1.0 EXECUTIVE SUMMARY

Project G.U.N.D.A.M. is a robot with the main purpose of solving a maze built to the discretion and specifics of the builders. The maze will be drawn out for the user to visually see a representation of the maze currently being explored and solved. The robot will also have the bonus functions of finding any and all paths inside of a maze once it has been solved and traversing a path created by the user of the system.

The G.U.N.D.A.M. will consist of four sensors. Two ultrasonic sensors will serve as the range finders on the front and rear of the chassis. Two IR sensors will serve as the range finders on the side of the chassis. These sensors will double as both detecting paths within the maze as well as assisting in regulating the straightness of the path taken by the robot.

The robot will also consist of a purchased chassis that will house the four sensors, numerous microcontrollers stacked on top of each other, including a Beagleboard-xM, a Power Regulating PCB, a battery, a Motor-Microcontroller, Sensor microcontrollers, a wireless module, etc., and will have two motors to power it. The Beagleboard will interface all of the other microcontrollers to each other and will serve as the main power-house of all the maze solving and maze drawing algorithms.

The robot will also have a communication subsystem that will consist of two microcontrollers and two RF transceivers. The communication will primarily consist of the G.U.N.D.A.M. relaying information about the path taken through the maze and in the end calculating the shortest path to the exit. The system will also employ an encryption algorithm to secure the data from being stolen or tampered with while in transmission. Finally, the communication subsystem will also have scalability in data rate, modulation, power consumption, and range.

2.1 Function of Robot Control System

Introduction for the Design

The function of this robot is to successfully navigate a maze and draw out a map of where that robot has gone in the maze. The idea of this system is to design a way of navigating and uncovering a complex network, where larger objects cannot go or where it would be too dangerous to send a person; this has application in land mine detection and avoidance, as well as searching the relationship of underground networks. For this project, we decided not to go with GPS, given that GPS cannot always be used.

Parts of the Robot

There are many parts that will form the whole of the robot. First, there is a chassis with two motors, four wheels, and two axels. Since this robot needs a way for us to control the motors as we wish, a microcontroller needs to be designed to allow us to do this.

Second, there needs to be a power supply to the robot. Ideally, we would like to be able to plug the robot in and still use it, while also having it run on a battery when not plugged in. To accomplish this, a switching circuit will be implemented to switch between the two. It will also regulate the supply voltages to the rest of the devices on the robot.

Third, there needs to be sensors that will let the robot know its current location and to draw data from on its movement and direction; a microcontroller will be set up to calculate and store all the data that goes with the sensor circuits.

Fourth, we will have a wireless device that can interface the robot with a laptop computer in order to draw out a diagram of where the robot has gone on the laptop computer.

Fifth, a microcontroller needs to be designed that interfaces all of the little microcontrollers with each other, as well as running algorithms for the robot to use in navigating the maze. The following diagram gives a rough idea of all the parts mentioned here and what other parts they will interface with.

BeagleBoard-xM Microcontroller

a. Runs Maze Solving Algorithms
b. Receives data from the Sensors
c. Sends data through USB Wifi to a Laptop

Chassis

a. Two Motors
b. Four wheels
c. Two axels

USB Wireless

a. Transmits data to a Laptop

Voltage Regulating PCB

a. Power Motors
b. Power Sensors
c. Power Microcontrollers
d. Switches Power between Outlet Power and a Battery

Sensor PCBs

a. Outputs Current Distance
b. Utilizes a microcontroller

Putting It All Together

What this robot will end up doing once it's completed is to navigate a maze by analyzing the data that the sensors give it and trying out techniques for solving a maze, draw out a map of where it has been in the maze, and send that map to an algorithm on a laptop computer so that we can all see what the robot has done.

So the Beagleboard microcontroller takes in inputs from the sensors, uses that information to determine where the robot should move and where it has been by telling the motor-microcontroller what to do; the Beagleboard microcontroller also will transmit where it has been (and the current map that it has created of that) through a wireless device to a laptop computer.

2.2 G.U.N.D.A.M Software

The main focus of the software portion of this project is to have some way of translating the information sent to the computer by our G.U.N.D.A.M. into a drawn out map of the maze it has finished traversing. This involves two things: how do we store this information and how our map will actually be drawn. There are two ways this can be accomplished: we can either draw out the map in real time as our GUNDAM is making its way through the maze or we can draw it out after our GUNDAM has found the correct path through the maze. Depending on how our wireless module works and what we can pull form the GUNDAM at any given time as well as how our software is able to interact with constantly changing information, drawing out the map after the GUNDAM has completed its course seems to be the simpler option of the two.

2.3 Communication System

The communication between the G.U.N.D.A.M. robot and a computer will be accomplished with two separate devices. The communication subsystem on the G.U.N.D.A.M. itself will be integrated with the main microcontroller, which will interface all other devices on the robot. The device the robot will communicate with will be connected to a computer through a universal serial bus (USB) interface. The task of designing an efficient communication subsystem involves choosing a modulation/demodulation scheme that can provide the least amount of interference and the fastest data transfer rate for our needs. The communication subsystem will also implement data encryption for further secure data transfer between the G.U.N.D.A.M. and a computer. The data encryption will be implemented on two microcontrollers on the G.U.N.D.A.M. and the computer.

The device that will be integrated into the G.U.N.D.A.M. to communicate with a computer will consist of a transceiver to send and receiver bits from the device on the computer. It will also have a microcontroller to encrypt the data bits going to the transceiver and decrypt the bits coming from the transceiver, as well as handle the way in which the computer and G.U.N.D.A.M. communicate with each other. Similarly, the device on the computer will have a transceiver and microcontroller to preform the same tasks of data transfer and encryption on the other end of the communication stream. The device will communicate with the computer through a USB interface. The software on the computer will handle the data transferred to the computer and use it to draw out the path the G.U.N.D.A.M. is taking through the maze.

The communication subsystem on the G.U.N.D.A.M. will be capable of both transmitting to the computer and receiving data from the computer. The computer will also be capable of both transmitting data to the G.U.N.D.A.M. and receiving data from the robot. These features may ultimately not be necessary since the G.U.N.D.A.M. will calculate the quickest path out of the maze with its on board microcontroller, therefore the communication subsystem will only need to send data to the computer and the computer will only need to receive data from the G.U.N.D.A.M. to draw out the maze. However, if it is later determined that the on board processing power or memory capacity is not enough for the robot to calculate a path out of the maze in the fastest and most efficient way, then the G.U.N.D.A.M. will need to have the capability of both sending and receiving data from the computer. This configuration would then leave room for the robot to concentrate processing power and memory to get accurate and real-time measurements from the on board sensors while navigating through the maze.

2.4 Sensors

The G.U.N.D.A.M. will require distance sensors to detect how far the robot is from the walls of the maze and these sensors will also aid in navigation and steering through the maze. The sensors should have to be able to detect the short distance the robot will be between the walls and itself accurately. Ultrasonic sensors provide and accurate, cost effective, and reliable way to determine the distances required. Similarly, another choice for short distance measurements can be infrared sensor. Infrared sensors can also be accurate and cost effective if the material that is used to measure distance from is not able to absorb much of the infrared light. Another disadvantage with infrared sensors is that they are not reliable in an outside setting but in general they can prove to be more cost effective than ultrasonic sensors. Therefore, since our application requires an indoor setting, both ultrasonic and infrared sensors can be used as a fair trade off between cost and accuracy. The purpose of using distance sensors is not only limited to navigation of the maze, they will also function to help draw the maze in real-time on a computer screen. For accurate drawing of the maze there must also be accurate measurements of the position the robot is in with respect to the boundaries of the maze walls.

The robot will contain at least four of these sensors to give an accurate 3-dimensional view of the location of the robot with respect to the walls of the maze. As the robot traverses the maze it will use the speed of the motors, which will be kept constant, and the duration of the movement to calculate the distance it has traveled. Since the electric motors will likely not run at the same speed while going forward or in reverse, there must be a correction mechanism that allows the robot to travel in a straight line. In the case the robot will drift to the left as it moves along a path, the measurements on the right side of the robot will be larger than the left because of the gap between the left side of the robot and the wall is closing and the gap between the right side of the robot and the wall is increasing. This variation in measurements can be used to steer the robot to the right until the gap on the left and right side of the robot are equal or within a certain threshold that will allow it to move in a relatively straight line. The G.U.N.D.A.M. will also use the measurements it gets from left, right, front, and back sensors to determine the shape of the path it is currently in. Based on these measurements the main microcontroller will tell the motors which way to turn. For example, if the robot is traversing a straight path the front and back sensors will measure a further distance than left and right sensors. Similarly, if the robot reaches a turn the front sensor and the left or right will measure a smaller distance than the back sensor. For the sensors to function this way the values will be read and stored on a microcontroller, which will store information that converts the voltage output of the sensors to usable distances. These distances will then in turn be used with a moving algorithm that will be in charge of navigating the robot through the maze; however, this will be separate from the maze-solving algorithm required to solve the maze.

3.1 IDE RESEARCH

In terms of the GUI's looked into to help draw out the maze, C++ and Java were the two main languages looked into. C++ was a desirable path to take in terms of programming as that has been the language of choice for past programming experience. The Win32 API would have been used to draw out these windows along with the visual interfaces. This did not turn out to be the path chosen, however, as the available sources in terms of learning how to utilize this method were very poor. Because of this, JAVA was the next natural choice for programming languages.

JAVA will also most likely be the language of choice when it comes to constructing the maze as I have the most experience using this software in terms of its GUI. There are a large assortment of tutorials and videos online to brush up on basic GUI components such as frames and buttons to allow for a refresher on how exactly the interface will be constructed. The map drawing software will allow for the user to not only view the path to leave the maze but also draw out a user specified path they wish the robot to take once the map has been drawn out.

JAVA has a very streamlined set of methods and functions when it comes to creating windows and simply drawing out simple images. While the C++ method had a large number of extra files needed in order to create something as simple as a window, JAVA has a lot of built in classes that are solely for the creation of windows, buttons, and frames. Combining this with the large number of tutorials out there to help build anything from a simple window to an actual game using the provided classes and methods, JAVA seems like a naturally obvious choice.

The interface for JAVA will consist of a Frame and Window to house the main component of our program, the drawn maze. The panel will also consist of buttons to switch from either solving the maze automatically, allowing the user to decide a path to be taken by the robot, or choose a method of solving the maze manually. Throughout this, the maze will be dynamically drawn during certain calls to a text file that the robot will be writing to. The maze will start off by solving the maze by making solely left turns. When the algorithm on the computer side determines that the robot has taken too long to solve the maze or detects a return to a point that has already been traversed, then the computer will send a signal to the robot to switch up how it is solving the maze.

3.2 MAZE SOLVING RESEARCH

There are a number of methods employed to solve a maze to minimize the amount of loops, dead ends, and failures upon traversal of the maze. While a human being can rely on a mix of these search practices to solve a maze, a robot needs a set of rules to follow to allow it to solve mazes of different sizes and types. This aspect of a robot also allows it to solve much more complex mazes that could give an average human a lot of trouble.

The most popular method of maze solving is the wall following method. This method dictates that the solver follow the maze along a set wall, either the left or right hand, and consistently choose this path to take upon any turns and forks in the road. This method, however, only works for mazes in which all of the walls are interconnected either with each other or the walls.

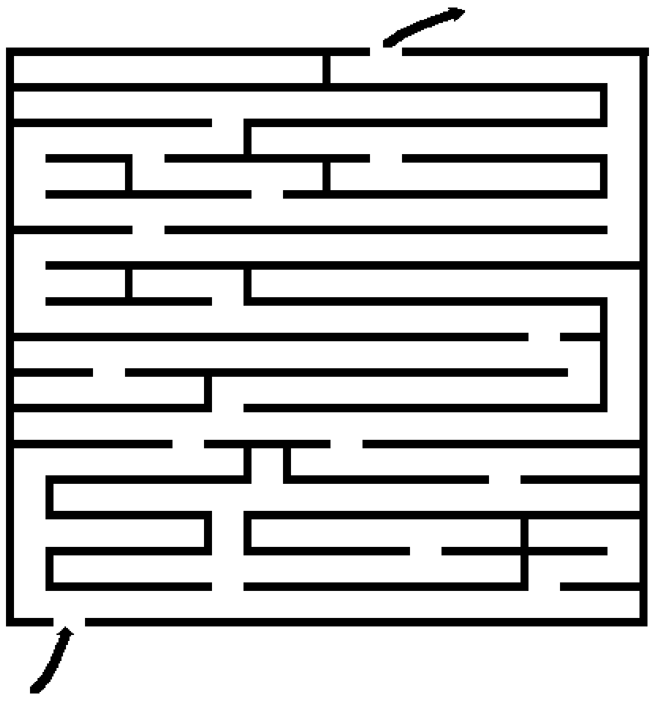


Figure 1: Wall-follower capable maze

If, however, the maze contains its entrance and/or exit somewhere in the center of the maze, if the paths lie on different, crossing levels, or if the maze has walls that stand on their own somewhere along the maze, then the maze follower method has no guarantee of working. These are all possibilities that can be tested upon traversal of the maze by the robot or software. While our particular mazes will contain no alternating levels, there is still the potential to create a maze with solitary walls or entrances that lie within the maze instead of on the borders. Both of these cases can be checked by the software as it draws out its maze.

Tremaux's Algorithm is a potential fall back if the wall-follower method proves to be a failure (stuck in a curved dead end, entrance/exit in center, or back at the entrance). Tremaux's uses a method that would work well with nodes placed at the turns and forks (a process gone into much deeper in later sections of this paper). A line is drawn along a path taken and given a value, 1 for the first time having moved along this path and 2 if the solver is walking along the path for a second time.

As each fork is reached, the solver must make several analyses. If the fork has a path that has not been traversed, then the solver will choose that path to travel along. If the solver is coming across an already marked path and its current path has only been traversed one, the solver will move back along its path in the opposite direction and mark the path with a 2. If the path it is currently on is already marked with a 2 however, then the solver will choose the path with the smaller value to travel along, marking it as it travels. The paths with a value of 1 represent the path to the exit. This method prevents the possibility of loops and also has the added benefit of returning to the start of the maze if no exit exists. This problem will not be an issue on our own constructed mazes, but allows the robot to return to the user in real world applications if no exit exists out of an area of particular interest.

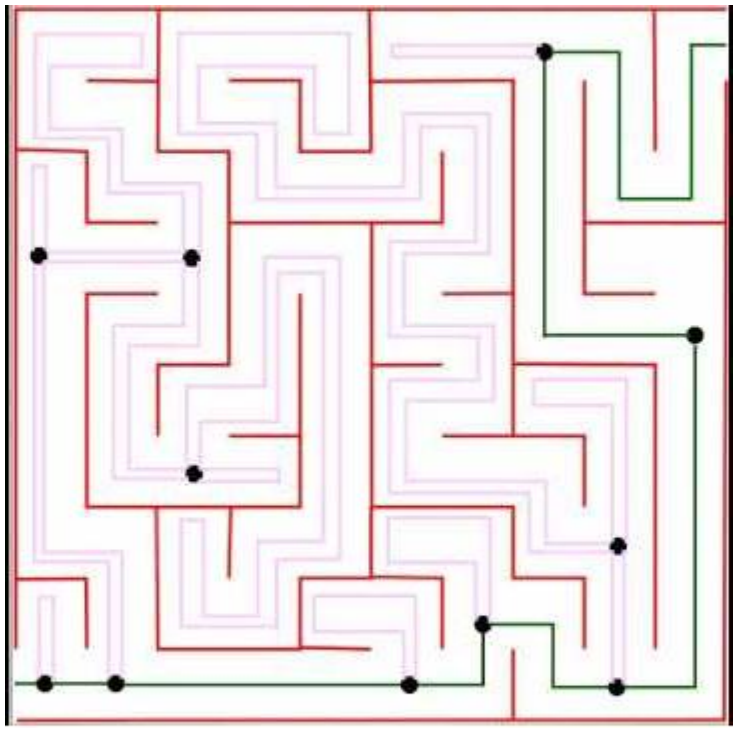


Figure 2: Tremaux Algorithm. The green path represents paths marked with a 1 while the red path represents paths marked with a 2.

These methods hardly represent all of the methods for solving a maze. They do, however, represent the methods to solving a maze without prior knowledge of the layout. This is a much more realistic approach to solving this maze if it were to be expected to find its way outside of more complex areas outside of mazes. For instance, we have no way of knowing the layout of a cave prior to entering it so the algorithms used to find our way out or to an exit rely on set rules and principles that take this into account.

3.3 SEARCH ALGORITHM RESEARCH

There is the question of how we will search through our maze to find a path specified by our user (one of the many features of the GUDNAM). There are quite a few search algorithms available that place priority on different aspects of a search: speed, shortest path, thoroughness. Along with this, there is a question of how we will implement such searches. Many of them rely on points along the path that can be analyzed. To solve this issue, nodes will be used to represent important points in the maze.

But how will we spread out these nodes in our maze? A dense mesh of nodes is one possibility. However, this can put a large strain on the software. To keep track of all these nodes can prove to be too much of a memory hog as well as unnecessary if particular nodes in a mesh are never utilized. This does provide a more natural looking movement throughout the maze, however. If a path is chosen in the maze, we are much more guaranteed to find a node sufficiently close to our destination.

There is the possibility of only creating nodes at important points in our maze. This can result in movement of our GUNDAM that appears highly unusual. For instance, if there is a straight shot from node A to C, but node B is along the way, the GUNDAM may choose to visit node B along its path when this is not an optimal decision. Also, with a smaller collection of nodes, a destination node may be difficult to reach if it is not close to a node we have created upon exploration of the maze. The solution to this may be to only apply nodes at openings and turns. This provides the result of natural movement along straight ways. Also, the creation of nodes upon selection of a destination provides a position on the map for the node to travel to.

Once the nodes have been established, a method of search through them needs to be established. Depth first search seems like a very easy first choice. The system of depth first search is to select any node that has not been visited and to continue along this path until a dead end or the destination has been reached. If a dead end is reached, we backtrack until we reach a node we have not visited. This search is very thorough but only applies well to small mazes and environments. Once the maze or environment reaches a certain size, the application of this search system becomes unreasonable. This will take up too much time and may cause an unwanted delay during the search process.

A solution to this is to apply a limit to the depth of the search. If the destination has not been reached within a certain amount of node analyses, we lower the depth we have decided to search and continue. There is a major drawback to this system. What if our destination just happens to be along the longest path of the maze? If this is the case, the destination may never be found using this technique. Of course, the opposite may be true without the utilization of this technique. If the destination is on the shortest path but we have gotten stuck on an exceptionally large search depth, we are forced to go through an unnecessarily long search tree.

These two problems make depth first search a bad choice for moving through our maze. Speed is an important factor as the smallest delay between communication of our

GUNDAM and our software is desired to prevent any awkward pauses. There are many other search algorithms with their own faults. Breadth first search searches every neighbor node of the current node before moving on a step more. The problem with this algorithm is another time issue. It forces us to check every single node within the vicinity of our start and ending nodes. Again, while this is okay for small mazes and even the maze we may conduct initial testing on, this becomes unreasonable for larger mazes.

One factor that breadth and depth have in common that may attribute to this issue is the fact that they do not consider the cost of moving from one node to another. All nods are created equal under this analysis, whether the node is an inch away or a mile away makes no difference to these techniques. Djikstra’s not only takes these into account but also has the intended result of always finding the shortest path to the destination, something Breadth and Depth had no guarantee of doing. Unfortunately, this method also has the consequence of checking a high number of nodes during its analysis.

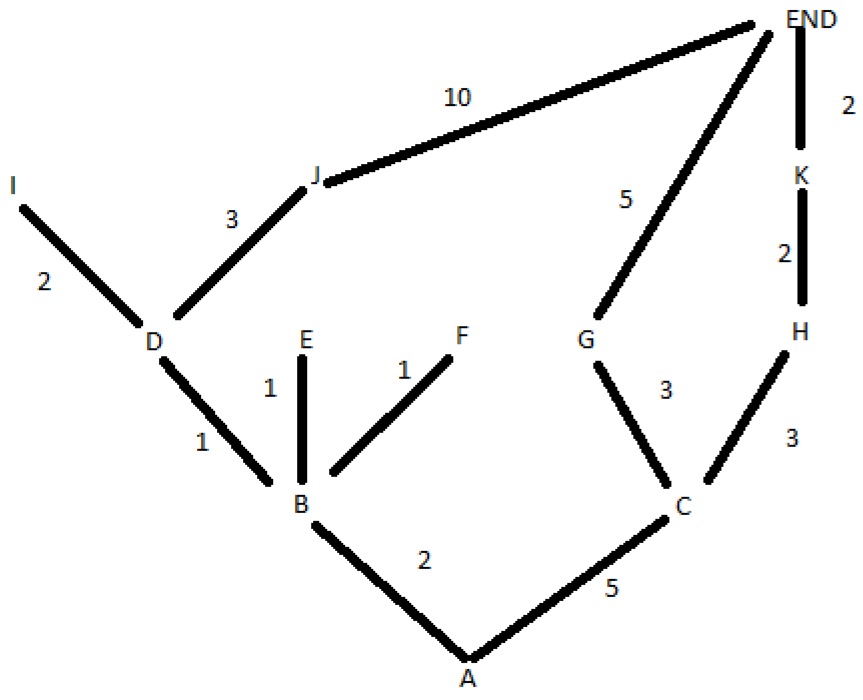


Figure 3: Map With Costs Displayed

As a more solid example, the previous image displays a map with the costs displayed on each of the links. These numbers represent how much it would be to move from one node to the node that link connects it to. These costs can represent anything from a monetary number to actual distance. For the sake of the GUNDAM, it makes the most sense for these numbers to represent the distance to the node.

Djikstra’s will first select the path from A to B, as this represents the smallest cost when compared to A to C. (It is important to note that A represents the starting point of this map). The next selection can be a tossup between C, D, E, and F. By adding up all the possible values, we find that we have an equal cost for D, E, and F. This results in the need to randomly choose a node between the three. We will be met with the possibility of choosing a node that cannot reach the end at all. These types of connections are counted as an infinite distance, causing the cost to be much higher than any node that actually can reach the End point.

It is important to note that all of the nodes that are connected to visited nodes are analyzed upon each step. This brings up the issue mentioned previously, this algorithm does nothing to alleviate the issue of visiting a high number of nodes throughout its process. While the shortest path is always found, it comes at a large time investment when larger mazes are introduced.

The problem with many of these searches is that there is no account taken for the cost of reaching the destination from a node, just the node to its neighbor. By considering the overall distance to the destination, we ensure not only the shortest path but a lack of excessive node searches. We choose the most apparently optimal node to travel to and only move back if we find that our search reaches a dead end. A* (A star) provides all of these benefits. We simply choose a node that has the shortest cost which is the cost of reaching the neighbor node plus the perceived cost of reaching the destination from the neighbor. This second cost is not an exact number, but rather an estimation. It is not aware of any possible boundaries such as walls and other obstacles until it is directly in contact with such an issue. However, the results of this algorithm make it the search method of choice for the GUNDAM’s path creation.

3.4 RESEARCH ON USING EAGLE

Introduction

Since we need to develop a PCB of the smaller microcontrollers, it was decided that Eagle would be worth using. In order to use Eagle adequately to design the components that we needed, some things needed to be learned throughout the process.

How Eagle Works

Eagle has a library of components that can be placed into a circuit schematic and connected together; each component has a diagram of what that part will look like on the actual PCB that includes the size of all its parts and pins. Once a schematic is made, the PCB can be designed automatically by EAGLE; and the parts can be rearranged and tagged to your specifications and liking.

Eagle Libraries

There are a number of useful Eagle libraries that needed to be used in drawing up schematics.

SUPPLY LIBRARY

This library includes all of the power supply labels so that all you have to do is place each label where you need them to be in the circuit. It keeps confusion of what lines connect where and to what to a minimum.

CONNECTOR LIBRARY

These libraries were used to find pin headers, female headers, switches, and power cord sockets.

OTHER LIBRARIES

There are other parts that can be found for just about anything that someone might need. Some of these that were used in our PCBs included capacitors, mosfets, transformers, diodes, LEDs, op-amps, resistors, and microprocessors. Eagle lets you set the value of each of these components to our specifications.

Once the PCB is Made

Once everything is designed and laid out into a Gerber file, EAGLE allows us to interface our design with a third-party PCB maker that can either just make the PCB board for our schematic to solder all the components on or they will solder everything for an extra cost.

3.5 RESEARCH ON MICROCONTROLLER SERIAL INTERFACE

In order to interface and program a microcontroller, we needed to come up with a way to use the serial interface inherent to a lot of them. This way, if we want to use a cheap microprocessor, we can program it to our specifications. After looking at different methods to do this, we came up with the idea of implementing a USB to UART chip in any design. This way we can easily transfer data from our computer to the microcontroller through USB. One part that is used in the motor controller to do this is the FT232RL.

The way that the UART Protocol works can be a bit confusing, but for our purposes, we only needed this as a way to program the microcontroller. So we don’t care about communicating the FT232RL with the microcontroller in any way while is it running. What instead happens is that the microcontroller is reset before data is sent through USB to it and then reset again once the data transfer is complete. This way there is no potential problem in having data ignored by the microcontroller because we force it to be ready when we need it to. This type of protocol is a serial communication.

The other protocol that was considered is SPI. This protocol also works serially, but it's a bit different in some key ways. Chips can be purchased to connect to the ATMEGA8 through SPI that will create a USB interface. The way we would use it if we ever found a use for implementing it would be a situation where we need to send and receive data at the same time. This saves on design cost by reducing the need for more parts, pins, and software to handle it all.

3.6 RESEARCH ON MICROCONTROLLER DESIGN

There are multiple ways to implement this project. One way is to design a microcontroller that interfaces with all the other parts. Another way is to design multiple, smaller microcontrollers and then have them connect to a bigger microcontroller; this in theory leaves more room for modularity should problems arise within individual components or should we want to design more than we had planned. Another option even is to buy the microcontrollers for our design purposes so that there would be less potential for erring on the design side of things.

We decided on a trade-off between the two, where we would have one powerful, purchased microcontroller that is able to run a task-intensive algorithmic analysis of everything it is doing, while leaving the design process to the smaller and simpler microcontrollers.

3.7 RESEARCH ON CHASSIS DESIGNS

There are a couple chassis designs that we had to consider in implementation of our robot design. One chassis that we considered had treads, each powered by two separate motors; a second chassis had two back wheels, each wheel powered also by two separate motors; a third chassis was the same as the second with the exception that it was quite long, so size became a factor. The following table outlines what it was that we were looking for in a chassis.

Power	**Decent Acceleration**
Size	**Not too long since the robot needs to turn 360 degrees**
Wheels/Treads	**A two axel system with four wheels for precise control**
PCB Mountable	**Needs to be capable of stacking PCBs**
Cost	**Low Cost**

The following chassis designs were considered and are noted for what they fell short on in red and what features were fulfilled in green in the following tables.

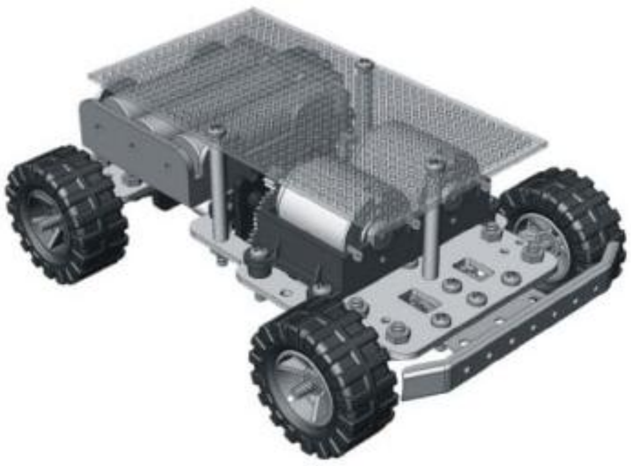
Chassis One – Machine Science Mobile Robot Kit

Power	Two Motors - Good Acceleration
Size	Max Length is Too Long (> 150mm)
Wheels/Treads	One axel with two wheels that slides on a ball bearing
PCB Mountable	Yes
Cost	Low Cost - $110

Chassis Two - 70108 Tracked Vehicle Chassis

Power	Has One Motor
Size	The max length is okay (< 150mm)
Wheels/Treads	Control is precise with treads
PCB Mountable	No
Cost	Low Cost - $17

Chassis Three – Mr. Basic Model TR-3



Power	**Two Motors - Good Acceleration**
Size	**Max Length is Okay (145*108*55mm)**
Wheels/Treads	**Two axels with two motors powering each one**
PCB Mountable	**Yes**
Cost	**Low Cost - $35**

3.8 RADIO FREQUENCY TRANSCIEVERS

For communication through the air to work with minimal interference a proper modulating and demodulation scheme must be chosen for our specific application. Since the robot will be in close proximity to the computer device it will be communicating with, the modulation method used will not need to have the advantage of distance if that were to be a trade-off over speed. There are many modulation schemes to consider such as the analog modulation methods of AM, FM, and PM or the digital modulation methods of PSK, QPSK, FSK, ASK, and QAM. The analog modulation methods provide a straightforward way to send analog signals such as voice over the air. However, digital modulation schemes involve digital-to-analog conversion of signals when transmitting and analog-to-digital conversion of signals when receiving.

The robot will send and receive bits of information, which makes the use of digital modulation schemes ideal for the communication subsystem. The bits will also be encrypted so the modulation scheme will require the least amount of interference possible to be able to accurately decrypt the information on the device receiving the bits. Both digital and analog amplitude modulations, such as AM and ASK, are susceptible to interference or noise but only require a simple design, however signal loss is difficult to detect. Frequency modulation schemes such as, FM and FSK provides less interference but multiple frequencies must be used for modulation and demodulation, which provides a difficulty in detection of signal loss since all frequencies used must be detected independently, in addition, more complex circuitry is required to implement than AM because it has to detect frequency changes in the carrier signal. A phase modulation scheme namely PM, PSK, or QAM, seems to provide the low susceptibility to interference and noise, like frequency modulation. However, only one frequency is used, which makes it easier to detect errors than frequency modulation schemes but more complex circuitry must be used to detect phase changes. One exception to the advantages of phase modulation is QAM, which is QPSK with amplitude modulation. Although more data can be transferred with this modulation scheme, this method is susceptible to noise due to the disadvantage of using amplitude modulation.

There are also spread-spectrum techniques, such as FHSS, DSSS, THSS, and CSS used to transmit a signal on a bandwidth is spread larger in the frequency domain to produce a signal with a wider bandwidth. The advantages of using this technique for transmission are that ability for multiple access and multiple functions in a particular wireless network. However, since our robot will only be communicating with one user, implementation of this technique will be superfluous for our application.

For the communication subsystem a digital modulation scheme that provides susceptibility to noise and easy signal loss detection is ideal. Quadrature phase shift keying provides all these requirements and 2-bits of data can be encoded, since 4 different phase changes are used for modulation in contrast with PSK, which only uses two phase changes to encode data.

Designing a communication subsystem from a modulation scheme can be difficult and ultimately futile since there are many options of integrated circuits available to provide low-power and cost effective solutions to transmitting and receiving data between two entities, i.e. the G.U.N.D.A.M. and a computer. There are plenty of RF transceiver manufacturers to choose from but not all of them provide modules that are tailored to our needs. As stated previously, an RF transceiver utilizing an amplitude modulation scheme is much more susceptible to noise than an RF transceiver using a frequency modulation scheme, however; there are more factors to consider since our RF transceiver will be applied to a mobile robot, namely power consumption. Although an RF module employing an FSK modulation technique it also requires more power to operate, which will reduce the battery like of the robot.

An RF module that has low-power consumption and is less susceptible to interference is the Semtech SX1211. It is described to be the lowest power integrated UHF transceiver in the Semtech product line. This module is a single-chip transceiver that operates in the frequency ranges between 863-870, 902-928, and 950-960MHz. The SX1211 has low power consumption of 3mA in receiver mode and 25mA at 10mdB in transmitter mode. Some of the features are configurable such as the modulation scheme can be either FSK or OOK. It also gives the options of programmable RF output up to +12.5mdB. The voltage needed for the power supply is 2.1-3.6V, which can be two AAA batteries. The bit rate achievable of up to 200kbps is high enough for our application. The data output and input is operates on an SPI bus, therefore, the MCU used by the communication subsystem must be able to handle this data transmission protocol. Although this module is highly versatile, main draw back is the small 5x5mm TQFN package, which would make it difficult to integrate into our G.U.N.D.A.M. design without acquiring the proper tools and in turn ultimately increasing the overall cost of the project.

Texas Instruments has a wide choice for RF transceivers with the differences being in operating frequencies, modulation techniques, and power consumption among others. The most critical factor influencing the choice of an RF transceiver is low-cost and power consumption, since our robot will be in close proximity to the computer, thus making range capabilities less significant. A possible choice for an RF transceiver is the CC1101 module. This module operates in the sub-1GHz frequency range, in contrast with the 2.4GHz range, which other TI products operate at. An advantage with operating in this range is for the module being able to provide better range for the same output power and current consumption. The frequency band range is lower than the SX211 from 300-348, 387-464, and 799-928. Power consumption is somewhat higher than the SX12111 with a receiver mode current 14mA, however; it has a larger maximum data rate at 500kbps.

Although having a higher data rate for the RF transceiver may not seem necessary, having the option will make this project much more scalable if the need arises for a higher data rate. The CC1101 also has a lot of versatility with programmable power output and data rate, which both effect range. For example, at an operating frequency of 915MHz, a power output of 0dBm, and data rate of 250kbps, the effective range if about 950ft, which is far beyond what is required of the robot. Therefore, we would be able to operate at the maximum data rate provided that the power consumption is within range. The operating current can be even further reduced using TI's TPS61730 step down converter. Using the TPS61730, the current can be reduced by 12mA from 34mA if the CC1101 is transmitting at maximum output power. As with the SX1211 the CC1101 module offers a SPI interface for communication. The CC1101 is more favorable than the SX1211 because of the scalability factor of having increased data rate with a minimal increase in power consumption.

The final RF transceiver module considered was the Linx Technologies' LT series module. Each transceiver module number corresponds to the operating frequency, which ranges from 315, 418, and 433MHz. Some of the advantages the LT series has over the other RF modules are the ease of integration into the robot design with an SMD package and the lack of external RF components needed for operation, which both lower assembly cost. Some of the disadvantages compared to the previous RF modules are the significant decrease in maximum data rate at 10kps and the AM/OOK modulation technique, which can make this module more susceptible to interference. However, the main advantage of the LT series RF module is that it is relatively simple to create a serial RS232 link between the modules and the G.U.N.D.A.M. with a voltage level converter, such as MAX232 from Maxim Semiconductor. A serial link is needed to send data between the microcontroller and the RF transceiver on the G.U.N.D.A.M. and a USB link is needed to send data between the RF transceiver and the computer. Similarly, Linx offers a USB module SDM-USB-QS-S that can interface directly with the RF transceiver module.

An additional and vital RF component to consider is an antenna that will fit into the G.U.N.D.A.M. without too many drawbacks. Although a lower operating frequency would increase the range of the device, a larger antenna would be needed to effectively reduce noise. Since an antenna would be quite large especially at the lower frequencies it is common practice to reduce the antennas length to half or a quarter of the length required for a particular wavelength. There are many antenna designs to consider including PCB or loop-style antennas, whip antennas, chip antennas, and specialty antennas.

PCB antennas are advantageous because they are built into the RF transceiver's board itself, however; some of the disadvantages are that it requires more board area than the RF module will need, the size will be less than ideal for operating at low frequencies, and it requires more resources to build, which could end up increasing overall cost of the design. Chip and specialty antennas can prove to be less expensive but there is a disadvantage of the reduction in range. A whip antenna has the advantage of having longer range and lower cost but if there were size limitations in the application than this antenna would be less than ideal. This antenna provides reliability and would make the G.U.N.D.A.M.'s communication subsystem more robust with respect to interference from other RF devices in our testing environment.

3.9 MICROCONTROLLER FOR RF COMMUNICATION

The G.U.N.D.A.M.'s communication subsystem will require a microcontroller to be able to handle the data being transmitted and received and implement an encryption algorithm. There are two main features needed for a microcontroller to be suitable for this application, which are the ability to communicate with an RF transceiver and another main microcontroller on the G.U.N.D.A.M.'s side. The microcontroller should also be able to communicate with an RF transceiver and read/write from a USB interface on the computer's side. To accomplish the first set of requirements the microcontroller needs to have two serial communication channels that can run simultaneously. The first channel will transmit and receive data from the RF transceiver and the second channel will transmit and receive data from the G.U.N.D.A.M.'s main microcontroller.

The MSP430 microcontrollers from Texas Instruments provide a Universal Serial Communication Interface that can handle UART, IrDA, I^{2}C, and SPI. Since most of the RF transceivers considered send and receive data using an SPI bus, it would make sense to used on channel as an SPI interface for the purpose of transmitting and receiving data to and from the computer. The MSP430 is also able to simultaneously use a second channel for serial communication and the I^{2}C interface provides a way to communicate with a second microcontroller, this being the main microcontroller in charge of the sensors and motor functions. In addition to these advantages, the MSP430G2xx microcontrollers provide a very low power and low cost solution to this application. Depending on the variation of the MSP430G2xx used it can have from 0.5KB-16KB of flash, 128-512B of RAM, an operating voltage from 1.8-3.6V, and up to 16MHz CPU speed. As noted earlier, the RF transceivers can run off two AAA batteries and this includes the MSP430 microcontroller as well.

3.10 ENCRYPTION ALGORITHMS

The G.U.N.D.A.M. communication subsystem will implement an encryption algorithm when receiving and transmitting data with the computer. Similar to modulation schemes there are many encryption algorithms to investigate for our application needs. One important thing to consider in encryption algorithms is the key and block sizes because while larger sizes provide increased security it will also require more memory. For the G.U.N.D.A.M. communication subsystem four encryption algorithms were considered: AES, SEED, RC5, and CAST. SEED is mostly used in the nation of South Korea and like other ciphers considered are block ciphers that have fixed length groups of bits. This encryption consists of a 16-round Feistel network 128-bit blocks and 128-bit key.

Rivest Cipher or RC5, in contrasts, has a variable block size ranging from 32, 64, or 128 bits, a key size ranging from 0 to 2040 bits, and a number of rounds ranging from 0 to 255 but similar to SEED it employs a Feistel network. CAST-128 is also a block cipher with a 12 to 16 round Feistel network, a 64-bit block size, and a key size between 40 to 128 bits. This encryption has been widely used in a number of products and for both commercial and non-commercial uses.

Advanced Encryption Standard is perhaps the most widely used encryption algorithm worldwide and has actually been adopted by the U.S. government. AES uses a fixed block size of 128 bits, variable key sizes of 128, 192, or 256 bits, and 10, 12, or 14 round substitution-permutation network depending on the key size. Cryptanalysis of AES with 128-bit key size has shown that there is high computational complexity using biclique attacks and related-key attacks. Along with its wide usage and susceptibility to brute-force attacks, this makes AES-128 an ideal encryption algorithm for the communication subsystem in the robot. There are two options for implementing AES-128 in the G.U.N.D.A.M., either hardware or software. However, a hardware implementation will require a specialized chip or RF transceiver, which could end up increasing the overall design cost, but a software implementation would be as secure and would not add to the cost of the design.

3.11 WIRELESS PROTOCOLS

All wireless protocols have their advantages and disadvantages even with the many advances in these protocols throughout the years. For the robot's communication subsystem the features required are low interference with other wireless systems and signals in the vicinity, a medium range of transmission, and low power consumption. Two main wireless protocols were examined for the G.U.N.D.A.M.'s communication subsystem, namely Wi-Fi and Bluetooth. IEEE 802.11 also known as Wi-Fi is a standard use for communication between computers in a wireless network. There are four versions of 802.11 protocols that are used widely today, in the other that they emerged they are 802.11a, 802.11b, 802.11c, 802.11n, and 802.11ac is currently under development. The following chart summarizes the main differences between them.

802.11	Frequency (GHz)	Max Data rate (Mbps)	Indoor Range (ft)
a	2.4	54	115
b	5	54	115
c	2.4	72.2	125
n	2.4/5	150	230

.Table 1: Specification of 802.11 protocols

The range and data rate for these different Wi-Fi protocols seem ideal for our application needs, however; any implementation of Wi-Fi causes high power consumption, which is less than ideal for a mobile moving device. IEEE 802.15 also known as Bluetooth, on the other hand, is widely used in applications that require low power consumption but there are range limitations when compared to Wi-Fi. The power consumption for Bluetooth can range from 1-100mW depending on the class of the device. Similarly, the range of transmission can be between 16-300ft and the data rate can fall between 0.7Mbps and 3Mbps also depending on the class of the device. Naturally, Bluetooth has a large advantage over Wi-Fi in terms of our applications needs.

As for cost Bluetooth has proven to be one of the lowest cost communication protocols to implement in a device. Many of the features of Wi-Fi, Bluetooth, and other existing protocols may not be necessary in the communication of two devices due to the type of information being transmitted between the two devices. Another obstacle inherent in the previous protocols is that they often come with an RF transceiver tailored to that certain protocol such as Bluetooth and Wi-Fi. Therefore, an off-the-shelf, cheaper, and more versatile RF transceiver couldn't be used or would be far more difficult to implement for the robot.

Some protocols are more suitable for embedded systems, such as ZigBee and SimpliciTi. ZigBee is a communication protocol designed primarily for low-power digital radios, which is also made to be simpler and less expensive than Wi-Fi and Bluetooth protocols. ZigBee nodes or RF transceivers implementing the ZigBee protocol have the advantage of being more responsive than other protocols, thus having a low average power consumption and giving the application a longer battery life. ZigBee can be implemented in many of Texas Instruments' RF transceivers but they are among those that consume the most power compared to the low-power CC1101.

Similarly, SimpliciTi is a low-power protocol aimed at embedded applications that require simple and small networks. SimpliciTI, like ZigBee, can be implemented and is specifically designed to run on TI RF transceivers in conjunction with MSP430 MCUs. The main advantages of this protocol are the low power that is needed to implement and the direct device-to-device communication capabilities. For our application both of these features are necessary in an efficient communication subsystem. However, as simple as ZigBee and SimpliciTI is, there can be a lot of unnecessary features or overhead that can be taken care of on the hardware side. A custom protocol similar to ZigBee or SimpliciTI gives the freedom to implement the features needed for the G.U.N.D.A.M. and the advantage of further reliability and security.

3.12 DISTANCE SENSORS

The G.U.N.D.A.M. requires distance sensors to be able to traverse through the maze. The paths of the maze that the robot will try to solve will have fixed width but variable lengths. To be able to stay on the center of the path consistently both reliable motors and chassis must be used that are capable of moving the robot in a straight line or a system of error correction that includes distance sensors. Since most motors and chassis set ups aren't capable of moving in a straight line and often drift to the left or right, a pair of sensors must be used on either side of the robot to ensure that an equal distance from each side of the wall is kept will moving forward or backwards down a path. The maze will mostly likely be a small practical build with narrow passages so the sensors have to be able to have a very short distance threshold.

There are two types of range finders, Ultrasonic and Infrared, that are low cost and have low power consumption but are also able to measure distance accurately in narrow paths. However, there are advantages and disadvantages in the use of these two sensors. Infrared sensors can have digital or analog output but the sensors that provide a binary output are meant to detect an object or obstacle not the distance of the object from the sensor. Infrared sensors have a wide range of minimum and maximum distances they are able to measure but there is also an inherent disadvantage in using infrared light to detect the distance of an object and that it the interference of infrared light outside or inside lighting. There are also some cases where the light the infrared sensor sends is not reflected back, but absorbed, from the object it is trying to detect, thus rendering the sensor useless. In essence, infrared range finders work through the process of triangulation where a pulse of light is emitted and reflected back at an angle, which the sensor uses to determine the distance from the object. These specifications can make infrared range finders impractical for large distances but very useful for smaller distances that other range finders aren't able to detect.

Ultrasonic sensors work in a different way than infrared sensors in that they emit sound waves to detect distances instead of light. An advantage ultrasonic sensors has over infrared sensors is that they can be used outside since there is no chance of interference with light. In addition, most objects do not easily absorb sound waves and the effectiveness of the ultrasonic sensor's ability to detect distances isn't as likely to be affected by the object being used. However, a large disadvantage that is inherent in ultrasonic sensors is that objects that are not flat or have many ridges can causes an echo, which could cause inaccurate distance measurements. The advantages infrared sensors have over ultrasonic sensors is that the infrared sensor's accuracy doesn't suffer from the shape of the object because it has a narrow beam and infrared sensors are generally available are lower costs than ultrasonic sensors. Also, designing and manufacturing accurate infrared sensors can be difficult but the same is not true for ultrasonic sensors.

The robot will need accurate sensors to be able to navigate efficiently and quickly through the maze, which would make ultrasonic sensors an ideal choice. However, infrared sensors provide a low-cost solution to distance measurements. Since the narrow paths will be flat and made of light-reflecting material it makes sense to use infrared sensors on the left and right side of the robot and use ultrasonic sensors on the front and back of the robot to reduce the overall cost. The trade-off of accuracy over cost is minimal in this case since the maze will be designed around the requirements of the sensors to make accurate measurements. The limitations of the sensors will also apply to the overall limitations of the robot, for example, infrared sensors cannot be used outdoors because of the interference of all the infrared light emitted by the sun and even in some indoor settings the sensor can prove to be somewhat inaccurate.

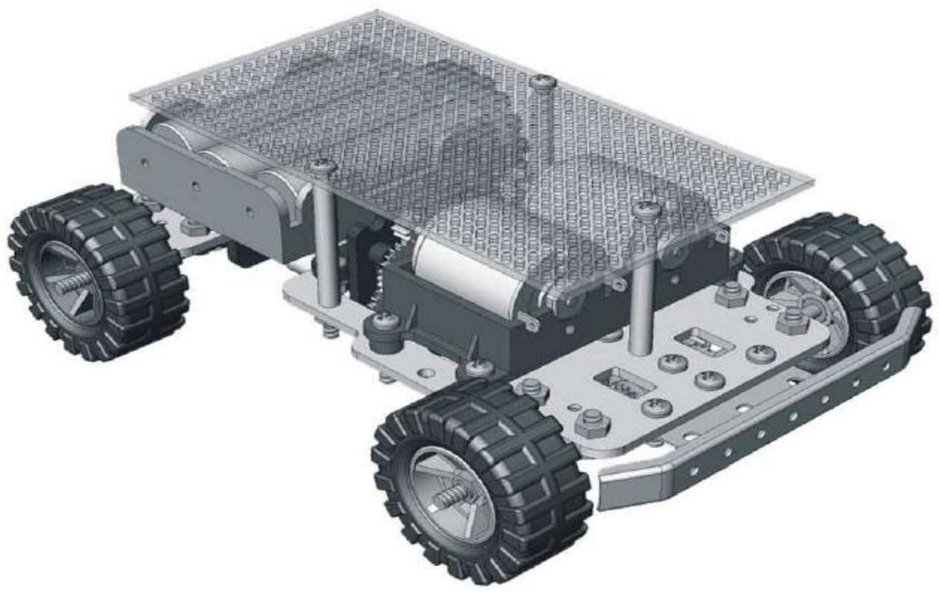
4.1 CHASSIS DESIGN/LAYOUT

Introduction

Rather than build a chassis for our robot, we decided to buy one instead. The reason for this is that mechanical engineering is outside the scope of our knowledge for this project. Some of the things that we decided we would need in a chassis are a way to attach PCBs, motors, a battery, and have a fully functional axel and wheel system that could be taken advantage of.

Concept of the Chassis

In order for the chassis to fit our specifications of solving a maze in a quick amount of time, the chassis needs to be capable of housing big motors that can induce a quick acceleration and stable velocity when required; this means that for our purposes, a chassis with treads is not going to be suitable. Instead we found a chassis that has four wheels, which has two axels for the front and back of the robot, capable of obtaining high accelerations. We found a chassis that can accomplish this, although the placement of the battery is small and inconvenient. However, we have some freedom on where we can place parts, since the chassis is assembled in a modular manner and since parts can be stacked up high over one another. The robot we chose is the Mr. Basic – Model TR-3 (shown below).

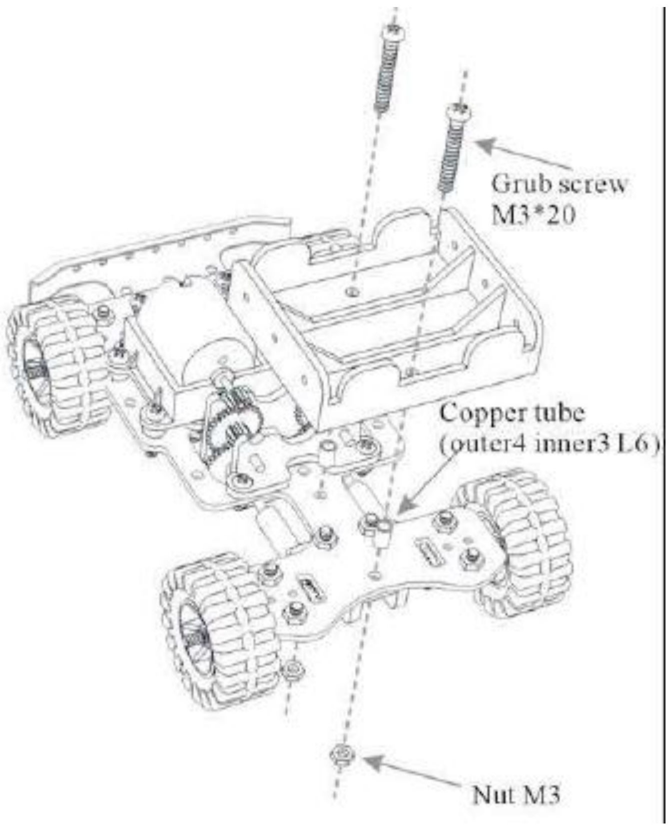


Parts of the Design

As can be seen from the picture of the fully assembled Chassis, each part of the chassis needs to be assembled; this is a benefit because if any of the parts need to be changed, modified, or upgraded, it will be easier to do. This chassis also comes with its own motors. Unfortunately, the motors are from China and the only information I can get on them is that they can operate in between 4.5v to 9v and are “strong and powerful”. If they aren’t powerful enough, we will replace them.

Motors (Motor260)	**Operating Voltage**	**Required Acceleration**
Min	**4.5v**	**1** $\frac{m}{s^2}$
Max	**9v**	**5** $\frac{m}{s^2}$

The battery on the chassis will be replaced with something that is rechargeable and will supply power to all of the robot’s circuitry and motors.



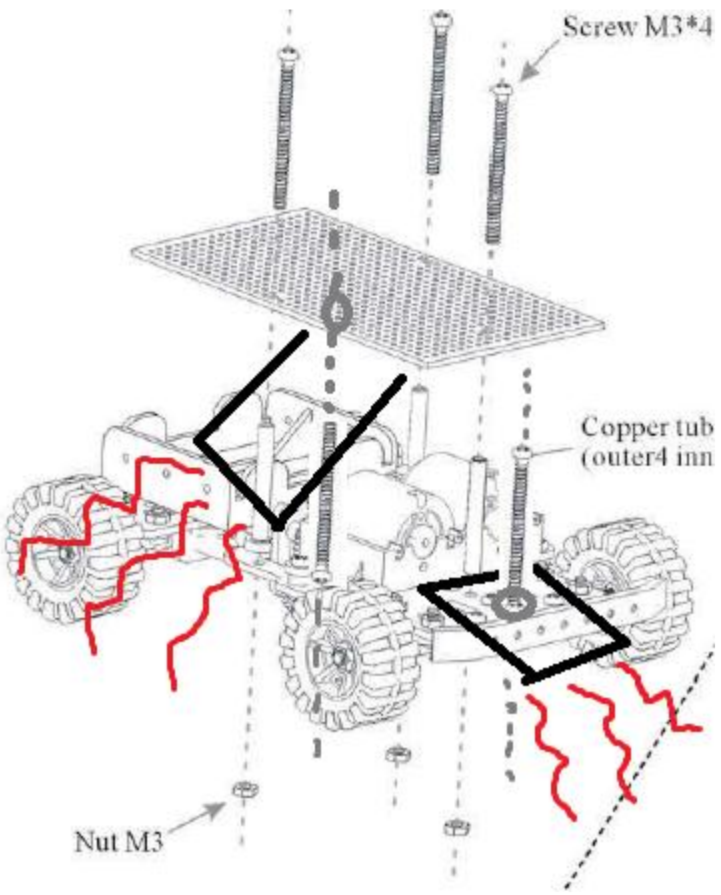
As can be seen above, since the battery holder is held on by two screws, this just has to be replaced with a cage that can accommodate the screws and the battery that we wish to use. For a rechargeable and more efficient energy supply, we've picked a Poly Quest PQ-0800LP-3S battery.

PQ-0800LP-3S Battery

Model	**PQ-0800LP-3S**
Format	**Triple (3S)**
Capacity	**800mAh**
Discharge	**18C (14.4A) continuous, 27C (21.6A) burst**
Voltage	**11.1V**
Size	**107 x 26 x 19mm**
Weight	**2.50oz.**
Connector	**Wire and PQ/Hyperion tap**

The placement of the PCBs for this robot will go directly on top of the PCB plate holder. There are holes to allow for positioning of our PCBs, as well as to give the ability to stack layers of PCBs.

The placement of the sensors will require more consideration. Since we have three sensors, which will be on their own PCB and will have their own microcontroller, we need to position them so that they can be securely held into place. To resolve this, it is decided that we can place the PCBs under the PCB plate holder and screw them up into it, as shown in the following.

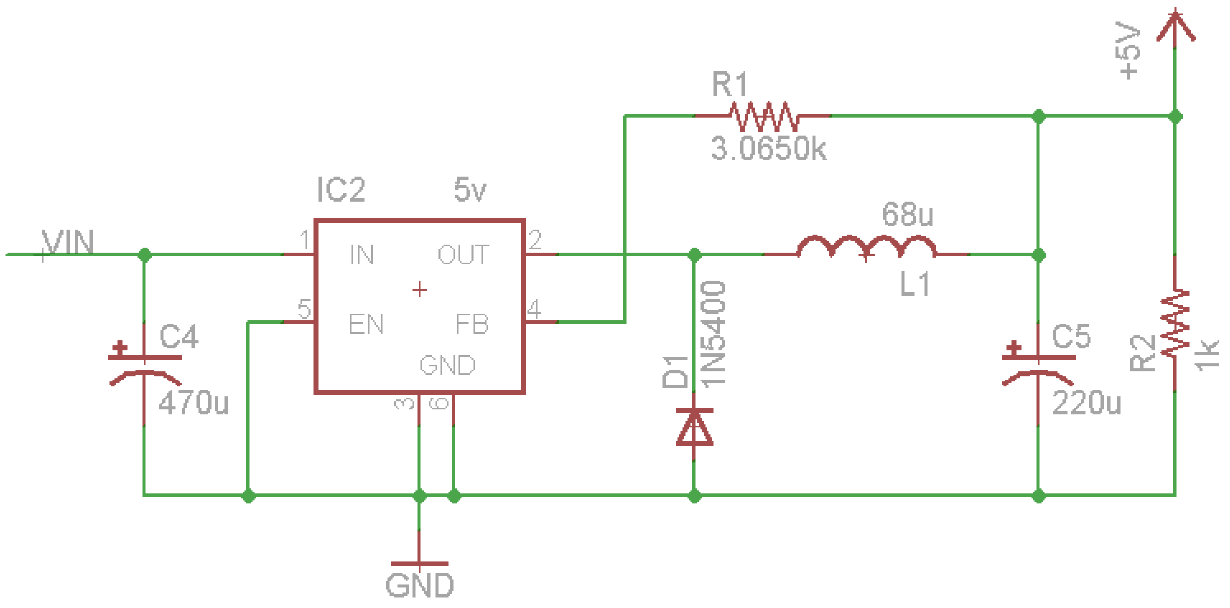


4.2 BEAGLEBOARD-xM DESIGN

It was determined that we were going to need a way to run sophisticated maze searching algorithms, as well as do heavy calculations. Part of the robot's goal is to create a 2-dimensional visualization of where it has been and use that information to visualize where it has been and what kind of maze it is dealing with; because of this we needed a microcontroller with a great deal of power. It was decided that an ARM processor would supply us with the processing power that we would need. Since the scope of designing a microcontroller around an ARM processor is outside the scope of our knowledge and was not something we could learn quickly, give our time restraints, we opted to buy a designed microcontroller, the Beagleboard-xM. The features of the board that are going to be utilized are shown in the following table.

OS	**Angstrom Linux**
CPU	**1Ghz Arm Processor**
Static Memory	**MicroSD**
Dynamic Memory	**512 MB LPDDR**
Ports	**4 USB Ports**
Embedded Interfaces	**I2C, I2S, SPI, MMC/SD**
Screen Interfaces	**DVI**

It's worth noting that this board is rated to run at no more than 2A. Supposedly, from a firsthand source online, with a USB wireless adapter plugged in, it will not go above 1A. In order to supply the current needed for this board, a switching regulator will be used to take the input voltage from the battery and supply the proper 5v to the board, efficiently. The circuit is shown below.



The way the circuit is set up from the datasheet of the LM2596S-ADJ, we needed to set up the following values for each component

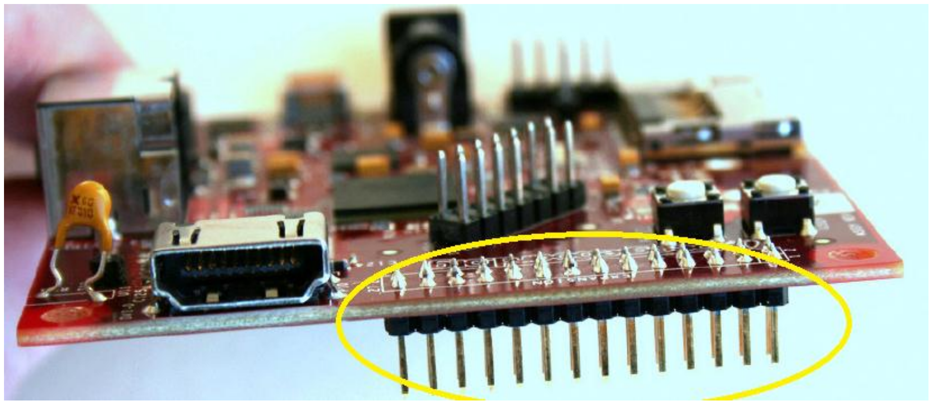
R_1	**? kΩ**
R_2	**1kΩ**
C_{IN}	**470μF**
C_{OUT}	**220μF**
D1	**1N5400**
L1	**68μH**
V_{out}	$1.23\ (1 + \frac{R_2}{R_1})$

To make $\mathrm{V}_{\mathrm{out}}$ be the 5 volts that is required, R_1 was calculated to be 3.065kΩ.

$$\boldsymbol{V_{out}} = \mathbf{1.23\ (1 + \frac{R_2}{R_1}) = 1.23\ (1 + \frac{1000}{3065}) = 5\ volts}$$

Now the Beagleboard has Angstrom Linux stored on the 4gb MircoSD card. When the board is booted up, there is a fully functional linux environment from which to control and interface all the other microcontrollers in the project.

There is also an interface on the board that we would like to use for handling input and output and interrupts. The way this can be handled is through the ARM processor's GPIO pins. The beagleboard has a dedicated expansion slot for this. A female connector has to be soldered into the board in order to handle this, but once done it should supply us with what we need to communicate with our other embedded circuits.



How Will the Beagleboard Software Function

This beagleboard will have to do a lot of things. First it has to receive data from the sensors; second it has to give commands to the Motor-Microcontroller; third, it has to transmit data wirelessly over a USB adapter; and lastly, it has to use an algorithm that dynamically attempts to solve the maze it is in, while also drawing that maze out. The following Pseudo-Code illustrates what will be happening.
//The beagleboard needs to be able to handle receiving input from the sensors as the //input is created. This means it will need to handle reading in data from three sensors.

```
while (1) { //main execution loop
        Read in current distance from sensor 1
        Read in current distance from sensor 2
        Read in current distance from sensor 3

        //An algorithm will be run that uses that data and calculates what direction to face
        //and how fast to go
        Calculate position
        Calculate current speed
        Calculate current acceleration

        Determine Motor Control Actions
        Set Pins to command Motor-Microcontroller

        //An algorithm will also be run that will take the current information from the
        sensors and normalize a location on a map
        Determine new map uncovered
        Draw Map

        //We'll have to send that data to a laptop in order to view the map
        Transmit data of the map wirelessly
}
```

4.3 BATTERY SWITCHING VOLTAGE REGULATOR DESIGN

Introduction

We need to be able to supply power to all of our devices with a regulated voltage and enough current for each device. The best way to do this would be to put the voltage regulators needed for all individual microcontrollers and put it all onto one PCB. This way we can help make part of the design process more modular and narrow down any problems or errors in the later design that we might happen upon.

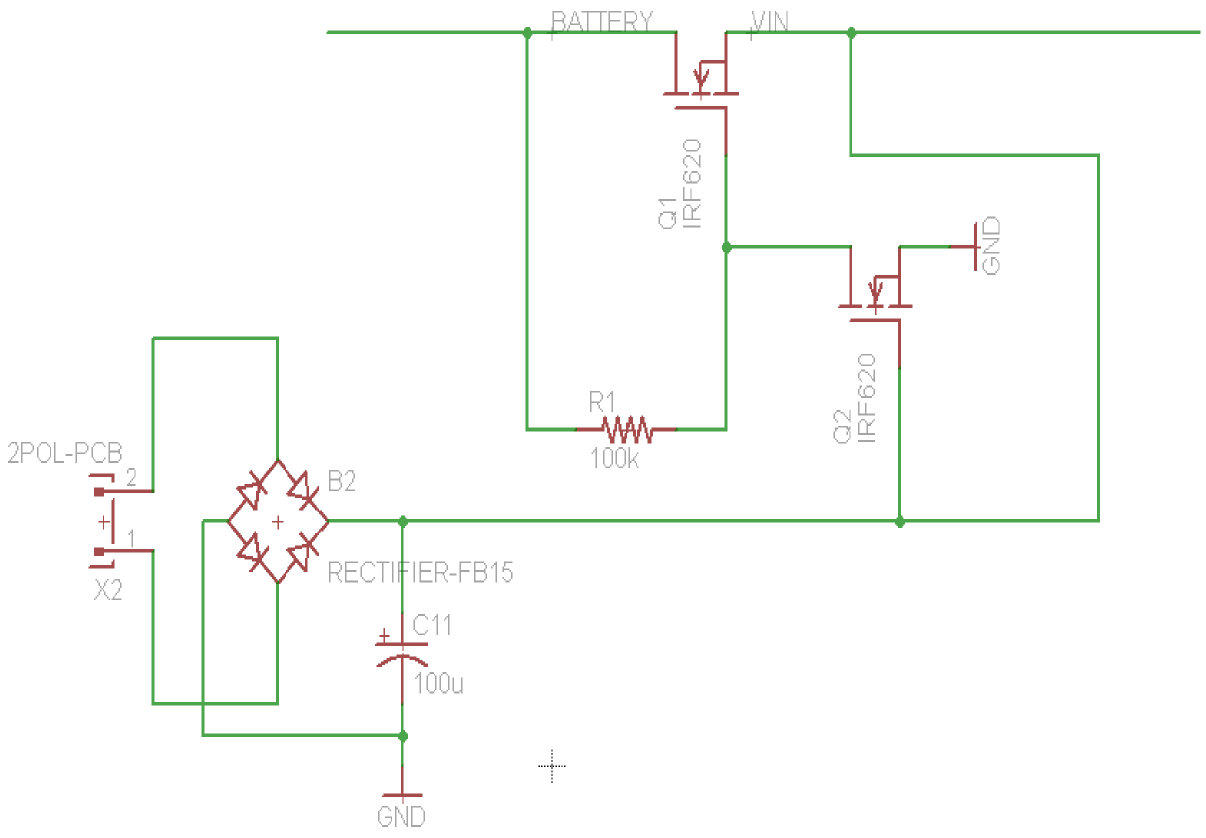
Concept of the Voltage Regulator

Provide a stable and clear power for all devices, without having to worry about implementing a separate regulator into each PCB design. This also helps save on energy cost. We also need to be able to switch between using energy from the battery and using energy from a wall outlet.

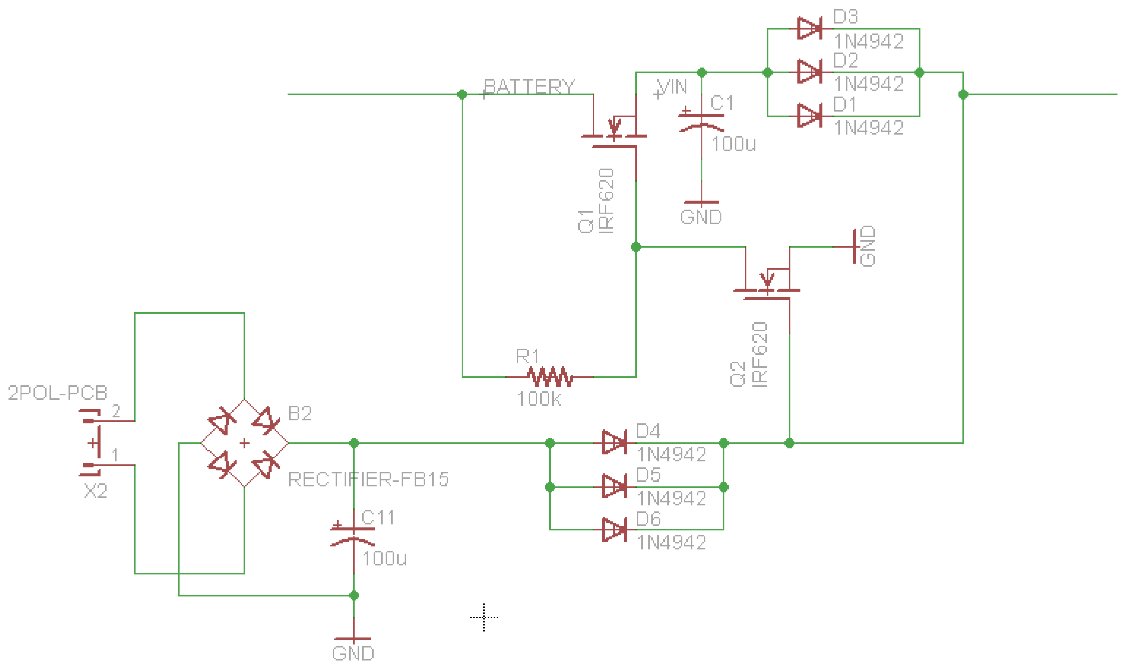
Parts of the Design

In order to implement this design, a switching circuit needs to be designed for each regulator. The best way to do this is to use two Power Mosfets for each regulator that connects both the battery voltage and the outlet voltage in a switching manner. When there is no outlet power, the one mosfet will act as a closed circuit and consume energy from the battery, while the other will be open circuit; when there is outlet power connected to the circuit, the mosfet will act as an open circuit, disconnecting the battery voltage source from the circuit, while the other mosfet acts a close circuit, consuming energy only from the outlet power and not the battery.

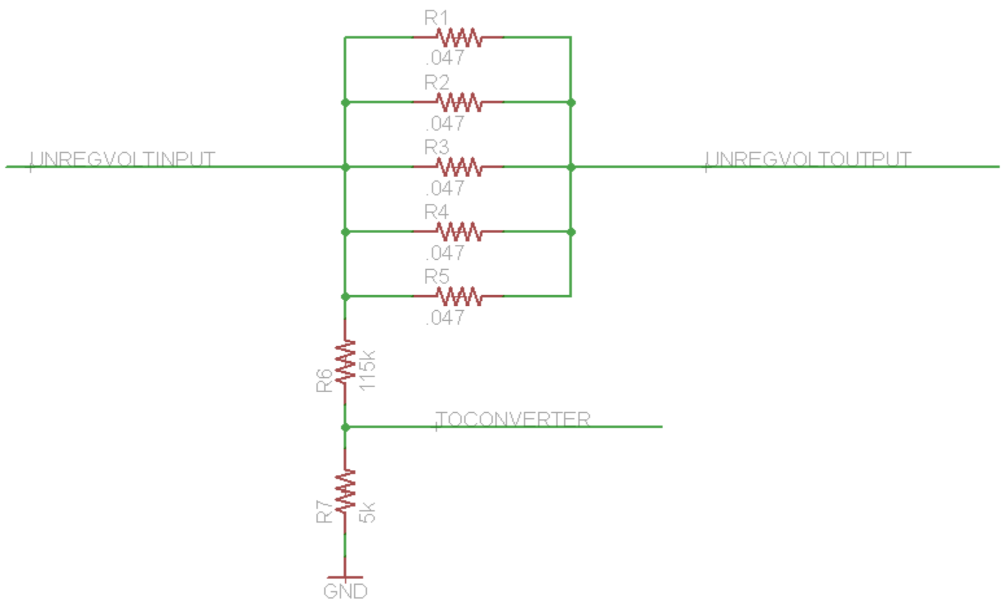
Battery Power	**Off**	**On**
Outlet Power	**On**	**Off**
Mosfet1	**Off**	**On**
Mosfet2	**On**	**Off**
Result	**Outlet Powered**	**Battery Powered**



In the above schematic, the way that we can switch between using power from the outlet and power from the battery can be done using the two power mosfets. In order to ensure that when the circuit is in switching mode that there is no drop in power, a high capacitance should be placed on the output of the switching. Also, to ensure that power from the battery does not go into the wall outlet or that power from the wall outlet does not go into the battery, two diodes should be placed as well. Our final schematic is shown below.



Another feature I would like to add on top of this is the ability to calculate and view how much energy is being consumed by the robot. I like the idea of having such a feature because it will help us make sure we are drawing a reasonable amount of power and that there are no problems with the amount of power that is being drawn. In order to save on part costs, instead of utilizing an extra micro-controller that will calculate everything, I'm making this a part of the motor-microcontroller, since the code that it runs is pretty straight forward and adding this to its clock cycles shouldn't be a problem.



From the above schematic, the resistors used are low ohmic, high power resistors. Each resistor is capable of .5 Watts; so the reason they are in parallel is to increase the maximum power to (5*.5 = 2.5) 2.5 Watts. These resistors only have to be placed across the voltage line coming out of the Switching Voltage Regulator circuit before the voltage gets regulated. This way, the actual unregulated power is the power we are measuring and not the total regulated power that the devices end up using. The line on the bottom of the circuit (TOCONVERTER) is to be connected to one of the Analog-to-Digital input pins of the Motor-microcontroller on the Atmega8. The reason for using the voltage divider is that the Atmega8 will only convert an analog signal in the range of 0 to 5 volts; so since the maximum voltage will be 120v from outlet power, then having a 115kΩ resistor in series with the 5kΩ will guarantee at most a 5 volt signal going to the Atmega8. Then by knowing the resistance across the parallel resistor network and knowing the voltage across the R7 resistor, we can get an almost instantaneous update on how much power we are consuming at each moment in time using the following equations.

$V_{Parallel\_Network}= V_{R7}+ 23*V_{R7}$

$I_{Parallel\_Network} = \frac{V_{Parallel\_Network}}{R_{Parallel\_Network}}$

$P_{Usage} \approx V_{Parallel\_Network} * I_{Parallel\_Network}$

So by adding the following pseudo code into what is already in the Motor-Microcontroller software, we can also keep a running tally of how much power is being consumed from all the circuits combined.

```
//As the Motor-Microcontroller goes through each loop iteration it will always calculate
//the newly collected power since the time it was collected previously
Initialize Routine {

        …. //Everything the Motor-Microcontroller is already doing in this routine
        Set $R_{Parallel\_Network}$ = Measured Resistance of the Parallel Resistors
        Set Beginning_Time = the current time
        Set Time = the current time
        Set Energy = 0 //This is going to be how many Joules we've consumed in total
                       //while operating
}

Main Loop {

        …. //Everything the Motor-Microcontroller is already doing in one loop
        Set $V_{R7}$ = The Current voltage found Across Resistor $R_7$
        Set $V_{Parallel\_Network}$ = $V_{R7}$+ 23*$V_{R7}$
        Set $I_{Parallel\_Network} = \frac{V_{Parallel\_Network}}{R_{Parallel\_Network}}$
        Set $P_{Usage} \approx V_{Parallel\_Network} * I_{Parallel\_Network}$
        Set Change_In_Time = current_time – Time
        Set Energy += $P_{Usage}$ * Change_In_Time //Add consumed energy to total
        Set Time = the current time
}

Print_Average_Wattage_of_Device {
        Set Average_Wattage = Energy / (the current time – Beginning_Time)
        Send Average_Wattage to Beagleboard
}
```

4.4 MOTOR-MICROCONTROLLER DESIGN

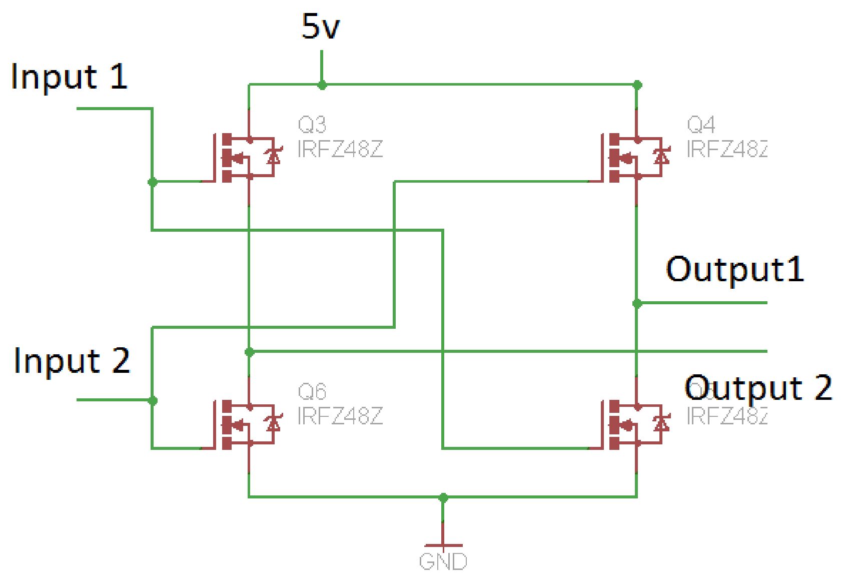
Introduction

In controlling the motors for the robot, a circuit had to be designed as a microcontroller that could take in various commands from the Master Microcontroller, as well as inputs from any extra sensors, and then drive the motors properly.

Concept of the Motors

There are two motors in this design. One motor will control the right side of the robot and the other motor will control the left side of the robot. To get a motor to move, a Pulse Width Modulation signal is generated at 5v across that motor. By making the duty cycle longer or smaller for each pulse, we can control how fast or slow the robot will move; and by reversing the voltage polarity across the motor, we can make it turn in the reverse direction.

In order to control which direction we want the motors to spin, a set of mosfets are used in the following configuration.



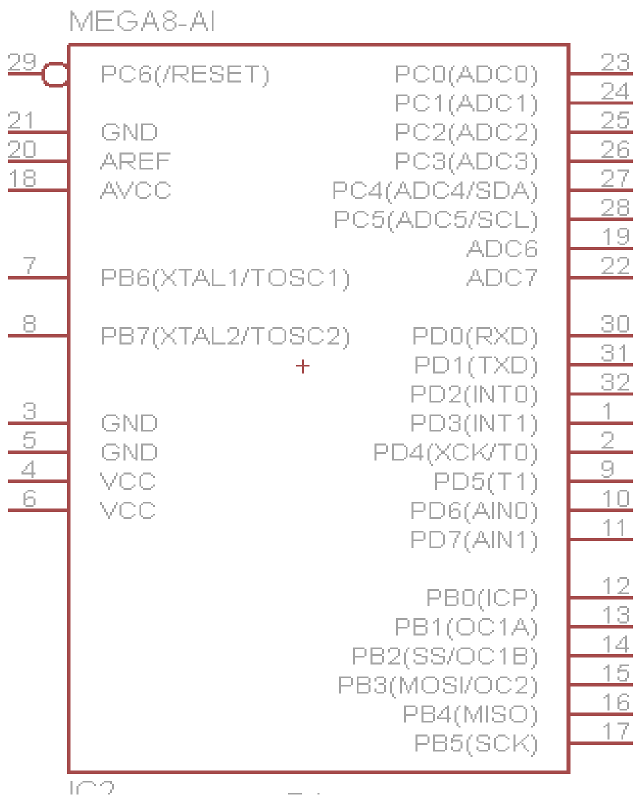
The Motor would be placed across ‘Output 1’ and ‘Output 2’; then depending on what the inputs are, ‘Output 1’ will act as an input voltage to the motor, while ‘Output 2’ will act as the ground or ‘Output 1’ will act as the ground, while ‘Output 2’ acts as the input voltage. Since the mosfets used are N-channel Power Mosfets, if 5 volts are supplied to the gate, they will invert, allowing current to flow. The following table represents this

Input 1	**Input 2**	**Output 1**	**Output 2**	**Output**
5v	**0v**	**GND**	**5v**	**Turn Right**
0v	**5v**	**5v**	**GND**	**Turn Left**
0v	**0v**	**0v**	**0v**	**BRAKE**
5v	**5v**	**Not Allowed**	**Not Allowed**	**Short Circuit**

*Note: In order to prevent damage from a short circuit, a polyfuse is going to be placed right before the GND.

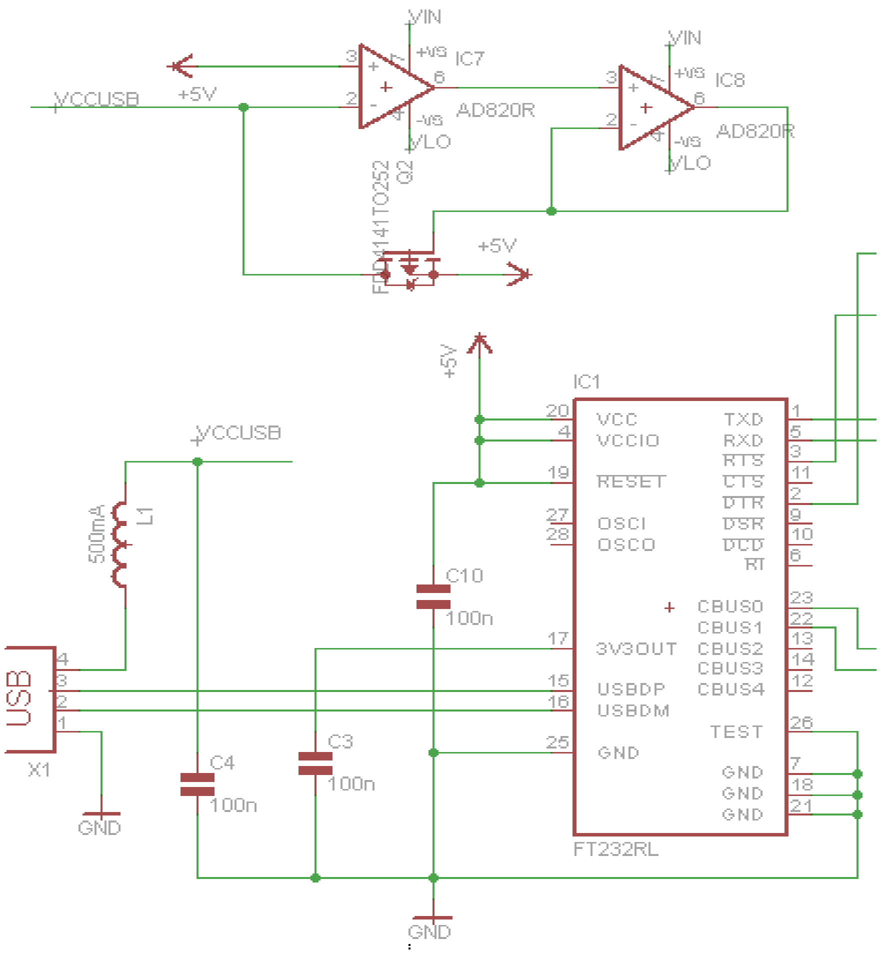
Parts of the Design

One part that is used in this microcontroller is an ATMEGA8. This part was chosen because it can be programmed quite easily using an Arduino IDE to interface with the chip through USB. It also has a good number of pins that can be used as interrupts and to control the motors of the robot.



As can be seen above, the ATMEGA8 supports SPI (Port B) and UART (Port D) to interface communication between the microprocessor. It also has two pins to allow for a crystal oscillator that will provide the clock for the microprocessor. Port C can be used for digital-to-analog conversion. Since the digital to analog converter is not needed and the Port B pins are unused, all the pins on the right side of the chip, other than PD0 and PD1, can be used as I/O and interrupt pins.

Another part that is implemented is the FT232RL. This was part was chosen to create a USB interface between the ATMEGA8 and its UART interface, so that we could program code into the ATMEGA8.

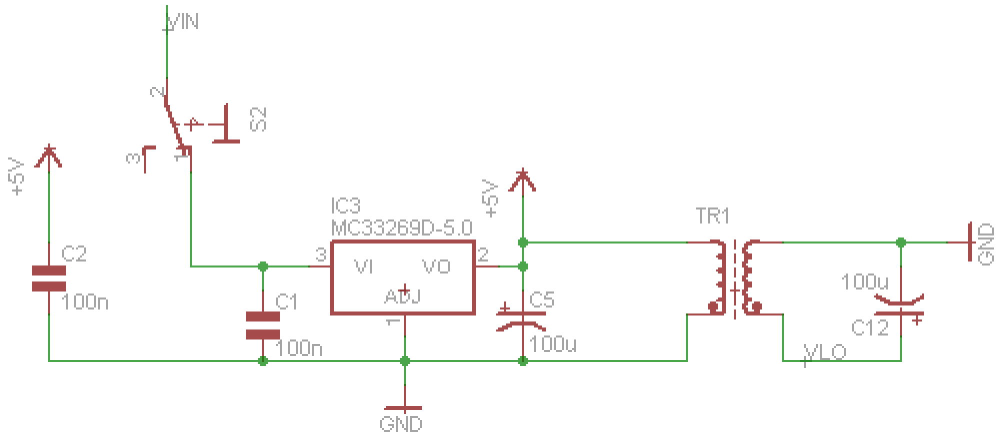


From the diagram above, there are some things to note about how this was set up. First, since the USB protocol can supply its own 5v supply, there is a circuit with two op-amps and a mosfet that switches between power from USB and power from another source (a battery or charger). If no external power is supplied, the P-channel mosfet will get no voltage to its gate and will be on, connecting power from USB; if external power is supplied, the P-channel mosfet will then turn off, allowing that external source to power everything.

There is also decoupling capacitors and a polyfuse that acts as a resettable fuse, which will prevent any devices from getting destroyed if too much current is drawn; the decoupling capacitors are used to ensure that voltages remain in the ranges they intended to be and help reduce any noise or distortion that might be created by electromagnetic interference or any voltages that are not intended.

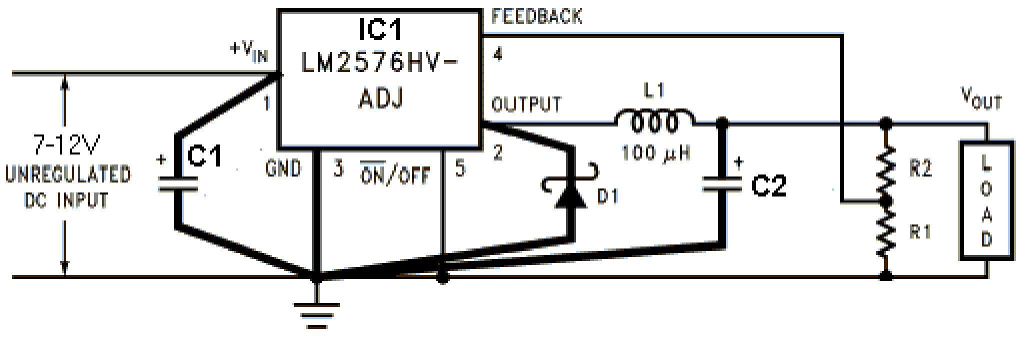
The TXD and RXD lines connect to the TXD and RXD lines in the ATMEGA8 with 1k Ohms of resistance between them. CBUS0 and CBUS1 work as LED lines to enable LEDs when the TXD and RXD lines are being used.
To supply power to these parts, there needs to be a 5 volt power supply. If the microcontroller is plugged in, it will run off the USB power, but if not then it will use a

regulator that will generate a positive and negative power supply of +5v and -5v respectively.



In the diagram above, an MC33269D-5.0 is used to generate a constant 5v power source. The only requirement is that at least an 8v source has to be supplied to it through VIN. A couple things to note about this design is that there are decoupling capacitors used to keep everything stable, as well as a transformer that generates a -5v power supply from the +5v power supply line.

In case the power requirement of the robot becomes too great, this can be upgraded to a switching regulator, which is more expensive, but will help save power in the design.



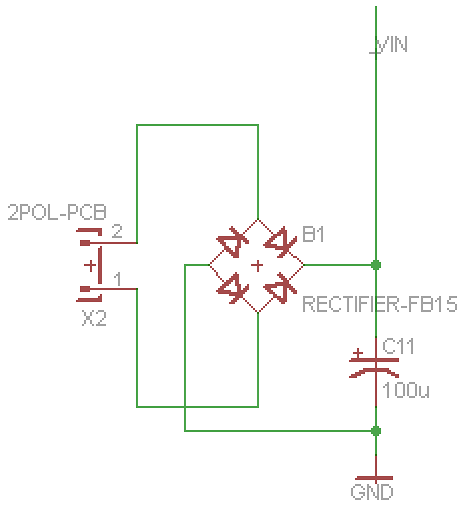
- Taken from datasheet for LM2576HV-ADJ

The way the switching regulator would save power is that it stores energy in an inductor and capacitor on the output. Then it can use a pulse width modulated signal to enable and disable current flow to the output while keeping a steady 5v output. This requires less current over time and less wasted power from the input.

	Input Power	**Output Power**	**Efficiency**
Unswitched	**12v*5A = 60W**	**5v*5A = 25W**	**25/60 = .416**
Switched	**12v*3A = 36W**	**5V*5A = 25W**	**25/36 = .691**

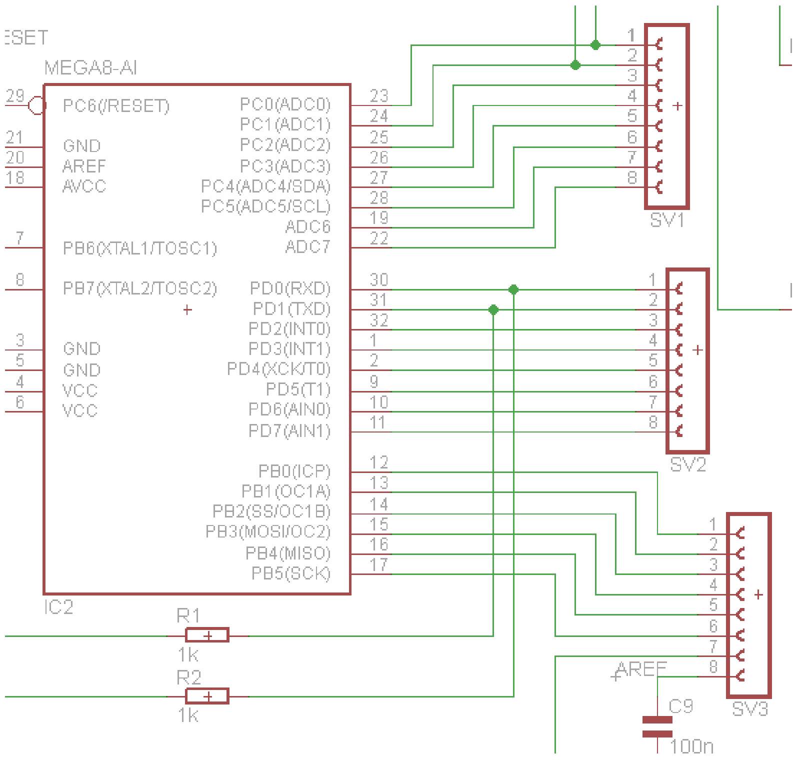
From the diagram above, it's clear that a switching regulator saves power. Some claim efficiencies around 90%.

Another part that is need for this microcontroller is a full-wave rectifier to VIN. The following schematic illustrates the design.

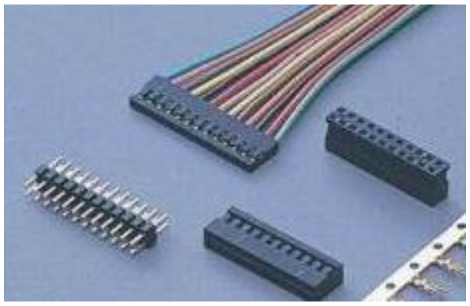


As can be seen from the above diagram, this rectifier uses a four diode bridge in order to create a full-wave rectified signal for VIN. The capacitor used is to keep the voltage from dropping to 0 volts and to keep it from changing as much. The diodes used are low voltage and should keep the GND across the capacitor at around 0 volts.

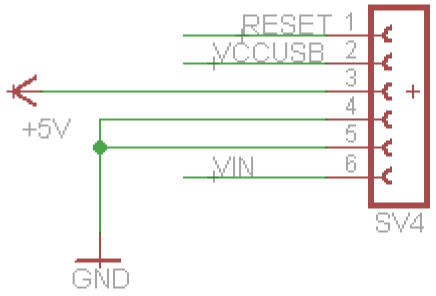
Another important part to look at in this circuit is the pins.



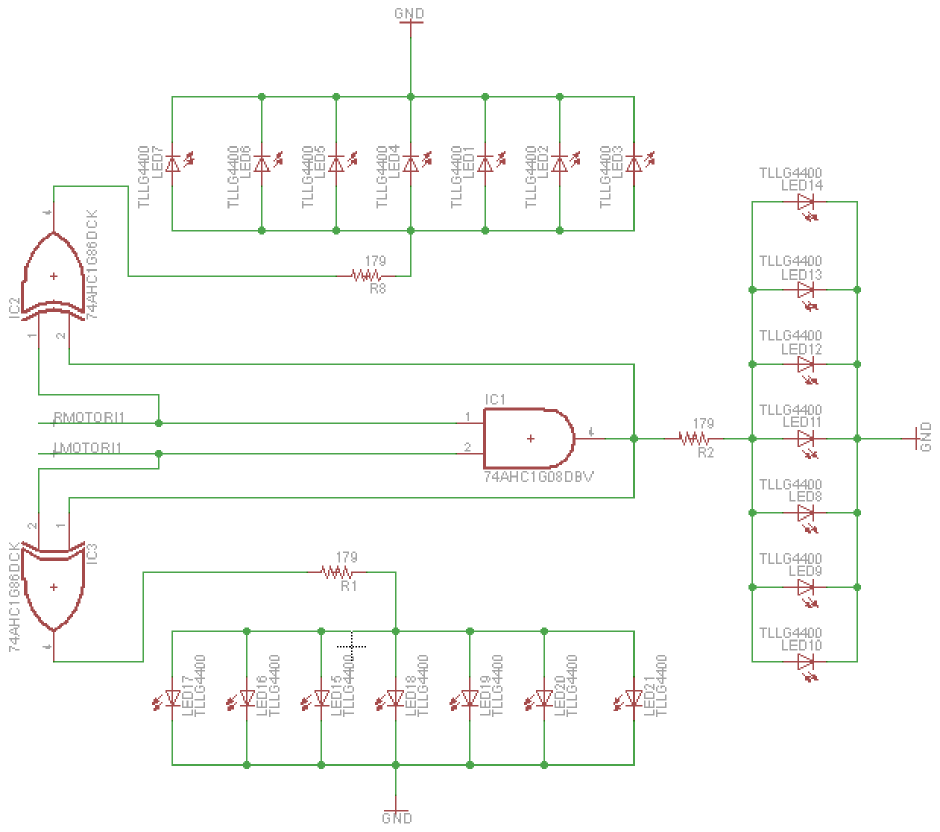
Each pin in the following diagram can be used as an input or output pin, while INT0 and INT1 act as two interrupt pins for this device. Each pin will use a female socket header so that it will be easy to secure wires to these pins using a standard male dupont connector. The following diagram illustrates this.



There is also a set of extra supply pins for an easy way to add on useful circuits, as shown in the following.



There is also going to be a set of LEDs to signal on the chassis whether or not the robot is turning right, turning left, or going forward. The LEDs will utilize the outputs of the Motor-Microcontroller to the motor mosfets to enable and disable networks of LEDs. A network of LEDs will look like the following.



What the schematic is doing is utilizing two input lines of four input lines that control the direction of rotation of each motor. Two input lines each go to a motor to control it, but only one input line from each is needed to switch on the LEDs. When both of those input lines are high, the robot is moving forward and only the forward LEDs will become active. The LEDs used are low power and become activated at 2mA. Their maximum power rating is 7mA, so I opted for a resistor that would allow 4mA to pass through each LED. To calculate what the resistor values needed to be, the following equation was used.

Input_Voltage = 5v
//7 LEDs are in parallel at 4mA each, so a resistor has to pass through 4*7mA = 28mA
Input_Current = 28mA
Resistance = 5/.028 ≈ 179Ω

Power Consumption of One LED Network	**28mA*5v = .14W**
Total Power Consumption of all three LED networks	**.14W*3 = .42W**

As of this moment the power consumption doesn't seem to be a problem, but if it is later on, each network will be reduced to three LEDs, yielding a total power consumption rate of .18W.

Microcontroller Software

The Motor Microcontroller needs to be able to accept commands from an external source and use that to control the motors. To do this, we need three logic lines. Two pins can act as data inputs, while the third pin can act as an input interrupt.

Input Pin 1	**Input Pin 2**	**Output**	**Robot Effect**
High	**Low**	**Right Motor Turns Right; Left Motor Turns Left**	**Robot Turns Right**
Low	**High**	**Right Motor Turns Left; Left Motor Turn Right**	**Robot Turns Left**
High	**High**	**Right Motor Turns Right; Left Motor Turns Right**	**Robot Moves Forward**
Low	**Low**	**Right Motor Off; Left Motor Off**	**Robot Stops**

The way this is supposed to work is that the microcontroller will only change the values of its outputs that control the mosfets when it receives an interrupt request; when it receives the request, it will use the data on Input Pin 1 and Input Pin 2 to change those mosfet outputs. So the only time the outputs to the motors will change is when an external interrupt is received on the microcontroller. The mosfets for each pin then switch between four states (go forward, turn left, turn right, and stop) for the motors.

The pseudo code for the Motor Microcontroller will operate along the following lines.

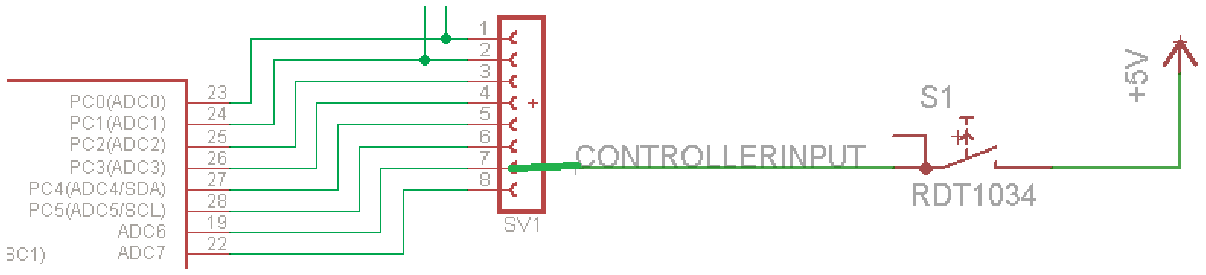
```
//Initialize registers
Initialize {
    Input1;    Input2;    Mosfet1Output1;Mosfet1Output2;    Mosfet2Output1;
    Mosfet2Output2;
    Set Mosfet Group Input and Output Pins
        &Input1 = Pin Used
        &Input2 = Pin Used
        &Mosfet1Output1 = Pin Used
        &Mosfet1Output2 = Pin Used
        &Mosfet1Output1 = Pin Used
        &Mosfet1Output2 = Pin Used
    Set Mosfet Group outputs to a valid combination
        Mosfet1Output1 = 1; Mosfet1Output2 = 0; Turn RIGHT Motor One
        Mosfet2Output1 = 1; Mosfet2Output2 = 0; Turn RIGHT Motor Two
            OR
        Mosfet1Output1 = 0; Mosfet1Output2 = 1; Turn LEFT Motor One
        Mosfet2Output1 = 0; Mosfet2Output2 = 1; Turn LEFT Motor Two
```

```
                  OR
            Mosfet1Output1 = 0; Mosfet1Output2 = 0; BRAKE Motor One
            Mosfet2Output1 = 0; Mosfet2Output2 = 0; BRAKE Motor Two
}

//Loop – Main
Loop {
      Wait for an interrupt request on the Interrupt Pin
      IF Interrupt_Received
            GoTo InterruptProcedure;
      Else
            ;
      }

//Handle the Interrupt when it is received
InterruptProcedure {
      IF Input1 == 0 && Input2 == 1 then              //Turn LEFT
            Set Mosfet1Output1 = 0; Set Mosfet1Output2 = 1;
            Set Mosfet2Output1 = 0; Set Mosfet2Output2 = 1;
      ELSE IF Input1 == 1 && Input2 == 0 then         //Turn RIGHT
            Set Mosfet1Output1 = 1; Set Mosfet1Output2 = 0;
            Set Mosfet2Output1 = 1; Set Mosfet2Output2 = 0;
      ELSE IF Input1 == 0 && Input2 == 0 then         //STOP
            Set Mosfet1Output1 = 0; Set Mosfet1Output2 = 0;
            Set Mosfet2Output1 = 0; Set Mosfet2Output2 = 0;
      ELSE IF Input1 == 1 && Input2 == 1 then         //Go FORWARD
            Set Mosfet1Output1 = 1; Set Mosfet1Output2 = 0;
            Set Mosfet2Output1 = 0; Set Mosfet2Output2 = 1;
      Return from Interrupt;
}
```

Other Considerations

Since the sensors may have limitations on how they function at close range, this motor-microcontroller is capable of intercepting a signal from a device that lets it know there is a problem and to brake the motion of the motors. Such a device can be simple, such as a bumper that when hit acts as a push button switch, switching on a circuit that informs the motor-microcontroller to stop what it's doing or to move into a different position. We're not sure if we will need this, but if it comes down to it being necessary, a Mushroom Push Button Switch will most likely be used.

Such a switch is long enough that it can signal before the sensors get too close to an object; and its button switch is big enough that any surface that comes in contact with it will most likely cause it to trigger. For a circuit, all that has to be done is to allow this to act as an open and close circuit, drawing a high voltage across the line when it is closed and no voltage (or zero volts) when it is opened. This line can be an input line on the motor-microcontroller, allowing it to know when to stop the motors or orient the robot in a new direction. A schematic is shown for illustrative purposes.



5.1 MAZE SOLVING DESIGN

The GUNDAM can perhaps send information to the computer at intervals. This information can be saved in a simple text file with each string representing a separate set of actions accomplished during a pre-programmed interval. Another option is to have the code make a call for information from the GUNDAM at set intervals and translate this information into additions to the map in progress. The robot could store information on its own and, when called, send any new information it has gathered concerning the outlay of the maze. The robot will determine whether to make a turn by measuring the distance from the walls using the ultrasonic sensors attached to its sides and front. For example, if the distance to the right suddenly becomes larger than a predetermined length, the GUNDAM will take note that there is a turn on its right and, depending on the method employed for navigating the maze, either turn into this path or continue forward.

Perhaps the portion of solving the maze that may present issues is actually entering the maze itself. This process will have its own function to determine what type of opening the maze has and what needs to be drawn on the software's side. If the maze opening simply consists of a straight path, the GUNDAM will send the following string "0 1 0." This lets the software on the PC side know that there is only one path to follow. The two zeroes represent the left and right path, which are closed off or non-existent. If there is only a left, the robot will send a "1 0 0." A sole right path has the signal "0 0 1" and all three options open has the signal "1 1 1." The robot will always follow the left wall as its initial method of maze solving. If straight is the only option, the robot will continue straight until it reaches its first left turn. If right is the only option, the robot will turn right and then proceed to follow the left wall. If left is the only option, the robot will turn left and follow the left wall. If presented with two options in which a left turn is an option, a left will always be chosen. If presented with two options without a left turn, the robot will choose to move straight. If the robot finds that all three options are available, a left turn will be chosen.

Another part of the software is actually switching between the separate maze solving algorithms. With the failure of wall following, Tremaux's algorithm will be entered. This will result in the robot no longer holding the task of running through an algorithm. This is due to the heavy memory load running Tremaux can have on the limited memory the robot has. Instead, the PC will take the role away from the GUNDAM and print out instructions for it to follow. This will include storing nodes with their locations and neighbors, storing paths taken and the nodes between the paths, and storing the amount of times certain paths have been traversed.

The instructions printed during this phase of maze solving will involve printing out a simple string of the direction the GUNDAM will need to move in. This will all be relative to the heading of the robot, not actual north, south, east, west directions. If the next node lies behind the robot's current heading, a string reading "behind" will be printed out. This sting will feed into a method with the sole purpose of rotating the GUNDAM 180 degrees. A reading of left will cause the GUNDAM to rotate 270 (or -90) degrees. A reading of right will cause the robot to rotate 90 degrees. With forward, the GUNDAM will simply retain its current heading. After reading this set of instructions, the robot will proceed to move forward until it reaches a new node. The location of this node will already be known to the PC side of the software. The robot simply needs to make its arrival at the NODE by printing out a NEXT to the text file.

5.2 MAZE CREATION DESIGN

Another function of the software is to allow the robot to find all paths within the maze, or at least a large portion of them. This has the unique situation of having no exact destination in mind. Rather, the maze will be charged with the task of finding any pathways within the maze that were not encountered during the task of solving maze. While this may not be important in maze solving, this feature would be a plus for more of an exploratory robot.

The method of exploring unreached paths would rely on the nodes. Each node contains information on its neighbors, but as an added piece of information, each node will recall whether a path it lies on the opening of has been traversed. Upon selection of this task, the nodes will be iterated through to perform a check on all paths that it has access to. If an unexplored path has been found, a set of instructions to reach this node will be computed and passed through to the robot through a set of text file changes explained on the communication design section of this report.

Once the GUNDAM has reached this node, it will chose at random a path to take (if there is more than one unexplored path from this node) and make its way down the nodes similar to a search tree. To elaborate, the GUNDAM will reach the first node down this unexplored path and the node will take note of all of its potential paths to traverse. Once the GUNDAM has reached an end to its exploration due to a dead end, it will move to the most recent node prior to the dead end with paths yet to be explored. It will continue this until the most recent node is the node that originally brought it down on its path of exploration. It will then move to another node with unexplored paths that were encountered during its maze solving.

5.3 PATH CREATION DESIGN

The final piece of the software is choosing a path for the GUNDAM to take. There will be two ways to relay this information. The user can either draw out a path from the GUNDAM’s current position to a final destination or they can choose a destination on a piece of the already solved maze and work out how the robot can make its way to that point in the maze. Both of these methods rely on the map not only being drawn but positioning nodes at important points on the map. This will be how the software will actually work out the path to the maze. Whether or not the path must be the shortest path or just a path is still to be determined, but the shortest path is the desired result of choosing a destination.

The nodes for the maze will be dropped at turns and forks in the road. There will be no need to position nodes in the middle of long straight ways as this provides no useful information on forming the path for the GUNDAM to take. Each node will contain information on all of its neighbor nodes' positions. This information will include the distance from the node currently being analyzed. While running the actual search through the nodes to form the path, an A* search algorithm will be used to find the final path.

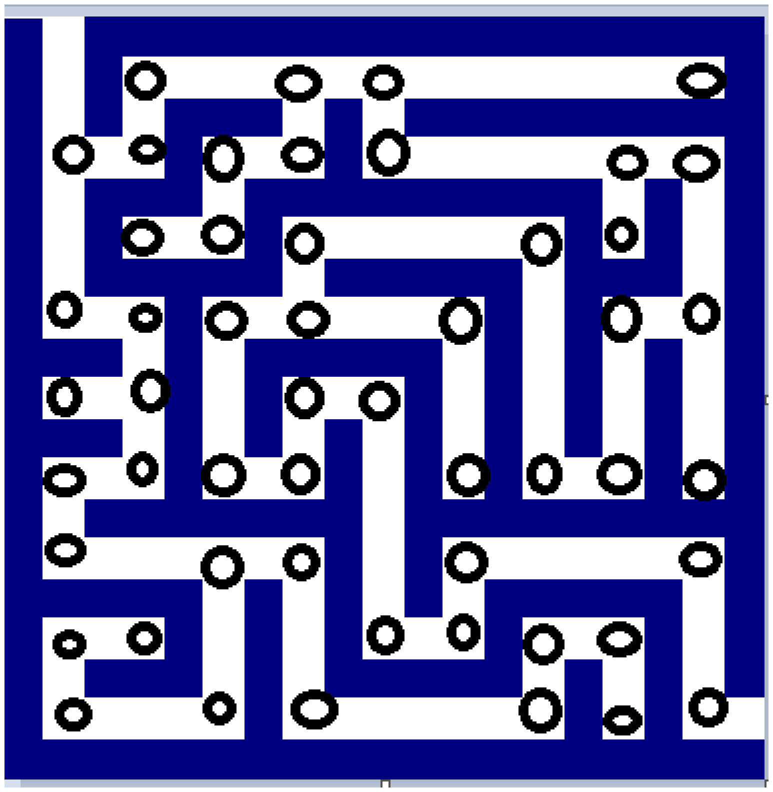


Figure 4: Map with nodes

A* provides a method of finding the shortest path while preventing a problem of searching far too many nodes to reach this solution. While this would not be a problem for smaller maps and mazes, the presence of many turns and forks may lead to an unnecessarily long search phase before the GUNDAM can actually find a path.

This path is fond by taking into account both the distance of the node from its neighbors as well as the distance from the neighbor node in question from the destination node. There are both recursive and loop methods for implementing this function. Since recursion leaves room for a lot of possible errors and mistakes hard to catch, the while loop method will be used to implement the A* search algorithm.

The information given back to the GUNDAM will be nodes needed to be traveled to as well as distances to travel before they will reach a particular node. The GUNDAM can also be aware of reaching a node by simply the presence of a turn or fork in the road as these are the only locations on the map that will have nodes. This is also another reason for the lack of nodes on straight ways. An inclusion of nodes in this area can result in putting an unnecessary load on the A^* search. To solve the issue of the user possible picking a point on a straight way nowhere near any official nodes, a temporary node will be created to allow for a search to that particular location.

There is an issue of selecting a location that is impossible to be reached from any point in the maze. The problem is how this can possibly be known by the software that this is the case. A possibility and desired solution is to verify that the position is either sufficiently close to a node or lies on the path from one node to another. If neither of these are the case, it is safe to assume that this location does not lie anywhere on a path in the maze. If the user chooses a point that is just on the outskirts of the maze, the software will simply move the robot to the point closest to that position. If the point lies far from any path or node, it is unreachable.

If the user has laid out a path for the GUNDAM to take, A^* is unnecessary. Instead, we must find nodes on the path laid out by the user. This provides a cleaner path for the GUNDAM to take. Of course, we could also simply allow the user to individually choose nodes to form their path.

All of these methods have one factor that can bring about issues: loops. A loop within the maze can cause confusion during the maze solving algorithm and, perhaps much more detrimental, to its maze completion algorithm.

During the maze solving algorithm, a check can be performed upon reaching each node to see if it has been encountered. If it has, then a loop has been performed. If loops are allowed in the current maze solving algorithm, then we can proceed as needed. If a loop does not apply to the type of maze we are assuming it to be however, we know that a change to the algorithm being implemented must be done.

During the maze completion algorithm, the same check will be performed. A loop during this method has different flags that need to be checked from the maze solving algorithm. We must check whether or not the node encountered has a path left to check. If so, then the loop can be ignored.

5.4 COMMUNICATION DESIGN

What remains is how this information will be communicated to the robot itself. This can be solved by having the robot open a text file, similar to the software’s interaction with the robot, at certain periods. It makes the most sense to accomplish this at the creation of a node or whenever the GUNDAM is idle. While the GUNDAM is in the process of updating the part of the maze it has traversed, it will conduct a check for the presence of a desired path to take. This adds to a list of states and flags to check:

- Should I continue with the same solving algorithm?
- What new solving algorithm should I use?
- Is there a new destination of interest?
- Is the path already laid out by the user?
- If not, what is A^* discovered path?

The software in turn is checking the following flags:

- Has the maze been solved?
- Has the GUNDAM done a loop?
- Is the current solving algorithm performing as it should for the assumed maze type?

There is the problem of figuring out if the text file has been updated or not. The solution to this will be a series of messages printed out in the text file to let the program and GUNDAM know whether or not the map has been solved and if the program requires the GUNDAM to switch up its search algorithm. First, the GUNDAM will print out a message to the text file indicating that it has begun to try and solve the maze: “START.”

The program will periodically read this text file. If the phrase “START” is still the content of the text file, it knows that the GUNDAM has yet to reach some sort of crossroad or turn. A timer within the program will also begin. This will not use real time but rather update an integer upon every check of the text file.

The moment the GUNDAM encounters a fork or turn, the text file will be updated with the information concerning how far it has traveled as well as a LOG along with its corresponding number (“LOG 1” for the first entry and so on). If the LOG text is still present, then it knows the program has yet to read its new entry. If it is not present and is instead replaced by a predetermined text input, it knows it is safe to remove the current text and update its new movements. In the case of LOG still being present, the GUNDAM will instead update the LOG with information on its new turn while keeping the information the program has yet to read. The end of these texts files will be “END” to let the program know it has reached the end of the current log.

The presence of LOG along with a number lets the program know that there is a new entry to read. Once the program interprets this information and stores them in variables and objects to redraw the map, the program updates the text file to read “CONTINUE.” This is a notification to let the GUNDAM know it has read all the required information from the previous log.

When the GUDNAM has finished its run through the maze, it will update the text file with the LOG message as well. The difference between this log from other logs is the fact that the end of this file will read “SOLVED” instead of “END.” This lets the program know that the GUNDAM is updating the text file not because of the presence of a turn or fork, but because it has found the exit of the maze. From this point, the GUNDAM will continue to read from a separate text file at periodic intervals to know whether the user of the program wishes to guide them through a pre-determined path through the maze.

There is the question of what exactly each line of text following the LOG will take in actual implementation. Each line will be in the following format:

LOG <distance traveled> <sensors activated>

The distance traveled will be calculated based on testing done to determine how far certain pulses cause the robot to travel. Using this information, a good estimate on how far the robot has traveled before coming upon a turn will be printed inside of the distance traveled location. The sensors will have their own I/O ports on the robot itself. This makes it easy to determine which sensors are reporting a distance much farther than the path width has been set to be. An activated sensor represents a path or turn that has been found by the sensor. If the sensor in front of the robot is also activated, this means that a path forward is also present. This will be important to note as this means that a dead end will not be drawn on the map on the PC side software falsely.

For a better understanding of how this works, we will take the hypothetical of the robot traveling down a path and coming upon a dead end with a turn on the right and dead ends on the left and straight ahead. Upon having a signal on its right sensor (an opening found on the right), the GUNDAM will first calculate how far it has traveled based on the pulses that have been sent to its servos. The method for determining this will be found through testing. For this example, we will say the robot has traveled a total of five feet before reaching this point in the maze. The input from all three sensors will then be taken to determine which sensors have found openings and which have found walls. The following message will be printed into the created for solving the maze:

LOG 5 0 0 1

The LOG lets the software know that a new path has been found. The 5 represents the distance traveled before taking on the new path or continuing on its current trajectory. The first two zeroes represent walls found by the left and forward facing sensors. The 1 represent the path found on the right. With the software’s printing of CONTINUE, the GUNDAM will follow the current solving algorithm.

If the initial left wall follower method happens to not be the proper algorithm to use due to an entrance residing within the maze, this will be detected on the PC side of the system during its drawing method. An input of “TREMAUX” will be printed onto a text file the GUNDAM is periodically checking during each turn. If this string is read, it lets the robot know that the current algorithm is not to be followed and that Tremaux's algorithm is to now be used.

Tremaux's method will utilize the software on the PC rather than software built into the robot. The robot will follow directions printed to the text file by the PC. These instructions will simply be a direction to move from the current position. The PC side software will remember situations such as number of times a certain path has been traversed (0, 1, or 2 times) and which nodes have paths yet to be used. An issue with this switching method is that it will be necessary to return to the beginning of our maze. This is due to the fact that many of the maze solving algorithms out there assume that the solver is beginning the method in question from entrance into the maze. If this is not the case, then there is no guarantee that the method of choice will yield expected results.

While the GUNDAM is traversing the maze, the user will not be allowed to request a path for it to take. This is to make the function simpler as well as for logical reasons. Solving the maze provides the minimum surface area for possible paths and destinations.that the user can choose from. There is the question, however, of how exactly this path formation text file will be formatted.

One possibility is to simply lay out the turns and distances for the GUNDAM to take in a simple long file. The problem with this approach is that we may have to end up storing this data somehow in the GUNDAM itself. We may already be competing for space on the memory of the robot itself, and the opportunity to not have to put any memory strain on it is always a plus.

Alternatively, we can use the same method but instead of having it recall the entire path, we have some way to have it remember what it has and hasn't read. One set of directions can be read and acted upon. Once the instructions have been followed, the GUNDAM will ignore the lines of text it has read and work on the next instruction. This can be accomplished by having the GUNDAM insert a single line of text after each instruction. This would require the GUNDAM to also either remove the prior lines it has inserted to avoid confusion or remember how many lines it has inserted. To be even simpler still, we can simply recall how many lines have been read on the GUNDAM's side and move down that many lines in the text file itself, updating this number with every read line.

A final possibility is to update the text file on the programs side. Each instruction will be printed out one line at a time into the text file. When the robot has read one instruction, it will replace the text in the file with a "NEXT." If a "NEXT" is read by the program, it knows its time to print out the next instruction. If the GUNDAM reads its own "NEXT" message when it comes to a fork or turn, it will remain idle and periodically check the file again for its next instruction.

When the program is done printing its final instruction, it will continue to read the file for a "NEXT" message. This will let the program know the robot has read its final instruction. It will then update the file to read "DONE." This lets the robot know that there are no more instructions to be read and as a result, it has finished the path desired of it. The check for "NEXT" before the "DONE" message is to prevent the program from changing the information in the text file before the robot can read its final destination. This would cause it to falsely perceive that it has reached its destination.

This last method will be the method of choice for our GUNDAM and program path communication. It provides a familiar interface that mimics the method of solving the maze, only in a reverse and modified way. The other designs are mentioned as they are fall backs to use if this method fails to work efficiently or at all.

5.5 INTERFACING RF TRANSCIEVER with MCU

All the modulation and demodulation schemes explored are able to accomplish the task of sending and receiving data between the G.U.N.D.A.M. and a computer, however; some schemes provide better features for our application. Frequency and phase modulation provide the ability to transmit a signal with minimal noise and inference, which is ideal in our environment. Phase modulation, in contrast with frequency modulation, in particular provides a way to more easily detect if the signal is lost and give the ability to retransmit any data that is lost. Phase shift keying is a digital modulation scheme that is ideal for the G.U.N.D.A.M.'s communication subsystem not only for the reasons previously discussed, but because the data transmitted will be binary data. However, even with the increase spectral efficiency and data rate the frequency modulation techniques over amplitude modulation techniques have higher power consumption.

To implement this design an RF transceiver of low-cost and low-power consumption will be used on the G.U.N.D.A.M. and the computer communication device. This transceiver will be the low power IC chip from Texas Instruments, CC1101, that has the ability to modulate and demodulate a binary signal using either 2-FSK, 4-FSK, GFSK, and MSK. This RF transceiver is very versatile and has many programmable features, such as frequency, power output, channel filter bandwidth, and Carrier Sense indictor among many others. It also has a high sensitivity of -116dBm at 0.6kbuad and operating at 433MHz, which only results in a 1% packet error rate. The CC1101 RF transceiver module has very low power consumption at 14.7mA in receive mode with 1.2kbaud and operating at 868MHz. However, this can be even further reduced with the use of the TPS62730 step down converter connected between the transceiver and the power supply. The following figure shows the input and output voltage of the step down converter.

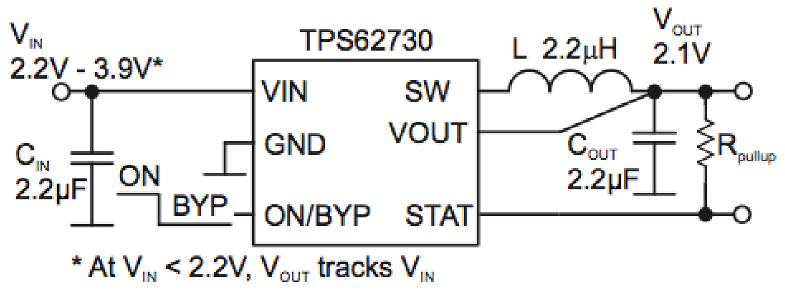


Figure 5: Step Down Converter schematic

This device will be a middleman between the power supply and the RF module, the input voltage V_{IN} will come from the power supply and the output voltage V_{OUT} will be the input voltage of the CC1101 module. Connecting the CC1101 directly to a 3.6V power supply would draw 34mA of current at maximum transmitting power output, however; with the TPS62730 output at 2.1V and a 3.6V power supply the current drawn would fall to 22mA. The CC1101 must have external RF components to operate; the following figure shows the components needed to operate at 315/433MHz.

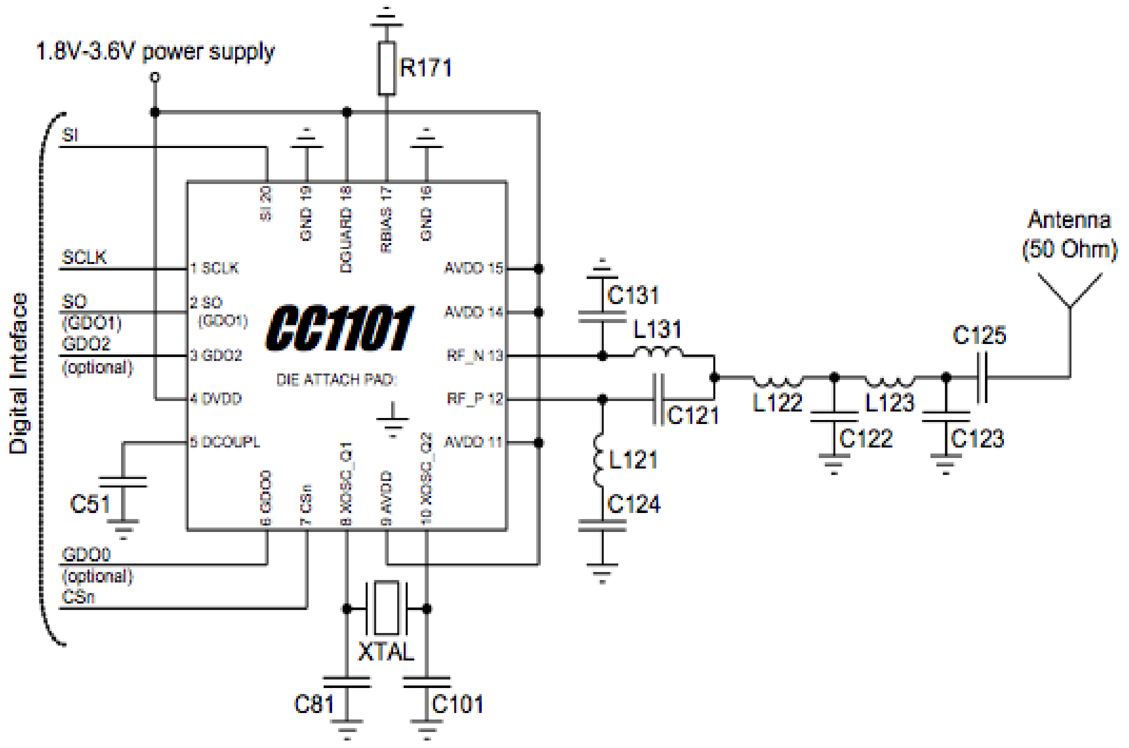


Figure 6: Schematic of CC1101 configured for 315/433MHz

The RF transceiver module will transmit and receive at different frequencies to increase efficiency and decrease interference during communication between the G.UN.D.A.M.'s module and the computer's. In the figure above the MCU will be connected to the RF transceiver through the pins on Digital Interface illustrated. The CC1101 is configured with this SPI interface, which is also used to transmit and receive data. Meaning the MCU will be in charge of programming the different modes available on the CC1101 and transmitting and receiving data through the same interface. The configuration registers on the CC1101 are located on the SPI addresses from 0x00 to 0x2E, which must be configured and verified by the MCU during start up phase before any transmission is done. The RF module has two 64-byte FIFO buffers for transmitting and receiving data. Depending on how the R/W bit is set when interfacing through the SPI, the FIFO can be accessed through a single address. As stated before the MCU will use four I/O pins for the SPI configuration interface, namely SI, SO, SCLK, and CSn. The CC1101 also has three pins GDO0, GDO1, which is shared with SO, and GDO2 that can output the internal status information useful for controlling the module. The MCU can be programmed to generate interrupts based on the status pins' output. One of the most useful programmable features on the CC1101 is the data rate, which can be configured through the configuration registers mentioned above.

The CC1101 has on built-in hardware support for packet orient radio protocols, which can reduce the amount of software needed on the MCU to send and receive data. Since packet handling is done by the hardware all the MCU needs to do is read the bytes transmitted from the FIFO and write bytes to the FIFO for transmission. The module also supports constant packet length and variable length protocols but for our application a constant packet length will simplify the code and reduce the chances of errors, which will in turn make debugging much simpler. The packet handling on the MCU is reduced to the number of bytes that can be read from or written to the transmitting and receiving buffers. The configuration pins can be used to determine when a packet has been received or transmitted and there is a configuration register that can be used to determine how many bytes are in the transmitting and receiving buffer. This feature makes it useful for the MCU to determine when the transmitting buffer is overflowing and the receiving buffer is under flowing, both of which will cause a R/W error that can be handled through interrupts. All of these can be set as interrupts on the MCU to reduce processing time.

An additional programmable feature on the CC1101 is the modulation scheme used. Since the CC1101 supports amplitude, frequency, and phase shit modulation all that is required is choosing a modulation scheme that fits our application best. As discussed before phase shift keying is the best modulation scheme that meets all our application needs and the CC1101 provides MSK, which is identical to offset QPSK, the original consideration when designing the G.U.N.D.A.M.'s communication subsystem. The CC1101 has a received signal strength indicator that can be utilized during the testing phase to determine the range the G.U.N.D.A.M. can operate from the computer and still have reliable data transmission. Yet another important programmable feature is the output power of the RF transceiver. The output power chosen will determine the typical current consumption, which will affect the range. During the testing phase an output power can be chosen based on a reliable effective range determined to be sufficient.

There is a possibility of a problem arising with using a frequency range of 315/433MHz for transmitting and receiving data, that is there can be other communication systems operating at these frequencies in our testing environment. Thus to reduce interference a lower frequency band which is not as commonly used by most modern wireless systems in our likely testing environment was chosen. In addition, FHSS will be used to make the G.U.N.D.A.M.'s communication subsystem more robust with respect to inference from other devices operating at the same frequencies. There are a number of configuration registers used for calibration on the CC1101 that can be read by the MCU to implement a proper frequency hopping function.

Another component to be considered in the process to optimize RF performance is a good antenna design. A whip antenna was determined to be ideal for our application because it requires no additional costs in PCB design, which makes them generally cheap to acquire. The whip antenna will enable the robot range to be maximized when the application doesn't have any strict size limitations, which is true for the G.U.N.D.A.M.'s communication subsystem. The length of the antenna depends on the frequency the RF transceiver will operate at. The CC1101 can operate in a large band of frequencies, however; a lower frequency band will provide an even longer range and less interference from modern communication devices, such as cellphones or those that employ Wi-Fi and Bluetooth, some of which are likely to be present in the G.U.N.D.A.M.'s testing environment. The frequency band chosen for the RF transceiver is 315/433MHz, meaning that this RF transceiver will require an antenna that is a quarter of the smallest wavelength utilized to keep a reasonable size and performance, which turns out to be approximately 23.8cm.

5.6 WIRELESS PROTOCOL DESIGN

In addition to finding a digital modulation scheme and an RF transceiver that can transmit bits of data efficiently through the air with minimal loss, the G.U.N.D.A.M. will require a communication protocol to control the way in which information is transmitted back and forth. A frequency that is out of the range of the most widely used protocols in use today, namely 802.11 and 802.15, must be chosen to reduce interference between the robot and the computer. As previously discussed an operating frequency range of

315/433MHz would reduce interference from most devices in the G.U.N.D.A.M.'s testing environment. Since no other users will be communicating besides the robot and the computer, there is no need to implement multiple access or collision avoidance techniques. However, to make the communication subsystem more robust a FHSS technique will be implemented by the MCU to reduce interference.

The main features required from the protocol is the ability to sense the presence of another transmitting device and send acknowledgements so the device goes into listening mode. A secondary but important feature to the protocol will be an implementation of the AES-128 encryption algorithm. This encryption algorithm has gained popularity due to its susceptibility to brute-force and other attacks. The implementation of AES-128 encryption algorithm will require the sender to encrypt the data using a public key and the receiver to decrypt the data using a private key. Both the G.U.N.D.A.M. and the computer device will share a public key to send data to each other but each will have their own private key to decrypt the data they receive.

The first requirement of the wireless protocol, which will be implemented by the MCU, will be to handle transmission between two nodes in a manner that they don't talk over each other, in other words, neither of the two nodes must transmit or receive data at the same time. To accomplish this there will be a routine in the MCU that will periodically check whether the device is transmitting or receiving data. This will be important in determining the next step because initially both the G.U.N.D.A.M. and the computer module will be in listening/receive mode that will wait for new transmission. Data transmission will begin when the G.U.N.D.A.M. has information it needs to transmit to the computer. In this case, the G.U.N.D.A.M.'s MCU will encode the data in AES-128 and pass it to the RF transceiver to be sent to the computer.

The computer will initially be in listening mode to be able to receive the encoded information, which it will then decode and pass it on to the computer. In this application, the G.U.N.D.A.M. will be the only device sending data and the computer will only be receiving data. However, the design allows for two-way communication between the G.U.N.D.A.M. and the computer for scalability purposes. Although the G.U.N.D.A.M. and the computer dongle will only implement one-way transmission they will each use acknowledgements to ensure that data was received without errors and no retransmission is needed. To accomplish this error-checking test, the G.U.N.D.A.M.'s RF transceiver will need to switch to listening mode to receive a packet of acknowledgement from the computer dongle. Similarly, the computer dongle will switch to transmitting mode for a short period of time to send the packet acknowledgement. This coordination will be accomplished by routines in the MCU and a timer to ensure that there is no collision or loss of data.

The second requirement of the wireless protocol will involve the MCU having the capability to also handle the encryption of data before it is transmitted over-the-air to either the G.U.N.D.A.M. or the computer. To reduce latency and errors, the acknowledgement packets will not be encrypted since they don't contain any relevant data to what the G.U.N.D.A.M. is communicating to the computer. A separate routine on the G.U.N.D.A.M. and computer's MCU will handle which packets will be encrypted and which won't. For the packets that are encrypted the MCUs will have an encryption and decryption routine. The function will take plain text data from the main microcontroller on the G.U.N.D.A.M. or the computer on the receiving dongle and encrypt it using the AES-128 encryption technique. To encrypt the plain text the MCU on the G.U.N.D.A.M. will use a pre-programmed key to encrypt the data and the MCU on the computer dongle will use the same pre-programmed key to decrypt the data and send it to the computer so the GUI interface on the computer can use the data to draw out the maze. Although not needed, the G.U.N.D.A.M. and the computer dongle will both have the capability of encrypting and decrypting plaintext for scalability reasons.

5.7 COMMUNICATION SUBSYSTEM

The design of the G.U.N.D.A.M.'s communication subsystem will be centered on a microcontroller that can process the signals transmitted from the computer, which requires demodulation and decryption. In addition, to transmit to the computer the microcontroller will be required to encrypt the data, which will be the path taken or other information about the maze, from the main microcontroller and modulate for transmission. The microcontroller in charge of taking the bits received from the transceiver will be an MSP430G2xx. Since the MSP430G2xx has the ability to convert analog-to-digital signals a simple AM or FM transceiver can be used for transmissions, but the microcontroller can also accept digital input so using a digital RF transceiver via an SPI interface, which will carry an advantage in reducing the amount of processing the MSP430 has to do.

The microcontroller will also be in charge of implementing a simple wireless protocol including AES-128 encryption to communicate with the computer and visa versa. To accomplish this, the code written on the microcontroller will be split into two main parts. One for the purpose of taking data from the main microcontroller in charge of controlling the robot itself and transmitting that data to the computer. In the process of transmitting that data the microcontroller will encrypt it using AES-128 encryption algorithm. The second for the purpose of establishing an initial connection and manage this connection with the device on the computer.

The MSP430G2553 microcontroller has the capability of communicating with several devices simultaneously through the use of two-channel serial communication interface, namely USCI_A and USCI_B blocks. The figure below shows the pin layout for the MSP430G2553 microcontroller.

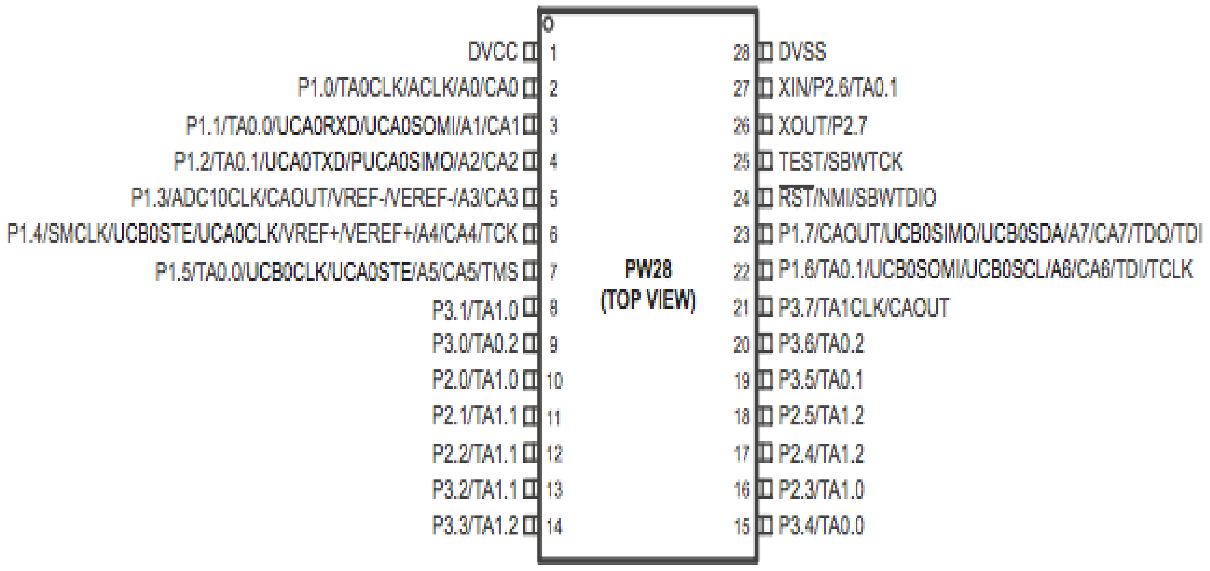

DVCC 1
P1.0/TA0CLK/ACLK/A0/CA0 2
P1.1/TA0.0/UCA0RXD/UCA0SOMI/A1/CA1 3
P1.2/TA0.1/UCA0TXD/PUCA0SIMO/A2/CA2 4
P1.3/ADC10CLK/CAOUT/VREF-/VEREF-/A3/CA3 5
P1.4/SMCLK/UCB0STE/UCA0CLK/VREF+/VEREF+/A4/CA4/TCK 6
P1.5/TA0.0/UCB0CLK/UCA0STE/A5/CA5/TMS 7
P3.1/TA1.0 8
P3.0/TA0.2 9
P2.0/TA1.0 10
P2.1/TA1.1 11
P2.2/TA1.1 12
P3.2/TA1.1 13
P3.3/TA1.2 14
PW28
(TOP VIEW)
28 DVSS
27 XIN/P2.6/TA0.1
26 XOUT/P2.7
25 TEST/SBWTCK
24 RST/NMI/SBWTDIO
23 P1.7/CAOUT/UCB0SIMO/UCB0SDA/A7/CA7/TDO/TDI
22 P1.6/TA0.1/UCB0SOMI/UCB0SCL/A6/CA6/TDI/TCLK
21 P3.7/TA1CLK/CAOUT
20 P3.6/TA0.2
19 P3.5/TA0.1
18 P2.5/TA1.2
17 P2.4/TA1.2
16 P2.3/TA1.0
15 P3.4/TA0.0

Figure 7: MSP430 pin layout.

The CC1101 module will connect the pins labeled Digital Interface in the figure below to the MCU’s USCI_A channel, which are pins P1.1, P1.2, P1.4, and P1.5. In addition, the MSP430 microcontroller will be connected to the robot’s main microcontroller, which in charge of the sensors’ input and output, through an I^{2}C interface. This connection will be accomplished through the MSP430’s USCI_B channel, which are pins P1.4, P1.5, P1.6, P1.7. On the G.U.N.D.A.M., the MCU has to be able to communicate with the RF transceiver through an SPI interface and simultaneously send that data through an I^{2}C interface that will be connected with the main microcontroller. The MCU on the computer dongle has to be able to communicate with the RF transceiver through an SPI interface and simultaneously send that data to the computer through an UART interface. Luckily, the MSP430 MCU has the capability to meet both of these application needs.

The device that will be on the computer has the same functions as the one on the G.U.N.D.A.M., however; the demodulated and decrypted data will be sent to the computer through a USB interface. The MSP430 has a serial interface to communicate with a computer but this would not work with all computers since many don't have serial ports and it would be ideal to be able to communicate with any computer that has a USB port and can run the software. To combat this problem a CP2101 chip will be used to convert UART to USB, which will allow the MSP430 to send and receive data from the computer. The CP2101, manufactured by Silicon Labs, is a single chip USB to UART Bridge. The computer will recognize this device as a virtual COM port, which in turn can be used by the GUI running on the computer to draw out the map of the maze. The bridge in turn will be connected through TXD and RXD pins, illustrated bellow, to an MSP430's UART interface, which is the USCI_A channel. The following figure shows the bridge pin layout.

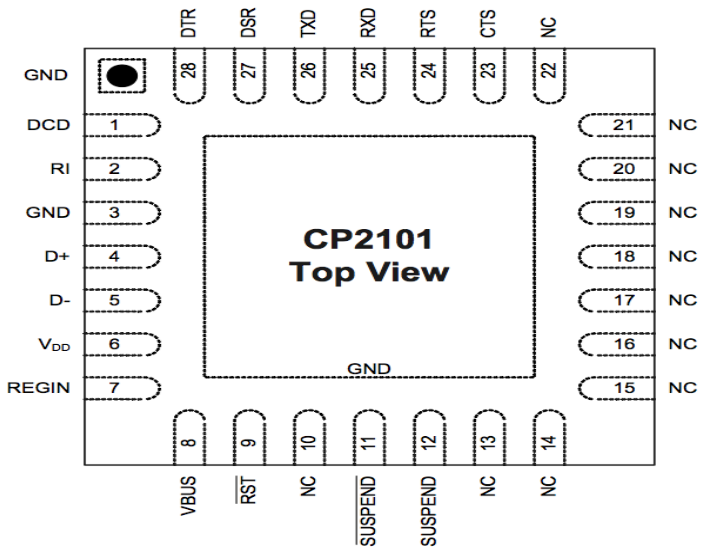


Figure 8: CP2101 pin layout

The CP2101 has baud rates from 300bps to 921.6kbps so it can handle the transmission speed on the RF transceiver and MCU. It also has a 512 byte transmit and receive buffer, which is over the 64-byte TX and RX FIFO of the RF transceiver, although the MCU will be the middleman between the RF transceiver and the CP2101.

5.8 RANGE FINDER SYSTEM

The G.U.N.D.A.M. will have two ultrasonic sensors on the front and back of the robot to detect longer distances in the narrow paths of the maze. It will also have two infrared sensors on the sides to detect smaller distances. Most ultrasonic sensors have a minimum distance of 2cm and infrared sensors have a minimum distance of 4cm. To combat inaccuracies in distance measurements the robot's main microcontroller will be required to keep the robot centered within the minimum distances that the sensors can measure. Infrared sensors have simple circuits but accurate infrared sensors are difficult to manufacture without the proper tools. Below is a figure of the Sharp GP2D120 infrared range finder functioning on a flat surface obstacle.

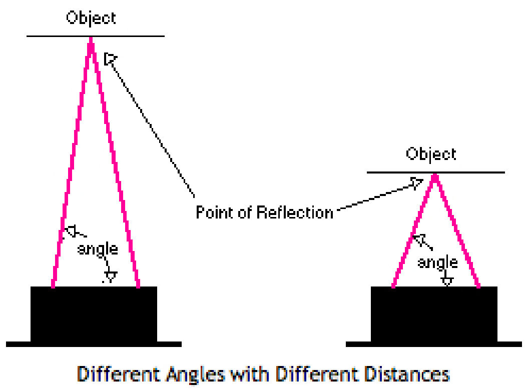


Figure 9: Function of Sharp GP2D120 infrared range finder.

This particular IR range finder has a minimum and maximum detection range from 4cm to 30cm. It has an operating voltage and current of 4.5V to 5.5V, 33 to 50mA. The response time is 30ms with an analog output. The voltages can simply be converted to a useful distance for the microcontroller using the chart below.

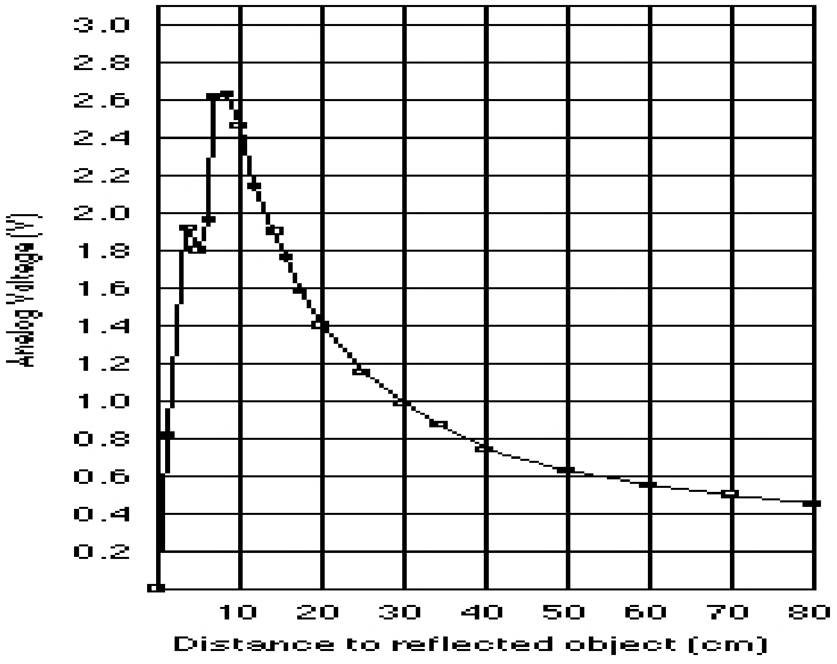


Figure 10: Voltage output vs. Distance of GP2D120 IR range finder.

Ultrasonic sensors on the other hand are less susceptible to inaccurate measurements due to the wider emitting beam. Building an ultrasonic sensor with individual parts and a circuit schematic can prove to be more cost efficient than purchasing a pre-built sensor. Since ultrasonic sensor output will be an analog signal there will have to be calculations done to determine the distance measurements based on the frequency of the transmitted beam and the time the beam takes to reflect back and be detected. Below is circuit schematic of an ultrasonic distance measurement circuit, however; the PIC12C508 will be replaced with the main microcontroller.

The range finder sensors will be mounted on the robot chassis itself and will interface with the main microcontroller, which will take its values and store them to calculate the distances and path traversed through the maze. The two infrared sensors mounted on the sides of the robot will be as close to the ground as possible and they will be housed in a shield to prevent interference from the indoor lighting at the receiving end of the sensor. The ultrasonic sensors on the front and back of the G.U.N.D.A.M. will be mounted further from the ground to reduce the possibility of the sound absorption from a carpet floor that could possibly be a part of the G.U.N.D.A.M.'s testing environment. Both the ultrasonic and infrared sensor will be at the midsection of the horizontal axis in the robots dimensions to increase the accuracy when calculating the position and location in the maze.

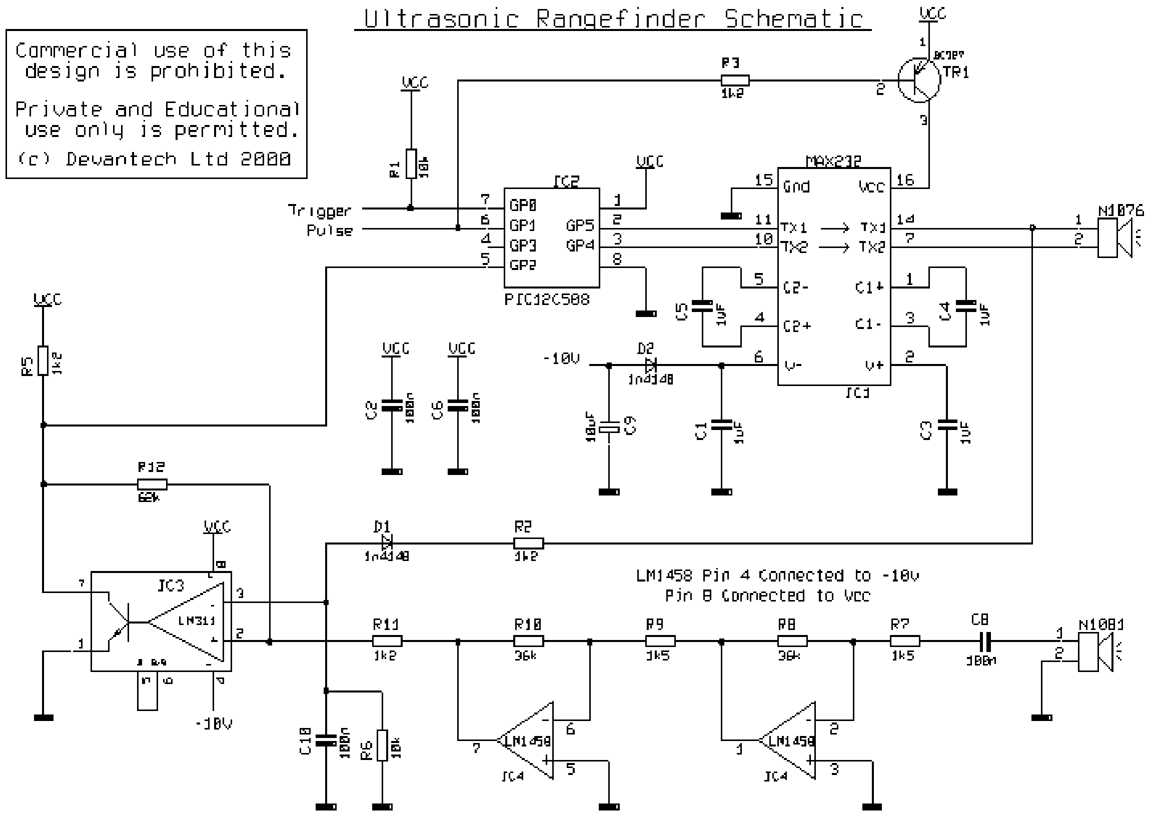

Ultrasonic Rangefinder Schematic
Commercial use of this design is prohibited.
Private and Educational use only is permitted.
(c) Devantech Ltd 2000
Trigger Pulse
PIC12C508
IC2
MAX232
IC1
LM311
IC3
LM1458
IC4
LM1458 Pin 4 Connected to -10v
Pin 8 Connected to Vcc
N1076
N1081
TR1
VCC
-10V

Figure 11: Circuit Schematic of Ultrasonic sensor

5.9 PHYSICAL MAZE DESIGN

The physical maze will be made of one of three materials, which are easy to manufacture for our application. Wood would be easy to cut and put together when it comes to the design. However, during the testing phase the wood can prove to be problematic if the sensors give inaccurate readings due to the properties of the wood and the same is true for metal and plastic. Since our robot will be a maze-solving robot, the maze has to be designed in a way that it can be taken apart and rebuilt to form a different maze. A technique that could prove useful in designing the maze is to create a number of separate blocks that each connect to each other like a jigsaw puzzle. The maze would consist of a number of separate blocks each with notches on both ends that are used to attach to another block. By using many blocks in series a simple or even complex maze can be constructed in a matter of minutes. The problem with using metal as a material is the difficulty in constructing the metal blocks, which also causes any changes required to be difficult as well. Plastic is a good alternative to wood because it is not as difficult to manipulate as metal but it can increase the cost of the overall design over wood. Ultimately, the determining factor of the material of the maze construction will be the result of the range sensors' accuracy in each environment. The following figure demonstrates the different types of joints that can be used in construction of the maze.

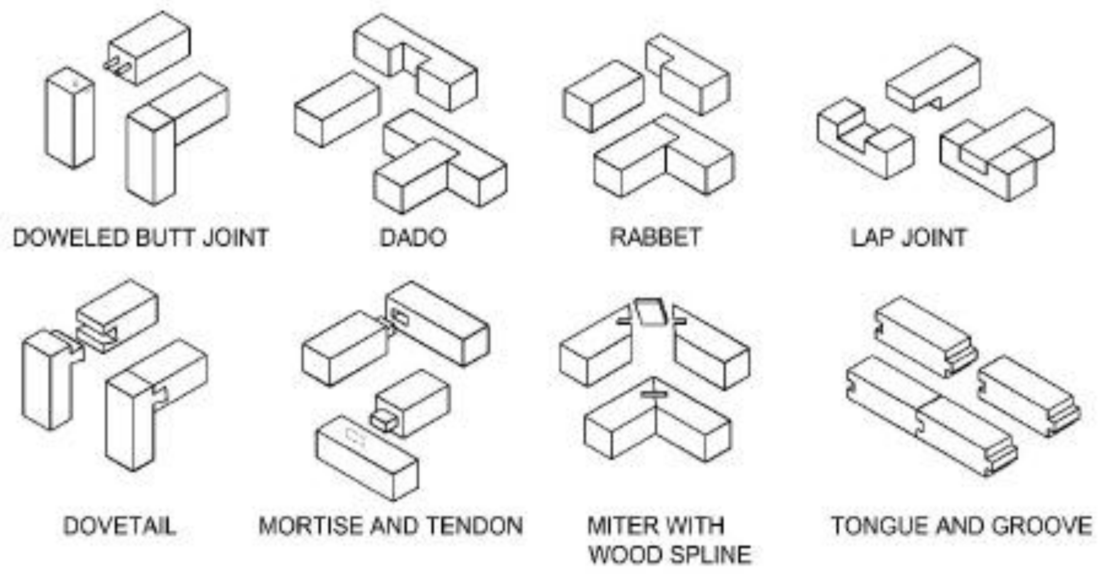


Figure 12: Wood joints used to construct maze

The best joints to use in this application are the ones that can hold the wood together by friction alone and without the use of any adhesive. Using adhesive would make the maze permanent, which is counterproductive since the testing environment has to have variability for the functions of the robot to be properly tested.

6.0 HARDWARE AND SOFTWARE DESIGN SUMMARY

The software side of our project will consist of two portions: the laptop and the chassis. The laptop side will have the responsibility of drawing out our maze, handling the implementation of the Tremaux algorithm, taking in input of the path chosen by a user, detecting a wall following fail due to a wall located on its own, and handling the task of finding all paths within our maze.

The interface will consist of a frame with multiple components. The first component is a window, which will hold the actual maze to be drawn out by our software. Options for several tasks will be presented in the form of buttons. The first option will be to solve the maze and results in both finding the exit as well as rendering all other options invalid until the maze has been solved. A second option will handle the decision to draw out a path and results in accepting input from the user to select nodes to make up a path for the robot. The third option consists of finding any and all paths within the maze itself.

The robot side software will handle the implementation of the wall follower method and detecting cases in which the wall following method will fail due to an entrance within the maze. This software also handles the inputs and outputs from the sensors as well as sending pulses to the servos for movement and turns.

DATA STRUCTURES

Node	• X location • Y location • Neighbors	• Float • Float • List(nodes)
Walls	• X location • Y location • Width • Height	• Float • Float • Float • float
GUNDAM	• Position • Heading	• Float • float
Neighbors	• X location • Y location • Visited	• Float • Float • List
Visited	• (bool) • Amount	• Bool • Int
Path	• Source Node • Destination Node	• Node • Node
Right Sensor	• Distance	• Float
Left Sensor	• Distance	• Float
Forward Sensor	• Distance	• Float
Rear Sensor	• Distance	• Float
Left Servo	• Pulse	• Struct
Right Servo	• Pulse	• Struct
Pulse	• Voltage • Time	• Float • Int

The G.U.N.D.A.M. will have four distance sensors mounted on the left, right, front, and back. The distance sensors mounted on the left and right of the robot will be infrared sensors. These sensors will be mounted closer to the ground and will have a shield to reduce interference from any light sources in the testing environment. The distance sensors to be mounted on the front and back of the robot will be ultrasonic sensors. The ultrasonic range finder will be mounted as far from the ground that the robot chassis will allow to prevent the sound pulses to be absorbed by any sound absorbing material in the testing environment, such as a carpet. During the testing phase the sensors will be tested for minimum and maximum distances. These sensors and the others will be connected to the main microcontroller through I/O interface.

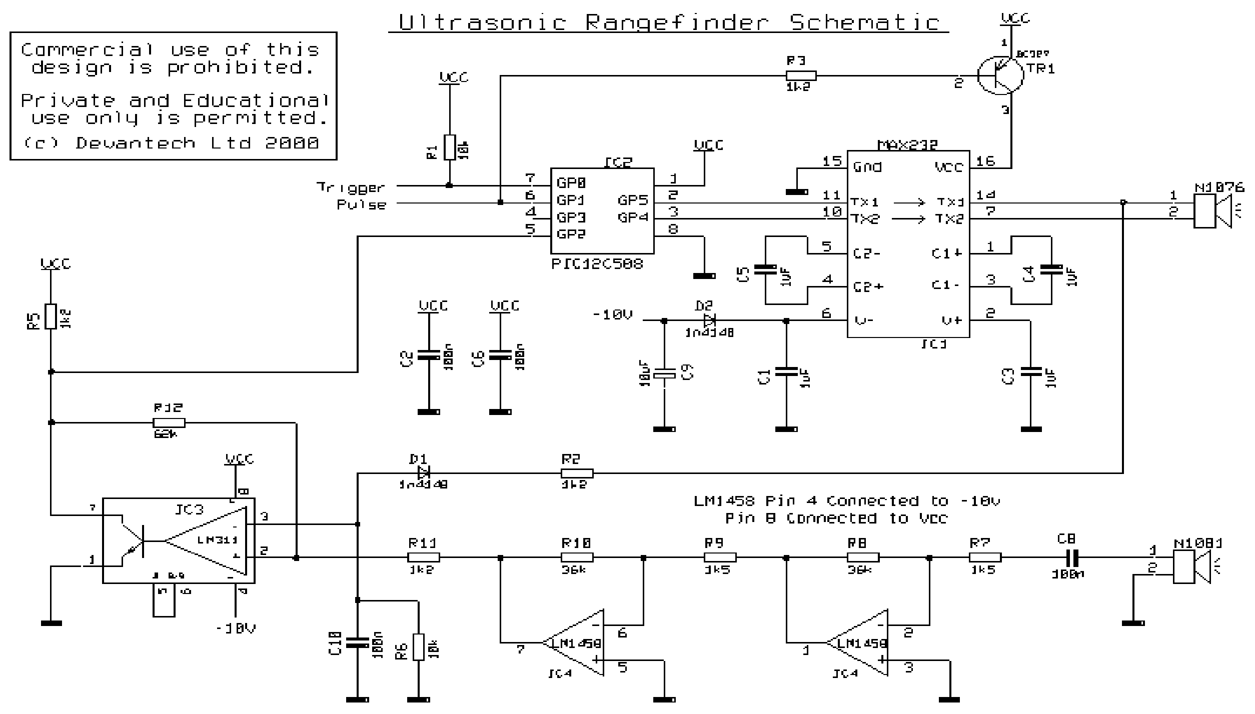

Ultrasonic Rangefinder Schematic
Commercial use of this design is prohibited.
Private and Educational use only is permitted.
(c) Devantech Ltd 2000
LM1458 Pin 4 Connected to -10v
Pin 8 Connected to Vcc
PIC12C508
MAX232
LM311
LM1458
Trigger Pulse

Figure 13: Circuit Schematic of Ultrasonic sensor

The communication between the laptop and the chassis will be accomplished through the use of wireless devices and it will consist of two parts. The first part of the wireless system will be mounted on the robot chassis and will include an MSP430 microcontroller and a CC1101 RF transceiver. The microcontroller will communicate with the RF transceiver through an SPI interface.

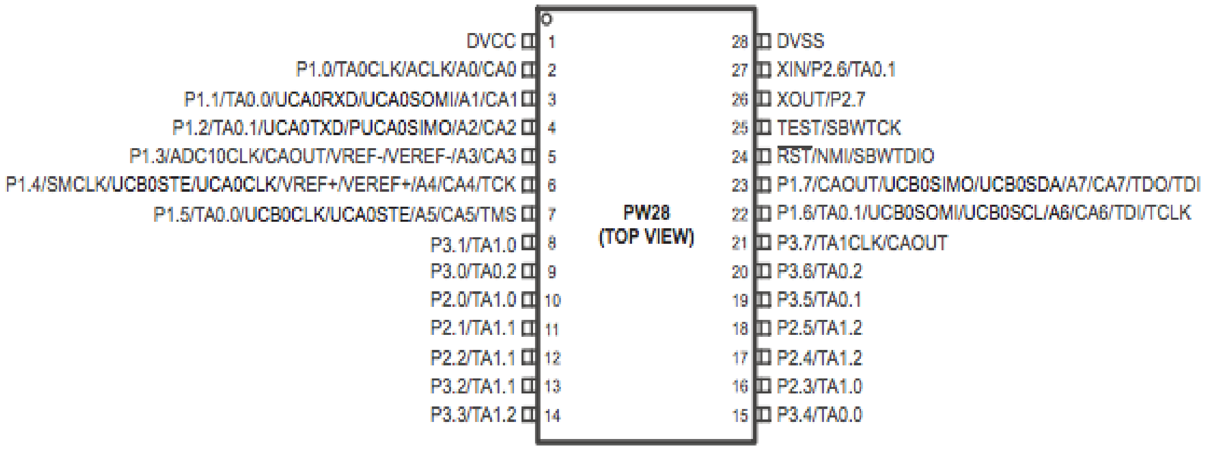

PW28
(TOP VIEW)
DVCC 1
P1.0/TA0CLK/ACLK/A0/CA0 2
P1.1/TA0.0/UCA0RXD/UCA0SOMI/A1/CA1 3
P1.2/TA0.1/UCA0TXD/PUCA0SIMO/A2/CA2 4
P1.3/ADC10CLK/CAOUT/VREF-/VEREF-/A3/CA3 5
P1.4/SMCLK/UCB0STE/UCA0CLK/VREF+/VEREF+/A4/CA4/TCK 6
P1.5/TA0.0/UCB0CLK/UCA0STE/A5/CA5/TMS 7
P3.1/TA1.0 8
P3.0/TA0.2 9
P2.0/TA1.0 10
P2.1/TA1.1 11
P2.2/TA1.1 12
P3.2/TA1.1 13
P3.3/TA1.2 14
28 DVSS
27 XIN/P2.6/TA0.1
26 XOUT/P2.7
25 TEST/SBWTCK
24 RST/NMI/SBWTDIO
23 P1.7/CAOUT/UCB0SIMO/UCB0SDA/A7/CA7/TDO/TDI
22 P1.6/TA0.1/UCB0SOMI/UCB0SCL/A6/CA6/TDI/TCLK
21 P3.7/TA1CLK/CAOUT
20 P3.6/TA0.2
19 P3.5/TA0.1
18 P2.5/TA1.2
17 P2.4/TA1.2
16 P2.3/TA1.0
15 P3.4/TA0.0

Figure 14: MSP430 pin layout.

The MSP430 has two serial communication channels that can run at the same time. The CC1101 module will connect the pins labeled Digital Interface in the figure below to the MCU's USCI_A channel, which are pins P1.1, P1.2, P1.4, and P1.5.

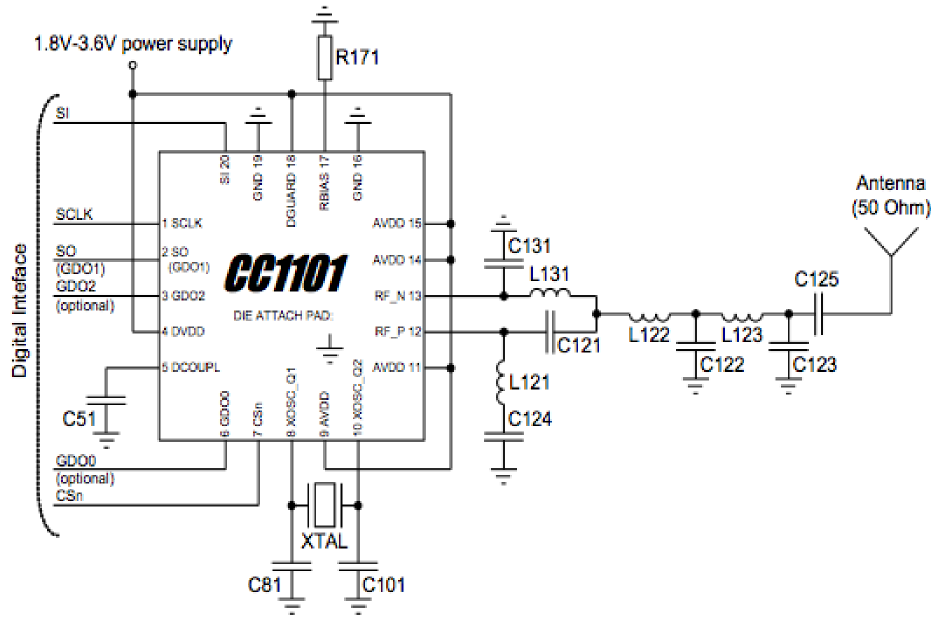


Figure 15: CC1101 RF transceiver pin layout

In addition, the MSP430 microcontroller will be connected to the robot's main microcontroller, which in charge of the sensors' input and output, through an I^2C interface. This connection will be accomplished through the MSP430's USCI_B channel, which are pins P1.4, P1.5, P1.6, P1.7. Both of these channels can be configured to support different serial connections. The figure below shows the pin layout of the TPS52730 step down converter that will be used to reduce the power consumption in the CC1101 RF transceiver. This device will be a middleman between the power supply and the RF module, the input voltage V_{IN} will come from the power supply and the output voltage V_{OUT} will be the input voltage of the CC1101 module.

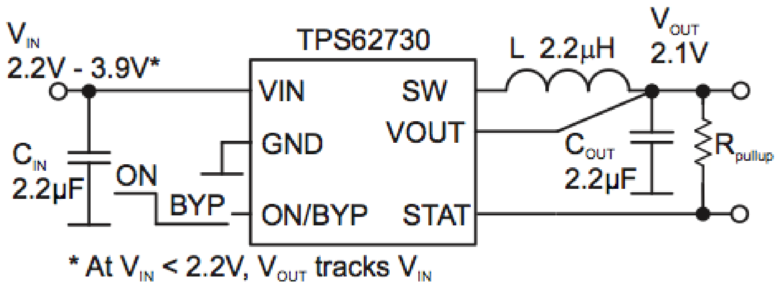


Figure 16: Step Down Converter layout

Similarly, a connection must be established between the laptop and a USB dongle. This will be accomplished with a CP2101 UART to USB Bridge. The laptop will have drivers that will create a virtual COM port to allow communication between it and the bridge. The bridge in turn will be connected through TXD and RXD pins, illustrated bellow, to an MSP430's UART interface, which is the USCI_A channel as described above.

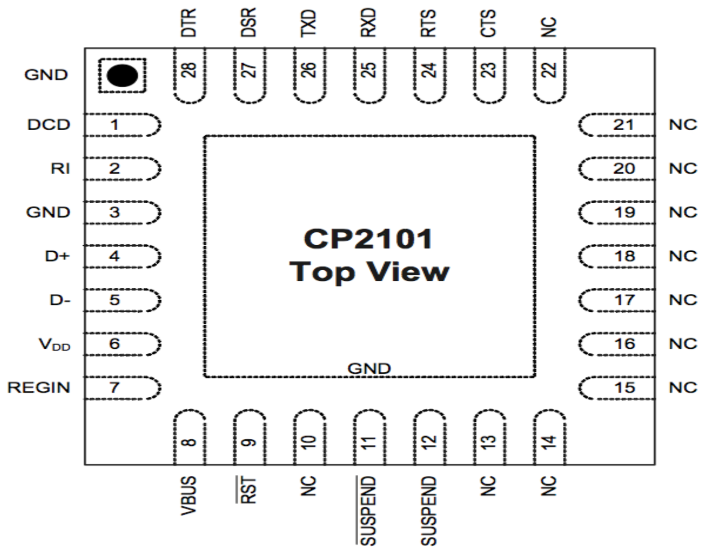
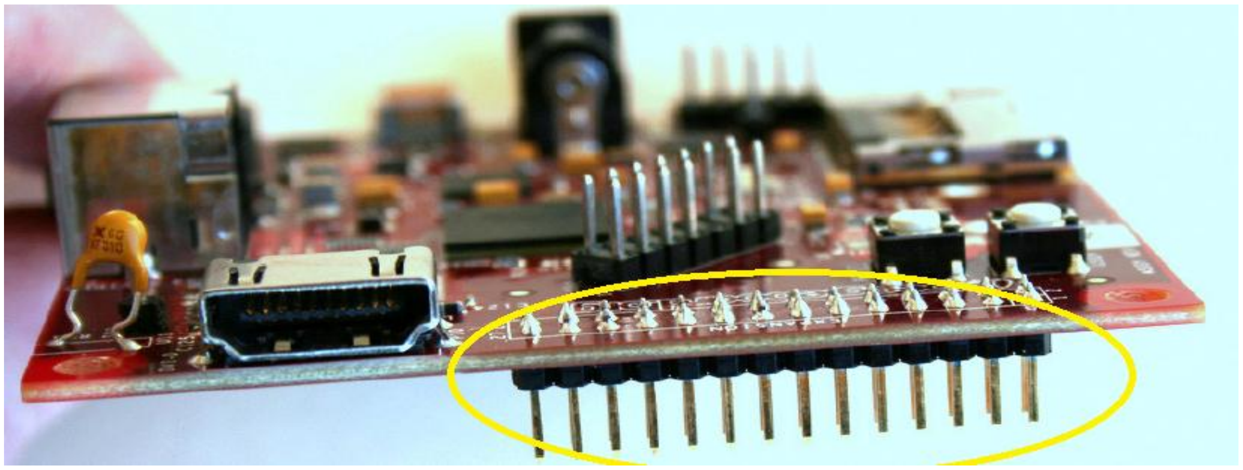


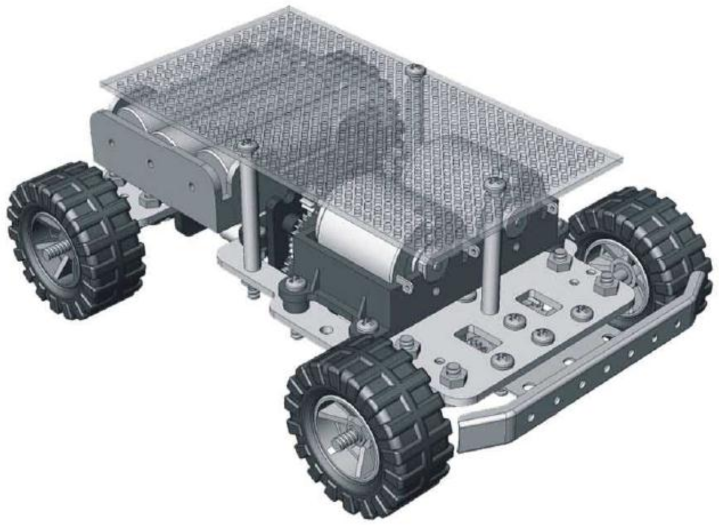
Figure 17: CP2101 pin layout

The MCU mounted on the robot's chassis will also take information from the main microcontroller that is to be relayed to the laptop and encrypt it using an AES-128 encryption technique. The information will then arrive at the MSP430 on the laptops, which will then decrypt the data using an AES-128 encryption key and relay the information to the bridge and ultimately the laptop. The laptop will take the information that will be in plaintext after decryption and use that information to draw the maze on the GUI designed to create the maze path and show the solution to the maze once the robot has solved it.

Communication between all microcontrollers will be handled through the Beagleboard with its expansion header pins.


Beagleboard Pins

All the microcontroller PCBs, sensors, and battery will be housed on the modified chassis. The battery will not be included because it is to be replaced as described in the design section.


Robot Chassis - Mr. Basic – Model TR-3

The motor-microcontroller is described in the design section, with schematic in the appendix, and will interface the motor controls with the beagleboard. It uses an Atmega8 microprocessor and a serial-to-usb UART chip – the FT232RL.

There is also a Battery Switching Voltage Regulator circuit that switches between outlet power and the battery. It’s also described in detail in the design section. Its schematic is also in the appendix.

7.1 GUNDAM MOVEMENT METHODS

The GUNDAM needs to be tested on different movement methods that it will use to perform basic movement. This simply consists of testing moving in two different directions and rotation. The function of moving forward will simply be tested by building a method to do just that. The same goes for moving backwards. There is a need to determine the proper pulses for each servo separately in order to ensure that our robot moves in as straight a path as possible. These values will be recorded at the end of forward and reverse testing.

The issue of testing rotation, however, is much more involved. A basic understanding of how many pulses or for how long pulses should be will need to be reached in order to ensure that the rotations will bring the GUNDAM on a straight path, or as straight a path as we are allowed through programming. The GUNDAM will first be put into a circle with the degrees measured out. Zero degrees will be its heading and 180 will be its direct rear. Ninety degrees represents a full turn to its direct right, while 270 (or -90) will represent its direct left.

Pulses to one servo will then be done with 5 second gaps between each. This will allow us to properly calculate how many pulses are done to achieve certain degrees of turning. The number of turns required to turn a full 90 degrees will then be saved. The same must be done for the other servo. It is important to note that both servos will not have the exact same responses to the pulses. This is due to human error and design of the servos themselves. To combat this, the same pulse test on the different servo is necessary to determine how a turn in the opposite direction will be achieved.

There will be a test to determine if turning while moving in a straight path will be feasible. Such a function would allow much smoother movement through the maze rather than a stop and turn approach. In order to test this, a method will be written to move the robot a set distance before finally pulsing one servo more than the other. This will result in one servo rotating faster than a separate servo.

If moving while turning is feasible, it will be necessary to determine the pulse characteristics necessary to achieve a turn around a corner and how wide a corner is allowed to make the said turn feasible. To test this, A plank will be placed for the GUNDAM to turn around. This process will simply consist of a process of narrowing down the numbers needed to make the turn. A general guess will be made for the necessary turn that will no doubt be too wide or tight to make the turn around the plank. If too wide, the servo closest to the plank needs its speed lowered or the servo farthest from the plank needs its speed increased. If the turn is too tight, the reverse is necessary. The pulses characteristics that achieve the desired turn will be recorded. The point of the turn will be logged to provide a basis for the dimensions of the maze.

7.2 BILL OF MATERIALS

The following table shows the bill of materials for the application circuit of the CC1101 RF transceiver at 315/433MHz.

Component	Value at 315MHz	Value at 433MHz	Manufacturer
C51	100 nF	100 nF	Murata GRM155C
C81	27 pF	27 pF	Murata GRM155C
C101	27 pF	27 pF	Murata GRM155C
C121	6.8 pF	3.9 pF	Murata GRM155C
C122	12 pF	8.2 pF	Murata GRM155C
C123	6.8 pF	5.6 pF	Murata GRM155C
C124	220 pF	220 pF	Murata GRM155C
C125	220 pF	220 pF	Murata GRM155C
C131	6.8 pF	3.9 pF	Murata GRM155C
L121	33 nH	27 nH	Murata LQG15HS
L122	18 nH	22 nH	Murata LQG15HS
L123	33 nH	27 nH	Murata LQG15HS
L131	33 nH	27 nH	Murata LQG15HS
R171	56 kΩ	Koa RK73	
XTAL	26 MHz	26 MHz	NX3225GA

Table 2: Bill of Materials for CC1101.

Chassis	$35
Beagleboard (with Cables and MicroSD card)	$195.18
Battery Switching Voltage Regulator PCB	TBD
Motor-Microcontroller PCB	TBD
LED Network	TBD

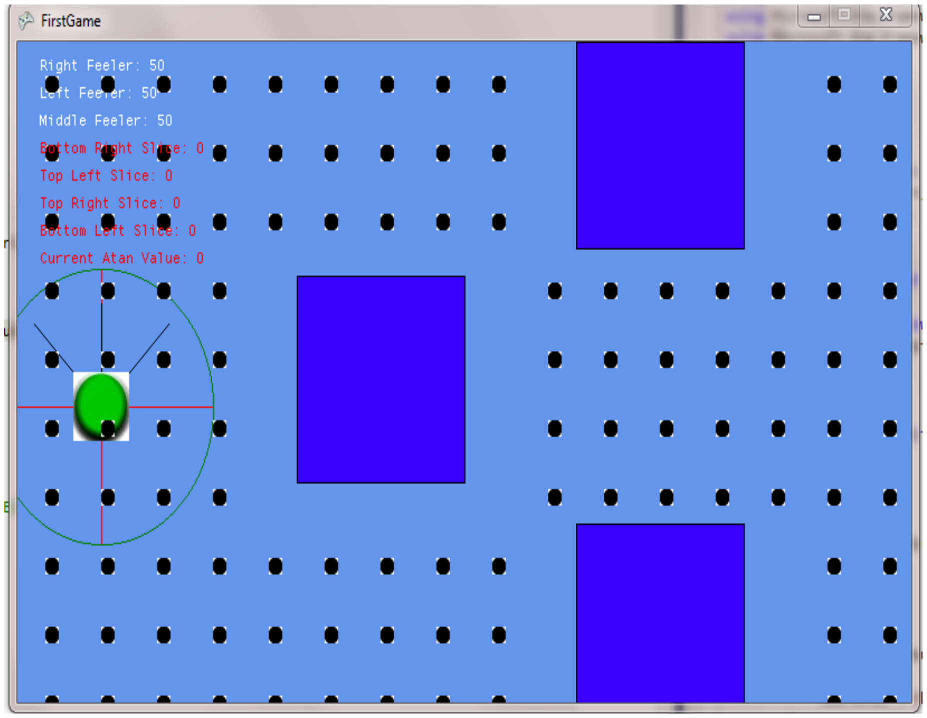
The following table shows the bill of materials for the parts used in the communication subsystem.

Part	Quantity	Price	Manufacturer
CC1101	2	$5.88	Texas Instruments
MSP430G2553	2	$2.02	Texas Instruments
CP2101	1	~	Silicon Labs
Whip Antenna	2	$10.90	Wilson
GP2D120	2	$29.00	Sharp
Ultrasonic trans.	2	~	~
TPS62730	2	$3.76	Texas Instruments
MAX232	1	$0.41	Texas Instruments
LM458	2	~	~
LM311	1	$0.19	Texas Instruments

Table 3: Bill of Materials for Communication subsystem

8.1 TESTING MAZE NAVIGATION

In order to ensure maze navigation upon request of the user would work sufficiently, a test of the A* algorithm would need to be conducted. While not written in JAVA, a program to run A* was made in Visual Studio (C#) to show the algorithm in action. The test field consists of a simple plain background with three walls placed throughout the field to construct a crude maze like environment. Across this environment, nodes were automatically generated on any space that was not populated by any of the three walls and also a sufficient distance from the walls to ensure that the center of the “robot” could take its position at said node without overlapping over a wall. While this problem in the simulation only causes a strange looking behavior, this occurrence in our real world application would cause the robot to attempt to inhabit space it could not realistically lie in.



```
Right Feeler: 50
Left Feeler: 50
Middle Feeler: 50
Bottom Right Slice: 0
Top Left Slice: 0
Top Right Slice: 0
Bottom Left Slice: 0
Current Atan Value: 0
```

Figure 18: Test map with nodes

One issue that was brought up upon testing of this system was the fact that walls needed to be stored someplace in memory to allow the nodes to be placed along the outside of the walls. This memory issue brought to light the issue of nodes possibly being placed outside of the maze boundaries. This issue can be resolved by storing locations of walls and nodes in their own respective arrays as discussed earlier. The method relies on putting the map on its own x-y plane. Each node will have its own x-y coordinate on the drawn out map itself. The walls also have their own position. It is important to note that the x-y position represents the center of the node and the upper left hand corner of each wall.

Several lists were created to implement the A* algorithm. There is the already established list of nodes, “nodes”, a list to house any nodes that are currently being analyzed and have not yet been discarded or implemented in the path, “half_nodes”, a list of nodes that have been completely analyzed, “nodes_analyzed”, and a list of nodes that are determined to be part of the current shortest path, “map.” The list of nodes currently being analyzed is housed with the source node. During implementation, the source node represents the node at the position of the robot or the location the user has selected as the beginning of the path.

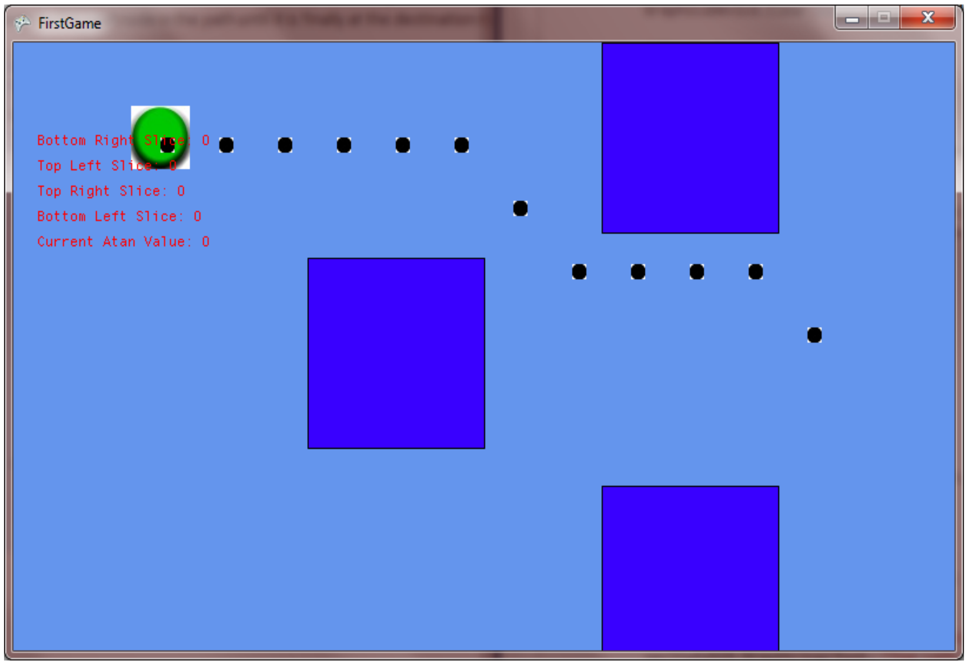
Once the initial node has been placed in half_nodes, a counter is incremented to keep track of the number of nodes inside the list and loop is entered that is only left once the count of half_nodes reaches zero. An float integer called tent_current is initialized to be -1. This is a flag to the loop to start the current node as the node containing the smallest score. This score is the distance from the current node to the source node added to the nodes heuristic and is set to tent_current.

There exists two functions that handle these scores: getG() and getHeuristic(). “getG()” calls back to an element of the node simply labeled g. These are all initialized as 0. This is for the first call to the function of creating a path. The G score represents the distance from the current node to the source and this score is 0 if the current node is also the source node. Upon all other calls through the loop, the G score is set to simply calculating the distance formula from the analyzed node to the neighbor in question. Because we are using x-y coordinates as one of the elements of the node object, this number is easy to calculate.

The heuristic is set the instant a destination node has been chosen. This number will never change as the destination node remains at a constant, unlike the currently analyzed node and its neighbors. This heuristic does not follow along the path of different nodes to be calculated. Because of this, another simple application of the distance formula is applied. There may, however, be a case in which the destination node has no neighbors. If this is the case, the heuristic is set to infinity.

If tent_current is greater than -1, the neighbors are searched through and their respective g + h score, or f score, is calculated. If this score is determined to be greater than tent_current, it is passed over. If the score is less than tent_current, it is replaced as the new node to be a part of the map. Once a node is discarded as not part of the shortest path, it is removed from half_nodes and placed into nodes_analyzed. The counter for half_nodes is also decremented. If the goal is reached, we have found our shortest path. Any node within the nodes_analyzed is placed inside the map list. This list is then passed to a sepearte function called complete_path. It is important to note that the lists in question simply contained indexes to the original nodes list.

The complete_path function’s job is to translate these indexes to a list of nodes that form the path. Originally, these nodes were placed in the list without any reference to the order they were in for the path to be completed. The method for going through these nodes was to simply choose the closest node to the current node that was not already traversed. A problem that was not encountered during the testing but was realized during it was the possibility of the closest node not necessarily being a node that could actually be reached from the current node. To combat this, the actual A* search had to be reworked. The indexes stored would be placed in the list in the order they were discovered. Once this was accomplished, the list would be searched through again during the complete_path function and the respective nodes would be added to the path_list in the order they are on the actual path. This would result in a much better analysis of the path without the issue of an unreachable node. The resulting path is pictured below.



```
Bottom Right Slice: 0
Top Left Slice: 0
Top Right Slice: 0
Bottom Left Slice: 0
Current Atan Value: 0
```

Figure 19: Completed Path

An issue with nodes disappearing during the traversal of the path seems to come up. The actual consequence of this disappearing node is not immediately apparent as the “robot” seems to make its way through each node regardless of whether it has been drawn out or not. Whether this will have adverse effects on the actual implementation in the robot is yet to be determined.

8.2 MAZE DRAWING TESTING

The bulk of the software side resides in drawing out the maze. This naturally leads to the bulk of the testing residing in this portion of the software. There need to be tests for the different interface options on the maze drawing window, the translation of the text for the different possibilities of the maze, the necessary scaling of the maze, and the drawing out of different types of maze to ensure that different combinations do not result in unforeseen layouts.

The first test is simply ensuring that we are able to display a window for the map to be drawn in. This will be a basic test of producing a method with several frames, windows, and buttons displayed for the user to view or interact with. The buttons need to be tested to ensure that they are capable of calling different methods and changing the expectations of the program as a whole. For instance, if the button for solving the maze or discovering all of its routes has been selected, the software is expected to call the function to actually draw out the maze and, thus, read from and write to a particular file for instructions on how the maze is written. This will be tested by printing out a different text input to the file for the button for solving a maze and the button for filling out a maze. The test for the button that will simply print out instructions will be tested identically.

Next, there need to be tests to ensure that we can draw out a basic layout and recall the said layout for future methods of drawing out paths. The first test will have the basic function of drawing out a line 50 pixels long and 5 pixels thick. The wall will be drawn at location (25, 25) of the frame. This information will then be written and stored within an array. This array, realistically, will hold all wall elements. Each wall element will hold its location (the starting point of where it was drawn), its length, its width, and its ending point. The length and width can be calculated with just its starting and ending point, but storing the length and width will provide a simpler way of recalling such information.

Once the single wall has been drawn on the map, a basic maze layout will be drawn to ensure that all of the walls are properly stored within the array. The maze will have an exit and entrance located on the borders for the reasons listed during the design portion of this essay. Each wall will have specific places to reside on the maze, as well as predetermined widths and lengths. This is all to test that the physical appearance of mazes drawn out on the interface. A successful test will consist of a clean maze layout with all walls stored and accounted for.

The next step is to place nodes at all locations that are important for the development of a path. As explained earlier, these locations are at turns and forks in a path. These nodes will, similar to the walls, have locations already known and stored in their own array. Nodes will not be visible during the final implementation of the software, but nodes will remain visible during testing to make debugging of the system much easier.

After the testing to ensure that mazes can be drawn, it is important to be able to translate known locations into instructions the GUNDAM will print out onto the text file for the program to periodically read. The first test is ensuring the maze drawing algorithm is able to initialize a maze. This step is important as upon entering the maze, it must be aware if a wall is immediately encountered in front of it or to its sides. The sensor’s will come into play for this in the actual implementation. During this test, we will test all possibilities for the entrance to a maze. It is important to note that the width of the maze will be a predetermined value that is important when actually drawing out our maze. The first possibility to test is a situation in which the only way to go is forward. A message will be sent with the following information: “0 1 0.” This message should result in an entrance drawn out with a two walls on either side extending one foot. This will serve as the default distance the entrance must start with before a turn will be built into the maze.

The second possibility consists of a path only available to the right. This situation utilizes the predetermined width of the maze. Using this width, we are able to determine how far the initial wall to the left will be drawn before the turn to the right is made for the left wall. The third possibility consists of a path only to the left. A similar result to a sole right path should be seen on the map with the difference of the wall to the right extending out to the width of a path before turning left. A test for the left and right being available should be the next test for our maze. In this situation, it is important that the software leaves the path to the right open with a node left behind for all relevant search algorithms. A similar result should be noted for a path straight and left, straight and right, and all three paths open.

A final debug test of a complete block should also be tested by the program. If such a situation is found, it is clear that the maze has no exit and no available paths to take. This test also serves to ensure that the GUNDAM is capable of taking notes of dead ends. Dead ends have two different meanings depending on the method we are currently in and the state of the maze. If currently in the entering maze method, the robot responds as stated before. If in the process of solving a maze with nodes still available with unexplored paths, we simply need to make our way back to the node. If all nodes are completely solved, then the maze has no exit. How we will test these possibilities will be further explored in the maze-solving-testing section of this paper.

8.3 MAZE SOLVING TESTING

The robot will need to be tested in its ability to solve the maze. This testing relies on various aspects of our GUDNAM to be tested individually. These components are the ultrasonic sensor, the different algorithms, and the communication between the GUNDAM and the software in terms of what algorithm it will use.

The ultrasonic sensor's first test will be simply to estimate the range of the sensors. While there will be an advertised range for the sensors, it is important to find the true limits of our sensors to prevent a case of the maze we test our GUNDAM on having walls that are too far or too close to be detected by our sensors. The initial test will be to determine how close our sensor can reach a wall before its input becomes unreliable. This test will serve also as a reference to what our buffer should be before we decide to simply make our robot turn or assume a dead end. The sensor will be set to output the distances at various intervals and also whenever the distance happens to change. The range at which the value ceases to be reasonably accurate is the value we will use as a base for our buffer before considering a wall too close. Some inches will be added to this value to prevent a hard boundary.

The same will be done for determining how far we can go with our sensor before it outputs a max distance. The sensor will face a wide wall and slowly move away from it at a steady rate. The output of the distance will also be printed out on our microcontroller programming screen to determine this distance. It is important to note that this is to test the actual input that the software will receive, not what the sensor is said to be capable of once our sensor has been built.

To test the flags our sensor will be setting off, a program was written while programming a microcontroller to simply output a message at certain points. A distance from a wall was set arbitrarily as a signal to print out a generic "Here's a wall!" message. Once this was accomplished, a method was written to send a signal to the servo motors. This was to test the ability to translate a signal from the sensor to action from the servos. Of course, this will be changed to specifically cause the servo motors to turn the robot a certain distance rather than simply turning both servos.

The next test will be communication between the GUNDAM and the software on the PC. The GUNDAM needs to be able to access a document on the PC that is solely for updating the drawn out map, a document solely for receiving instructions on movement, and a document for which algorithm to use when solving the maze. The three documents will be created and stored in their own folder. A simply program to open these files and input the text "TEXTGUNDAM" will first be done to ensure that this communication has been successful.

A test for the software to write to the text files will also be done to allow for the updating of instructions for the robot to follow. This will simply follow the same procedure as writing to the text files by the GUNDAM. An input of "TEXTPC" will be written to the text files.

A test for sharing the writing to the files by the GUNDAM and the PC will be conducted to ensure that the software on both sides will follow the rules of taking turns to write to the text files without interrupting each other's processes unintentionally. In order to conduct this test, a process of periodically reading the text file will be conducted on the GUNDAM's side for the text of either "GO" or "WAIT". The text file is initialized to contain the phrase "GOPC." The software on the PC will then be written to open up the text file the robot is reading and input the text "GOGUNDAM." The GUNDAM, upon reading this text, will input the phrase "WAITPC." The PC should, upon reading this phrase should know to wait for the GUNDAM's next input into the text file. The GUNDAM, upon reading WAITPC on its next pass will input "GOPC." The PC side software should then input the phrase "WAITGUNDAM" letting the GUNDAM know to wait for the software's input. This loop of action will be done for a minute to ensure that the software are able to both modify a text file as well as be aware of the fact that a text file is awaiting input from software on the other side.

After text input and turns between software has been tested, the ability to modify the movement of the GUNDAM by the software must be tested to ensure that the software is capable of modifying the solving algorithms of the GUNDAM. The GUNDAM will periodically check a text file without actually modifying its contents. The software will randomly choose which servo to send a signal to. This decision will simply be in the form of "LEFT" or "RIGHT", representing the side of the robot the servo resides on. If a left is printed out inside the text file, the left servo will turn, resulting in a clockwise spinning of the robot. If the text file contains the phrase "RIGHT", the robot should spin counter clockwise. This correlation will be achieved due to test methods that will be created inside of the microcontroller of the GUNDAM. One method is activated by a branch to its contents upon reading of the LEFT phrase. A similar reaction is intended from a reading of the RIGHT phrase. This also has the added effect of testing that the methods written for rotating the servos achieve just that.

A test will then be made to ensure that the robot is capable of maintaining a method of movement until receiving different instructions from the PC side software. A method will be written that will keep the robot moving straight in a certain direction. The PC software will then write a text line of "STOP" into a specified text file. The GUNDAM will, of course, periodically check the text file for this message and the message of "GO." Upon receiving the STOP message, the GUNDAM will enter a method of standing still while continuing to check the text file. This method will be entered between 50 pulses during movement and every 2 seconds while not moving. The PC software, after a set time, will input the message of GO and the robot should then move forward once more.

A similar method will be done for movement, but instead, the signal for a request for more movement will be sent by the GUNDAM rather than assumed by it. The GUNDAM will move for a predetermined amount of time before printing its own message for the

PC labeled “TURN.” The software will read this input during its reading loop and randomly choose a direction to turn. For this case, we will say LEFT has been chosen. During this process, the GUNDAM will not move. The GUNDAM, upon reading LEFT during its reading loop, will enter its method for turning left 90 degrees.

The bulk of the maze solving testing will now be tested in several different types of mazes. Each maze will have different properties to test for different algorithms and cases within a maze. The first test will simply consist of a single path with one entrance and one exit.

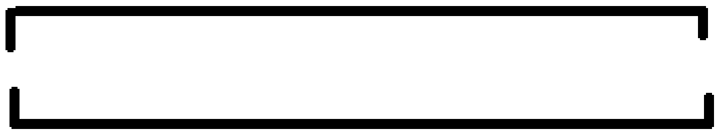


Figure 20: Simple, single path testing maze

This test serves to show that the robot is capable of recognizing when it has found an exit. As exits within all mazes constructed for this project will have the same properties, the recognition of an exit only needs to be tested once.

The robot then needs to be tested on its capability of following the left wall. This routine serves as the initial solving method of the robot whenever it begins its maze solving routine. A simple, single turn maze will be constructed with one left turn. If working correctly, this should result in the robot turning left and reaching the exit. The same test needs to be done for one right turn. If this is done correctly, the robot should turn right and then come across the exit.

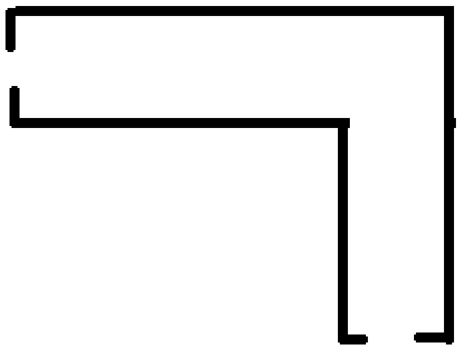


Figure 21: Single turn testing maze

While these test its simple ability to follow the left wall, it needs to be ensured that these turns are a deliberate choice and not a result of chance. To test this, a proper left wall will be constructed to follow as well as alternate choices that break from following the left wall. If presented with a left turn and a right turn, the proper response to this case should be a left turn. If presented with a left turn and a straight path, the proper response should also be a left turn. If presented with a straight path and a right turn, the proper response should be continuing along the straight path. If presented with all three paths open, a left turn should be the result.

While this handles simple testing of left wall following, it needs to be tested that the software is capable of recognizing a situation in which left wall following will not work. As explained before, this occurs whenever the entrance to a maze does not lie on the borders. To test this, utilization of the text file needs to come into play. If the exit is in fact on the border, the rear sensor will report a signal of a distance much larger than the known width of the maze. A second test will then be constructed in which the rear sensor reports a distance less than “the known width of the maze – length of the chassis” but equal to or greater than “the minimum range of the rear sensor.” This means that the robot has started its search somewhere within the maze (i.e. the entrance resides within the maze rather than its borders). These cases need to result in an input to the text file by the software side. For the purpose of this test, we will input this text file ourselves first to ensure that the robot no longer simply follows the left wall and instead waits for instructions by the software to traverse the maze.

We will first build a maze with a central position with four directions available for traversal by the robot. The robot will be entered into the Tremaux algorithm manually for this particular test. Once in this particular algorithm, we will input the text “forward.” This should result in the GUNDAM moving forward to the next node, coming to a complete stop, and inputting the phrase NEXT to the text file. Once the tester reads this input, a text of “back” will be written into the text file to ensure that the GUNDAM rotates a full 180 degrees and returns to its starting node. The text input of left and right will be separately written to ensure that the robot rotates the proper amount of degrees and moves to the next available node. The final implementation of this will be to ensure that the robot inputs SOLVED rather than NEXT. To test this result, the GUNDAM will be made to move into a node that leads into an exit.

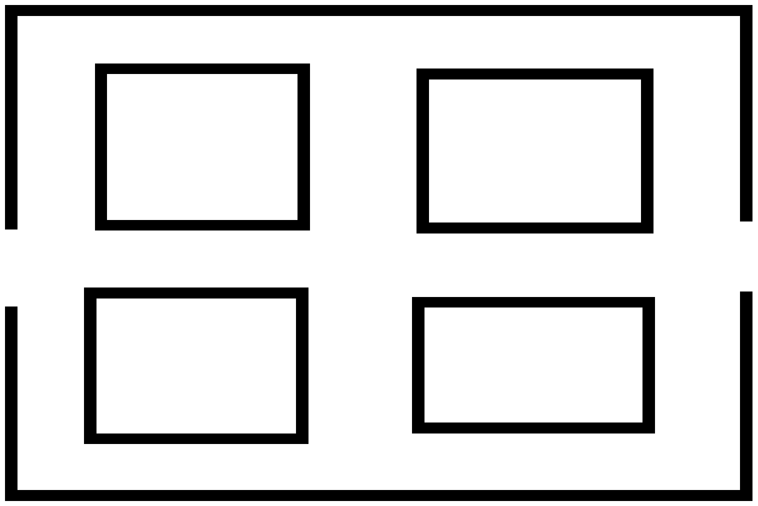


Figure 22: Tremaux testing maze

After basic mazes have been tested, a more representative maze will be built to ensure that these methods are properly utilized. The role of the PC side of the software will be taken over by the tester to allow for the isolation of testing the robot on its own. The first complex maze will be solvable through the use of left turns. As such, the only output from the robot should be information on the path it has taken and the distance traveled. These outputs will be able to be analyzed by the tester upon each turn of the robot. It is up to the tester to input the correct text string expected by the GUNDAM before it can continue on with its search algorithm.

The next complex maze will involve an entrance located within the maze. This test implementation will involve a maze with much more turns but will not be too complex as too make memory of paths taken along with their values too much for the tester. It will be up to the tester to input proper directions for the robot to take that correspond with Tremaux’s algorithm. Once the maze has been solved, an input of SOLVED by the robot will be expected once again.

8.4 GUNDAM/Laptop Link Testing

An important aspect of the project is the ability for the GUNDAM to be capable of communicating with the PC side software and vice versa. There need to be tests conducted to ensure that the modification of the text files occurs with no unforeseen faults such as interruptions mid reading of the files and failure to process a change within the text file. Tests for these have been constructed, yet without the added hassle of drawing out the proper maze according to the instructions written. These instructions were previously written out by the tester. It is now important to test how well the

GUNDAM is able to solve through a maze while updating the PC side software with relevant information which will then be translated into the proper maze.

To test this, a first initial test will be conducted combining all prior tests into a single function. The single path maze will be used once again to test the simple drawing out of paths and walls and recognizing an entrance and exit. The text file’s contents will be outputted by the software on each read to ensure that the expected contents are in fact inside. This also will make it easier to determine if the next step implemented by the software was indeed correct. The maze drawing itself will also be simultaneously tested here. The result should be a robot making stops at each interval, a node visibly placed at said intervals, and the maze path properly drawn out.

The next test needs to be on a maze with a solution that will leave portions of the maze undrawn. It is important to ensure that these paths are not closed off by the maze drawing algorithm and the robot does, in fact, follow the proper algorithm of wall following. Once again, nodes must be dropped at their proper locations and stops must be made at each turn and fork.

The next test will be on a maze unsolvable by wall following, the result of these tests will rely on first enacting the wall following method. The software needs to be tested to ensure that it is able to detect either a starting point within the maze or a wall that lies on its own with no connections to walls that have an origin at a wall. The software should then, upon discovery of each of these cases (which will be tested separately), revert to Tremaux with the associated message printed into the text file. This message will, of course, be outputted by the software to make sure it has done so. The software should then enter its Tremaux algorithm and properly sort through the nodes to be searched through. The same maze used on the manual Tremaux will be used in this case as well. This has the benefit of the solution already available as something to compare the software’s output to.

8.5 MAZE COMPLETION TESTING

The function of finding all paths of the maze is another function to be tested for the GUNDAM and PC software as both will be utilized for this particular method. Each physical test will be done on prior complex mazes used for testing the solving function of the robot. Before testing physically however, it is much easier to simply create a maze within the system with nodes placed at their proper locations and watch a virtual implementation of the method. Each node will light up to signify a visit to its location by the algorithm. Upon visit to a location that is purposefully labeled undiscovered, a node should be placed at this location to represent a newly found break in the path. Once the maze has been fully explored, all nodes should be lit up a different color from their starting color. Along with lighting up nodes, an output representing where it plans to move next should be outputted to a file for the tester to ensure that the algorithm is in fact purposefully choosing the next and correct path. The weights of each path (how often they have been traversed) will also be written along their trajectory to better aid in the debugging phase.

As stated earlier, nodes are placed at every turn. These nodes are stored on the laptop side along with information concerning whether a turn has been taken or not. The test will consist of dropping the maze at a node location on the map and starting the relevant method. The robot should then proceed through all unexplored paths of the maze itself. It is important to make sure that each movement of the robot corresponds to the movement the algorithm has deemed the next step in its process. Upon finding all paths, the proper output to the text file should also be ensured along with the robot ceasing to move and instead awaiting input.

8.6 PATH FORMATION TESTING

The final function to be tested is the ability to choose a path for the GUNDAM to take. To test this, a similar method to maze exploration will be used. In this test, all nodes will be manually placed in the virtual maze with a similar lighting up procedure for visited nodes. The difference will be that only nodes along the path will be visible once a path has been chosen. Afterward, each node should light up once reached by the algorithm. An added factor to the test will be to draw a line for the path. This is to help ensure that the proper method of moving from one node to the next is actually being used rather than an attempt by the software to move the robot through a wall. This will also ensure that unreachable nodes are not being included in the path. During the movement through the path, the text file will also be checked to ensure that proper instructions are being fed to the GUNDAM at all times.

Once this has been tested, it is important to test these instructions with the GUNDAM. The robot will need to traverse the maze using the proper nodes in the proper order, avoiding all walls and unreachable nodes. Paths of differing complexity will be tested. A simple path from two neighboring nodes will first be tested to ensure basic movement. Next, turns will be added to ensure that the robot is capable of changing direction while traversing a predetermined path. A selection of a position close to a node also needs to be tested. This is to ensure that the software is capable of taking a location not previously stored as a node and create a new node for the user and robot to use in making its path. A test in selecting a node outside of the maze will also be done to ensure that a proper error message is printed out for the user. The final test will be to implement a complex path from one side of the maze to the other involving a good selection of turns and distance. This will ensure that the A* path algorithm has no hiccups when presented with a large number of nodes to work with.

8.7 PCB, Chassis, Motor, Microcontroller Software Testing Procedures

Introduction

Some testing procedures need to be done on the fabricated PCBs, the robot chassis, the chassis motors, the software, and the integrity of the placement of parts on the chassis. This will insure that there are less unexpected problems under operation of the robot and will help us make sure that everything is designed to how we need everything to be.

PCB Testing Procedures

Part of the procedure for testing the PCBs is to make sure that all the parts specified are included in the printed circuit board and connected the way specified. Once this is verified, the boards can be powered up. If something goes wrong and a part burns out, then we'll have to determine which part that was and why, most likely by using a multi-meter to test the voltage across each component in order to understand what happened and why. If the board for some reason doesn't power up at all, then we'll look for a problem with the power supply of that PCB. If the board powers up and there are no problems with any of the components, then each input and output pin needs to be checked to make sure that they work. This can be done by making them high or low using the microcontroller for those pins. A simple testing algorithm can be used on the micro-controller in order to make sure all the pins work right.

```
//Put all pins High
Pin1 = high
Pin2 = high
…
PinN = high
```

While each pin is high, we will check that they are high by using a multi-meter. If that checks out, then we will check whether or not all the pins go low as well using a similar testing algorithm.

```
//Put all pins Low
Pin1 = low
Pin2 = low
…
PinN = low
```

Any pins that are not displaying the correct high or low output, depending on the test, will be checked off as not usable, since this means there is something wrong with either the micro-controller or something in the circuitry is wrong or has burned out those pins.

For the Switching Voltage Regulator, the battery will have to be checked, each voltage regulator will have to be checked for the proper voltage, and the switching mechanism of alternating between outlet power and battery power needs to be checked by checking with a multi-meter for open and closed connections around the two mosfets.

Robot Chassis Testing Procedure

The Chassis needs to be checked for structural integrity. The wheels need to be checked to make sure the bearings work; if there is something wrong with them, they can possibly be replaced. The axels need to be checked to make sure they are working as intended; if any of the gears that control the axel for either the right or left wheels is broken or won't turn, the chassis needs to be replaced.

As for the parts themselves that are fastened onto the chassis, if any of those parts are loose, they need to be properly fixated to where they need to go. Loose parts that can't be drilled need to be set with plates that can fasten them to the chassis and parts that can be drilled need to be screwed firmly into place.

Chassis Motors

The motors themselves need to be tested within their operating range to determine if they are working as intended. We also need to test how the motors compare to each other; they both won't operate the same, given the same Pulse Width Modulation for a given voltage. So we need to recalibrate the motors for different Pulse Widths, so that they have the same degree of operation with each other. For a given operating voltage, we'll have to note these differences and implement them into the controller.

Voltage	**5v**
Motor 1 (Acceleration)	**M1 (To Be Measured)**
Motor 2 (Acceleration)	**M2 (To Be Measured)**
Calibration Difference	$|\mathbf{M2} - \mathbf{M1}|$

We'll also have to check that the motor-microcontroller is controlling the motors correctly without any problems.

Software Testing Procedure

The testing of the software will come after all the other testing and calibration is done. The first phase of the software testing will involve testing whether or not the software does what it is supposed to do; this will require testing the individual algorithms to see if they operate as intended and testing the individual routines to make sure that whatever outputs they are supposed to create, make sense.

Specifically, the motor-microcontroller will need to be checked to make sure that the calculations for determining how much energy is being consumed provide accurate and reasonable answers; to do this, every clock cycle can send data out through USB to a computer or to the Beagleboard, where it can be checked for accuracy. We'll also have to check that the Pulse Width Modulation signals to the motors are operating as it is. given commands from the Beagleboard; this will require that the digital logic behind the motor-microcontroller is checked and that the Beagleboard is sending out the correct and intended signals/commands. The best way to do this is to first test it without using the Beagleboard, so that we can know whether there is a problem with the micro-controller or the Beagleboard. A table for testing whether or not the proper output is found is shown below.

Input Pin 1	**Input Pin 2**	**Output**	**Measured Output**
High	**Low**	**Output 1 High & Output 2 Low**	
Low	**High**	**Output 1 Low & Output 2 High**	
High	**High**	**Not Accepted**	
Low	**Low**	**Output 1 Low & Output 2 Low**	

For the Beagleboard, since this board is not being designed, testing will not have to be so extensive. If the board boots up and loads the Angstrom Linux OS, then that will be the first test. The only thing to then test is whether or not the expansion header works on pin high or pin low as explained earlier and to test whether or not the USB interface works with a wireless USB stick.

8.8 Testing the RF Transceivers

First, testing of the CC1101 RF transceiver must be done without encryption algorithm applied to the transmitted data. To test proper operation an oscilloscope can be used for spectral analysis of the modulated signal or on the receiver's end we can look at the digital signal and determine if the proper bits that were sent are being received. Another way of testing proper function of the RF Transceivers is to use the MSP430 MCU through a SPI interface. The MSP430 MCU value line has an evaluation board that can be interfaced with a computer for debugging and programming purposes. Testing can be performed from the MSP430 by reading the 4-wire SPI interface on the CC1101. If the pins read the proper values than the CC1101 can be configured with the specifications detailed in the RF transceiver design. However, sending data over the CC1101 won't be sufficient and a second CC1101 with an SPI interfacing with an MCU will be needed to read the data.

Another option for testing the operation of the CC1101 involves a chip status byte that can be read by the MCU. When the header byte, data byte, or command strobe is sent on the SPI interface from the MCU, the chip status byte is placed by the CC1101 on the SO pin, which in turn can be read by the MCU to determine the state of the CC1101. Since packets are handled on the hardware of the RF module, the testing procedure to ensure packets are being sent is simplified. On the receiving end MCU will read the receive buffer and output it in a debug window connected to the computer. This can be duplicated with the other MCU and RF module to ensure that both nodes are capable of talking to each other. The CC1101 module also has status registers that allow the MCU to read the amount of data in either the TX and RX FIFO. To ensure the RF transceiver connected to the computer can properly communicate through the UART to USB Bridge, a virtual COM port connected must be established through the CP2101. Through the use of a terminal program on the computer, the bridge can be tested for correct data transmission. A test program on the MCU connected to the RF module on the computer will be run to send plain text data from the MCU through the bridge.

Another testing procedure that is required to ensure the proper function of the RF Transceiver module in conjunction with the whip antenna it uses for extended range. The CC1101 module has an RSSI status register that holds an estimated value of the signal power level in the chosen channel. However, for the RSSI status register to be of any use it has to be converted from 2’s complement number to dBm. A function can be written on the MCU to handle this conversion; first the hexadecimal number must be converted to decimal, next based on the decimal number an offset formula can be applied to convert to a dBm value and output it to the screen. For example, at an operating frequency of 433MHz the RSSI offset is 74, if the RSSI decimal value is greater than or equal to 128 then the RSSI value in dBm is equal to the RSSI decimal value minus 256 divided by 2 minus the RSSI offset of 74. Similarly, if the RSSI decimal value is less than 128 then the RSSI value in dBm is equal to the RSSI decimal value divided by 2 minus the RSSI offset of 74. This value is useful in determining the efficiency of the RF transceiver on the robot at certain distances from the RF module on the computer.

8.9 Testing of custom communication protocol

Following the test of the transceiver it is necessary to test if the robot and the computer are properly communicating with each other. A way to test this is have a simple program on the microcontroller on the robot listen for transmissions and send acknowledgements. If the previous tests verify that the RF transceivers are sending and receiving data packets properly, than the MCU on the G.U.N.D.A.M. can send data packets to the computer’s RF module, which will initially be in listening mode. Once the computer’s RF module receives the data packet, it will need to send an acknowledgement to the G.U.N.D.A.M.'s RF module. For the protocol to work properly the G.U.N.D.A.M.'s RF module must go immediately into listening mode after transmitting packets for a short period of time and the computer's RF module has to transmit immediately after receiving packets. By calculating the latency of transmitting and receiving packets, the MCU on each end can ensure that packet acknowledgements have enough time to be sent and received, therefore proper synchronization between the two nodes must be established when taking latency into account. If each side can properly transmit and listen for acknowledgments than the custom wireless protocol valid operation can be verified.

8.10 Testing AES-128 encryption and decryption

The testing that the RF transceiver modules work properly is critical to testing if the communication protocol is working properly. Similarly, testing that the communication protocol is working properly is critical for the next test procedure, which involves the AES-128 encryption technique. The AES-120 encryption technique will be implemented using software on the MCU. Plaintext data will be sent to the G.U.N.D.A.M.'s MSP430 module from the main microcontroller in charge of the sensor data and maze navigation algorithms. First, it is vital to test whether or not the main microcontroller's data is received properly through the I^2C interference it will use to communicate with the MSP430 microcontroller. The main microcontroller will be in charge of formatting the data from the sensors and the motors in a way the GUI on the computer can interpret to draw the maze. The purpose of the communication subsystem is not to interpret but to send data from one node to the other. The MCU on the robot will take the plaintext information from the main microcontroller and encrypt it using AES-128 encryption. To test that proper encryption has occurred, the MCU can be connected to an evaluation board and the MCU will send the encrypted and unencrypted data to a debugging window on the computer. Once proper encryption has been established, the reverse must happen on the MCU connected to the computer dongle on the receiving end. In a similar manner, the MSP430 microcontroller will be connected to the computer will decrypt the data and if all the previous testing procedures succeed than the computer will be able to receive plaintext data through its established virtual COM port.

8.11 Testing of distance sensors

The two types of distance sensors on the G.U.N.D.A.M. will be ultrasonic and infrared sensors. The infrared sensors will be mounted on the left and right side of the robot. The way they work requires a flat surface to properly measure distances from an object. The maze will be designed to have flat walls throughout with not circular turns since a circular turn will cause the robot to move erratically considering the time it takes for the sensors to read the distances. The response time for the GP2D120 infrared range finder is 30ms, which means the main microcontroller cannot have a truly continuous value of the distances between the robot and the maze's walls. In addition, the infrared sensors must be tested on different surfaces and different colors, since the range can vary wildly if any of these factors are not kept constant. Some of materials the sensors can be tested on include wood, ceramic, plastic, or metal. The result of these tests can ultimately determine what material the maze will be made of. Initially, it is safe to assume that reflective surfaces with a light color would be best in determining the accuracy of the infrared distance sensors. From the graph in Figure 10, it is evident that the sensor isn't very reliable at short distances, such as 0-10cm. To combat this the robot will need to center itself in the path of the maze at a minimum distance of 10cm from each side of the wall.

The ultrasonic sensors in this design needs further testing to determine the effective range. Once the parts are acquired and the sensors are built they can be tested on using the same procedure described to test the infrared sensors. The range requirements for the ultrasonic sensors are more critical in the maximum range rather than the minimum range as with the infrared sensors. The reason being is that the position the ultrasonic sensors will be on the front and back of the robot. As the robot traverses the maze many of the paths will be long and narrow, which creates large distances on the front and back of the robot. Since ultrasonic sensors are generally more accurate in measuring longer distances, a test procedure to determine the maximum and minimum distances would be to mount the sensors on the robot chassis and manually drive the robot close to and far from an object to get the tolerance readings. The sensors will be read from using the main microcontroller so it is ideal to use this setup when testing the sensors against various materials.

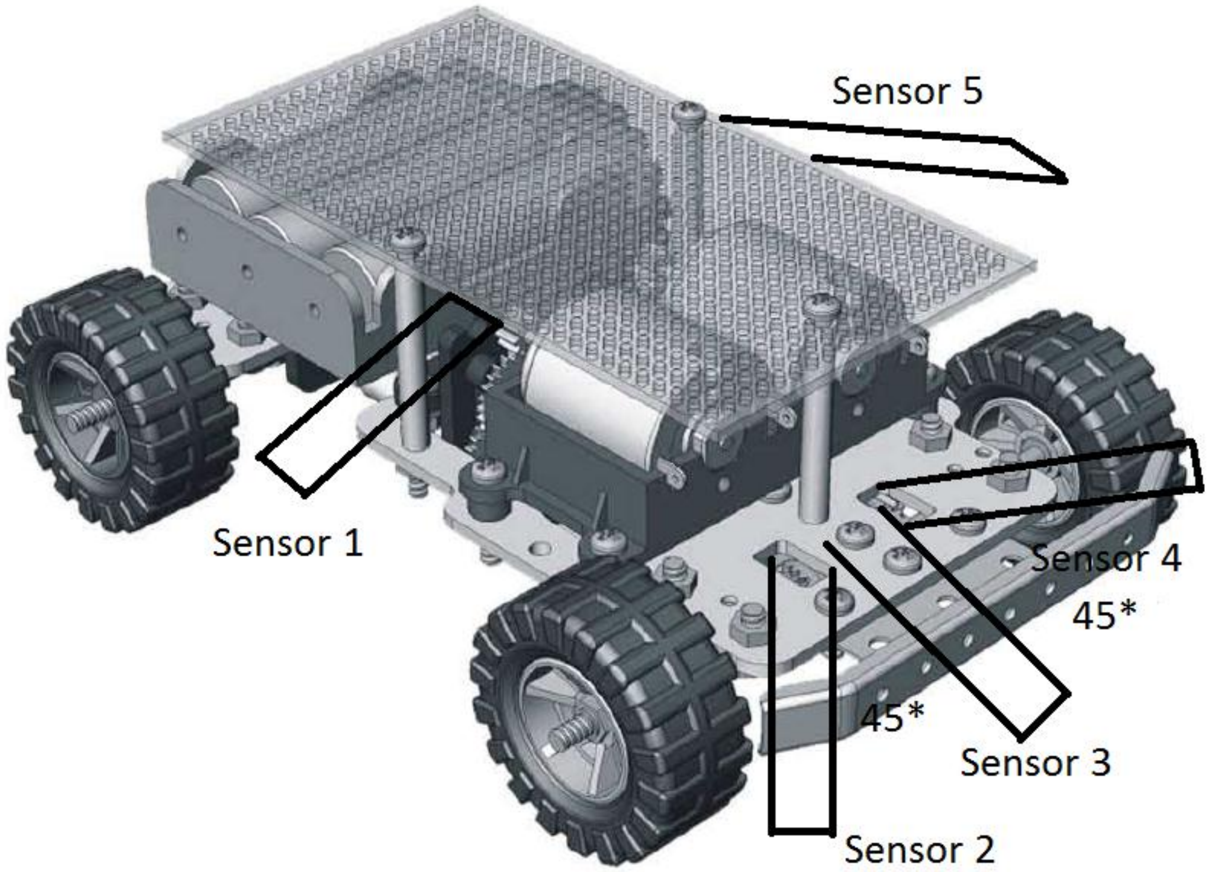
9.1 Enhancements

Introduction

Some enhancements are considered for this project, in the event there is time for implementing them. One enhancement would involve more sensors for better geometry detection and another enhancement would involve a robotic arm, in order to pick up or remove debris that might get in the way of the robot's path. There could also be a solar panel attached to the robot's power supply in order to help it conserve power and energy. Another idea is to have an LCD touch screen hooked up to the Beagleboard; and another idea is to have a small character screen that displays the energy consumption of the device while it is on.

Sensors

One option instead of having three sensors, one on each side and one on the front of the robot, is to have two more sensors on the front of the robot chassis. This way, we can angle two of those sensors away from the one in the middle, which would allow for better detection of what kind of surface is in front of us. It would work by using trigonometry to give us information about the curvature of the object in front of the robot, in order to determine what kind of path is ahead. Normally, the standard maze will not incorporate such complex corridors. The following illustrates the idea.

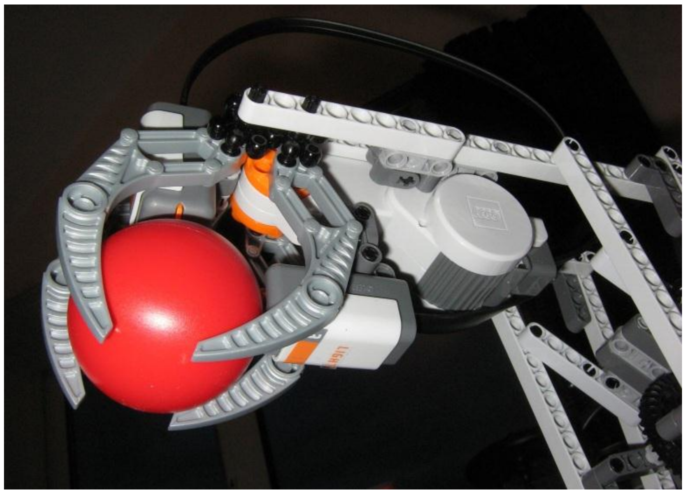


Then to calculate what kind of surface is in front of us, simple trigonometry can be used.

X_Length = Sensor 3 - Sensor 4 * cos(45*)
Y_Length = Sensor 4 * sin(45*)
Length_of_Edge = $\sqrt{\text{X\_Length}^2 + \text{Y\_Length}^2}$
Angle_Away_From_Horizontal_of_Robot = $tan^{-1}\,(\frac{X\_Length}{Y\_Length})$

Robotic Arm

The idea behind having a robotic arm is that it would be able to detect debris that is in front of it and then try to pick it up and move it aside or put it in a basket. Some debris would not be intended for picking up and the robot would have to determine whether it can pick up or move debris on its own. For the robot to detect an object there would need to be a video camera that sends its feed to an algorithm that analyzes that video to determine what it is looking at. This, in theory, wouldn’t be too hard to implement if all the debris was a particular shade of color and all the arm had to do was grab it like a net and then put it in a basket or move it aside. Most robot arms seem to have a clamp, but to pick up an object the arm would have to be more than just a clamp. The following illustrates what is being referred to here.



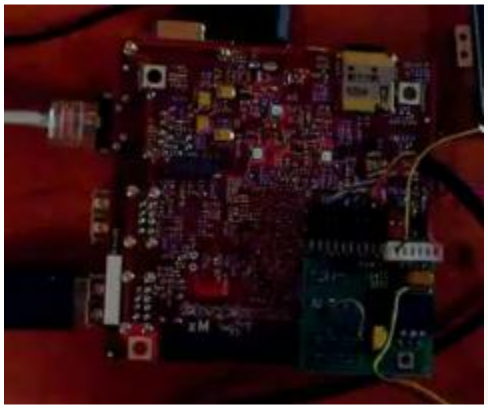
In the above, there are four clamps with double joints that could go underneath an object and completely grab it. The other option is to take a single clamp robot arm and attach scoopers to each clamp so that as the arm clamps down it scoops up on object instead. It would be the same effect as the above.

Solar Panel

Another option even is to try and make an overhead solar panel for this design that could supply enough power to power the robot, as well as charge the battery when it is low on charge. Since this design doesn’t require too much energy, about 3A from a 5 volt source mostly, about 15 Watts for the solar panel would be enough to power everything. But the panels range anywhere from $50 to $100 and they seem to be a bit bigger than what the size of robot chassis can handle. However, it could probably be done with the right panel.

LCD Touch Screen

There is an expansion LCD touch Screen that can be purchased for the Beagleboard. It requires an add-on LVDS board to interface a thin LVDS LCD screen to the board through an expansion header. This can be purchased from Chalkboard Electronics.



The picture on the right is the LCD screen with the LVDS board; and the picture on the left is the Beagleboard with the LVDS board attached. The reason for showing this is to understand that the Beagleboard is about 1/4th the size of the LCD screen and cannot fit onto the robot chassis adequately. It is much too big.

The following table explains the features that make this idea impractical for our purposes.

Cost	**$134.99**
Current Usage	**~15 Watts**
Size	**Too big for the chassis; there is no room for its placement**

As can be seen it isn't a very reasonable feature to use in the design, but it would be a nice way to output the map of where the robot has gone onto the screen. If we can find a smaller screen at a reasonable price and have time to invest in adding an extra battery to power the LCD screen, this could be a viable option once everything else in the project is completed.

Small Character Screen

A character screen could also be used to display interesting information about the operation of the robot. One main feature might be to display the amount of energy that is currently being consumed since the start of operation. This could be periodically updated so as not to hamper the functioning of any of the microcontrollers. Another feature might be to display the current time or display debugging messages that can help pinpoint any problems that the software might be having or causing.

10.0 Milestones

Stage One

Determine what we want to do and research how to do it.

Stage Two

Design all the schematics that are required and get all the parts that are to be purchased and aren't a part of the design process. Also design pseudo-code for how all the algorithms are going to be designed and will function.

Both search algorithms will be coded and tested on pre made maps through code input. The A* search algorithm will also be coded up and tested for its capability of finding the shortest path through a set mesh of nodes.

Stage Three

Have all PCBs made and tested for their reliability. Have all the microcontroller code designed and implemented into the microcontrollers. Testing of the microcontroller code goes into this stage.

The software will be capable of drawing out maps through text input by the user as well as drawing out paths on pre-planned maps and discovering paths through said maps as well.

Stage Four

Everything needs to be put together. All the PCBs need to be positioned and fastened to the chassis. The sensors need to be fastened to the chassis and all the parts need to be communicating with each other.

The software will be capable of taking in inputs from the robot and translating said info into map patterns as well as feeding back directions to the robot itself.

Stage Five

The mazes need to be built and any fine-tuning of how the robot will function needs to be done here. If the robot needs more power, the motors should be replaced. Anything that is not operating as we wish it to be will need to be improved to aid in successful completion of the goal of the project.

Stage Six

If all systems are working correctly, any possible enhancements to the robot will be done at this time.

Appendix

Parts List

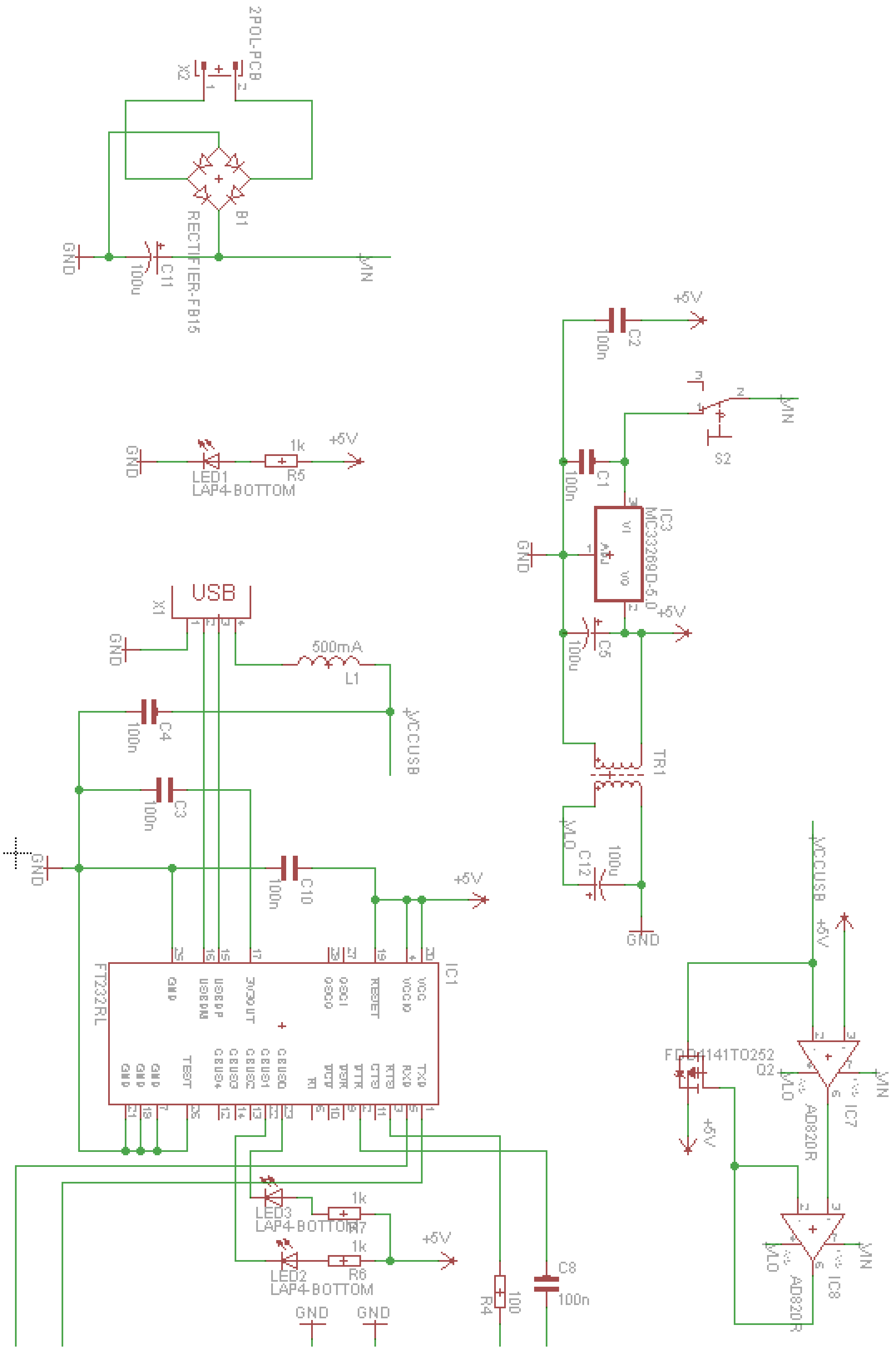
Component	Value at 315MHz	Value at 433MHz	Manufacturer
C51	100 nF	100 nF	Murata GRM155C
C81	27 pF	27 pF	Murata GRM155C
C101	27 pF	27 pF	Murata GRM155C
C121	6.8 pF	3.9 pF	Murata GRM155C
C122	12 pF	8.2 pF	Murata GRM155C
C123	6.8 pF	5.6 pF	Murata GRM155C
C124	220 pF	220 pF	Murata GRM155C
C125	220 pF	220 pF	Murata GRM155C
C131	6.8 pF	3.9 pF	Murata GRM155C
L121	33 nH	27 nH	Murata LQG15HS
L122	18 nH	22 nH	Murata LQG15HS
L123	33 nH	27 nH	Murata LQG15HS
L131	33 nH	27 nH	Murata LQG15HS
R171	56 kΩ	Koa RK73	
XTAL	26 MHz	26 MHz	NX3225GA

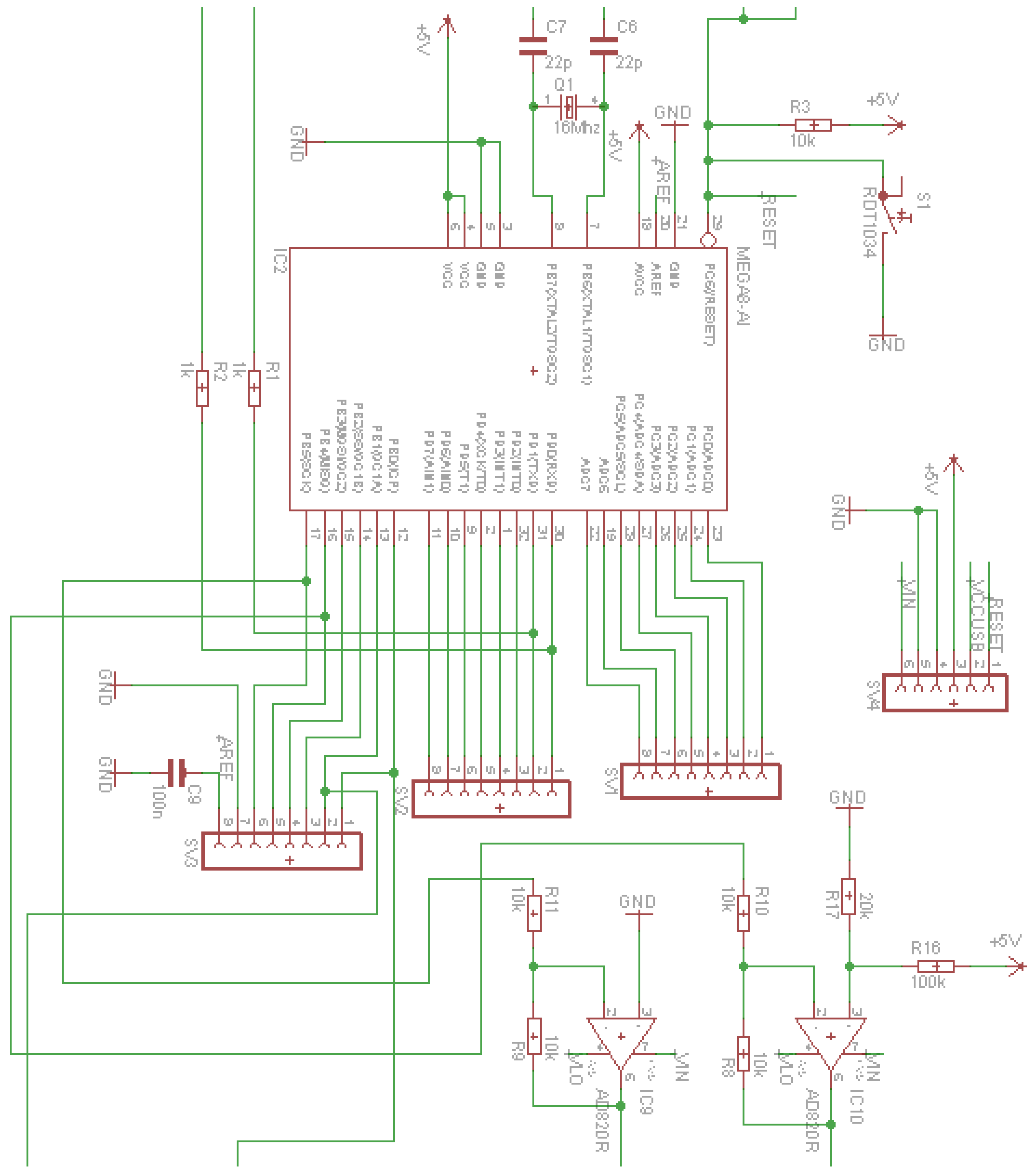
Part	Quantity	Price	Manufacturer
CC1101	2	$5.88	Texas Instruments
MSP430G2553	2	$2.02	Texas Instruments
CP2101	1	~	Silicon Labs
Whip Antenna	2	$10.90	Wilson
GP2D120	2	$29.00	Sharp
Ultrasonic trans.	2	~	~
TPS62730	2	$3.76	Texas Instruments
MAX232	1	$0.41	Texas Instruments
LM458	2	~	~
LM311	1	$0.19	Texas Instruments

Chassis	$35
Beagleboard (with Cables and MicroSD card)	$195.18
Battery Switching Voltage Regulator PCB	TBD
Motor-Microcontroller PCB	TBD
LED Network	TBD

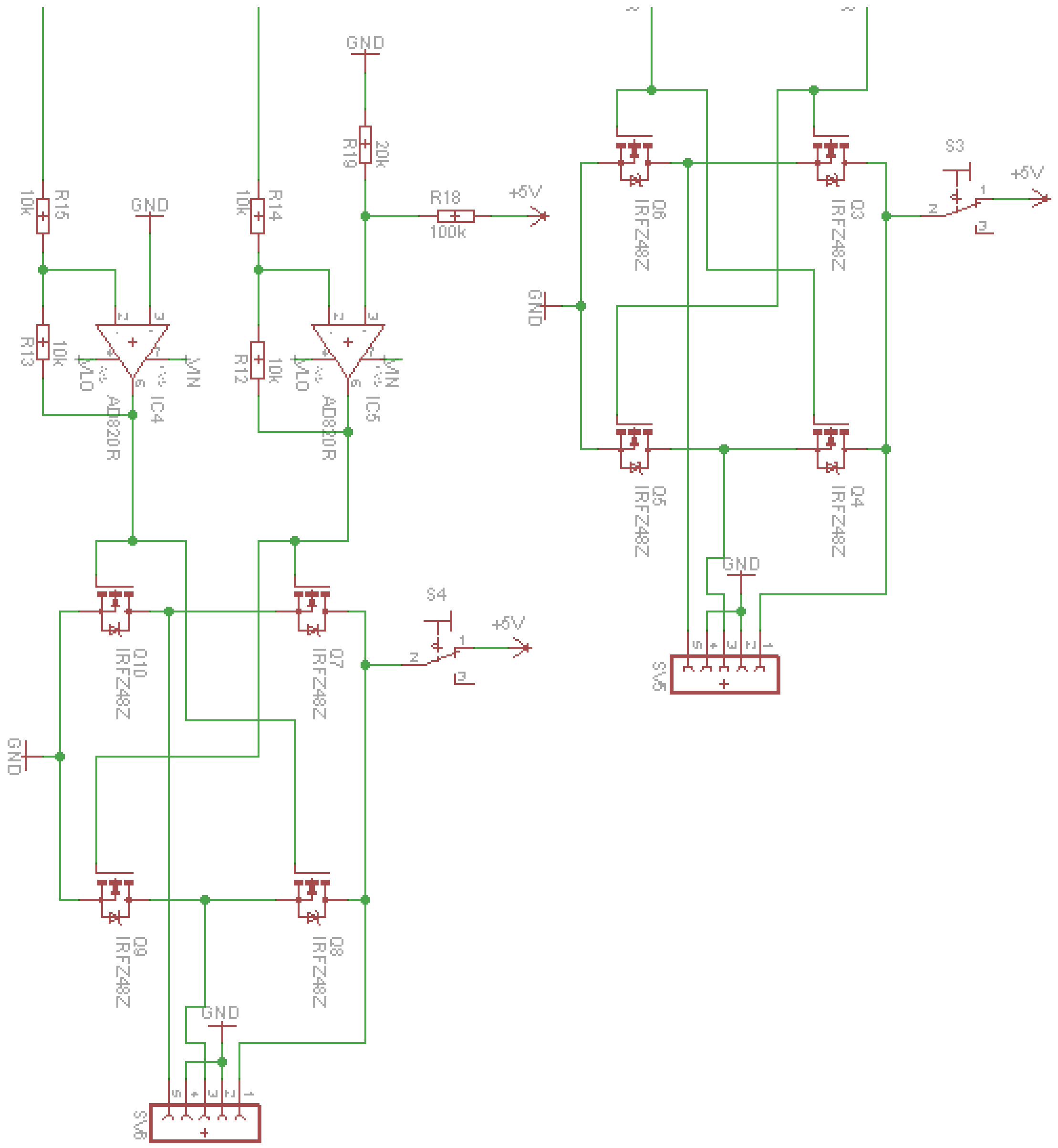
Schematics

Motor-Microcontroller

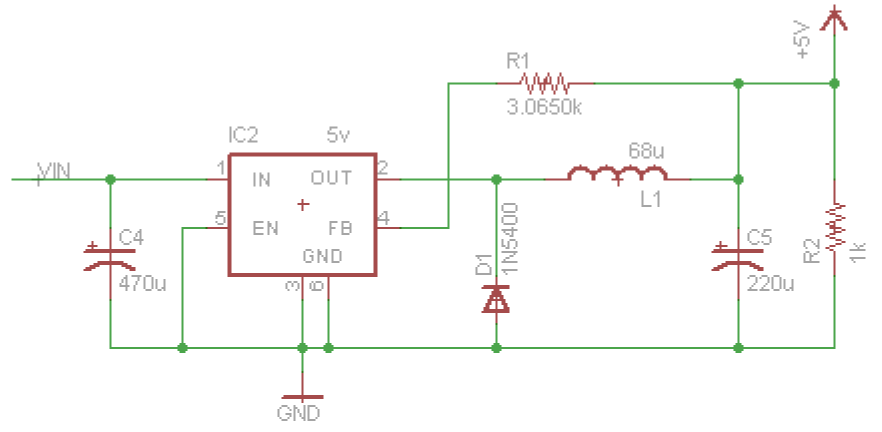




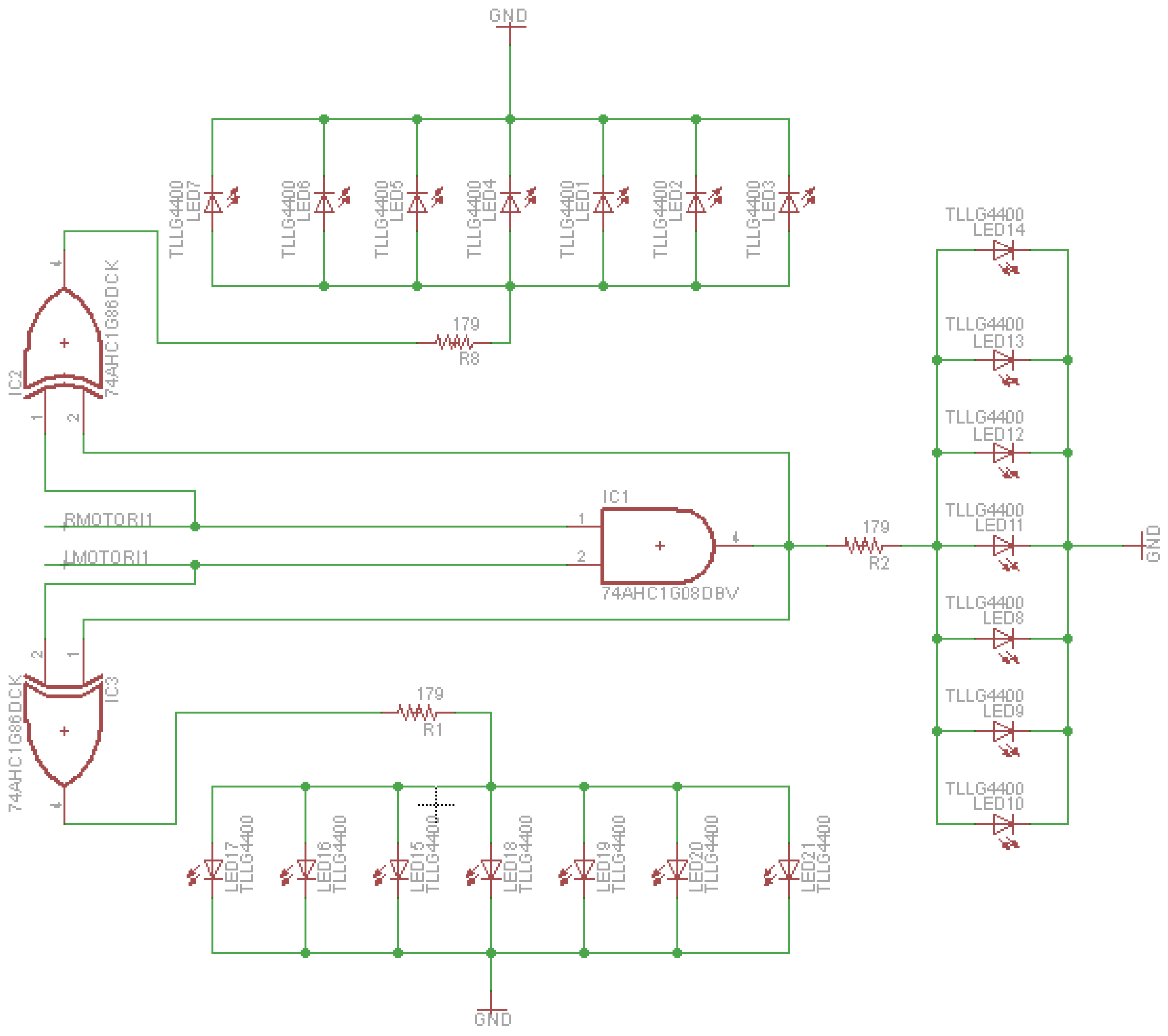
IC2
MEGA8-AI
C7
C6
22p
Q1
16Mhz
+5V
GND
AREF
RESET
R3
10k
S1
RDT1034
R1
R2
1k
SV1
SV2
SV3
SV4
C9
100n
R8
R9
R10
R11
R16
100k
R17
20k
10k
IC9
IC10
AD820R
VIN
VLO
VCCUSB


GND
R19
20k
R18
100k
+5V
R15
10k
R14
10k
R13
10k
R12
10k
IC4
AD820R
IC5
AD820R
Q6
IRFZ48Z
Q3
IRFZ48Z
Q5
IRFZ48Z
Q4
IRFZ48Z
S3
S4
SV5
SV6
Q10
IRFZ48Z
Q7
IRFZ48Z
Q9
IRFZ48Z
Q8
IRFZ48Z

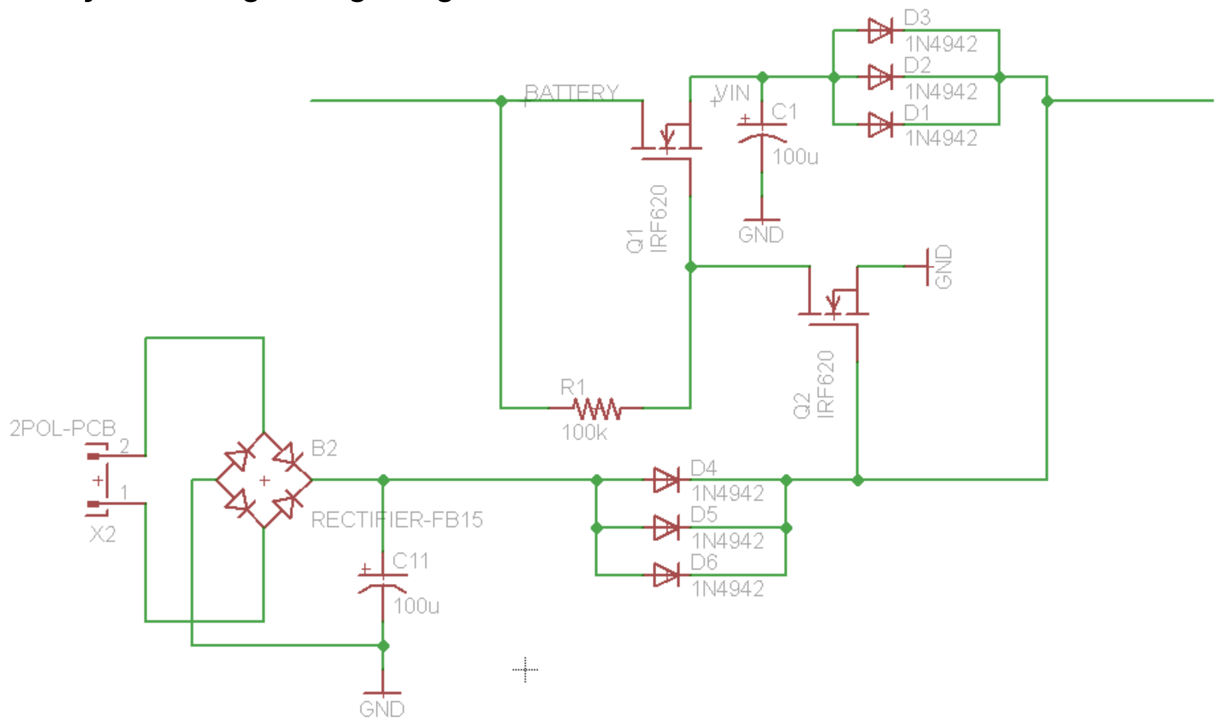
Beagleboard Switching Regulator



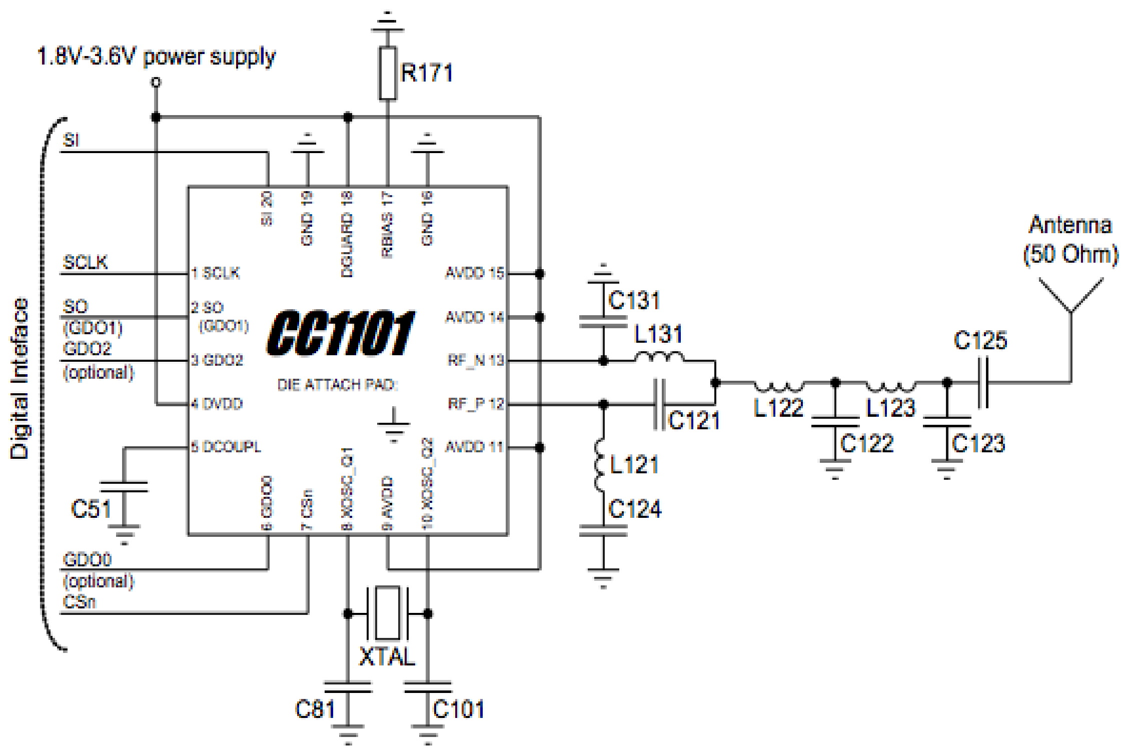
LED Network



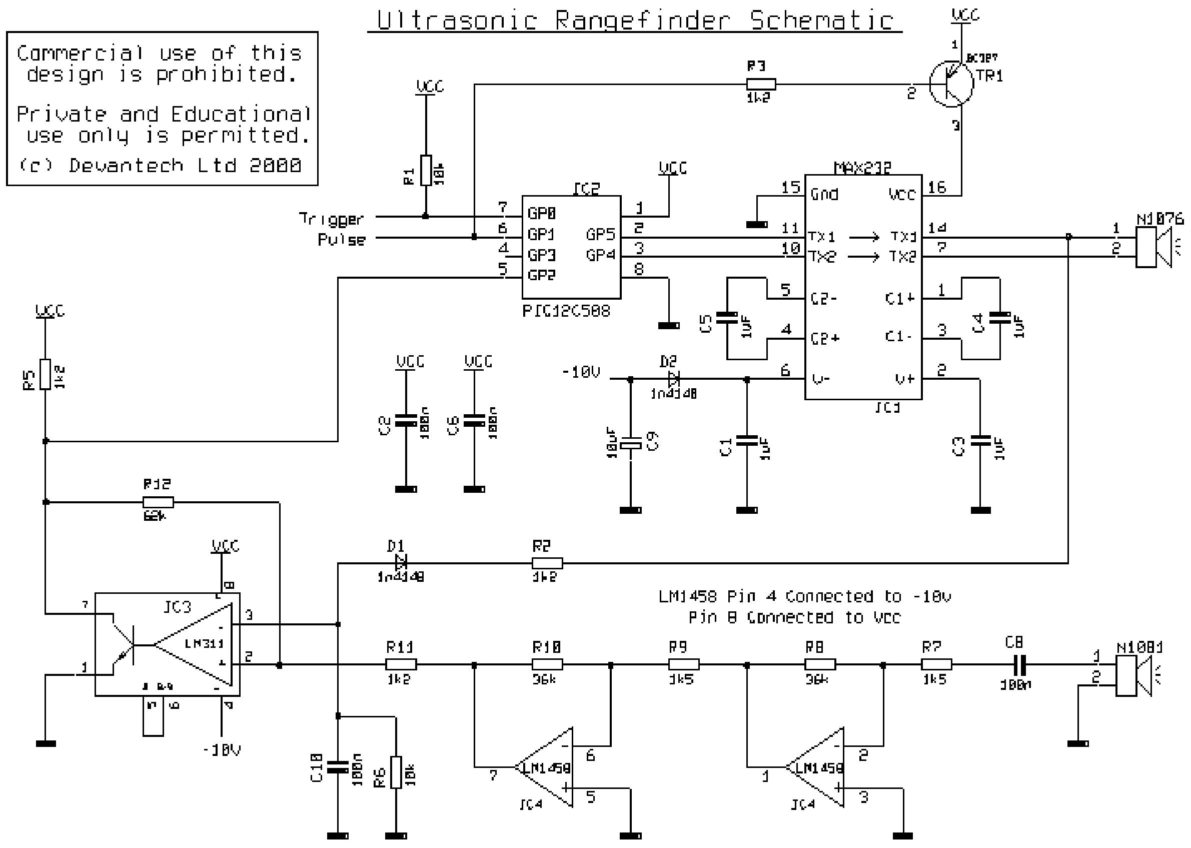
Battery Switching Voltage Regulator



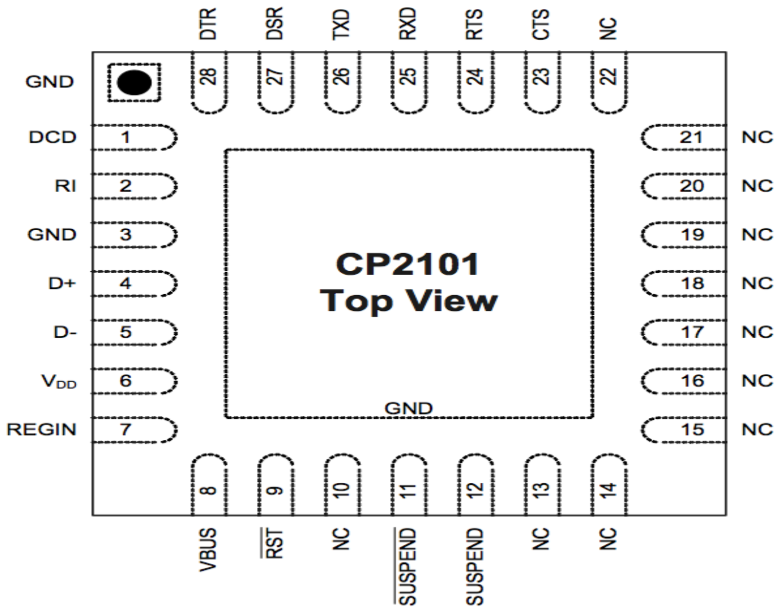
RF Transceiver



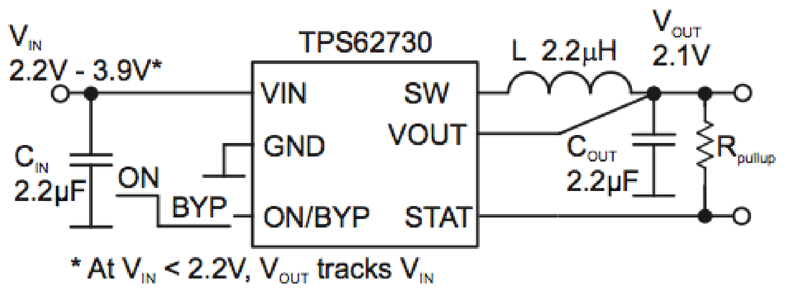
Ultrasonic Sensor


Ultrasonic Rangefinder Schematic
Commercial use of this design is prohibited.
Private and Educational use only is permitted.
(c) Devantech Ltd 2000
Trigger
Pulse
PIC12C508
MAX232
BC337
TR1
LM311
LM1458
LM1458 Pin 4 Connected to -10v
Pin 8 Connected to Vcc
N1076
N1081
-10V

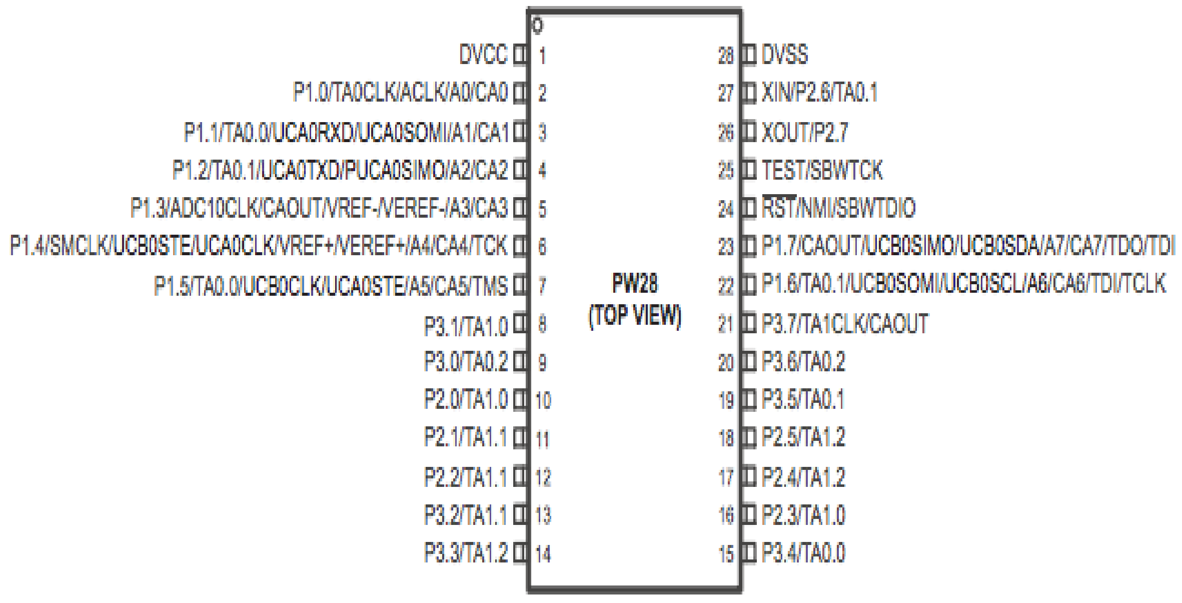
UART to USB Bridge



Step Down Voltage Regulator



MSP340 Microcontroller



DVCC 1
P1.0/TA0CLK/ACLK/A0/CA0 2
P1.1/TA0.0/UCA0RXD/UCA0SOMI/A1/CA1 3
P1.2/TA0.1/UCA0TXD/PUCA0SIMO/A2/CA2 4
P1.3/ADC10CLK/CAOUT/VREF-/VEREF-/A3/CA3 5
P1.4/SMCLK/UCB0STE/UCA0CLK/VREF+/VEREF+/A4/CA4/TCK 6
P1.5/TA0.0/UCB0CLK/UCA0STE/A5/CA5/TMS 7
P3.1/TA1.0 8
P3.0/TA0.2 9
P2.0/TA1.0 10
P2.1/TA1.1 11
P2.2/TA1.1 12
P3.2/TA1.1 13
P3.3/TA1.2 14
28 DVSS
27 XIN/P2.6/TA0.1
26 XOUT/P2.7
25 TEST/SBWTCK
24 RST/NMI/SBWTDIO
23 P1.7/CAOUT/UCB0SIMO/UCB0SDA/A7/CA7/TDO/TDI
22 P1.6/TA0.1/UCB0SOMI/UCB0SCL/A6/CA6/TDI/TCLK
21 P3.7/TA1CLK/CAOUT
20 P3.6/TA0.2
19 P3.5/TA0.1
18 P2.5/TA1.2
17 P2.4/TA1.2
16 P2.3/TA1.0
15 P3.4/TA0.0
PW28
(TOP VIEW)