# Chassis B

Prepared for:
Dr. Kevin Wedeward
Dr. William Rison

Written by:
Paul Ferrell
Adam Milner
Josette Turner
Jason Oberling

May 6, 2002

## Abstract

This paper describes in detail the robot developed by Team B Chassis group for the Junior Design class at New Mexico Tech. This was one part of a four part project. The complete four part robot was to autonomously find a golf ball and transport it to a hole. The piece described here involved most of the physical part of the robot, as well as its power system. This project was not completely finished in time, but the results of what was accomplished are promising.

# Table of Contents

## Introduction

We were given the task of building the chassis for an autonomous golfing robot. It was to carry the parts of three other groups, provide them with power, intelligently move them to their desired location via commands received from them, pick up a golf ball, and drop it accurately on command. We have had varying amounts of success.

The physical design and operation of the chassis was by far the most demanding piece. It is fast, powerful, aesthetically pleasing, and accommodates all of the groups with room to spare. It also needed to be sturdy, and move accurately. It has neither of these attributes.

The power system is robust and reliable. It has power to spare, and can run the entire robot with all parts attached for more than an hour. Low power and high power are separate and optically isolated systems.

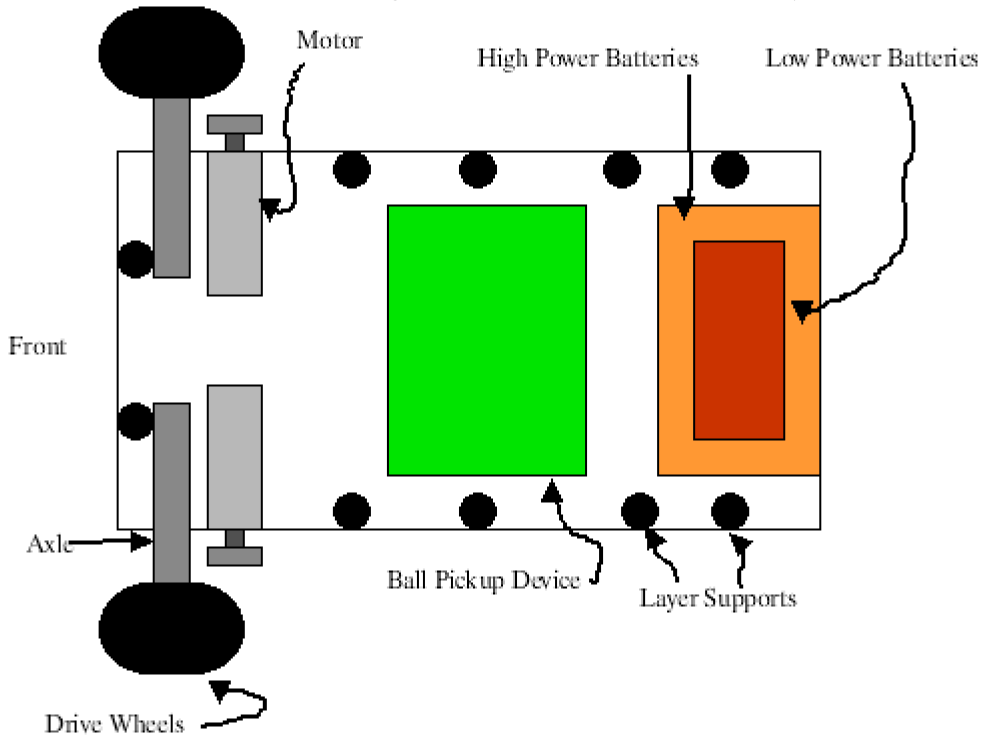
The general electronics have been well designed, and perform there function as needed. They include an H-Bridge board, and the ball pickup and high power control board.

A variety of programs were developed to run and test our robot. The main program is short, complex, and functional. Testing programs were also developed to debug various systems and allow the chassis to operate as a stand alone part. They all work as well.

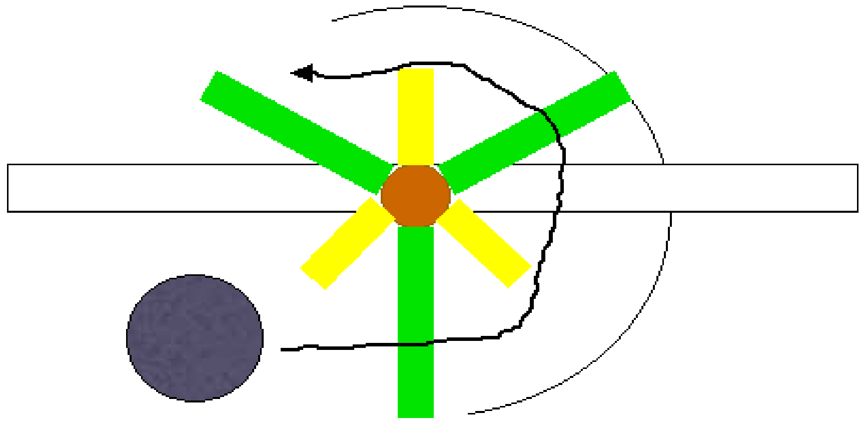
## Chassis

Our chassis is composed of four layers of Plexiglas seperated by supports made of all-thread aluminium rod. This gave us a very flexible design that could be expanded very easily. The use of Plexiglas reduced the chance of a short, but it did increase the amount of electrical noise on the robot.

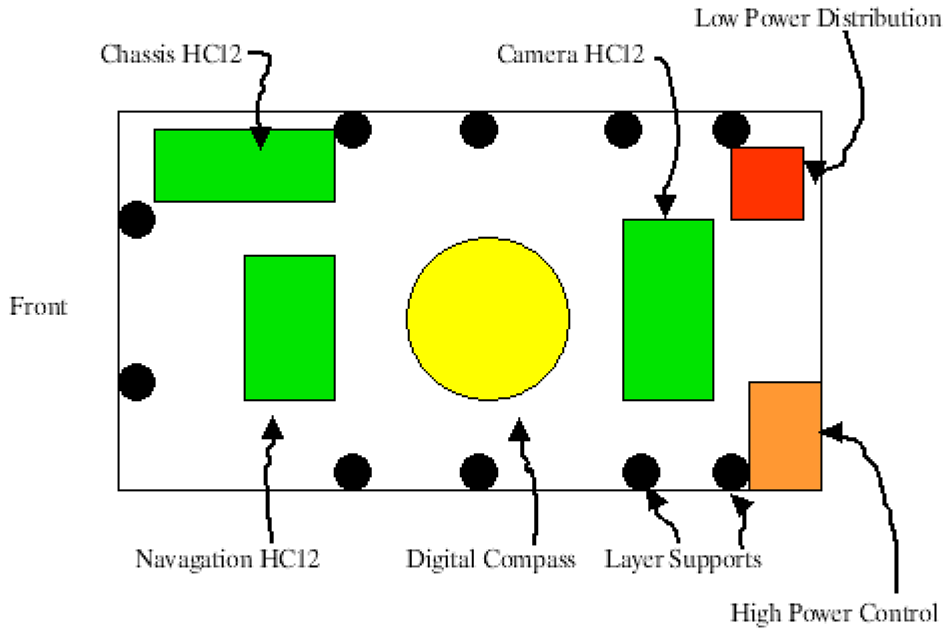
The first layer had the wheels, motors, ball pickup device and batteries mounted on it. This gave us a low center of gravity, increasing stability. The wheels were fixed to independant axles and were driven by a chain and sprocket system. The axles were mounted underneath the first layer to give the robot enough clearance to be able to drive over a golf ball. The drive wheels were off road RC truck wheels 5" in diameter. In the rear are coaster wheels also mounted under the first layer. The batteries were mounted in the read to balance out the weight of the motor/wheel assembly in the front.



The ball pickup device consisted of a revolving agitator brushing against a metal track. The agitator was driven by a motor and a pair of interlocking cogs. A golf ball would be brushed up by the agitator against the metal track and dropped off on top of the first layer.



The second layer held the majority of the electronics for the robot. The chassis control HC12, h-bridge board, high power control board, low power distribution board, the navagation HC12 and digital compass were all mounted on this layer.



The third layer is only half a large as the first and second layers and had the communications HC12 and transceiver. The final layer had the three ball/hole detection cameras mounted on it and the GPS receiver. This layer was four feet above the ground.

## Drive Motor Selection

Our design called for two drive motors to provide differential steering. In order to decide which motors to use, it was necessary to estimate the maximum weight of the robot, the drive wheel diameter, maximum speed and the maximum slope that the robot would need to climb.

The final estimates were as follows.
Mass: 15kg
Maximum slope: 15deg
Wheel Diameter: 15cm
Maximum Speed: 0.5m/s

From this it was possible to determine the minimum characteristics needed for our motors, as follows:
Speed: 200RPM
Torque: 12in-lb

However after measuring the slopes at the chipping green, it was discovered that at one place the maximum slope was ~30deg. This increased the needed per motor torque to 24in-lb.

We decided to use the Matsushita GMX-6MPO13A motor, because it was capable of providing 32in-lb @307RPM, the built in shaft encoder, and its price. The downsides to this motor were its size and large power requirements. However, the H-Bridges we had chosen were capable of supplying the stall current safely and the substantial torque provided allowed us to use larger batteries if it became necessary.

## H-Bridges

We used the National Semiconductor LMD18200T H-Bridge to interface between our HC12 and the drive motors. The LMD18200T was selected because it can handle 3A continuous, 6A peak at 24V, which is better than our motors required. Cost was not a factor as we were able to acquire several as samples. Because of the need for isolation between the low and high power systems, optical isolation was needed on the PWM and direction bits. We decided to use the Fairchild H11L1 optoisolators for this because they have a built in Schmitt trigger. However, since the H11L1 is an inverter, it was necessary to invert the signal before sending it to the H11L1. This was done using a standard 7804 hex inverter. It was also decided that it would be nice to have visual feedback on the signal being sent to the motors, so a pair of LED's per motor was used to display the PWM and direction signal.

## Ball Pickup Control

The ball pickup motor only needed to go in one direction, so a full H-Bridge was not necessary. We decided to use a simple MOSFET switch, using an IRF711 power MOSFET. Once again, optical isolation was needed because the pickup motor was running off high power, so a H11L1 optoisolator was used. This time however, the signal inversion was done in software.

## High Power

The high power system for the robot consisted of two 12V, 5 A-hr SEC 12-5 MICROLYTE Plus lead acid batteries wired in series to give a final voltage of 24V at 5 A-hr. The purpose of these batteries is to power the motors for movement and ball pickup. We chose lead acid batteries because of their high A-hr rating and relatively low cost. We purchased these batteries from Herbach and Rademan, an Internet surplus company, for $15.95 each. To charge these batteries, we simply bought a generic 12V battery charger. It was designed for a car battery, but it worked just fine for our purposes.

The obvious advantage of these batteries is their high A-hr rating. This rating gave our high power system around one and a half hours of runtime. This was possible because our motors only draw around 1.5A each, with a stall current of 4.5A, and the draw for our ball pickup motor was around 0.25A.
The biggest disadvantage is their weight. Each battery weighs between eight and ten pounds, making the total weight for batteries alone between sixteen and twenty pounds. This problem is easily solved with proper weight balance on the chassis, but if weight is a serious consideration then these are not the best batteries.

## Low Power

The purpose of the low power system is to supply the robot with clean, regulated power. This system consists of one 7.2V, ~1.5 A-hr NiCd battery pack, and one power distribution board to supply regulated power throughout the robot. We considered NiMH and Li-ion batteries, but decided on NiCd batteries because they are easy to find cheap due to the popularity of RC cars. This system was also optically isolated whenever we had low and high power on the same board. This was done to keep high power noise from affecting the low power system.

The power board was designed and implemented by us. We used two 5V regulators and six standardized, polarized connectors. The regulators were Texas Instruments REG104FA-5 1A low dropout 5V regulators. We ordered them through TI's free sample program, so had to pay nothing for them. Each regulator drove a row of three connectors, which were three-pin, standardized connectors. On the board itself, pin one was +5V, pin three was ground, and pin 2 was not connected.

The advantages of this system are that it supplies clean power to the robot, due to optical isolation of the two power systems. Noise generated by our motors cannot reach our low power because of this isolation. The NiCd battery also supplies approximately one hour of runtime for the entire robot. It is also easy and relatively cheap to implement. The regulators are inexpensive at $2.22 each, and the battery pack cost about $15.00, and came with a charger.

## Software Design

The software end of our project involved a variety of programs. The primary program was responsible for motor control, communications, and command decoding. Other small programs were developed to test communications on both the receiving and sending ends.

## Main Program

The main program is the only program needed by the chassis to control itself when fully integrated. Most of its features are described below, and here are some more. The main program is small. At less than ¾ of a kilobyte in size, it fits into byte erasable memory. This allows us to remove power without removing the program without the full complications of Flash memory. Most of the code is very general. You could control four motors instead of two, have a large queue of commands, and more with only a few simple changes.

There are some disadvantages to the way it was programmed as well. The size constraint was a serious limitation. If it were to be done again we would not worry about size and use Flash from the beginning. A more robust distance measuring system is needed to compensate for the imperfect machining so that it accurately moves the distance it is supposed to.

## Command Protocol and Decoding

In order to accommodate any sort of action the other groups needed to take, command protocol was set so that we could accommodate any sort of command they needed to send to us. It was later decided that this complexity was unnecessary; all we had to do was move.

In fact, all the chassis had to do was turn, and then move forward. Backwards movement was not desired. The final result was that once a command was received, the robot would turn the amount desired, and then move forward a limited distance. The robot would always be stopped when it was without instructions. When a set of movements has been completed, a ready line that connected to Navigation would go high signaling that we are ready for the next set of data.

The command protocol was quickly reduced to basic one command that is four bytes long. The first byte received is the distance that the chassis needs to move forward. The second is a general-purpose byte, and only two bits of it are used. The one bit (or "who bit") tells us who is telling us to move. The zero bit allows us to use 360° of accuracy for turning. The third byte received is how much we need to turn in degrees. The final byte is for error checking.

Once the distance is received, it is converted into the number of encoder ticks that would correspond to traveling that distance. There are actually two different units of distance that we use for traveling forward. For Navigation we travel forward in units of three centimeters. For camera group, we travel at an accuracy of one centimeter. In other words, a 0x66 from Nav. group tells us to go forward 3 meters, while the same from Camera group will only move you a meter. The "who bit" also allows us to set a different speed for each group. Navigation travels at maximum speed, while Camera group clunks along at about one-third speed.

Turning is a bit more complicated. In order to get 360° of accuracy the extra bit from the general-purpose byte is used. If it's on, we simply add 256 to the turning byte. After quickly deciding which direction to turn and adjusting the value accordingly, the program converts it into a number of encoder ticks as it did with the distance variable.

The final byte allows the robot to make sure it is doing what it is supposed to. It's always a 0xC4, and that particular value is never sent to the chassis except as the closing byte of a chassis communication.

The decoded commands are organized into an array of structs so that they can be stacked and accessed using a non-redundant set of code. Moving forward, turning, and the idle stop commands are all just elements in this array, and are handled in exactly the same way. The struct could easily be expanded to allow for queue of commands, but it was not needed for the final design.

The command protocol and decoding code is present in both the main program, and the slave test program.

## Communications

Communications utilized the standard Serial Peripheral Interface (SPI) of the HC12. We only received commands, so this part of our robot was fairly simple. Using the SPI interrupt, the program receives the bytes one at a time, and shifts them into an array. When it finds the 0xC4 byte as stated before, the command decoding sub-program is called.

Communications is simple, as long as you don't care about how well it works. We cared, so a great deal of time was spent getting it to work properly. The biggest problem seemed to be noise on the ready line. After long hours attempting to fix it, it was deduced that the problem was a interrupt timing issue having to do with how the ready bit was set.

The actual setup of this system is detailed in the main program comments. The communications code is present in the main program as well as the slave test program.

## Slave Test Program

The slave test code was used for basic debugging of communications for both stand alone purposes and integration. As stated earlier, it contains the command decoding code as well as the communications code. Its purpose is to allow for easy communications debugging without the complications of the main program. Dbug12 printf statements are used to easily check accuracy. This program was an invaluable tool, and would be more so if we ever fully attempted to get the entire robot integrated and working.

## Master Test Program

This program simulates communication with Navigation and allows the chassis to operate as a stand-alone component. It uses the real time interrupt to send bytes to either the slave test program or the main program. An array in the program contains a list of commands to send to the chassis. The program is relatively simple, and will not be gone over in detail.

## Motor Control

The motors are controlled using the input capture (IC), output capture (OC), pulse width modulation (PWM), and real time interrupt (RTI) sub-systems. The program controls their speed, limits their acceleration, is zero speed error proof, and has a smooth transition between positive and negative speeds.

The current speed is determined by the IC and OC systems. The input capture system finds the period between rising edges of the encoder for each of the motors. If the capture timer goes for more than one full cycle (which would cause random periods to be read), the output capture system catches it, and sets the read period to its max value. The end result is the availability of a constantly updated variable for each motor that contains the period of each encoder pulse divided by the timer clock period (4 μs) (This variable is called CyclsPrPls).

The PWM system controls the power sent to the motors via its duty cycle. This method is common knowledge and will not be detailed.

Every 65 ms, the RTI adjusts the speed. The current speed is found according to this formula:

$$\frac{10000}{Cycls\,\mathrm{Pr}\,Pls} = \frac{(4\mu s * 1250\,pulses\,/\,revolution)^{-1}}{Cylcs\,\mathrm{Pr}\,Pls} * 100 = \text{Current Speed.}$$

The speed is controlled by accumulating the adjusted error in the variable that controls how much power is sent to the motors. If the error is zero, no adjustment occurs. If the error is high, the PWM duty cycle would change drastically. The amount the PWM duty cycle can change is limited, however, therefore limiting how much the speed can change in a given amount of time. This acceleration controls gives the robot a theoretical zero to max speed time of 1.3 seconds, and can be easily adjusted.

The motors have a controlled zero speed, and can smoothly transition between forwards and backwards wheel movement.  Essentially, this is done by making the desired and current speeds signed variables, and then translating the result of the speed calculations into actual motor control.  The sign of each of the variables is contained in a separate variable and applied when needed.  As a the motor speed crosses the zero boundary, the direction bit for the H-bridges is flipped, causing all power to be applied in the opposite direction.  This eliminates jerky movement in the motors.  This combined with the zero speed determination form the output capture system allows the motors to attempt to counteract any attempt to move them  when they don't want to be moved.

## Conclusion

This has been an interesting experience. Doing something new without a prior idea of what works well physically was the most challenging part. We've learned how to be quick, accurate, and get things done right the first time. Communications among four group members is hard enough, and doing the same with a fourteen member team caused serious setbacks. This was a great experience. The chances of the entire thing being successfully completed were horrible.

I feel that we were generally successful with our part. While it isn't going to be picking up or dropping of a ball anytime soon, it's never going to find it either. It doesn't drive too well, but project-wide communications were never fully realized either. In contrast, software, electronics, and the power system work really well. Too bad all it can do is drive into walls.

## Budget

| | Quantity | Cost Ea. | Cost Total |
|---|---|---|---|
| SEC 12V Battery | 2 | $15.95 | $31.90 |
| Gearhead Motor | 2 | $29.95 | $59.90 |
| NiCad Battery | 1 | $14.95 | $14.95 |
| HC 12 | 1 | $0.00 | $0.00 |
| H-Bridge | 2 | $0.00 | $0.00 |
| Plexiglass | | | |
| 1/4 inch | 1 | $30.00 | $30.00 |
| Misc Hardware | | $50.00 | $50.00 |
| Wheels | | | |
| caster | 2 | $5.00 | $10.00 |
| drive wheels | 2 | $25.00 | $50.00 |
| Chains & Sprockets | | $55.92 | $55.92 |
| Misc Electronics | | $10.00 | $10.00 |
| Battery Charger | 1 | $20.00 | $20.00 |
| Pick up Motor | 1 | $0.00 | $0.00 |
| Voltage Reg (5V) | 1 | $0.00 | $0.00 |
| Total Cost | | | $332.67 |
| Budget Asked for | | | $250.00 |

## Power Budget

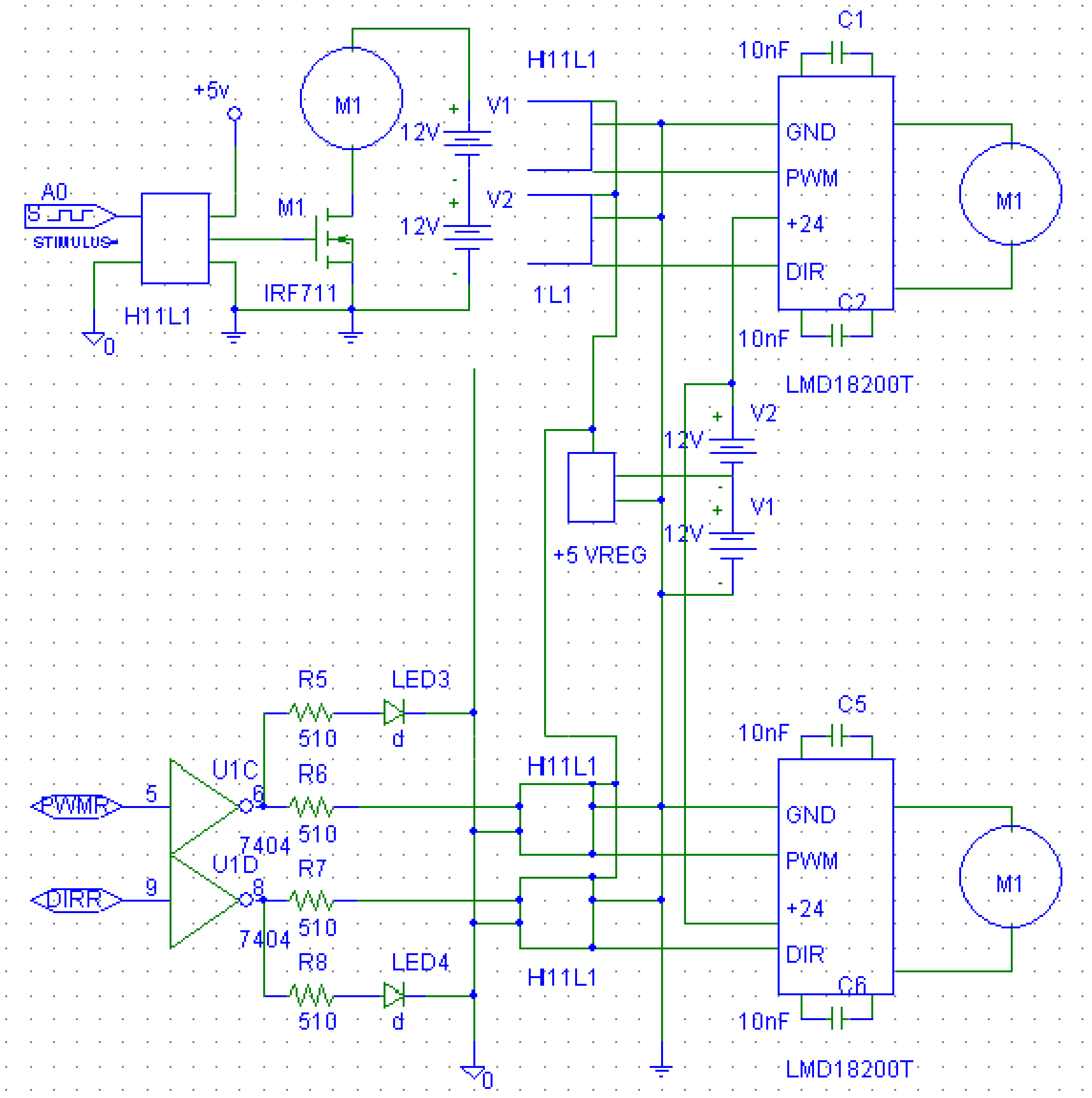
### High Power

| Part Description | Current Draw | Quantity | Total Draw |
|---|---|---|---|
| Drive Motor (24V) | ~1.5 A | 2 | ~3 A |
| Ball Pickup Motor (12V) | .5 | 1 | .5A |
| Total | | | ~3.5A |

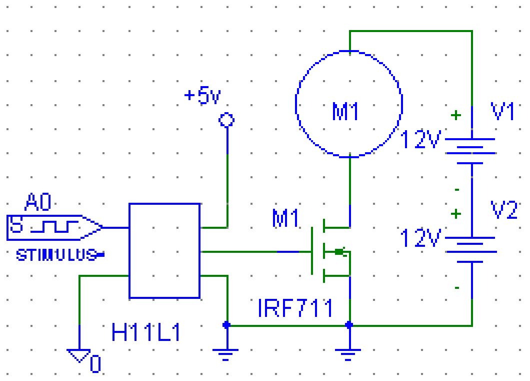
### Low Power

| Part/Module Description | Current Draw |
|---|---|
| HC12 | ~ 100 mA |
| Navigation | ~ 400 mA |
| Communication | ~ 120 mA |
| Ball/Hole Location | ~ 500 mA |
| Total | ~ 1.12 A |

## Circuit Diagrams



Drive Motor H-Bridge



Pickup Motor Control

## HC12 Source Code

```
        // Port Usage
        // PortT 0: Left Motor Encoder Input
        // PortT 1: Right Motor Encoder Input
        // PORTB 0 & 1: Left & Right Motor Encoder Input
        // PWM Port 2 & 3: L & R Motor PWM

        #include <hc12.h>



        void GetSpeed(char M);
        void SetComm();

        void rti_isr();  /* General Motor Control */
        void ic0_isr();  /* Left Motor Encoder decoder */
        void ic1_isr();  /* Right motor encoder decoder */
        void oc4_isr();  /* Left Motor Zero Speed check */
        void oc5_isr();  /* Right Motor Zero speed check */
        void spi_isr();  /* Communications reciever */

        #define VSt 0x0b10 //Start of Vector Table
        #define T5V 0x14  //Relative Positions from VSt
        #define T4V 0x16  //For these interrupts
        #define T2V 0x1A
        #define T1V 0x1C
        #define T0V 0x1E
        #define RTV 0x20
        #define SPV 0x08

        #define TR 26     //#of ticks per degree turn
        #define ACCELMAX 12 //max change in PWM duty per update cycle(65ms),
currently set to ~5%
        #define TKPCM 71  //#of ticks per centimeter straight movement


        volatile unsigned char BComm[4]; //raw communication data
        volatile signed char CurrCmnd;  //pointer to our current subcommand in the queue
        volatile unsigned int DistMvd;  //number of ticks we have moved in current
subcommand

        volatile struct Command  //this is what a single subcommand looks like, a
direction for each wheel, a speed and a distance
        {
         volatile signed char MDirL;
         volatile signed char MDirR;
```

```
 volatile unsigned int  Dist;
 volatile unsigned char Spd;
} Cmnd[3];

volatile struct MCtrl // this is all the control data about a single motor
{
 volatile unsigned int FirstPulse; /*Time of the first of two pulses */
 volatile unsigned int CyclPrPls;  /* Timer Cycles between pulses */
 volatile unsigned char * PWDTY_;
 volatile unsigned int * IC_;
 volatile unsigned int * OC_;
 volatile signed char CurrDir; /* Current Motor Direction (L,R) */
 volatile signed char DesDir;  /* Desired Motor Direction (L,R) */
} Motor[2];

/* Speed Setting */
volatile unsigned char DesSpeed = 0;  //we should start not moving

main()
{
 COPCTL = 0;

 // Setting up interrupt vector table
 *((void (**)())(VSt + T5V)) = oc5_isr;
 *((void (**)())(VSt + T4V)) = oc4_isr;
 *((void (**)())(VSt + T1V)) = ic1_isr;
 *((void (**)())(VSt + T0V)) = ic0_isr;
 *((void (**)())(VSt + RTV)) = rti_isr;
 *((void (**)())(VSt + SPV)) = spi_isr;

 // Defining the location of the registers in the struct
 Motor[0].PWDTY_ = &PWDTY2;
 Motor[1].PWDTY_ = &PWDTY3;
 Motor[0].IC_ = &TC0;
 Motor[1].IC_ = &TC1;
 Motor[0].OC_ = &TC4;
 Motor[1].OC_ = &TC5;


 //Do nothing command
 Cmnd[2].MDirL = 1;
 Cmnd[2].MDirR = 1;
 Cmnd[2].Dist = 0;
 Cmnd[2].Spd = 0;
 CurrCmnd = 2;
	/* SPI enabled, SPI interrupt enabled, Slave Mode, MSB first */
	SP0CR1 = 0xCC;

	/*Port B Input, 0 and 1 are encoder bits*/
	/*Port A is output, 2 and 3 are left and right direction bits, 7 in comm ready bit*/
	DDRB = 0x00;
	DDRA = 0xFF;

	/* Setting up PWM (2 & 3), (L & R) to control motor */

	/* Setting PWM 3 & 2 to 8 bit mode, and Clock B = 250kHz */
	// 00000101
	PWCLK = 0x05;

	/* Setting polarity to high for 2 & 3 */
	// Setting clock to B for PWM 2 & 3
	// 00001100
	PWPOL = 0x0C;

	/* Setting allignment to left */
	// 00000000
	PWCTL = 0x00;

	/* Select period of 200 for PWM 3 & 2 (Final f = 1000 Hz ) */
	PWPER3 = 249;
	PWPER2 = 249;
	PWDTY3 = 0;
	PWDTY2 = 0;

	/* Enable PWM 3 & 2 */
	PWEN = 0x0C;

	/* Setting up timer */
	TSCR = 0x80;			/* Enabling timer */
	TMSK2 = 0x05;  /* Setting pre-scaler to 5, 4us per tick */

	/* Setting up pin 0 & 1 (L & R) input capture interrupt */
	TIOS = 0x30;		/* Makes pins 0 & 1 & 2 input capture, 4 & 5 output
compare */
				/* Output compare pins 4 & 5 do not effect cooresponding output
pins */
	TCTL4 = 0x15;		/* pin 0, 1, 2 captures on rising edge */
	TMSK1 = TMSK1 | 0x37; /* Enable ECT 0 - 2, 4, 5  interrupts */
	TFLG1 = 0xFF;		/* Clear Interrupt flag */

	/* Setting up RTI */
```

```
            RTICTL = 0x87; /* RTI Enabled, occurs every 65 ms */
            RTIFLG = 0x80; /* Clearing RTI flag */

            enable();        /* Enable interrupts */

           while(1)  //this is the heart of it, once everything is setup, we sit in this loop
forever processing commands
            {
             DesSpeed = Cmnd[CurrCmnd].Spd;           /* Set Desired speed */
             Motor[0].DesDir = Cmnd[CurrCmnd].MDirL;  /* Set Desired Direction (1 =
forward) */
             Motor[1].DesDir = Cmnd[CurrCmnd].MDirR;  /* ( -1 =  Reverse ) */
             while (DistMvd < Cmnd[CurrCmnd].Dist);   // wait for the robot to finish its
current subcommand
             if (CurrCmnd == 2) PORTA = PORTA | 0x80; // tell the sender that we have
finished all subcommands
                                        // and are ready to recieve another set
             while (CurrCmnd == 2);                  // wait until we recieve another command
             DesSpeed = 0;                          // slow down to a stop
             while (Motor[0].CyclPrPls != 0xFFFF);    // wait for both motors to stop
             while (Motor[1].CyclPrPls != 0xFFFF);
             ++CurrCmnd;                            // goto the next subcommand
             DistMvd = 0;                          // reset the distance counter
            }
           }

           /* This subroutine sets the speed by altering the duty cycle of the PWM output */
           @interrupt void rti_isr()
           {
            signed int CurSpeed;
            signed int NewDty;
            signed int DtyChng;
            unsigned char M;

            for( M = 0; M <= 1; ++M) //we are going to do this for both motors
            {
             if (Motor[M].CyclPrPls != 0)
             {
              CurSpeed = (unsigned int)(10000/Motor[M].CyclPrPls);
              DtyChng = (signed int)((Motor[M].DesDir * DesSpeed - Motor[M].CurrDir *
CurSpeed)/3);
              if (DtyChng > ACCELMAX) DtyChng = ACCELMAX;   //this puts a hard cap
on how fast we can accelerate
              if (DtyChng < -(ACCELMAX)) DtyChng = -(ACCELMAX); //this way we can
avoid slipping during acceleration
              NewDty = (signed int)(DtyChng + *Motor[M].PWDTY_ * Motor[M].CurrDir);
              if (NewDty < -254) NewDty = -254; //PWM duty is a char, so we do this to
keep from overflowing
              if (NewDty > 254) NewDty = 254;
              if (NewDty < 0)  /* Moving Backwards */
              {
               NewDty = NewDty * -1;
               PORTA = PORTA & (0xFB - 0x04 * M);  /* Clears Direction Bit
(0=reverse)*/
              }
              else /* Forwards */
               PORTA = (PORTA | (0x04 + 0x04*M));   /* Sets Direction bit for the correct
motor */

               *Motor[M].PWDTY_ = (unsigned char)NewDty;
             }
            }
            RTIFLG = 0x80;
           }

           /* LEFT MOTOR */
           /* This interrupt does the following: */
           /* Utilizes the GetSpeed function to find the speed for the left motor */
           /* Determines Direction via PORTB 0 */
           @interrupt void ic0_isr()
           {
            Motor[0].CurrDir = (signed char)(1 - ((PORTB & 0x01) * 2)); /* Checks Left
motor Direction */

            GetSpeed(0);

            ++DistMvd; //we are checking distance on only the left motor

            TFLG1 = 0x01;  /* Clears Interupt flag 0 */
           }

           /* RIGHT MOTOR */
           /* This interrupt does the following: */
           /* Utilizes the GetSpeed function to find the speed for the right motor */
           /* Determines Direction via PORTB 1 */
           @interrupt void ic1_isr()
           {
            Motor[1].CurrDir = (signed char)(1 - (PORTB & 0x02)); /* Checks Right Motor
Direction */

            GetSpeed(1);
```

```
 TFLG1 = 0x02;  /* Clears Interupt flag 1 */
}

/* Stores the first pulse time in a global variable,*/
/* and finds the time difference of the second pulse. */
void GetSpeed(char M)
{
  Motor[M].CyclPrPls = *Motor[M].IC_ - Motor[M].FirstPulse;
  Motor[M].FirstPulse = *Motor[M].IC_;
  *Motor[M].OC_ = *Motor[M].IC_;
}

/* This allow for an approximately zero speed for each motor */
@interrupt void oc4_isr()
{
 Motor[0].CyclPrPls = 0xFFFF;
 TFLG1 = 0x10;
}

@interrupt void oc5_isr()
{
 Motor[1].CyclPrPls = 0xFFFF;
 TFLG1 = 0x20;
}

@interrupt void spi_isr()
{
 char I;

 for (I=1; I<4; I++) BComm[I-1] = BComm[I]; //move bytes down list
 I = SP0SR; //get SPI regiset ready to send data
 BComm[3] = SP0DR; //read new byte


 if (BComm[3] == 0xC4 ) { //check for magic number indicating complete
command
  PORTA = PORTA & 0x7F;// This lets us turn off A7 (the comm ready bit)
ASAP so as to avoid confusing the master
  SetComm();
 }
}

void SetComm(void) //this function translates from Navagation style commands
to our internal system
{
 int Angle;
```

```
        char Unit = (BComm[1] & 0x02);

        //A complete command consists of a turn, a forward and stop
        //We have already set the stop, so we just need to set the turn and the forward

        //Turn Sub Command
        Angle = ((BComm[1] & 0x01) * 256 + BComm[2]);
        Cmnd[0].Spd = 75;  //we turn at ~1/4 max speed
        if (Angle > (180))
        {
         Cmnd[0].MDirL = 1;
         Cmnd[0].MDirR = -1;
         Cmnd[0].Dist = (360 - Angle) * TR;
        }
        else
        {
         Cmnd[0].MDirL = -1;
         Cmnd[0].MDirR = 1;
         Cmnd[0].Dist = Angle * TR;
        }

       //Forward Sub Command
        Cmnd[1].MDirL = 1;
        Cmnd[1].MDirR = 1;
        Cmnd[1].Spd = 255 - Unit * 65;
        Cmnd[1].Dist = BComm[0] * TKPCM * (3 - Unit);
        CurrCmnd = -1;  //set the next subcommand to be done in the command loop to
be the first subcammand of the new command
        DistMvd = 0xFFF0; //tell the command loop to quit its current subcommand and
to go onto the first one of the new command


       }
```