ASSISTED-BALANCE BICYCLE

SENIOR DESIGN

FALL 2010

CYCLE 2 REPORT

12/06/2010

Daniel Dunbar, Daniel Golden, Mason Nixon, Brett Smith, and Bryan Wall

Table of Contents

I.	Executive Summary		3
II.	Bicycle Physics and Equipment Placement………………..	Bryan Wall	4
III.	Motor and Power Systems………………………………………..	Brett Smith	7
IV.	Structural Design/Sensors…………….………………………..	Mason Nixon	11
V.	Balancing Control System Theory……………………………..	Daniel Dunbar	20
VI.	Embedded System Architecture………………………………..	Daniel Golden	23
VII.	Administration………………………………………………………..	Daniel Golden	27
VIII.	Conclusion		29
IX.	Appendix		30

I. Executive Summary Mason

The most difficult part of learning to ride a bicycle for many people is creating enough momentum through pedaling in order to allow the bike to stabilize itself. Once a rider learns how to mount a bicycle and distribute his or her weight, the rest is up to the natural physics of the wheels, seat, pedals, brakes and handlebars. Imagine if this most difficult task was eased by taking away the necessity of stabilizing oneself.

Understanding the physics of a bicycle is much more in depth a problem than one might think considering how common the device is in everyday life. When a bike is falling out of balance, it is as if the bicycle is making a circle to the side of the fall, pulling the bike towards the center of that circle, sometimes referred to as the "circle of fall." In order to keep from falling into the circle of fall (i.e. crashing), the rider has to create a counter-pull out of the fall. There are two ways a rider can create this counter-pull to overcome the fall. One choice is to quickly speed up the bicycle. The other is to quickly tighten (shorten) the radius of the circle of fall. Experienced riders instinctively do both as required. What we propose to do is create the counter-pull necessary to allow for the rider to remain upright.

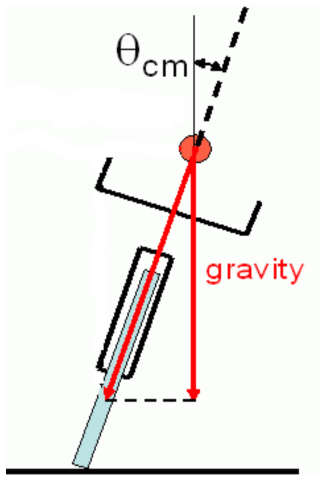
When someone learns to ride a bike using training wheels, they teach themselves how much force is required to propel themselves forward, how hard one must brake to stop, and also the general mechanics of the steering of a bike. There is a giant step from here to learn to balance oneself while accounting for those other factors once the training wheels have been removed. Our balance-assist can create the intermediary step and allow the rider to become more confident in their ability to control the bike and ultimately learn to ride the bike with no assist. We plan to have certain degrees of assist so that, as one progresses, they may be able to have less of a counter-pull automatically generated so that they may be able to compensate themselves.

Beyond teaching someone how to ride a bicycle, the balance assist also has application in rehabilitation of people with inner ear problems (which can cause a deficiency in the ability to self-balance) or other debilitating injuries and also in aiding the elderly who have lost their confidence in their ability to self-balance.

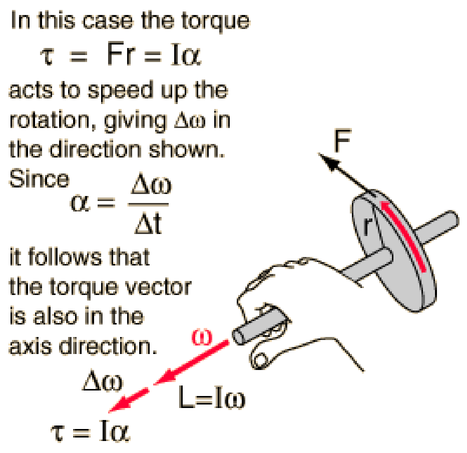
As a senior design project, this project encompasses many areas of engineering. General dynamic physics is involved in determining the motion of a bicycle. The problem of balancing a bicycle is directly comparable to the classic nonlinear control systems problem of the inverted pendulum. There is a great mechanical aspect of this problem in devising mounting methods and mass distribution of our balance assist. Lastly, there is a project management aspect of the project that involves time-management, money-management, and teamwork that are all well accounted for by our proposal.

II. Bicycle Physics and Equipment Placement — Bryan Wall

The primary principal that allows a falling object, such as a bicycle, to have its position and angular acceleration changed by an accelerating flywheel is something that is called “coupled forces”. This essentially means that as long as the flywheel is fixed to the bicycle, the moment, also called torque, which is generated by the angularly accelerating flywheel, is transferred to the angular acceleration of the bicycle which is then used to rotate the bicycle back to the vertical position. There are many variables that must be calculated to determine what angular acceleration of the flywheel is need to counteract the falling motion of the bike due to gravity including the torque created by the bicycle falling as well as the moment of inertia of the flywheel.



To calculate the torque of the bicycle falling we must first calculate the center of gravity, also known as center of mass, of the bicycle. The reason for this is because the force of the gravity pulling the bicycle down can be represented by a single force that is applied to the object at its center of mass. This is not so simple a task when trying to calculate the center of mass for an oddly shaped object such as a bicycle. However, we can easily calculate the center of mass of simple geometric shapes such as discs and rods. This is helpful due to the fact that a bicycle is essentially made up of two discs and a few rods. Therefore, the process to find the center of mass for a bicycle becomes fairly simple. All that has to be done is to find the center of mass of all the individual pieces of the bicycle. Then, those individual centers of masses can be added up using weighted averages to find the center of mass of the entire bicycle. Once that is done, the radial arm from the ground to the center of mass where the gravity acts on is known, and the torque of the falling bicycle can be calculated.


In this case the torque
$\tau = Fr = I\alpha$
acts to speed up the rotation, giving $\Delta\omega$ in the direction shown.
Since $\alpha = \frac{\Delta\omega}{\Delta t}$
it follows that the torque vector is also in the axis direction.
F
r
ω
$\Delta\omega$
$L=I\omega$
$\tau = I\alpha$

Once the torque of the falling bicycle is known, that force can then be compensated for by accelerating the flywheel in the opposite direction of the falling motion of the bicycle. The torque of the bicycle and the torque created by the flywheel are related by the equation:

$$M_b \times {R_b}^2 \times \alpha_b = M_f \times {R_f}^2 \times \alpha_f$$

M_b is mass of the bicycle, R_b is radius from the ground to the center of mass of the bicycle, α_b is the angular acceleration of the bicycle, and the terms on the right side of the equation are the mass, radius and angular acceleration of the flywheel. It thus becomes a matter of calculating the necessary variables to determine how much angular acceleration of the flywheel is needed to move the bike. As can be seen by the equation above, there are a few variables we can control to keep the flywheel angular acceleration as low as possible to keep it within the limits of the motor. Those condition are to keep the center of gravity of the bicycle as low as possible, have the ratio of the center of mass of the bicycle to center of mass of the flywheel as large as possible without overweighting the motor, and making the radius of the flywheel as large as possible.

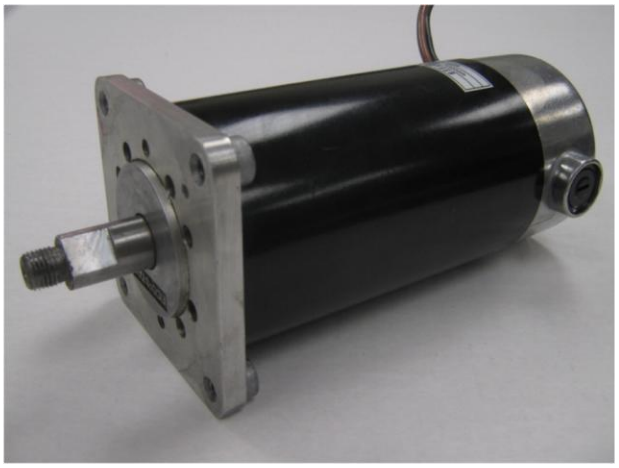
Using the above principle and concepts we can approximately model the falling of a bicycle and the effect that an angularly accelerating flywheel attached to the body of the bicycle will have on the bike. It is therefore possible to create a counter moment to the bicycle falling using a flywheel while keeping the values within the capabilities of the motor and the stress limits of the materials for the flywheel.

In order to keep the necessary mass, radius, and angular acceleration of the flywheel as small as possible compared to the torque of the bike, it is necessary to do all we can with equipment placing to keep the torque of the bike as low as possible. Since the torque of the bike is represented by the equation $T_b = M_b \times {R_b}^2 \times \alpha_b$, and our goal is to keep T as small as possible, it is easy to see that to do this we need to keep the center of gravity and the total mass of the bike as low as possible. Keeping these goals in mind we placed the battery just above the peddles which is as low as we could reasonably mount it. For the motor and flywheel placement we were limited by the fact that they needed to stay together and that if the flywheel was too close to the ground then it would reduce the angle at which the bike could fall without the flywheel hitting the ground. We therefore chose to mount the motor, flywheel, and remaining circuitry on a platform above the back tire behind the seat.

III. Motor and Power Systems Brett Smith

Motor

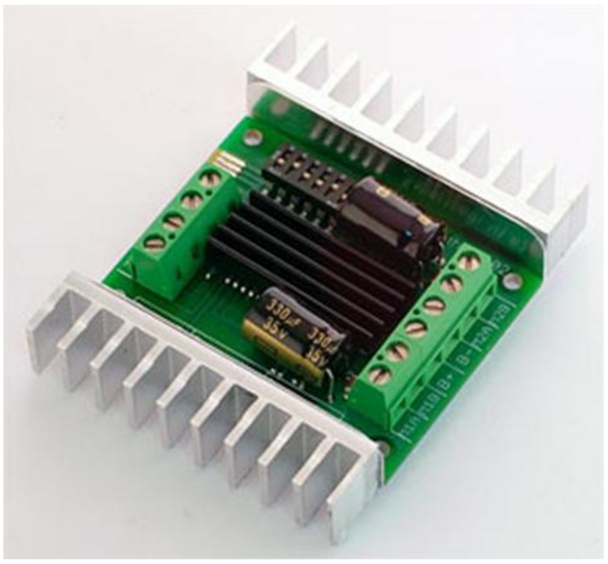
After considerable research, we chose a cordless drill motor for our design. We selected an 18V Bosch drill. This decision was beneficial for several reasons. For one, the drill motor is powerful. It is capable of spinning up to 1600 RPM and generating 500 in-lbs of torque. Secondly, it came with two lithium-ion batteries rated at 1.3 Ah so we would never have to wait on one to charge. Also, the drill came with a quick charger that can fully charge the battery in 30 minutes. Lastly, the drill has forward and reverse functionality built into it (more on that later). One problem we began to notice in the drill motor was the fact that it was geared. The motor had two different sections that fit together. The motor by itself was not powerful enough by itself for our purposes. However, the gears added another wrinkle to or control system. The transmission had a clutch of some sort that would brake almost instantaneously. This created a massive amount of unwanted torque. Also, as the drill turned over time, the motor and transmission sections would become loose and slide apart. For these reasons, we decided to pursue another motor. Calvin Cutshaw was generous enough to lend us a 24V motor that was no longer being used by the autonomous lawnmower team. It was rated up to 500W and 4300 RPM. It was much more powerful than the drill motor. It had a much smoother response when changing directions as well.



24V Motor

Motor Controller

As mentioned earlier, the drill has forward and reverse functionality built into it. However, after purchasing the drill, we discovered that it has a mechanical switch that allows for the reverse to be activated. This is undesirable for a couple reasons. Mechanical switching is much slower than electrical switching. Also, to be able to switch directions in software we would have to have some sort of actuator to flip the switch which would just overcomplicate the circuitry. We looked into designing an H-bridge circuit using transistors to allow reversing the voltage applied to the motor on the fly. This would be a rather cheap and cost effective method. The problem with this solution is that all the MOSFET transistors we saw had low current ratings. Our drill has been measured at up to 12 amps. In the end, we settled on a premade motor controller ordered off the internet. It is rated up to 15 amps (21 amps if a heat sink is added). This saves us room since it is all contained on a printed circuit board. Plus, it is very easy to interface with the microcontroller. One of the chips melted on our motor driver during the final week so we had to order a new one. We decided on a Sabertooth 25A motor controller. This upgrade provided us more cushion to effectively power our new 24V motor.

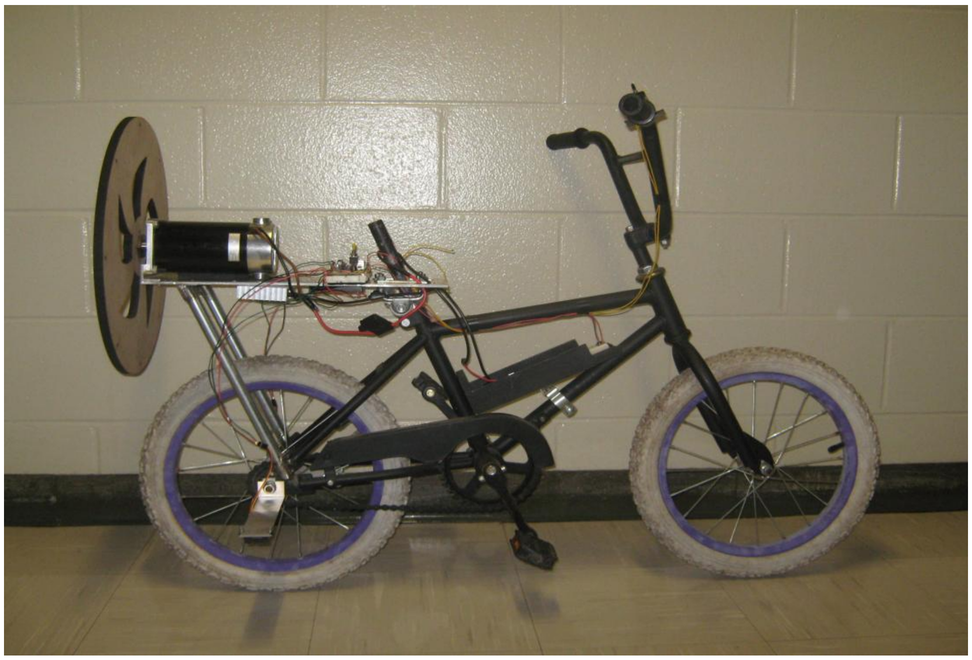


Sabertooth 2x25 Motor Driver

Wiring

As far as the wiring of everything, we have mounted most of the components on a breadboard that sits on a metal mounting plate positioned behind the bike seat. This is the best option for flexibility. The breadboard is attached to the plate by Velcro for easy removal. The motor controller, accelerometer, and microcontroller are all wired together on this breadboard. The 18V battery is mounted to a wooden shelf below the bike seat. We wanted to power everything with a single battery but still keep the sensitive electronics (microcontroller, accelerometer, IR sensors) separate from the motor. To accomplish this we acquired a DC/DC converter. It is capable of converting an 18V to 30V signal into a regulated 5V signal. Since the motor does draw considerable current, we had to use thicker wires than the 22 gauge used on the breadboard. The wires inside the drill are 14 gauge. To be on the safe side we went ahead and wired the motor controller to the battery and motor using 12 gauge. To make our lives easier, we implemented a switch that was mounted on the handlebars. This was used to cut power to the motor without having to constantly unhook the battery. Later, we also used this switch as a digital input for calibrating the balance point of our control system. By flipping the switch at the point that seemed the most balanced, we could set our reference point for the control system.

18V Lithium Ion Rechargeable Drill Battery



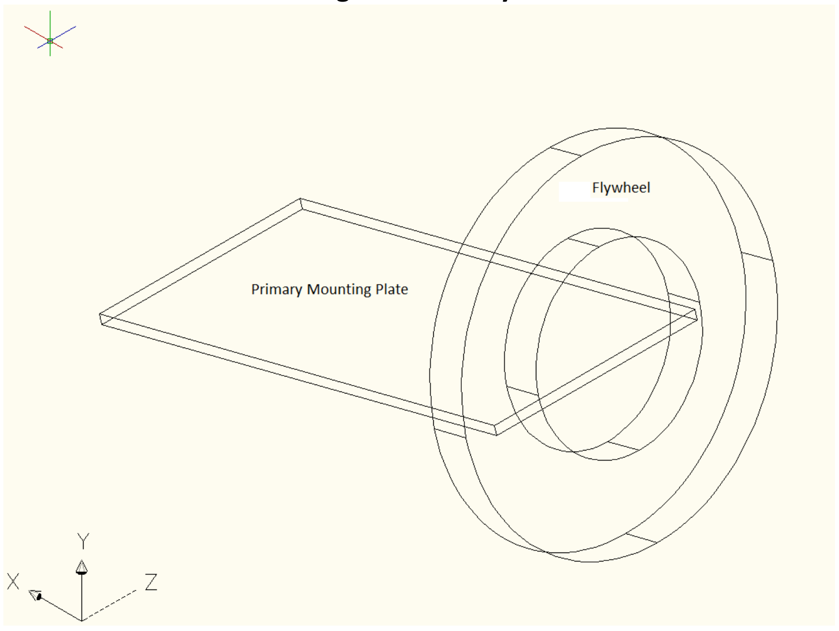
Final Version of Bike

IV. Structural Design/Sensors/Wiring — Mason Nixon

Structural Design

For the mounting hardware (See Figure 3 CAD below), we decided to use an in-house method for construction of the primary hardware mounting plate and mounting rods. We came up with the optimal dimensions for a primary mounting plate by taking into consideration the size of the flywheel and the amount of hardware we would need room for. Since the plate had to be long enough to space the flywheel away from the bicycle wheel, it was necessary to add additional supports at the rear of the bike. Calvin Cutshaw assisted in the cutting and welding of the plate, and we assisted by rethreading the U-mounts and sought a bicycle seat to fit the seat pipe.

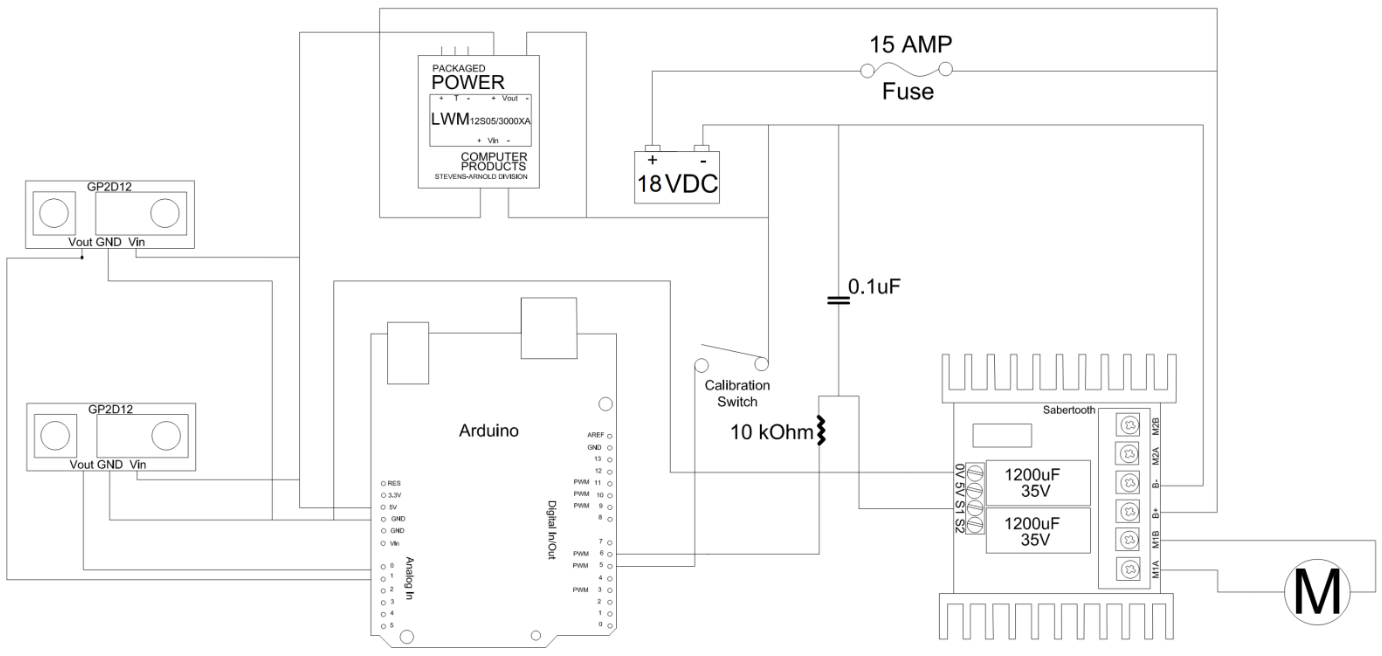
Mounting Plate and Flywheel



Once the plate was fixed and level, we added the solder-less breadboard and the motor. Most of the mounting we did at this point is meant as a temporary solution, not intended for use on the final product design. We designed a wooden motor mount to affix the motor to the primary plate. We then used foam and Velcro to attach the solder-less breadboard to the primary plate. The foam is used as a damper to counter the vibration due to the motor.

We were able to design an electrical schematic in AutoCAD 2009, which allowed for ease of updates and offered a professional look. On the breadboard itself, using the schematic shown in Figure 4 below, we were able to wire all of our components using minimal space. It was necessary to use two different batteries because at this time we did not believe it was proficient to design a power system that would operate off of one power source. We mounted the accelerometer as far from the motor as we could to minimize the vibration caused by the motor. We also mounted it on the opposite support of the primary plate for the same goal.

Electrical Hardware Schematic



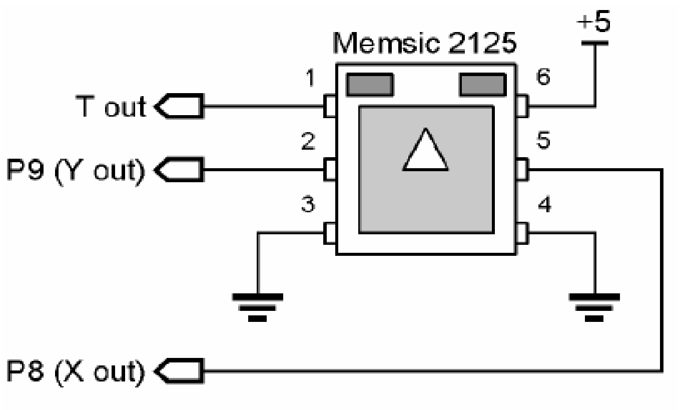
Note: All parts are NOT to scale.

Sensors

Accelerometer

At first, we were utilizing an accelerometer to detect our angular position. For the accelerometer we decided to use the Memsic 2125, which is a low cost, dual-axis thermal accelerometer capable of measuring tilt, acceleration, rotation, and vibration with a range of ±2 g. The Atmel Atmega328 can be paired with this accelerometer to test for an accurate readout and simulation of operation. The Memsic 2125 has a simple digital interface: two pins (one for each axis) emit pulses whose duration corresponds to the acceleration of that axis.

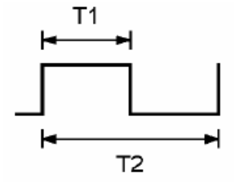
Figure 1. Essential Memsic 2125 Connections



Internally, the Memsic 2125 contains a small heater. This heater warms a "bubble" of air within the device. When gravitational forces act on this bubble it moves. This movement is detected by very sensitive thermopiles (temperature sensors) and the onboard electronics convert the bubble position [relative to g-forces] into pulse outputs for the X and Y axis.

The pulse outputs from the Memsic 2125 are set to a 50% duty cycle at 0 g. The duty cycle changes in proportion to acceleration and can be directly measured by the Atmel. Figure 2 shows the duty cycle output from the Memsic 2125 and the formula for calculating g force.

Figure 2. Memsic 2125 Pulse Output



A(g) = ((T1 / T2) – 0.5) / 12.5%

The T2 duration is calibrated to 10 milliseconds at 25° C (room temperature). Knowing this, we can convert the formula to the following embedded C code routine as an example:

```
const int xPin = 2;                    // X output of the accelerometer
const int yPin = 3;                    // Y output of the accelerometer
 // variables to read the pulse widths:
 int pulseX, pulseY;
 // variables to contain the resulting accelerations
 int accelerationX, accelerationY;

 // read pulse from x- and y-axes:
 pulseX = pulseIn(xPin,HIGH);
 pulseY = pulseIn(yPin,HIGH);

 accelerationX = ((pulseX / 10) - 500) * 8;
 accelerationY = ((pulseY / 10) - 500) * 8;
```

In the above code, by dividing the raw pulse input pulseX - Y by 10 milliseconds and subtracting 1000 times 0.5 and then dividing the entire quantity by 1 over 0.125. The accelerationX - Y are in milli-g's (Earth's gravity = 1g. or 1000 milli-g's).

```
  z1 = (accelerationX * 9);
 z2 = (accelerationY * 9);

 //Calculate the angle to 2 decimal places by making a long into a float.

 tiltX = (float)z1 / 100.0;
 tiltY = (float)z2 / 100.0;
```

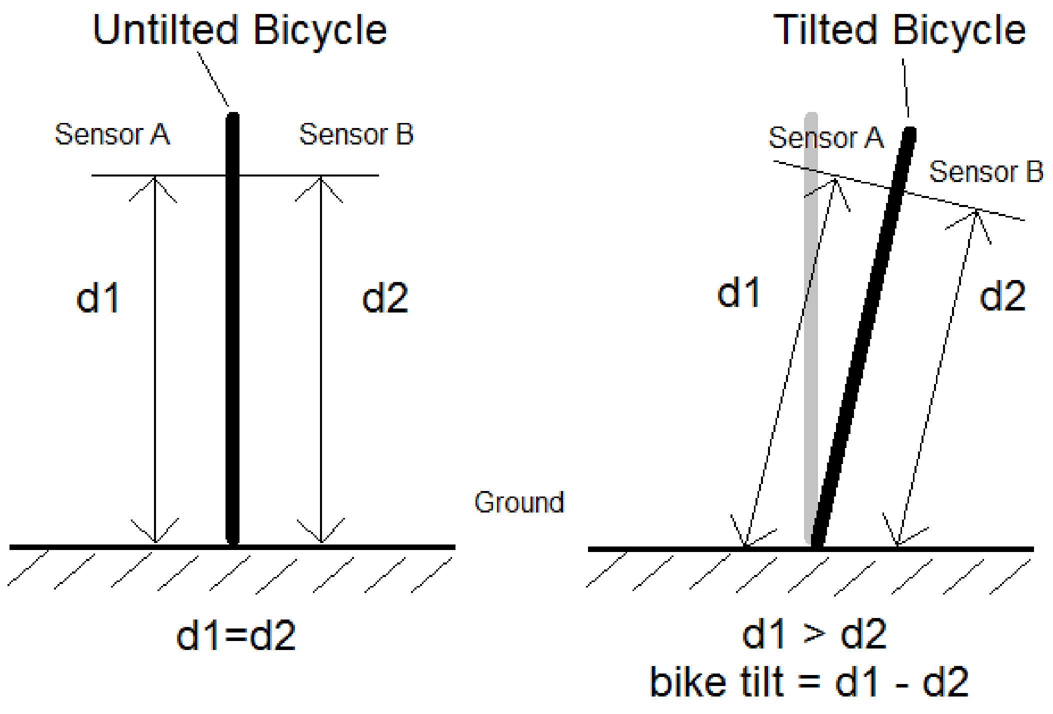
Since the output from the accelerometer is a number between 0 and 1 which corresponds to 0 to 90 degrees, one can multiply the acceleration calculated by 9 and divide by 100.0 and obtain an angle out to up to 4 digits. The Memsic would have provided suitable resolution for our feedback since the output is accurate to 1 milli-g and the Atmel Atmega328 microprocessor should provide a simple interface for implementing our control system.

IR Analog Range Sensors

The problem with the accelerometer is that when we apply a corrective acceleration with our flywheel, this creates an opposing angular acceleration reading in our accelerometer that could potentially give us a reading of a position that is inaccurate and cause instability in our control system. Also, we are limited by the digital PWM output that has a maximum speed of 10ms per reading. These inadequacies led us to determine that a more suitable sensor would be infrared range sensors. We decided on the Sharp analog IR GP2D12 range sensor. This sensor takes a continuous distance reading and returns a corresponding analog voltage proportional to the distance with a range of 10cm (~4") to 80cm (~30").

Placing two of these sensors at the same height and mounting them in fixed positions on the bicycle, we were able to determine any tilt in the bike by comparing their heights above the ground. If the bike leaned to one side, then the height reading of the sensor on the side of the tilt would be lower. Taking the difference of the two heights yields a slope that can be fed into our control system and used as feedback. This method, illustrated below, provides an effective means of feedback for control and allowed us to make extremely accurate corrective action. Also, since the sensor is analog, our system was sped up almost 5 times as fast, since we could read in data at about 2.75ms – limited only by the Atmel328.

Illustration of IR Sensor Concept



It is also illustrated in the code Daniel Golden wrote below:

```
int sensorValue;

int readIR(int sensor) {
      if (sensor == IR_LEFT){
            // read the value from the left IR sensor:
            ADMUX = _BV(REFS0) | 0;        // select channel ADC0
            ADCSRA |= _BV(ADSC);           // start the conversion
            loop_until_bit_is_clear(ADCSRA, ADSC);
// wait for conversion to complete
            sensorValue = ADC;
      }else if (sensor == IR_RIGHT){
            // read the value from the right IR sensor:
            ADMUX = _BV(REFS0) | 1;        // select channel ADC1
            ADCSRA |= _BV(ADSC);           // start the conversion
            loop_until_bit_is_clear(ADCSRA, ADSC);
// wait for conversion to complete
            sensorValue = ADC;
      }
      return sensorValue;
}

irDiff = readIRsmoothed(IR_LEFT, ir_LH_accum, &ir_start_cnt_LH) -
readIRsmoothed(IR_RIGHT, ir_RH_accum, &ir_start_cnt_RH);
// calculate new accelerometer data
controlsys(irDiff, &MotorDutyCycle, &integrator, &lasterror);
// process control system
```

Although a large step above the accelerometer, the infrared range sensor had many potential disadvantages as well. Of course, the most noteworthy is distortion from other sources of infrared light. Infrared from sunlight would be the largest potential attenuator, but incandescent lights, remote controls, and nearly anything that generates heat is a source of infrared radiation and could distort the bicycle control system. The second disadvantage to this approach is the need for a nearly perfectly lvel surface. Any discontinuity in the floor or ground that the sensor encounters could cause an otherwise balanced bike to be pushed to one side, since the control system could read the height difference as the beginning of a fall.

We were able to gain accurate tilt measurements from the IR sensors, but as noted in the above paragraph they were far from ideal. Our design could be optimized by a gyroscopic accelerometer, or an accelerometer that gives position measurements independent of angular acceleration. These sensors could of have been implemented, but due to time and cost constraints, we were unable to utilize them in our design.

Moving Average Filter

For smoothing of sensor data, it is necessary to implement a smoothing filter. One such filter with excellent results is the moving average filter (MAF). The moving average filter accumulates the inputs and averages them for each new point as described in the following equation:

$$y[i] = \frac{1}{M}\sum_{j=0}^{M-1} x[i+j]$$

From which the output, y, may be calculated from the input x and where M is the number of points averaged by the filter. As an example, in a 3 point moving average filter, point 10, for instance, in the output signal is given by:

$$y[10] = \frac{x[10] + x[11] + x[12]}{3}$$

The moving average filter is a form of convolution. As noted in *The Scientist and Engineer's Guide to Digital Signal Processing By Steven W. Smith, Ph.D*, “You should recognize that the moving average filter is a *convolution* using a very simple filter kernel. For example, a 5 point filter has the filter kernel: …0, 0, 1/5, 1/5, 1/5, 1/5, 1/5, 0, 0…. That is, the moving average filter is a convolution of the input signal with a *rectangular pulse* having an area of *one*.” [2]

The listing below shows our program with a 5-point filter that we utilize to implement the moving average filter.

```
//Moving Average Filter Implementation for LPF of sensor data

float sensorSmoothed(float accumulator[], int a) {
  int i=0;
  float maf;
  float sum=0;
  for(i=0;i<=3;i++){
    accumulator[i]=accumulator[i+1];
//Shift out the oldest
  }
  accumulator[4]=sensorIn();
//Retrieve most recent noisy data point
  for(i=0;i<=4;i++){
    sum+=accumulator[i]; //Sum of 5 terms
  }
  maf=sum/5; //Average of the 5 data points
  a=a+1; //Increment through first 5 points to initially fill
  if(a<5){
    return sensorIn();
  } //Wait for first 5 points to be accumulated
  else{
    return maf;
  }
}
```

MAF Noise Reduction vs. Step Response

The moving average filter has an extremely simple implementation. In commenting on this, Dr. Steven Smith says, “This situation is truly ironic. Not only is the moving average filter very good for many applications, it is *optimal* for a common problem, reducing random white noise while keeping the sharpest step response."[2] While our noise is not always random, the MAF is invaluable when trying to smooth and isolate a sensors data.

The Result of a Signal Filtered with a MAF[2]

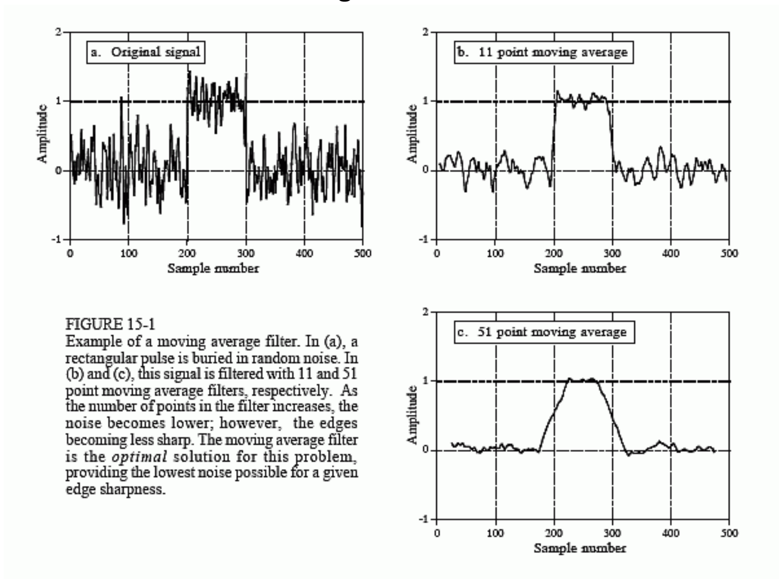


FIGURE 15-1
Example of a moving average filter. In (a), a rectangular pulse is buried in random noise. In (b) and (c), this signal is filtered with 11 and 51 point moving average filters, respectively. As the number of points in the filter increases, the noise becomes lower; however, the edges becoming less sharp. The moving average filter is the *optimal* solution for this problem, providing the lowest noise possible for a given edge sharpness.

Figure 15-1 demonstrates a MAF filtering a signal. The original noisy signal is show in (a). The result in the step response as shown in (b) is that the amplitude of the noise is depleted, while at the same time the edges taper off, which is not the most ideal of situations for a step response. In (c), the M value is increased by a factor of 5. The step response contains even less noise, but the sharpness in the edge is nearly lost. Although sharpness is lost, the MAF outperforms all other linear filters as far as "lowest noise for a given edge sharpness"[2]. Also, as noted in *The Scientist and Engineer's Guide to Digital Signal Processing*, "The amount of noise reduction is equal to the square-root of the number of points in the average. For example, a 100 point moving average filter reduces the noise by a factor of 10."[2]

MAF Frequency Response

The frequency response of an MAF is shown below in Figure 15-2. The MAF frequency response is simply the Fourier transform of a rectangular pulse as described below:

$$H[f] = \frac{\sin(\pi f M)}{M sin(\pi f)}$$

The above equation is for an M-point MAF from which the frequency, f, shifts between 0 and 0.5 and H[f=0]=1. As seen in the figure below, there is very little stopband attenuation and the response has a slow roll-off. This illustrates a major disadvantage of the MAF: it cannot isolate different bands of frequencies. To summarize, the moving average is more than adequate as a smoothing filter, however, as indicated in the frequency response, it is a horrible low pass filter. [2]

Frequency Response of MAF[2]

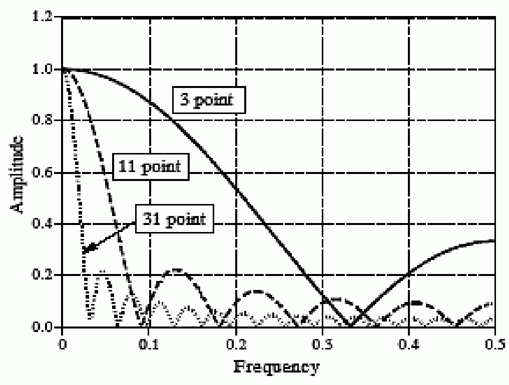
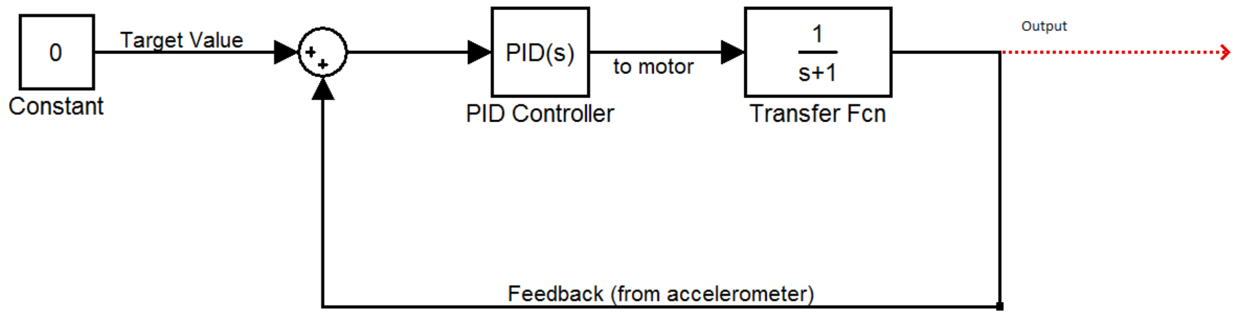


FIGURE 15-2
Frequency response of the moving average filter. The moving average is a very poor low-pass filter, due to its slow roll-off and poor stopband attenuation. These curves are generated by Eq. 15-2.

V. Balancing Control System Theory Daniel Dunbar

The goal of the control system is to balance the bicycle using feedback from the accelerometer or the IR sensors. This creates a closed loop control system, which is controlled by a PID controller. There are two inputs into the system, the desired value of zero (a completely balanced system) and the feedback from either the accelerometer or the IR sensors. We implemented both feedback designs on one bike. In the future, a common filter may be used to get a more accurate feedback system. These two inputs are added together to get an error, which is inputted into the PID controller. The output from the PID controller determines the voltage applied to the motor, which affects the speed and acceleration of the flywheel, thus balancing the bicycle.



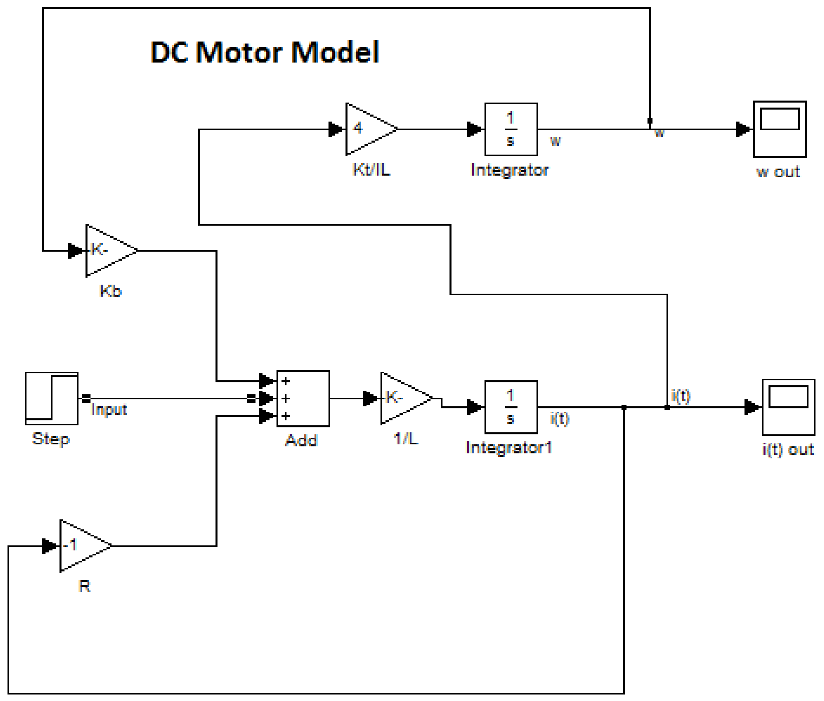
To model the control system, SIMULINK was chosen. SIMULINK is a component of the MATLAB package and is used to build systems from a block component level. This allows the user to create a complete system within SIMULINK using standard blocks used in control system theory. SIMULINK also has a PID tuner, which will automatically tune the gains for the P,I, and D parts of the controller. If an accurate model of the plant can be obtained or designed, this is an extremely effective tool that allows the user to look at the rise time, overshoot, etc. to find a controller that will suit the application best.

Also, SIMULINK allows the user to model certain components using differential equations. For example, a DC motor can be broken down to two differential equations:

$$\frac{d\omega(t)}{dt} = \frac{K_T}{I_L} i(t) \quad \text{and} \quad \frac{di(t)}{dt} = \frac{1}{L}[V_s - Ri(t) - K_b\omega(t)]$$

where K_T, I_L, R, L, and K_b are parameters of the motor.

From these equations, one can create a model using integrators, adders, and gain blocks. Thus, is on can obtain the motor parameters, one can create an accurate model of a motor within MATLAB and use the computational power of the program to do some useful analysis.



To model the control system initially in SIMULINK, an external simulated input had to be created. This input represents the external forces on the bicycle, such as a person leaning or simply gravity working on the system. This was added to the force created by the angular velocity of the motor to create the feedback portion of the system, which will be actualized by either the accelerometer or the IR sensors in the physical system. For the external input, several different patterns were created to try to accurately simulate some forces that might occur during physical testing.

In transition to the implementation phase of the control system, the original plan was to use MATLAB's built in Embedded C writer to write the code for the control system. However, when we tried this, the code was much too convoluted to be able to fine tune away from MATLAB, so we wrote the code ourselves. The SIMULINK model was not done in vain, though, because we are using it as a theoretical basis for our control system, and we are working to develop an accurate transfer function that lines up with our actual motor under load. When we achieve this, we will be able to use the SIMULINK model to tune the control system to optimize performance.

In tuning the control system, we realized that having to reprogram the microcontroller for any slight change in our constants was not only time consuming and tedious, but it also did not allow us to see immediate changes. To fix this problem we connected three potentiometers to our microcontroller's analog inputs. The pots acted as voltage dividers into the input, so we could vary the amount of voltage going into the input. This enabled us to read in different values in real time which we made correspond to the K_P, K_I, and K_D values, so we could tune our control system in real time instead of reprogramming the microcontroller every time we wanted to adjust a gain constant.

The code, as seen as *ControlSystem.c* on the CD, was built as a function called from the main program. This allows the program to be modularized for ease of reading and helps in the debug process. Also, the K_P, K_D, and K_I constants are at the top for easy access when tuning the control system. We made several changes from the Cycle 1 code. First, we added a calibration feature to calibrate our desired value to make up for misalignment of our sensors. Second, we added the IR sensor portion of the control system; when we used the IR sensors, we had to completely change our gain constant values because our input to the control system was different. Next, we added the potentiometer tuning system. Finally, we added pre-compiler if-statements to change between the accelerometer control system and the IR control system depending on define statements found elsewhere in the program.

VI. Embedded System Architecture — Daniel Golden

Previously, our architecture was designed for using an accelerometer as our control system feedback. As already explained, this was problematic since our accelerometer does not isolate dynamic acceleration of motor and bike from the static acceleration of gravity. So, for our second generation balancing system, we incorporated a control loop which reads Infrared sensor data from two sensors, smoothes the data, sends the difference of that data to the control system, and then sets a new corrective Duty Cycle and direction pin for use by our motor controller. This second generation control loop is shown in Figure 1 below.

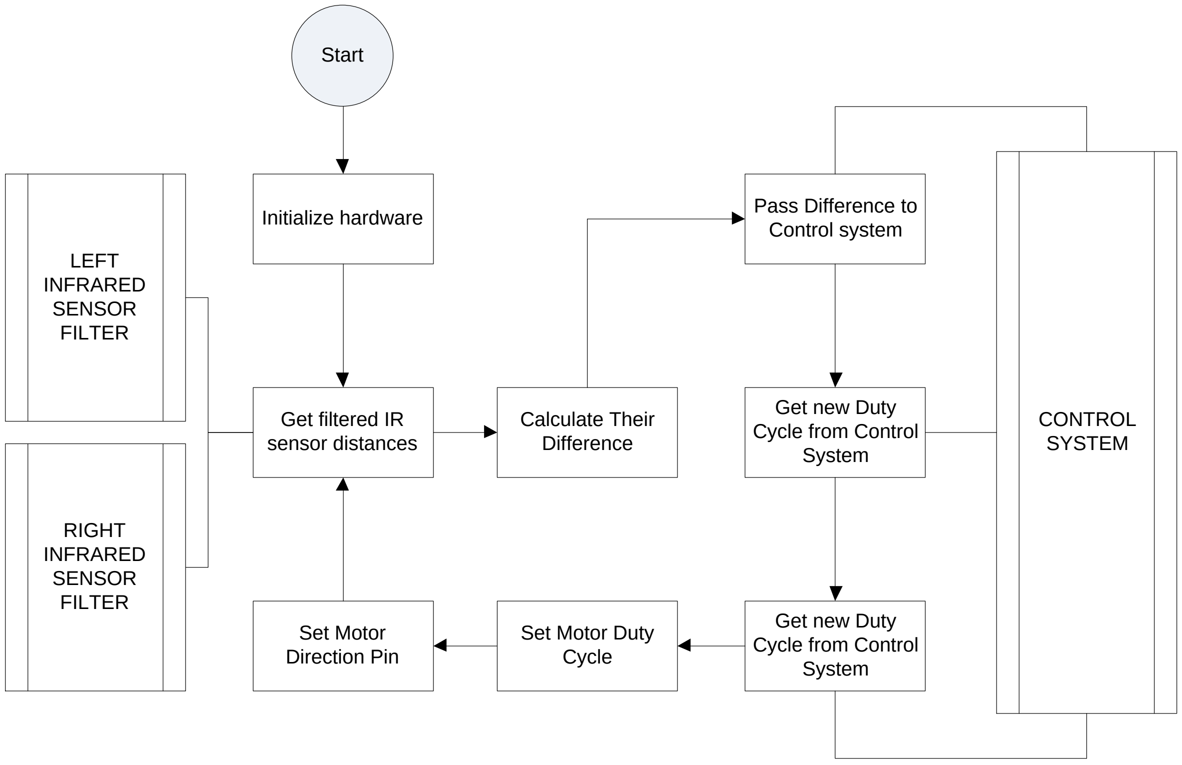

Start
Initialize hardware
LEFT INFRARED SENSOR FILTER
RIGHT INFRARED SENSOR FILTER
Get filtered IR sensor distances
Calculate Their Difference
Pass Difference to Control system
Get new Duty Cycle from Control System
Get new Duty Cycle from Control System
CONTROL SYSTEM
Set Motor Direction Pin
Set Motor Duty Cycle

Figure 1 - Embedded System Control Loop

Infrared Distance Filter

Both the left and the right infrared sensor filters are of the moving average type. The filter smoothes N points. N can be changed by changing the following line in the code in "includes.h":

```
        #define MAFSIZE      12       // size of the moving average filters
```

Control System

To execute the second generation control system we chose to simply encapsulate the entire system into a function that takes four arguments.

```
controlsys( int    IRdiff,
            int    *MotorDutyCycle,
            float  *integrator,
            float  *lasterror )
```

The first is the difference of the two IR sensors and the other three are pointers to variables in the calling method's scope which are updated by the control system function.

Hardware Initialization

In our first generation system, we were implementing an accelerometer.   However, since we are no longer using it, we have no need to remove the PWN measurement ability. On the other hand, our infrared sensors are using the analog inputs into our microcontroller. Therefore, to use these pins, we must set up our microcontroller's registers to perform this action.

Hardware Initalization Routine:

```
/* Set up timer0 for motor control */
TCCR0A |= _BV(COM0A1);  // inverted pwm outputs
TCCR0A |= _BV(WGM01) | _BV(WGM00);          // Phase correct mode - ICR0 is top
TCCR0B |= _BV(CS01);                        // prescale F_CLK_IO by 8

/* Set analog inputs for reading IR sensors. */
ADCSRA |= _BV(ADPS2)|_BV(ADPS1)|_BV(ADPS0);         // ADC prescale factor is 128
DIDR1 |= _BV(AIN1D) | _BV(AIN0D);       // disable digital input buffers on ADC pins
ADCSRA |= _BV(ADEN);                    // enable ADC conversions
ADMUX  |= _BV(REFS0);                   // AVCC with external capacitor at AREF pin

/* Set up pin 6 for motor controller PWM output. */
DDRD |= _BV(PD6);                       //Set PD6 for output
PORTD |= _BV(PD6);                      //Set  PD6 to Output High
```

Main Program Loop

The main program loop is the first code to execute on system power-on. This loop calls the hardware initialization routine, enables interrupts, and then enters the infinite control loop:

```
int main(){

        hardwareInit();                         // initialize hardware
        _delay_ms(50);
        sei();                                  // enable interrupts

        for(;;){// infinite program loop
                /* calculate new IR difference */
                irDiff = readIRsmoothed(IR_LEFT, ir_LH_accum, &ir_start_cnt_LH) –
                        readIRsmoothed(IR_RIGHT, ir_RH_accum, &ir_start_cnt_RH);

                /* process control system */
                controlsys(irDiff, &MotorDutyCycle, &integrator, &lasterror);
                OCR0A = MotorDutyCycle;         // set the output PWM
        }
        return 0;
}
```

Motor Control

In order to control the speed of our motor, we are generating a PWM signal using the 8-bit PWM mode of timer0. To adjust the motor's speed and direction, we write a number between 0 and 255 to OCR0A where 127 is the midpoint where the motor is stopped. The maximum motor speeds for both directions are obtained by writing either a 0 or a 255 to OCR0A. In other words, writing a value of 0 produces a duty cycle of 100% in one direction. Writing a value of 0xFF to OCR0A generates a duty cycle of 100% in the other direction.

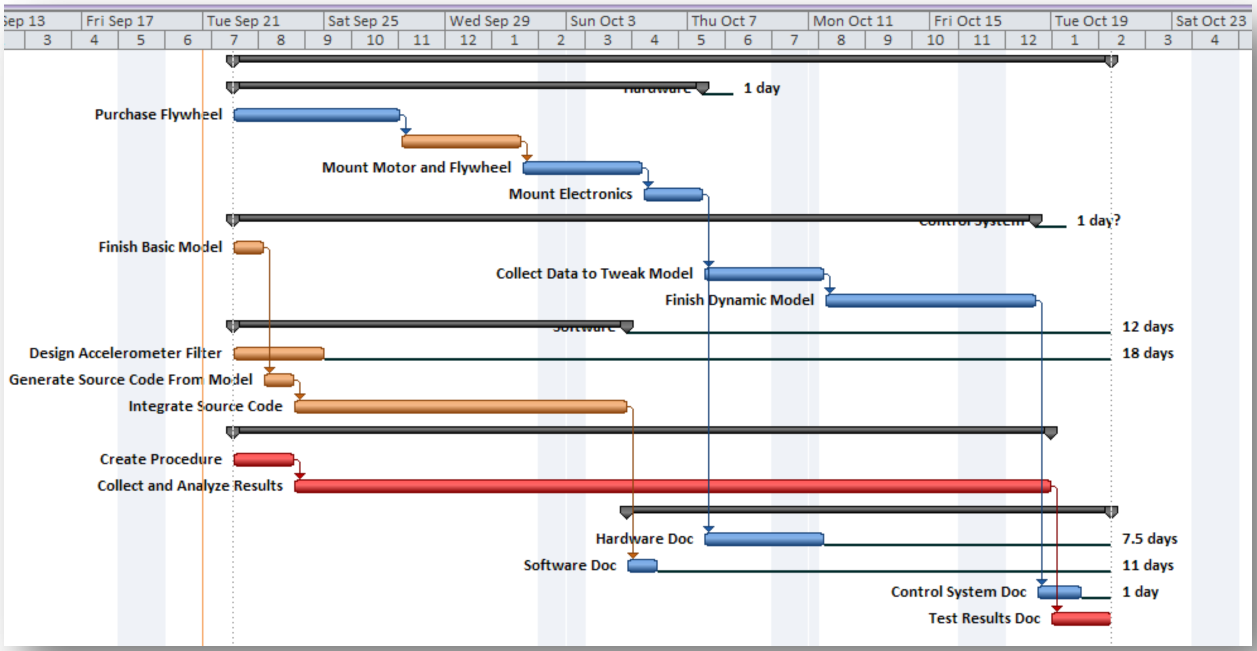
VII. Administration Daniel Golden

Budget

Item	Price
Batteries/Charger	$153.49
Motor	$60.00*
Flywheel	$100.00*
Bike	$50.00*
Microcontroller	$29.99
Motor Controller	$124.99
Mounting Hardware	$100.00*
IR Sensors/Accelerometer	$60.89
Developmental Costs	$30.00*
Total	**$709.36**

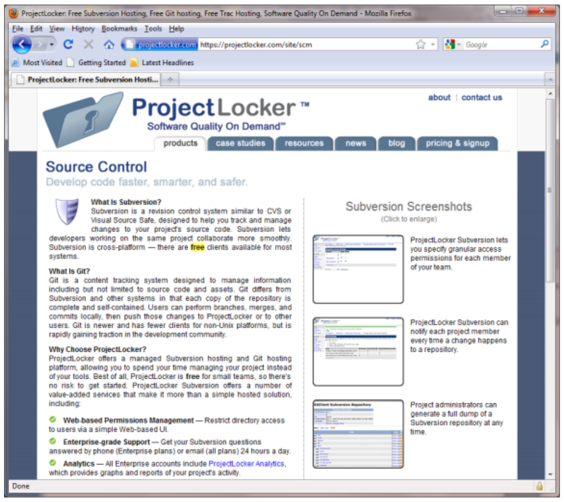
Timeline

For timeline creation, we used Microsoft Project. Project has a **Gantt Chart** tool, which allowed us to orderly schedule tasks – ultimately streamlining the design process. The following figure is an example of our timeline:



Organization

For source control organization and versioning, we chose to use Subversion. Our subversion host is ProjectLocker and our client software is TortoiseSVN. Both are free services, and work together to give us the ability to maintain our code on external servers and to revise or review our code remotely on multiple computers. The following are screenshots of the website services.


ProjectLocker: Free Subversion Hosting, Free Git hosting, Free Trac Hosting, Software Quality On Demand - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://projectlocker.com/site/scm
Most Visited
Getting Started
Latest Headlines
ProjectLocker: Free Subversion Hosti...
about | contact us
ProjectLocker™
Software Quality On Demand™
products
case studies
resources
news
blog
pricing & signup
Done

Source Control

Develop code faster, smarter, and safer.

What Is Subversion?
Subversion is a revision control system similar to CVS or Visual Source Safe, designed to help you track and manage changes to your project's source code. Subversion lets developers working on the same project collaborate more smoothly. Subversion is cross-platform — there are **free** clients available for most systems.

What Is Git?
Git is a content tracking system designed to manage information including but not limited to source code and assets. Git differs from Subversion and other systems in that each copy of the repository is complete and self-contained. Users can perform branches, merges, and commits locally, then push those changes to ProjectLocker or to other users. Git is newer and has fewer clients for non-Unix platforms, but is rapidly gaining traction in the development community.

Why Choose ProjectLocker?
ProjectLocker offers a managed Subversion hosting and Git hosting platform, allowing you to spend your time managing your project instead of your tools. Best of all, ProjectLocker is **free** for small teams, so there's no risk to get started. ProjectLocker Subversion offers a number of value-added services that make it more than a simple hosted solution, including:

- **Web-based Permissions Management** — Restrict directory access to users via a simple Web-based UI.
- **Enterprise-grade Support** — Get your Subversion questions answered by phone (Enterprise plans) or email (all plans) 24 hours a day.
- **Analytics** — All Enterprise accounts include ProjectLocker Analytics, which provides graphs and reports of your project's activity.

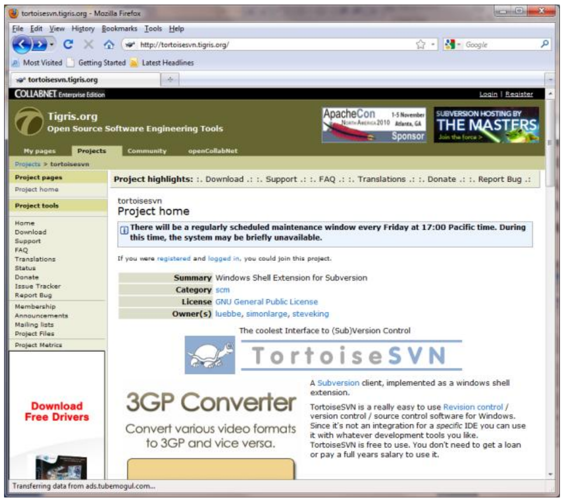
Subversion Screenshots

(Click to enlarge)

ProjectLocker Subversion lets you specify granular access permissions for each member of your team.

ProjectLocker Subversion can notify each project member every time a change happens to a repository.

Project administrators can generate a full dump of a Subversion repository at any time.


tortoisesvn.tigris.org - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://tortoisesvn.tigris.org/
Most Visited
Getting Started
Latest Headlines
tortoisesvn.tigris.org
COLLABNET Enterprise Edition
Login | Register
Tigris.org
Open Source Software Engineering Tools
ApacheCon North America 2010 1-5 November Atlanta, GA Sponsor
SUBVERSION HOSTING BY THE MASTERS Join the force >
My pages
Projects
Community
openCollabNet
Projects > tortoisesvn
Project pages
Project home
Project tools
Home
Download
Support
FAQ
Translations
Status
Donate
Issue Tracker
Report Bug
Membership
Announcements
Mailing lists
Project Files
Project Metrics
Download Free Drivers
3GP Converter
Convert various video formats to 3GP and vice versa.
TortoiseSVN
Transferring data from ads.tubemogul.com...

Project highlights: :. Download .: :. Support .: :. FAQ .: :. Translations .: :. Donate .: :. Report Bug .:

tortoisesvn

Project home

There will be a regularly scheduled maintenance window every Friday at 17:00 Pacific time. During this time, the system may be briefly unavailable.

If you were registered and logged in, you could join this project.

Summary	Windows Shell Extension for Subversion
Category	scm
License	GNU General Public License
Owner(s)	luebbe, simonlarge, steveking

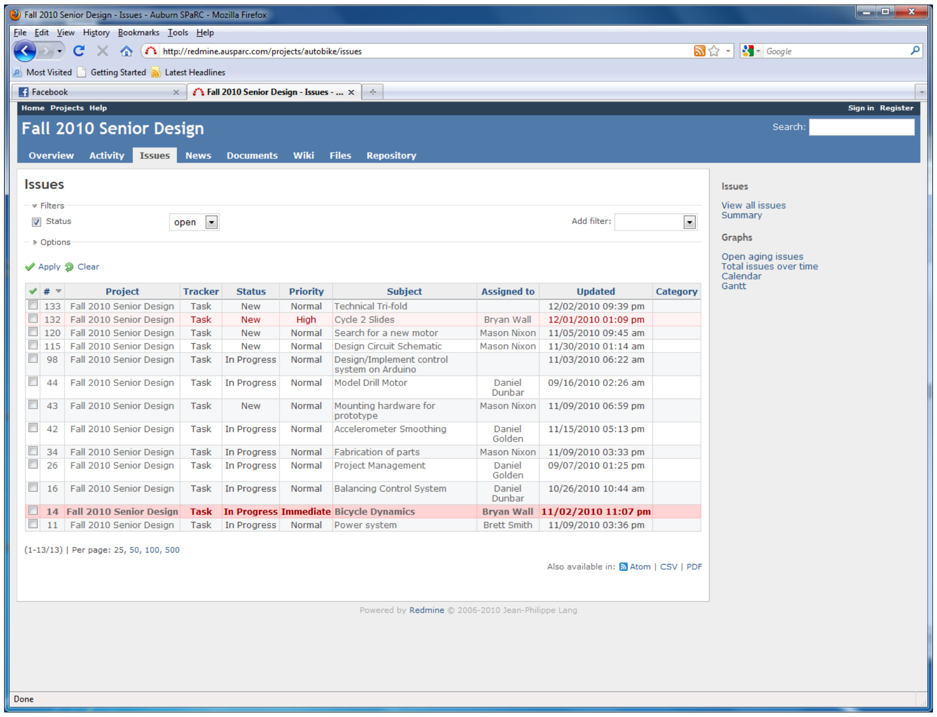
The coolest Interface to (Sub)Version Control

A Subversion client, implemented as a windows shell extension.

TortoiseSVN is a really easy to use Revision control / version control / source control software for Windows. Since it's not an integration for a *specific* IDE you can use it with whatever development tools you like. TortoiseSVN is free to use. You don't need to get a loan or pay a full years salary to use it.

Redmine – Content Management System

In order to facilitate organization, we decided to use a content management system called Redmine. There are several advantages to this system including issue tracking, revision control, Gantt chart, wiki, news, activity feed, activity notifications, and resource management. The following is a screenshot of our content management system:



Issues

Filters

Status: open

Options

Apply Clear

#	Project	Tracker	Status	Priority	Subject	Assigned to	Updated	Category
133	Fall 2010 Senior Design	Task	New	Normal	Technical Tri-fold		12/02/2010 09:39 pm	
132	Fall 2010 Senior Design	Task	New	High	Cycle 2 Slides	Bryan Wall	12/01/2010 01:09 pm	
120	Fall 2010 Senior Design	Task	New	Normal	Search for a new motor	Mason Nixon	11/05/2010 09:45 am	
115	Fall 2010 Senior Design	Task	New	Normal	Design Circuit Schematic	Mason Nixon	11/30/2010 01:14 am	
98	Fall 2010 Senior Design	Task	In Progress	Normal	Design/Implement control system on Arduino		11/03/2010 06:22 am	
44	Fall 2010 Senior Design	Task	In Progress	Normal	Model Drill Motor	Daniel Dunbar	09/16/2010 02:26 am	
43	Fall 2010 Senior Design	Task	New	Normal	Mounting hardware for prototype	Mason Nixon	11/09/2010 06:59 pm	
42	Fall 2010 Senior Design	Task	In Progress	Normal	Accelerometer Smoothing	Daniel Golden	11/15/2010 05:13 pm	
34	Fall 2010 Senior Design	Task	In Progress	Normal	Fabrication of parts	Mason Nixon	11/09/2010 03:33 pm	
26	Fall 2010 Senior Design	Task	In Progress	Normal	Project Management	Daniel Golden	09/07/2010 01:25 pm	
16	Fall 2010 Senior Design	Task	In Progress	Normal	Balancing Control System	Daniel Dunbar	10/26/2010 10:44 am	
14	Fall 2010 Senior Design	Task	In Progress	Immediate	Bicycle Dynamics	Bryan Wall	11/02/2010 11:07 pm	
11	Fall 2010 Senior Design	Task	In Progress	Normal	Power system	Brett Smith	11/09/2010 03:36 pm	

(1-13/13) | Per page: 25, 50, 100, 500

Also available in: Atom | CSV | PDF

Powered by Redmine © 2006-2010 Jean-Philippe Lang

Communication

For technical related communication our primary method was via Redmine. Redmine allowed us to chat about each issue separately from other issues – which is very important when there are multiple tasks being performed simultaneously. It also helps every person on the team to know what is going on.

A secondary mode of communication was via email. Our email address is aurascl@googlegroups.com. However, this mode of communication was almost solely for use to remind all members of meetings and coordinate scheduling.

VIII. Conclusion

To summarize, we propose to create a balance-assist for a bicycle that can be used on any standard bike. The applications are in learning to ride a bike, aiding the disabled, and aiding the elderly. Using control systems theory, we will mount a flywheel to a reversible motor, feedback accelerometer angle data into our control system, and output compensated motor commands to correct for the lean associated with a falling bike. Our design encompasses many aspects of engineering, primarily those listed in the senior design requirement as defined by Accreditation Board for Engineering and Technology (ABET).

IX. Appendix

Contributions

We all contributed to this project in many different areas. Below, we will list the main items that were worked on and by whom:

Daniel Dunbar – Daniel led the Control System design aspect of the project. He also worked on debugging the integration with the microcontroller, flywheel design, debugging the dc-to-dc converter, wiring the circuits, and various presentations of the project.

Daniel Golden – Golden Dan was the project manager for the whole project and did most of the management work scheduling meetings, forming agendas, etc. He also led in the microcontroller integration aspect of the project, including implementation code for the IR sensors. He also helped in flywheel design, control system tuning, motor acquisition, and poster design.

Mason Nixon – Mason led the sensor design aspect of the process, deciding on both the accelerometer and IR sensors we used and writing preliminary code to implement them. He also led in procuring a dc-to-dc converter, making a schematic of the wiring, wiring the circuits, and structure design. He helped in flywheel design, poster assembly, and access to tools found in the SPARC lab.

Brett Smith – Brett led in purchasing, documentation, such as weekly status reports and Cycle 1 and 2 binders, and power system design, including batteries, motors, and motor controllers. He helped in flywheel design, wiring the circuits, tuning the control system, structure design, and poster design.

Bryan Wall – Bryan led in mechanical design and flywheel design. He also helped in control system tuning, structure design, wiring the circuits, debugging power system and coding, and PowerPoint compilation.

Every member of this team played a big part in the overall project and had a hand in most, if not all, of the aspects, whether in the design phase or the implementation phase or both. Thus, the list above should not be taken as an exhaustive list of the effort of our team as a whole.

Acknowledgements

Special thanks go out to several people who made this project possible. Calvin Cutshaw and Linda Barresi were very helpful in turning our ideas for the bike into a reality. They were immensely helpful in attaching the mounting plate to the bike and devising methods of securing the motors to the plate. Daniel Golden's brother, David Golden, is an Industrial Design student. He designed our flywheels in Adobe Illustrator and had them made using a laser cutter. Last but not least we would like to recognize Mr. and Mrs. Stringfellow for their gracious monetary donation to our project.