {
                //start of depression
```

```
                detect.nextState = 3;
                detect.controlBits = detect.controlBits + 1;
                rstartWidth = rrsDist;
                startWidth = sDistance;
                }
        break;
case 3: //start of parking lot, start storing distance, set corresponding flag to 1
        if (abs(startWidth - sDistance) > 7)
                {
                detect.nextState = 4;
                }
        else
                {
                detect.nextState = 3;
                rendWidth = sDistance;
                detect.controlBits = detect.controlBits + 1;
                }
        break;
case 4: //possible end of parking lot
        if (abs(startWidth - sDistance) > 7)
                {
                detect.nextState = 5;
                endWidth = sDistance;
```

```
                        //fprintf(stdout,"endWidth: %d \n\r", endWidth) ;
                        }
                else
                        {
                        detect.nextState = 3;
                        detect.controlBits = detect.controlBits + 2;
                        }
                break;
        case 5: //end of parking lot measurment, determine if really a lot
                if (detect.controlBits > 4)
                        {
                        isLot = 1;
                        detect.nextState = 1;
                        }
                else
                        {
                        isLot = 0;
                        detect.nextState = 1;
                        }
                break;
        }
        detect.state = detect.nextState;
}
```

```
//************************************************************
//findParkingParameters
//************************************************************
void findParkingParameters(void)
{
	parkingLotWidth = rendWidth - endWidth;
	if ((endWidth > 20) && (endWidth < 30))
		angularLotWidth = 22;
	else
		angularLotWidth = 21 + (abs(endWidth - 10) * 0.3);
	//angularLotWidth = 16 + ((startWidth - 10)*0.3);
	//angularLotWidth = 0.8* startWidth;
	//angularLotWidth = (-1*(0.03*pow((float)rstartWidth,2))) + ((float)rstartWidth) +
7.75;
	//angularLotWidth = angularLotWidth * (float)rstartWidth;
	//if (angularLotWidth < 16)
	//	angularLotWidth = 18;
	//fprintf(stdout,"endWidth: %d \n\r", endWidth) ;
	//angularLotWidth1 = (pow(pow(startWidth, 2) + 2209, 0.5)/2);
	//fprintf(stdout,"angularLotWidth1: %d \n\r", angularLotWidth1);
	//angularLotWidth1 = angularLotWidth1 - 4;
	//fprintf(stdout,"angularLotWidth1: %d \n\r", angularLotWidth1);
}
```

2. Control Module-

```
#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include <util/delay.h>

#include "funcdef.h"
#include "structs.h"

//set up the debugging utility ASSERT
#define __ASSERT_USE_STDERR
#include <assert.h>
//UART
#include "uart.h"

// UART file descriptor
// putchar and getchar are in uart.c
FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);

//I like these definitions
#define begin {
#define end   }

#define t0 50
#define t1 50
#define t2 100
#define t3 100

//the subroutines
void initialize(void);   //all the usual mcu stuff
```

```
void turnLeft(void);
void turnRight(void);
void forward(void);
void reverse(void);
void stop(void);
void forwardLeft(void);
void forwardRight(void);
void reverseLeft(void);
void reverseRight(void);

//state machines
void lrStateMachine(void);
void fbStateMachine(void);
void masterStateMachine(void);

volatile int time0,time1,time2, time3;                    // system time, ms timer
volatile unsigned int distance, Din;                      // temporary variable that stores value from ADCH
volatile char enableMask, bridgeInput;

char isForward, isReverse, isLeft, isRight, isParkingLot, dum;          // flags
unsigned int delayCounter, timerCounter, simulationCounter, distCounter, startupCounter;// counters
int sDistance, fDistance, rDistance, parkingLotWidth;          // distance variables

int distanceSimulation[20] = {5, 5, 5, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 5, 5, 5, 5, 5, 5};

/*controlBits:
        00: stop
        01: Forward/Left
        10: Backward/Right
```

```
*/


//************************************************************
//ISRs
//************************************************************

//************************************************************
//Turns on motor on overflow
//This is the up time of the PWM
//************************************************************
ISR (TIMER1_COMPA_vect)
{
enableMask = 0x00;
}


//************************************************************
//Turns off motor on match with OCR0A
//This is the down time of the PWM
//************************************************************
ISR (TIMER1_OVF_vect)
{
enableMask = 0x02;
}


//************************************************************
//timer 0 overflow ISR
//************************************************************

ISR (TIMER0_COMPA_vect)
begin
  //Decrement the time if not already zero
```

```
  if (time0>0)  --time0;
  if (time1>0)  --time1;
  if (time2>0)  --time2;
  if (time3>0)  --time3;
end

//************************************************************
//end of ISRs
//************************************************************



//************************************************************
//Entry point and task scheduler loop
int main(void)
{
  initialize();
  // main task scheduler loop
  while(1)
  {
        PORTB = (bridgeInput) & (~enableMask);
        PORTD = ((~isParkingLot)<<2);
        if (isParkingLot)
              PORTD = 0xFB;
        else
              PORTD = 0xFF;
        //PORTD = dum;

        getDistance();

        //left-right
        if (time0 == 0)
              {
              time0=t0;
```

```
            //getDistance();
            lrStateMachine();
            }
        //forward reverse
        if (time1 == 0)
            {
            time1=t1;
            fbStateMachine();
            }
        //move car and park if space is found
        if (time2 == 0)
            {
            time2=t2;
            moveCar();
            //testCar();
            }

        /*if (time3 == 0)
            {
            time3 = t3;

            }*/
        masterStateMachine();

        //for testing movement function of car using buttons and Port A
        /*if(~PINA & 0x08) turnLeft();
        else if(~PINA & 0x04) turnRight();
        else if(~PINA & 0x02) forward();
        else if(~PINA & 0x01) reverse();
        else stop();*/

  } //end of while
}   //end of main
```

```
//*************************************************************
//getDistance
//gets distance from the 3 sensors mounted in the order - Front, side, rear
//*************************************************************
void getDistance()
{
//      fprintf(stdout,"distCounter: %d \n\r", distCounter);
//start conversion
        ADCSRA |= (1<<ADSC);

        //wait till the ADC is done
        while (ADCSRA & (1 << ADSC)) {};

        //read the value of the Voltage
        Din = ADCH;

        /* ADC Calculation:
        (1/distance)*slope + intercept = (Din * 2.56)/(256);
        1/distance = (Din/100 - intercept)/slope;
        1/distance = (Din - intercept*100)/(slope*100);
        distance = (slope*100)/(Din - intercept*100);

        Vin = Din/100
        Din = 100*Vin
        */

        //display distance on hyperterm
        if (distCounter == 0)
                {
                //forward 4-30cm sensor, sensor 2
                if(Din < 150)
                        {
                        if (Din < 120)
```

```
                {
                distance = 60;
                }
        else
                {
                distance= 1274/(Din - 6);
                }
        }
    else
        {
        distance= 976/(Din - 38);
        }
    ADMUX = 0xE1;
    //fprintf(stdout,"Front Distance = %d \n\r", distance) ;
    distCounter = distCounter + 1;
    fDistance = distance;
    }
else if (distCounter == 1)
    {
    //side 10-80cm sensor, sensor 2
    distance= 2414/(Din - 15);
    ADMUX = 0xE2;
    //fprintf(stdout,"Side Distance = %d \n\r", distance) ;
    distCounter = distCounter + 1;
    sDistance = distance;
    }
else
    {
    //rear 4-30cm sensor, sensor 1
    if(Din < 125)
        {
        if (Din < 20)
            {
            distance = 60;
```

```
                        }
                else
                        {
                        distance= 1181/(Din - 9);
                        }
                }
        else
                {
                distance= 896/(Din - 40);
                }
        ADMUX = 0xE0;
        //fprintf(stdout,"Rear Distance = %d \n\r", distance) ;
        distCounter = 0;
        rDistance = distance;
        }
}

//************************************************************
//masterStateMachine
//State machine which controls actual motion of the car
//************************************************************
void masterStateMachine(void)
{
   unsigned int fbBits, lrBits, masterBits;
   fbBits = fb.controlBits;            // (FB FB)
   lrBits = lr.controlBits;            // (LR LR)
   masterBits = (fbBits<<2) + (lrBits);  // (FB FB)(LR LR)

   /*
        01: Forward/Left
        10: Backward/Right
        */

   switch (masterBits)
```

```
    {
    case 0: stop(); break;
    case 1: turnLeft(); break;
    case 2: turnRight(); break;
    case 4: forward(); break;
    case 5: forwardLeft(); break;
    case 6: forwardRight(); break;
    case 8: reverse(); break;
    case 9: reverseLeft(); break;
    case 10: reverseRight(); break;
    case 3:
    case 7:
    case 11:
    case 12:
    case 13:
    case 14:
    case 15:
    default: break;
    }
}

//*************************************************************
//fbStateMachine
//State machine which determines the forward backward motion of the car
//*************************************************************
void fbStateMachine(void)
{
    switch (fb.state)
    {
    case 0: //Rest State
            if (isForward)
              fb.nextState = 1;
            else if (isReverse)
              fb.nextState = 3;
```

```
        else
          fb.nextState = 0;
                        fb.controlBits = 0;
        break;

case 1: //Go Forward
        if (isForward)
          {
          fb.nextState = 1;
          fb.controlBits = 1;
          }
        else
          {
          fb.nextState = 2;
                        timerCounter = 0;
          }
        break;

case 2: //0.5second Reverse to stop the car
                        //0.5second delay
                        if (timerCounter < 1)
                                {
                                timerCounter = timerCounter + 1;
                                fb.nextState = 2;
                                fb.controlBits =2;
                                }
                        else
                                {
                                fb.nextState = 0;
                                fb.controlBits = 0;
                                }
        break;

case 3: //Reverse
```

```
        if (isReverse)
          {
          fb.nextState = 3;
          fb.controlBits = 2;
          }
        else
          {
          fb.nextState = 4;
                        timerCounter = 0;
          }
        break;

   case 4: //0.1second Forward to stop the car
                        //0.1second delay
                        if (timerCounter < 1)
                                {
                                timerCounter = timerCounter + 1;
                                fb.nextState = 4;
                                fb.controlBits =1;
                                }
                        else
                                {
                                fb.nextState = 0;
                                fb.controlBits = 0;
                                }
        break;
   }
        fb.state = fb.nextState;
}

//*************************************************************
//lrStateMachine
//State machine which determines the left right motion of the car
//*************************************************************
```

```
void lrStateMachine(void)
{
   switch (lr.state)
   {
   case 0: //Rest State
         if (isLeft)
           lr.nextState = 1;
         else if (isRight)
           lr.nextState = 3;
         else
           lr.nextState = 0;
                        lr.controlBits = 0;
         break;

   case 1: //Move Left
         if (isLeft)
           {
           lr.nextState = 1;
           lr.controlBits = 1;
           }
         else
           {
           lr.nextState = 2;
           }
         break;

   case 2: //1second Reverse to stop the car
                        //1second delay
         lr.nextState = 0;
                        lr.controlBits = 0;
         break;

   case 3: //Move Right
         if (isRight)
```

```
            {
            lr.nextState = 3;
            lr.controlBits = 2;
            }
        else
            {
            lr.nextState = 4;
            }
        break;

   case 4: //1second Forward to stop the car
                        //1second delay
        lr.nextState = 0;
                        lr.controlBits = 0;
        break;
   }
        lr.state = lr.nextState;
}

//*************************************************************
// setMovement
// sets the appropriate flags to move the car in desired direction
//*************************************************************
void setMovement(int a, int b, int c, int d)
{
        isForward = a;
        isReverse = b;
        isLeft = c;
        isRight = d;
}

/* H-Bridge Pin Out and Port B Configuration:
in1,in3 turns on hl (PB7,PB3)
in2,in4 turns on hr (PB6,PB2)
```

```
EnA = PB5
EnB = PB1
Out1 - Out2 : Left/Right
Out3 - Out4 : Forward/Backward
(lower # in a (1,2) or (3,4) combination goes to the live wire
*/

//*************************************************************
void turnLeft(void)
begin
        bridgeInput = 0xA0;   //turn left:       hl,lr: 10,00
        PORTC= 0xF7;
end

//*************************************************************
void turnRight(void)
begin
        bridgeInput = 0x60;    //turn right:     hr,ll: 01,00
        PORTC= 0xFB;
end

//*************************************************************
void forward(void)
begin
        bridgeInput = 0x0A;   //forward:                hl,lr: 00,10
        PORTC= 0xFD;
end

//*************************************************************
void reverse(void)
begin
        bridgeInput = 0x06;    //reverse:               hr,ll: 00,01
        PORTC= 0xFE;
end
```

```
//*************************************************************
void stop(void)
begin
        bridgeInput =0x00;          //stop:                 00,00
        PORTC= 0xFF;
end

//*************************************************************
void forwardLeft(void)
begin
        bridgeInput = (0x0A)|(0xA0);
   //forward:          hl,lr: 00,10
   //turn left:    hl,lr: 10,00
        PORTC= 0xF5;
end

//*************************************************************
void forwardRight(void)
begin
        bridgeInput = (0x0A)|(0x60);
   //forward:          hl,lr: 00,10
        //turn right:     hr,ll: 01,00
        PORTC= 0xF9;
end

//*************************************************************
void reverseLeft(void)
begin
        bridgeInput = (0x06)|(0xA0);
        //reverse:              hr,ll: 00,01
        //turn left:      hl,lr: 10,00
        PORTC= 0xF6;
end
```

```
//*************************************************************
void reverseRight(void)
begin
        bridgeInput = (0x06)|(0x60);
        //reverse:                hr,ll: 00,01
        //turn right:     hr,ll: 01,00
        PORTC= 0xFA;
end


//*************************************************************
void initialize(void)
begin

 /*************** TIMER INITIALIZAIONS *********************/
 //TIMER0
 //set up timer 0 for 1 mSec timebase
 TIMSK= (1<<OCIE0);        //turn on timer 0 cmp match ISR
 OCR0 = 249;               //set the compare re to 250 time ticks
 //set prescalar to divide by 64
 TCCR0= 3; //0b00001011;
 // turn on clear-on-match
 TCCR0= (1<<WGM01) ;

 //TIMER1
 //sets motor speed to zero
 OCR1A = 38000;
 //turns on interrupt vectors for timer0
 TIMSK = (1<<OCIE1A)|(1<<TOIE1) ;       //turn on ISR
 // timer 0 prescalar to 64
 TCCR1B = 1;

 /*************** ADC INITIALIZAIONS ***********************/
```

```
//Set up ADC
ADMUX = 0xE2; //Internal 2.56 reference voltage, with capactiance at AREF
ADCSRA = 0x87; //enable on, start conversion off, clock = SYSCLK/128

/*************** PORT INITIALIZAIONS *********************/
//Set up port A
//for pushbuttons to test moveement control of car
DDRA=0x00;
PINA=0;

//Set up port B
//output to H-Bridge
DDRB=0xFF;
PORTB=0x00;

//Set up port C
//LEDs for testing
DDRC=0xFF;
PORTC=0xFF;

//Set up port D
//detected parking lot
DDRD = 0xFF;
PORTD = 0x00;

/*************** SOFTWARE INITIALIZAIONS *****************/
//Structure instances
park.state = 0;
park.nextState  = 0;
park.controlBits = 0;

move.state = 0;
move.nextState  = 0;
move.controlBits = 0;
```

```
fb.state = 0;
fb.nextState  = 0;
fb.controlBits = 0;

lr.state = 0;
lr.nextState  = 0;
lr.controlBits = 0;

detect.state = 0;
detect.nextState  = 0;
detect.controlBits = 0;

//counters
simulationCounter = 0;
distCounter = 0;
delayCounter = 0;
timerCounter = 0;
startupCounter  = 0;

//distance
sDistance = 0;
rDistance = 0;
fDistance = 0;
parkingLotWidth = 0;

//booleans
isForward = 0;
isReverse = 0;
isLeft = 0;
isRight = 0;
isParkingLot = 0;

bridgeInput = 0x00;
```

```
  //UART initializations
  uart_init();
  stdout = stdin = stderr = &uart_str;
  fprintf(stdout,"Starting...\n\r");

  //crank up the ISRs
  sei();

end
            //=================================================
```

REFERENCES

Books:

- **8051 Microcontroller And Embedded Systems** by Muhammad Ali Mazidi
- **Printed Circuit Board Design** by Douglas Brooks

Websites:

- www.atmel.com
- www.extremeelectronics.co.in
- www.avrfreaks.com
- www.wikipedia.com
- www.8051projects.net
- www.avrbeginners.net
- www.sourceforge.net
- www.sparkfun.com

Data Sheets

- Atmega32- Internet Link: *www.atmel.com/atmel/acrobat/doc2503.pdf*
- L298 H-Bridge IC- Internet Link: *http://www.st.com/stonline/products/literature/ds/1773/l298.pdf*
- Voltage Regulator- Internet Link: *http://www.digchip.com/datasheets/download_datasheet.php?id=513599&part-number=LM340T5*
- TSOP IR Sensor- Internet Link: *http://graigroup.files.wordpress.com/2008/04/tsop-1738-based-proximity-sensor.pdf*