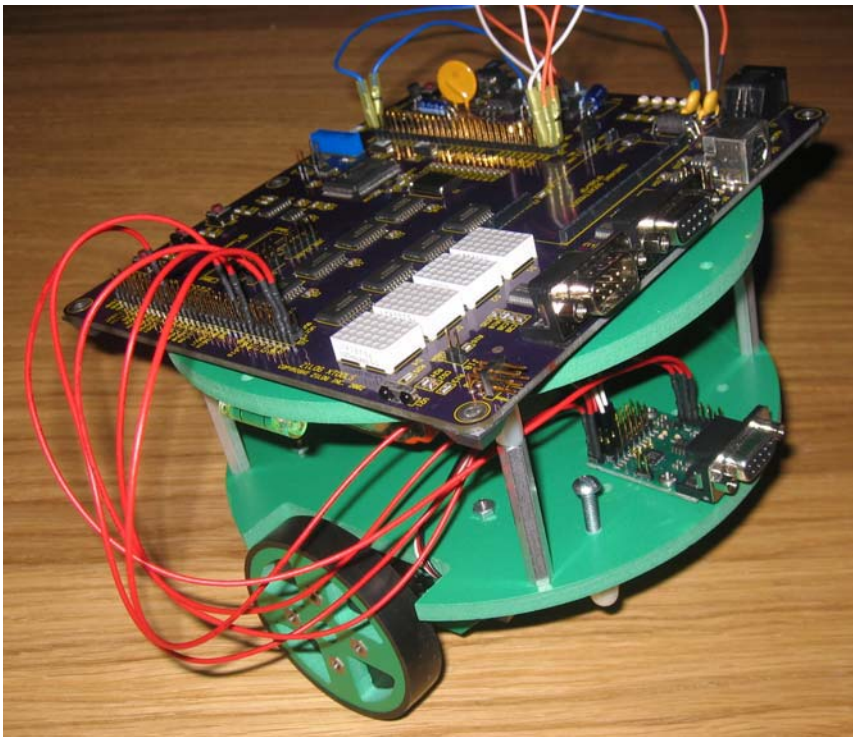
THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON DC

A Zilog ZNEO based Self-Balancing Robot with PID control.

Spencer Burdette
CS297: Embedded Systems
The George Washington University
Spring 2007



Abstract

This project undertakes the construction and implementation of a two-wheeled robot that is capable of balancing itself. The structural, mechanical, and electronic components of the bot are assembled in a manner that produces an inherently unstable platform that is highly susceptible to tipping in one axis.

The wheels of the robot are capable of independent rotation in two directions, each driven by a servo motor. A pololu servo motor controller provides current for the motors and generates the required pulse width modulated signals to position the servos. Information about the angle of the device relative to the ground (i.e. tilt) is obtained from two reflective object sensors mounted on the device. A Zilog ZNEO microcontroller receives sensor information from two analog-to-digital input ports and generates motor control signals on the serial output. The sensor information is fed to the microcontroller and is processed by a crude proportional, integral, derivative (PID) algorithm to generate compensating position control signals in order to balance the device.

Table of Contents

Bot Construction

The self balancing robot consists of a double-decker platform, two servo motors, two wheels, a servo motor controller, a microcontroller, two reflective object sensors, and various wires and connectors. The platform, wheels, and servo motors were purchased in a single package kit. Table 1 below is a detailed description of the components.

Component	Manufacturer	Supplier	Cost
Double decker base, wheels, and servo motors	Budget Robotics "Scooterbot"	www.budgetrobotics.com	$57.95
Motor controller	Pololu serial 8-servo controller	www.pololu.com	$26.95
Microcontroller	Zilog ZNEO Contest Kit	www.mouser.com	$49.99
Reflective object sensors	Fairchild QRB-1134 reflective object sensors	www.hobbyengineering.com	$2.25 (ea)
various wires, connectors, screws, and fasteners	various	various general stores (Home Depot, Target, etc.)	~$14.00

Table 1: Detailed list of robot components.

Selection of an appropriate base for the robot was a significant design decision. The base had to be relatively easy to construct as well as susceptible to tipping. The scooterbot platform from budgetrobotics.com proved to be the most attractive solution. The kit includes two decks, two wheels, two servo motors, the necessary wiring, and all of the required hardware. The assembly process is cataloged in photos in Appendix A, however the description below details the key steps.

Minor modifications were required to allow the robot to tip in one axis and to mount the sensors. First, the base of the device needs to be flipped before the upper deck is mounted to ensure that it rides more than an inch or so off of the ground (instead of the standard height of only a ¼ of an inch). This requires the 1" screws supporting the adjustable "skids" to be replaced with longer screws, specifically #8-32 1½" screws. The Fairchild sensors conveniently fit inside the bracket that mounts the servo motors to the base. This, too, requires the shorter mounting screws to be replaced with longer counterparts, specifically #4-40 1" screws.

The wires from the sensors and servo motors were routed along the bottom of the base to the top and tightened down with a zip tie. This prevents slack in the wires touching the ground or interfering with the sensors. The motor controller was mounted to the lower deck with screws that came with the scooterbot kit after drilling an additional hole. On the bottom of the top deck, a small bubble level (optional) and Velcro tabs for weights were mounted. The weights that I used for balancing were a battery pack and a heavy padlock. On the top surface of the upper deck, 4 velcro pads were attached to secure the microcontroller. The ZNEO board's center foot conveniently sits in the center hole of the top deck, while the 4 corner feet of the board hang off of the round deck.

Hardware Components

The hardware components that require interfacing are the microcontroller, the servo motor controller, and the sensors. The servo motors themselves connect directly to the servo controller, and thus do not require any wiring description.

The pololu servo motor controller is shown below in Figure 1. The controller handles the task of generating the varying with pulse width modulated signals that are required to position the two servo motors (the device can control up to 8 servo motors). It accepts 5 or 6 byte input commands that are sent 8 bits at a time with no parity and 1 stop bit (8N1) via the serial output pin. It supports commands to configure the controller and position the servos. The maximum baud rate is not deterministic, but is somewhere near 50K baud. This project worked reliably at 38K baud.

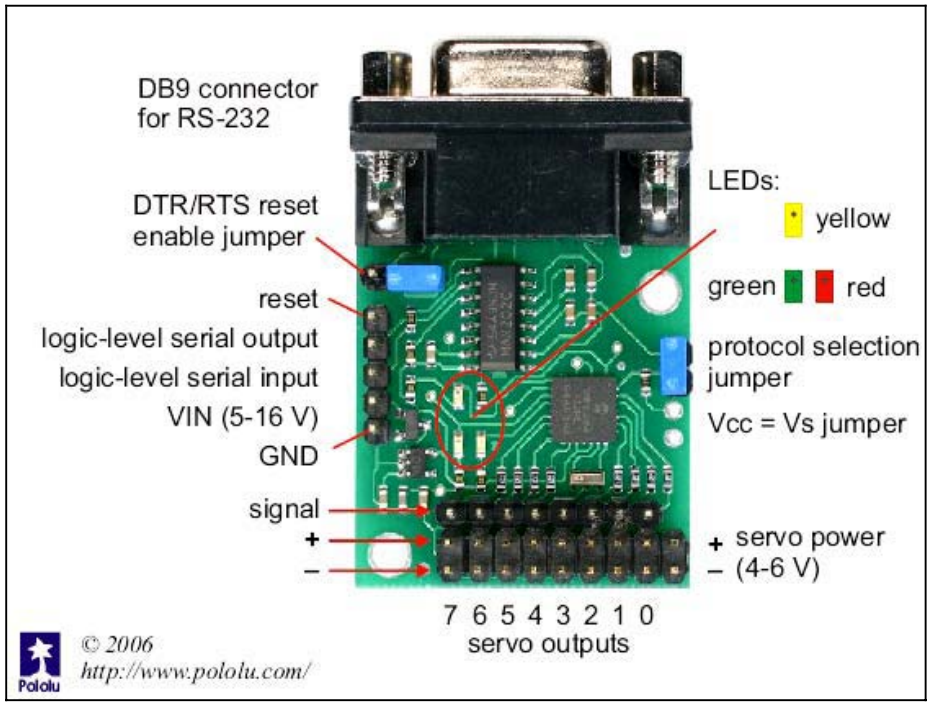


Figure 1: The Pololu serial servo motor controller.

The VIN, GND, servo power, servo ground, and logic level serial input pins are the only pins that are used. The DB9 connector is not used, and thus the similar pololu device that does not provide a DB9 connector would also have been a suitable choice. Note that the protocol selection jumper must be removed for operation with the self balancing bot.

Servo outputs 0 and 1 are used to control the two servo motors on the bot. It does not matter which output mates with which servo, since the bot does not necessarily have a concept of forward and aft.

The Fairchild QRB-1134 reflective object sensors consist of an infrared emitting diode and an NPN silicon phototransistor mounted side by side in a protective housing. The transistor varies the voltage on the collector based on the intensity of the reflected light from the LED that it detects. A 10K Ohm resistor is used to pullup the collector output to Vcc, therefore it is implied that the transisitor collector actually pulls down the voltage in operation. A schematic of the sensor is depicted below in Figure 2.

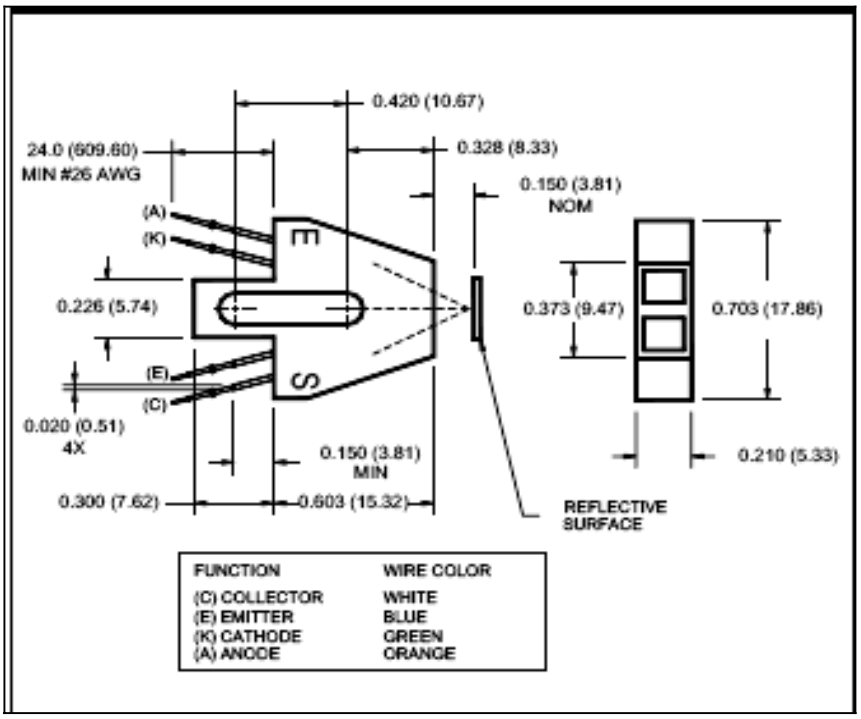


Figure 2: Schematic of the Fairchild QRB-1134 reflective object sensor.

The components were wired to the ZNEO controller as depicted below in Figure 3. Note that the wires of the reflective object sensors were modified into a connector with inline resistors as illustrated with photographs in Appendix B.

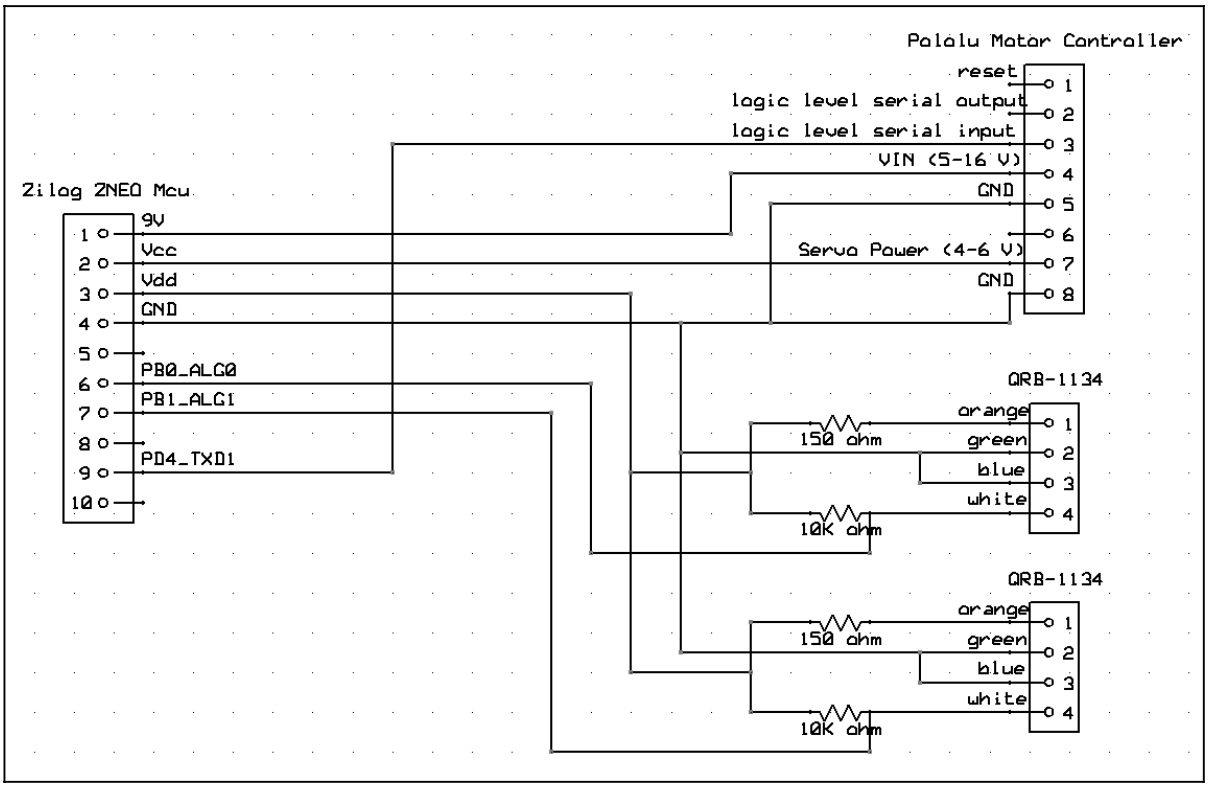


Figure 3: Wiring schematic for connecting the microcontroller to the motor controller and two reflective object sensors.

To further clarify the above diagram, the following is a tabular summary of the connections from the ZNEO board to the sensors and motor controller.

Device	Pin Description	ZNEO Pin connection
motor controller	VIN (9 -16V)	9V
	servo power (4-6V)	VCC
	GNDs	GND
	logic-level serial input	PD4_TXD1
QRB-1134 [1] and QRB-1134 [2]	LED anode (orange)	VDD
	LED cathode (green)	GND
	phototransistor emitter (blue)	GND
QRB-1134 [1]	phototransistor collector (white)	PB0_ALG1 (before resistor), VDD (after resistor)
QRB-1134 [2]	phototransistor collector (white)	PB0_ALG1 (before resistor), VDD (after resistor)

Table 2: Pin connections from the hardware devices to the microcontroller.

Software Modules

The nature of the project and the separation of hardware components leant itself nicely to a layered API. The API layers, in order of increasing abstraction, can be summarized as shown below. Each layer makes use of the previous layers in order to ultimately interface with the microcontroller peripherals and external devices.

- **Hardware abstraction layer** – timer, analog to digital conversion, button, LED, and serial interface components.
- **Device interaction layer** – motor controller and sensor input functionality.
- **Integration layer** –PID controller algorithms that processes input from the sensors and sends compensatory outputs to the servo controller.
- **User interface layer** – rich, menu-based UI available on the serial port in order to directly interact with the integration and device layers. Additionally, a "manual" interface is available on the ZNEO board itself (via the buttons and LED display) to configure the integration layer.

A diagram of the interaction among software layers is shown below in Figure 4.

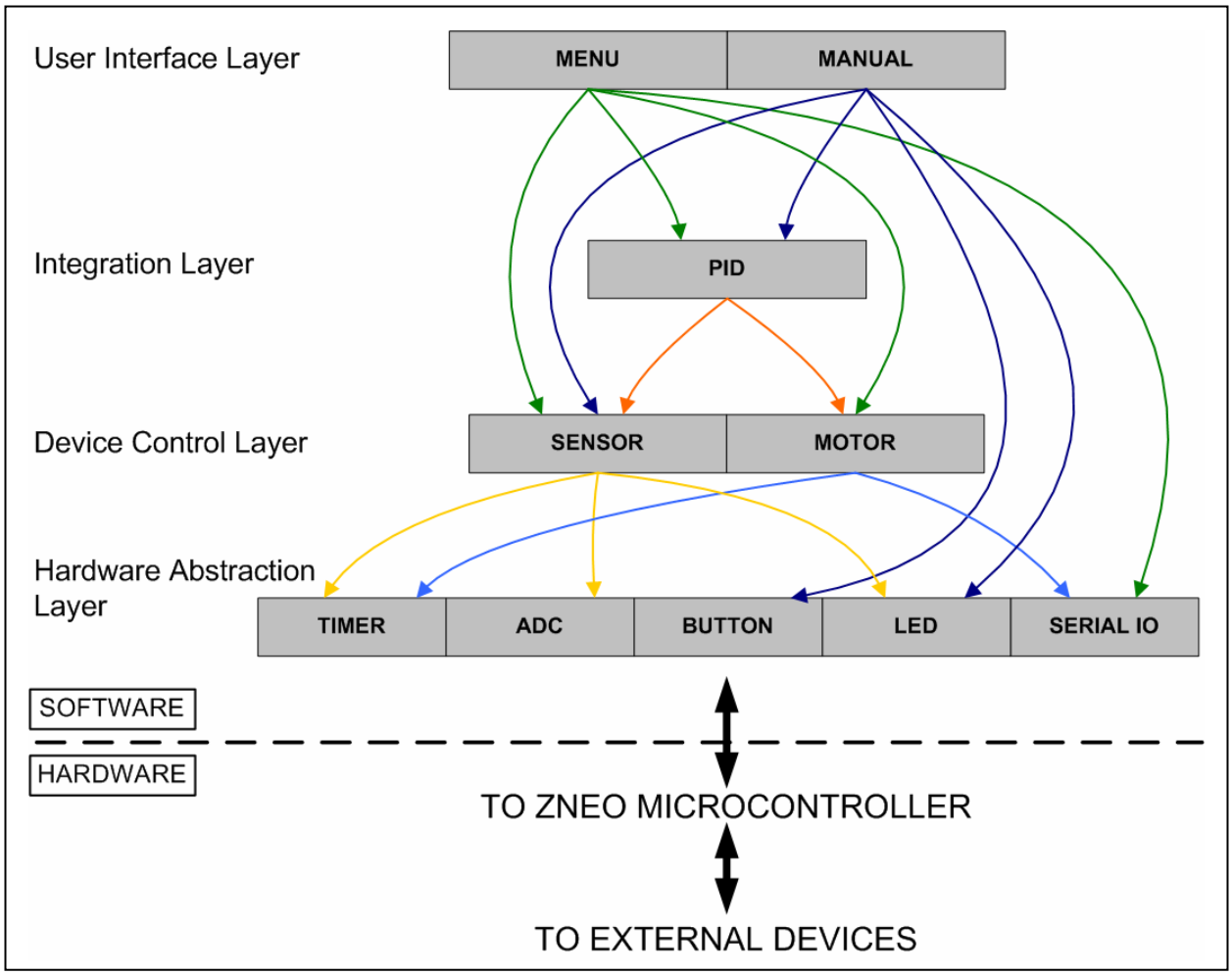


Figure 4: Interaction diagram of the software layers.

Note that the interaction among layers is not strictly hierarchical. Clearly, any layer is free to make use of the hardware abstraction layer, as that is its precise purpose. Notice also the menu component's interaction with the PID, motor, and sensor components. The menu interface was designed as a testing tool to allow the user to exercise the functionality provided by each of the underlying components. A detailed description of each layer is described in the following sections.

Hardware Abstraction Layer

The hardware abstraction layer handles the micro-controller specific port and register configurations that are required to interact with the onboard device peripherals. It exports an API to upper layers that is independent of the specific controller hardware. Ideally, it could allow an application that utilizes the layer to be run on an entirely different controller, presuming the device-specific glue code were properly ported.

The layer supports controlling the LEDs, analog-to-digital converter, timers, buttons, and serial interface. It can be expanded to support additional timer and serial modes, or to support additional ZNEO peripherals (such as the DMA or SPI interfaces, for example). Components that require interrupts typically accept a function pointer to their initialization function. For example, the button component initialization function has the following signature:

```
button_init_b0(unsigned int do_debounce, interrupt_cb icb);
```

The second argument is a function pointer to a function that is called when the given interrupt is triggered. This approach insulates the application writer from dealing with interrupt vector tables and interrupt service routine pragmas and the like. A summary of the useful hardware abstraction layer functions (in psuedocode) is as follows. Where the meaning is not immediately obvious from the name of the function, the respective source code headers in the project source code repository can be inspected.

ADC
```
adc_n_init(adc_port, is_continuous, priority, user_callback)
adc_get_analog_value(value_out)
```

Button
```
button_init_b<x>(do_debounce, user_callback)
```

Timer
```
timer_init_next(timeout, mode, priority, user_callback)
timer_n_init(timer_num, timeout, mode, priority, user_callback)
```

LED
```
led_init(refresh_microseconds)
set_display(display_text)
```

*SerialIO**
```
menu_get_input(input_destination, maximum_length)
```

**the serial input is currently married to the menu functionality, but should ideally be separated to its own component.*

Each of the components has been tested and has evolved through the completion of earlier labs for the class. This abstraction proves to be quite powerful in rapidly developing a prototype application. With the proper selection of functions from the hardware abstraction layer, a developer can effortlessly develop a test application that, for example, displays analog sensor inputs at a fixed interval on the LEDs in only a few lines of code.

Device Interaction Layer

The device interaction layer is responsible for retrieving inputs from the analog reflective object sensors and for sending commands to the motor controller. It also accepts and maintains configuration data for the sensors and motors. Like the hardware abstraction layer, the device interaction layer strives to present an API that insulates upper layers from the specific details of interacting with the devices.

The motor control layer provides a rich API that allows the setting and querying of a number of motor parameters. The caller of these functions is not responsible for formatting the commands and sending the data to the controller through the serial port.

Likewise, the sensor component provides the user with information about the raw sensor information by managing the ADC interaction itself. The sensor component can also calculate and report an error value, indicating the absolute lean of the robot. The range of the error is from -1000 to 1000, with 0 indicating level, a negative value indicating forward lean, and a positive value indicating an aft lean. Finally, the sensor layer manages calibration information for the two sensors, since it is ultimately responsible for determining the error based on this data. The following is an excerpt of useful functions from the interaction layer.

Motor

```
int motor_turn_on(which_servo)
int motor_turn_off(which_servo)
int motor_set_direction(which_servo, direction)
int motor_get_direction(which_servo)
int motor_set_position(which_servo, position)
int motor_get_position(which_servo)
```

Sensor

```
int sensor_set_sample_rate(sample_rate_us)
int sensor_calibrate_level()
int sensor_get_value(which_sensor, values_out)*
int sensor_is_grounded(values_in)
int sensor_calc_error(values_in)
```

**the sensor_get_value() function blocks until a new value is obtained from the ADC. This allows the application code to call the function in a continuous loop without worrying about timers and interrupts. The values are retrieved from the sensors at the sample rate set in sensor_set_sample_rate(). The function retrieves values from both sensors, despite the value of which_sensor, so therefore the function can block for up to 2 times sample_rate_us per invocation.*

Integration Layer

The integration layer is intended to intelligently respond to inputs from the sensor component by sending corrective output to the motor component. The manifestation of this functionality is a proportional, integral, and derivative controller. Briefly, a PID controller seeks to adjust the output of a device based on feedback from the inputs in order to achieve a specified goal or setpoint. Each component of the controller (proportional, integral, and derivative) calculates its contribution and provides its own adjustment in order to shape the behavior of the device. A more thorough explanation of the characteristics and algorithms associated with a PID controller can be found in the document "SystemControlWithFeedback.pdf."

The PID controller implements the algorithms for each of the three components. PID algorithms are, by nature, highly dependant on their respective gain factors. Properly tuned gains can stabilize the device output, while improperly tuned gains can dramatically deteriorate device stability. The integration layer supports the setting and querying of each of the component gains. It also serves as the main entry point for beginning control of the robot. Balance control is started and stopped with calls to the PID controller. The PID controller also supports the tuning of the device for a given number of iterations. The interaction layer functions are as follows:

PID

```
int pid_set_kp(kpin)
float pid_get_kp()

int pid_set_ki(kiin)
float pid_get_ki()

int pid_set_kd(kdin)
float pid_get_kd()

int pid_control_start(is_tune_mode, tune_iterations)
int pid_control_stop()*
```

**The pid_control_stop() function must be called from an interrupt context, since the pid_control_start() will execute on the processor without yielding until tune_iterations have elapsed.*

User Interface Layer

The user interface layer provides a means of controlling the functionality of the lower layer components at run time. This flexibility allows the developer to test the entire system without needing to recompile and reflash the controller between each test. The menu interface serves primarily as a tool to characterize, test, and validate the underlying component behaviors.

The menu based user interface provides a means to exercise the entire API provided by the motor, sensor, and PID components. Certain menus also display relevant information about the current state of the device. The user interacts with the controller across a serial link on the console port. A serial interface application such as hyperterm can display the options and retrieve inputs using the users PC. The main menu and motor, sensor, and PID control menus are demonstrated below.

```
Self-Balancing Bot Main Menu.
-----------------------------
  M Display motor control menu
  S Display sensor menu
  P Display PID controller menu
  R Run

Enter command [MSPR]:
```

Figure 5: User interface main menu.

```
Self-Balancing Bot Motor Menu
-----------------------------
  B Back to main menu.
  M Toggle which motor is being controlled. [BOTH]
  O Toggle servo motors on or off.  [OFF - OFF]
        D Set servo motor direction. [FWD - REV]
  R Set servo motor range. [15 - 15]
  S Set servo motor speed. [0 - 0]
  P Set servo motor position. [62 - 62]
  A Set servo motor position absolute. [0 - 0]
  N Set the servo motor neutral position. [0 - 0]
        C Oscillate motors.

Enter command [BMODRSPANC]:
```

Figure 6: Motor control menu.

```
Self-Balancing Bot Sensor Menu.
-----------------------------
  B Back to main menu.
  S Set sample interval. [50 ms]
  D Display sensor data on LEDs. [NONE]
  F Calibrate full forward tilt. [1023 - 1023]
  A Calibrate full aft tilt. [1023 - 1023]
  L Calibrate level. [845 - 719]
  C Calibrate corners. [115 - 109]
  T Set sensor tolerance band levels.[g:.05 l: .08: c: .02]

Enter command [BSDFALCT]:
```

Figure 7: Sensor interface menu.

```
Self-Balancing Bot PID Menu.
----------------------------
  B Back to main menu.
  S Set sample interval. [50 us]
  L Set system goal. [845 - 719]
  P Set proportional gain. [0.60]
  I Set integral gain. [0.02]
  D Set derivative gain. [0.15]
  T Tune gains.

Enter command [BSPIDRNE]:
```

Figure 8: PID control menu.

In addition to the serial-based menu interface, the project also required a means to tune the gain parameters on board the device. A "manual" mode was developed, in which the device is modified using the onboard buttons and LED display as a readout. Button 0 cycles the various input modes, and buttons 1 and 2 execute mode-specific actions as summarized below.

Mode (cycled by pressing button 0)	Button 1	Button 2
Calibrate forward tilt	start calibration	start calibration
Calibrate aft tilt	start calibration	start calibration
Calibrate corners	start calibration	start calibration
Calibrate level	start calibration	start calibration
Modify Kp	increment 0.02	decrement 0.02
Modify Ki	increment 0.002	decrement 0.002
Modify Kd	increment 0.002	decrement 0.002
Modify num cycles	increment 1	decrement 1
Start PID control	start	start
Enter console mode	start	start

Table 2: Summary of manual configuration modes and button functionality.

Implementation Issues

There were a number of implementation issues with the project, most of which were overcome during development. The issues corresponded to each of the three high level components: the motor controller, sensors, and PID controller.

Servo Motor Control Issues

A standard servo motor typically rotates through a maximum range of only 90 or 180 degrees. It employs a potentiometer or other feedback device in order to assure accurate positioning of the servo. The servo motors that ship with the scooterbot kit, however, have been modified for continuous rotation. This allows the servo to continuously rotate 360 degrees, at the expense of sacrificing accurate position control. The servo motors did not come with documentation, and the pololu servo controller provides a command interface with the assumption that it is controlling standard servos. This discrepancy was the initial impetus for creating the menu based interface in order to exercise and characterize the exact behavior of the customized servo motors.

First, a brief description of the servo controller command interface is required. A single servo command consists of a sequence of five or six bytes as follows:

start byte = 0x80	device ID = 0x01	command	servo num	data 1	data 2

The six available commands are sent as 0x00 through 0x05 and are: [*set parameters, set speed, set range, set position (7-bit), set position (8-bit), set position absolute, and set position neutral*]. The values for data1 and data2 are particular to each command, but in all commands the most significant (7^{th}) bit must be cleared to zero. The motor controller is not committed to a maximum or minimum baud rate, but is suspected to reach its limits around 50K baud. This project reliably communicates with the controller at 38K baud.

The development of the motor API logically follows from an inspection of the motor controller command set. Using the motor component API and the menu interface, it was determined that set neutral, set absolute, set speed, and set range commands have an unpredictable (or at least difficult to characterize and illogical) impact on the position of the motor controllers.

This process did, in fact, provide information as to how to properly control the motors. The servo motors rotate in a direction and speed proportional to the 7 bit position that is specified. The neutral value for each servo is 62, so a positional value of 61 will turn the motor very slowly in the forward direction while the value of 10 will turn the motor rapidly forward. Likewise, values of 63 and 120 will turn the motor slow and rapid (respectively) in the reverse direction.

At this point it was apparent that one of the servo motors was slightly offset from the neutral point. This was elicited when setting the left servo to the neutral point (62) and observing slight forward rotation. A compensating factor was implemented *within* the motor control API, so that the user could send identical positions for the left and right servos and obtain the expected identical output.

Another issue that was brought to light by the motor menu interface was a power consumption problem. When the motors were commanded to rapidly reverse directions, the ZNEO controller would occasionally reset. I hypothesized that the reason for this was that the motor controller was drawing a substantial amount of current that was sufficient to reset the device. I remedied this error by inserting a small delay between sending the commands to the left servo and right servo. Trial and error elicited a minimum delay of approximately 50 ms. This delay has significant impact on the PID controller and hence the overall stability of the device; since the controller can adjust the servo motor positions **at most** only 20

times per second. In general, however, the characterization of the motor behavior was attained and the servo motor issues were overcome.

Sensor Issues

The reflective object sensors are both rapidly responsive and highly sensitive, two characteristics that are desirable for this robot. The downside of the devices, however, remains their short range and their non-linear response.

The *very* short range of the devices can be a ¼" or lower, depending on the resistance value selected for the diode. When mounted on the robot, the forward and aft sensors entered a state of "saturation" (i.e. maximum values) before the robot was aligned level. This saturation, in effect, creates a dead zone where the device cannot detect any further tilt until the sensor re-enters its operating range. The solution to this problem was to raise the sensors view of the ground a small amount (about an eighth of an inch) by affixing a piece of cardboard to double sided tape along the sensor's path.

The most significant problem with the sensors is their non-linear response. The diagram below is excerpted from the QRB-1134 datasheet.

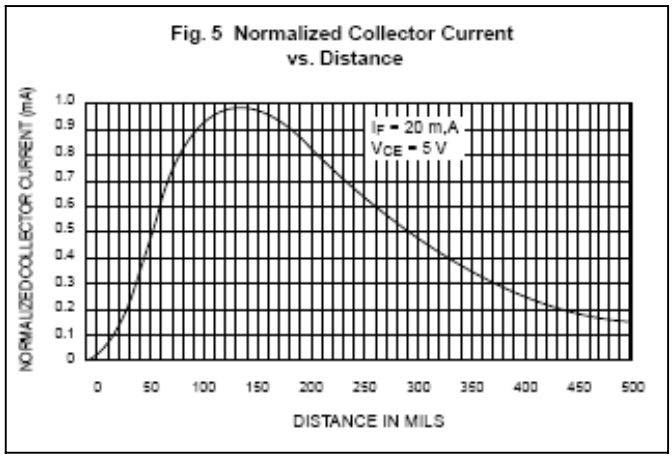


Figure 9: Reflective object non-linear sensor response, current vs. distance.

Had I been more experienced, I may have identified this as a potential problem prior to purchasing and utilizing the sensors. The nature of the problem is that at any given position along the sensors response curve, one cannot tell if it has passed the maximum response value or not. Take for example a voltage reading of 4 V (corresponding to a current above of .4ma, remember we used a 10 Kohm resistor and V=IR), looking across the graph the sensor could be reading a distance of 50 mils or 325 mils. Without more information, it is impossible to determine which value is the correct distance.

My initial solution was to investigate the position information on the other sensor whenever there was ambiguity with a reading. Clearly, only one sensor could be reporting a far-side or near-side (relative to the maximum) distance at a time. The problem with this approach, however, returns to the issue of the limited sensor range. For most of the operating range of one sensor, the other sensor is in a state of saturation, thus no information can be gleaned from its output.

The second solution appears obvious at first glance – if you know the starting position of the sensor, you can track it as it traverses back and forth across the maximum. This solution is sound in theory, but does unfortunately does not hold up in practice. This is attributed to the rapid change in position of the bot and the sample interval. There are often times where the maximum value is traversed in between sensor intervals, and the controller cannot determine if the sensor has traveled *through* the maximum or if it has simply reached the maximum and turned around. This led to the development of a series of tolerance bands about the maximum values which, once entered, assumed the device would be exiting the other

side and toggled an *is_past_corner* value. This value could be consulted to determine the true distance. Unfortunately, this solution proved problematic again due to the sample rate. Furthermore, an error detection and correction scheme was implemented which would repair the value of *is_past_corner* when known positions were reached (i.e. both sensors registering "level" values). This solution provided only marginal improvements.

Ultimately, the ambiguity of certain sensor readings proved to be the undoing of the PID controller. Within known ranges, the controller operated reliably and the sensors provided useful feedback. Beyond these ranges, however, the error could not be accurately calculated since the function could not confidently know the true position values.

PID Issues

Surprisingly, the PID controller turned out to be one of the simplest and most straightforward components to implement. The first issue to overcome was the discrepancy between the error domain and the motor control domain. As stated earlier, error values range between -1000 and 1000, while the servo motor controller requires values in the range of 0 – 127 (with 62 being neutral). This domain mapping was easily achieved by applying some mathematics.

There are two remaining issues with the controller that need to be identified. I am confident that these issues could have been solved empirically had the sensor problem been resolved. The first issue is related to the integral component. The function of the integral component is to sum the error over time and influence the output proportional to the sum. This tactic allows a controller to overcome steady-state error conditions by essentially "ramping up" the output when it is observed that the current output is not achieving the desired effect. Note that since the integral component is strictly a sum and the error can be positive or negative, that the reversing of direction has the effect of reducing the sum. The problem is that when the error reads zero (i.e. the bot is level), a pure integral algorithm will retain for a small time a remaining sum component which will force the output to knock the bot from its level stance. I believe a solution may be to entirely clear the integral sum when it is determined that the bot has reached (or oscillated beyond) level.

The other problem is related to the nature of the servo motors. In a pure PID implementation, the controller will not produce any output when the error is zero. With this project, however, recall that the servo motors have been modified for continuous rotation. This implies that they will continue rotating at their current speed if they do not receive any input. Clearly, this action quickly knocks the bot from level when the PID reads a zero error value. I believe that setting the servos to the neutral position upon a zero error will eliminate this issue and thus enhance stability.

Lessons Learned

Working on the implementation of a full scale PID controller taught me volumes about system control. I now have a clearer and more concrete understanding of the contributions of each of the PID components, as well as their interactions. It was very rewarding to develop the mathematical relationships between the sensing domain and control domain, and to watch the controller calculate compensating outputs.

The most essential lesson I have learned is to properly scrutinize a device data sheet before selecting the component for integration into a project. Had I been wearier of the potential issues, I may have rejected the reflective object sensors as potential tilt measuring sensors due to their non-linear output. I ultimately attribute the shortfall of this project to the nature of the sensors, so clearly this decision was a stinging lesson to learn.

Future Work

The components for the self balancing robot were obtained independently. Therefore, work can continue on the project despite the course being over. I have a high degree of confidence in my PID implementation, as well as my motor control and sensor APIs. I feel that with the proper adjustment (or replacement) of the tilt sensors, that I will achieve success in balancing the robot over an extended period of time.

My first attempt will be to increase the sample rate of the sensors. Typically, PID controllers only sample their input at the same rate as they generate output. This is logical, since after all there is little sense in wasting power to sample your environment if you are not going to immediately act on it. I adhered to this principle in my implementation when, in fact, it turns out there *is* a purpose to sampling faster than I calculate outputs. The benefit of an increased sample rate would be an increase in the resolution and accuracy of tracking the position on the response curve of the sensor data. If I can reliably and repeatedly detect when a sensor crosses its maximum value, I will be able to calculate accurate error values.

If the increased sensor sample rate does not work, the reflective object sensors will have to be replaced with a more suitable gyroscopic, angle, or tilt sensor. Ideally, the PID code can remain unchanged, since it conforms to the sensor API, and only the underlying sensor implementation will need to be changed to accommodate the replacement device.

Appendix A: Self Balancing Bot Construction

Figure A1: The unassembled scooterbot components.

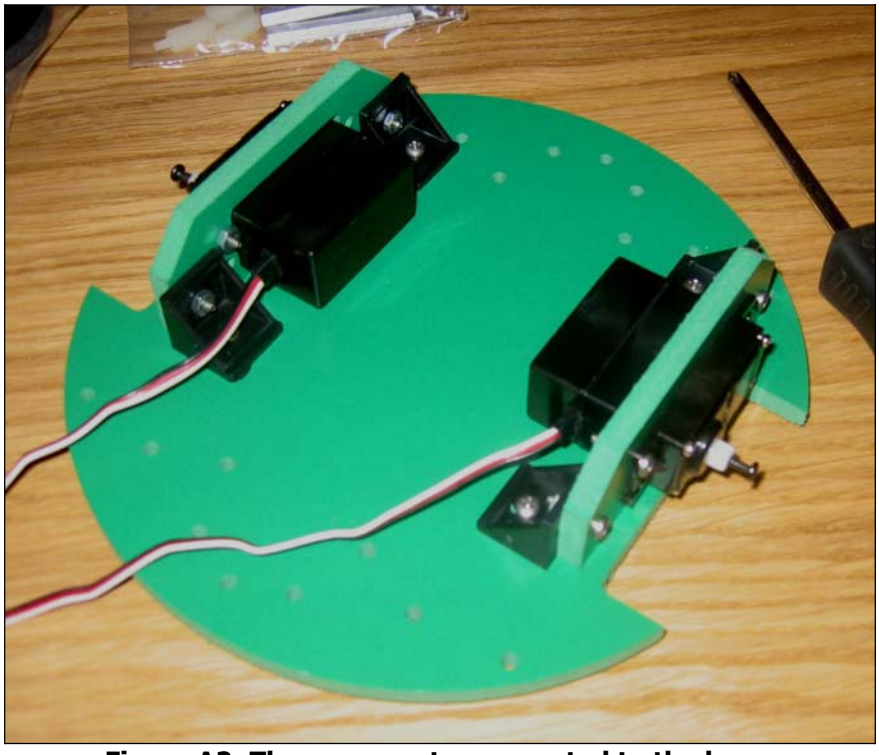


Figure A2: The servo motors mounted to the base.

Figure A3: Assembling the wheels.

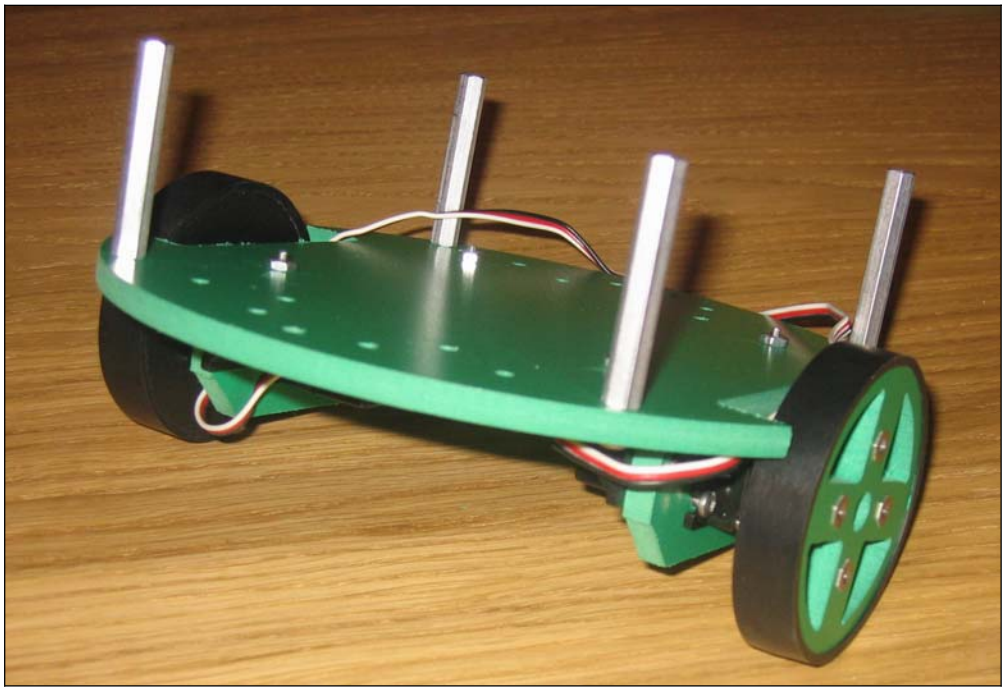


Figure A4: Inverting the base provides the necessary additional ground clearance.

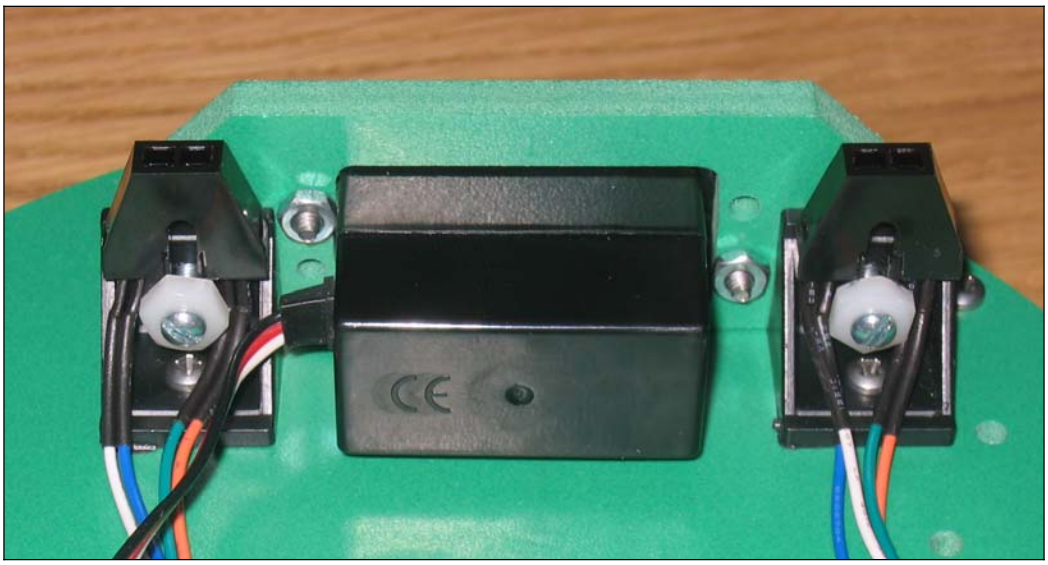


Figure A5: Photo reflective sensors mounted to the servo-mount brackets. View is of the underside of the base platform.

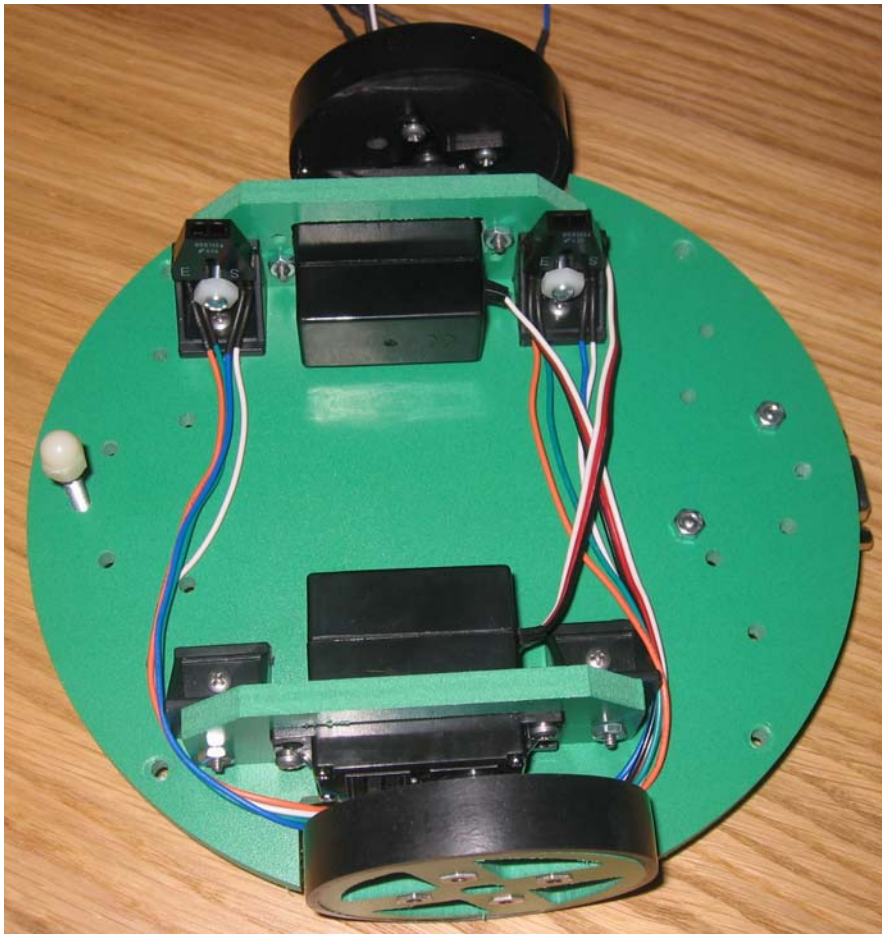


Figure A6: Routing the wires to the opposite wheel well along the underside of the bottom deck.

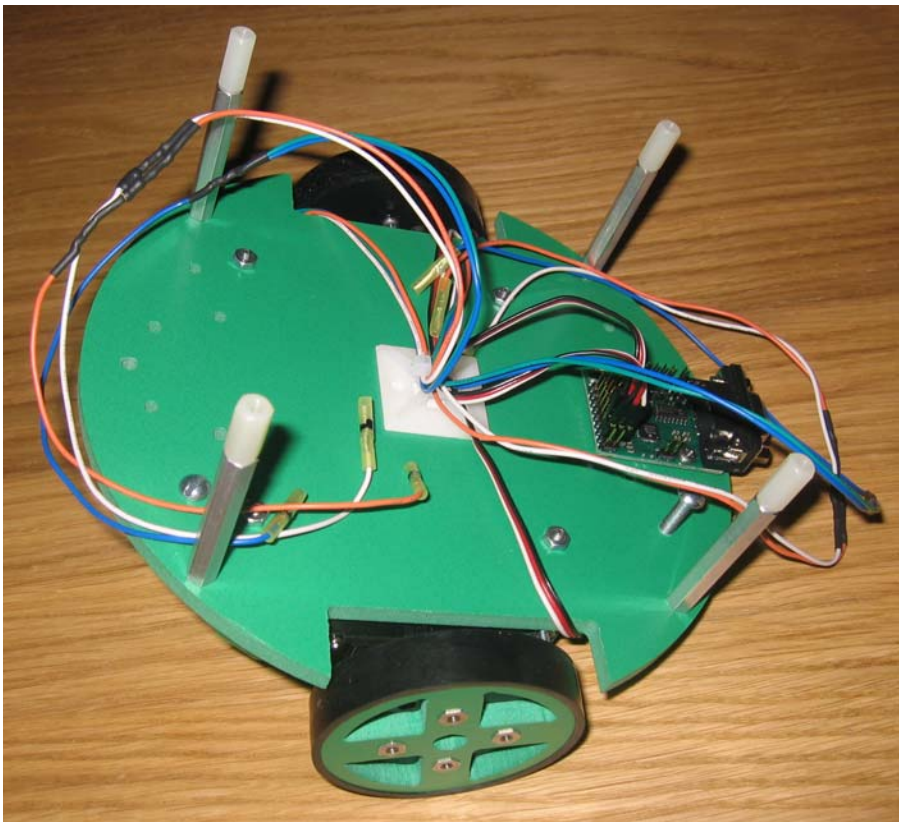


Figure A7: Top view of the bottom deck showing the mounted motor controller and wire tie-down. Note that the QRB-1134 connector assembly is described in Appendix B.

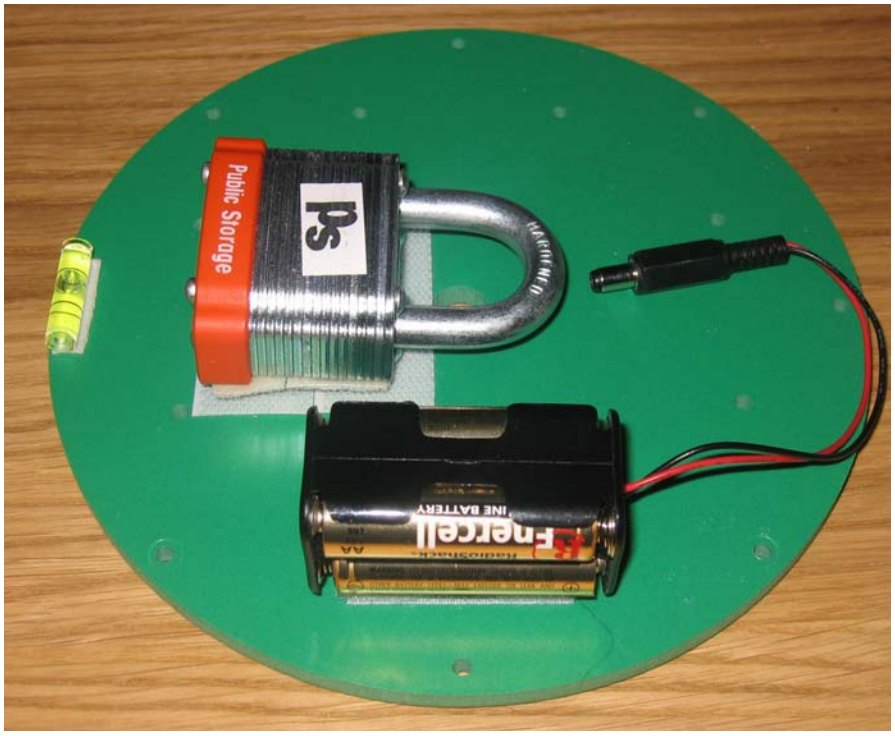


Figure A8: Mounting weights and optional bubble level with Velcro to underside of the top deck.

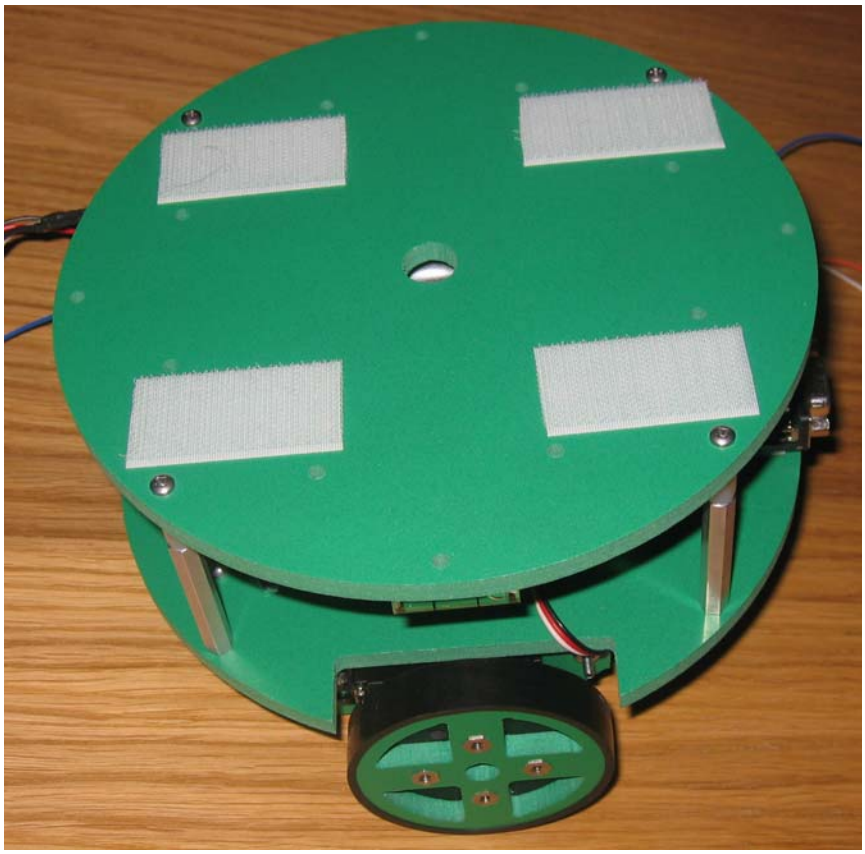


Figure A9: Top deck installed and Velcro strips ready to accept ZNEO microcontroller board.

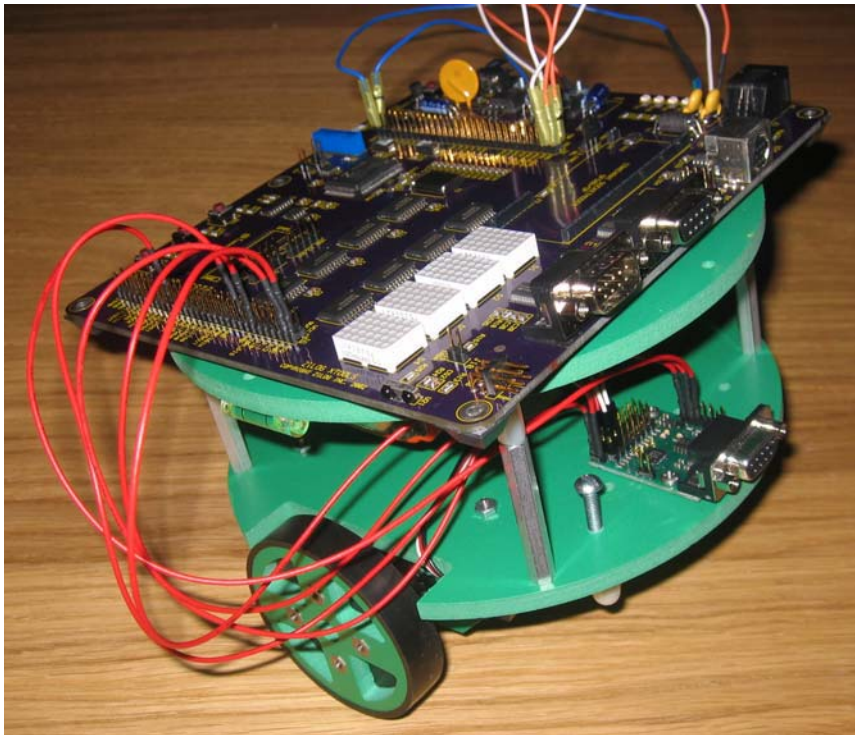


Figure A10: The final assembled and wired self balancing robot.

Appendix B: QRB-1134 Reflective Object sensor connector assembly

The following idea was influenced by a guide on www.ranchbots.com/club/papers/QRB1134%20quickconnect.pdf. Essentially, the connectors involve the soldering of two resistors inline with the power and signal wires of each sensor. The negative leads are also soldered together, and connectors are crimped to the wire ends.

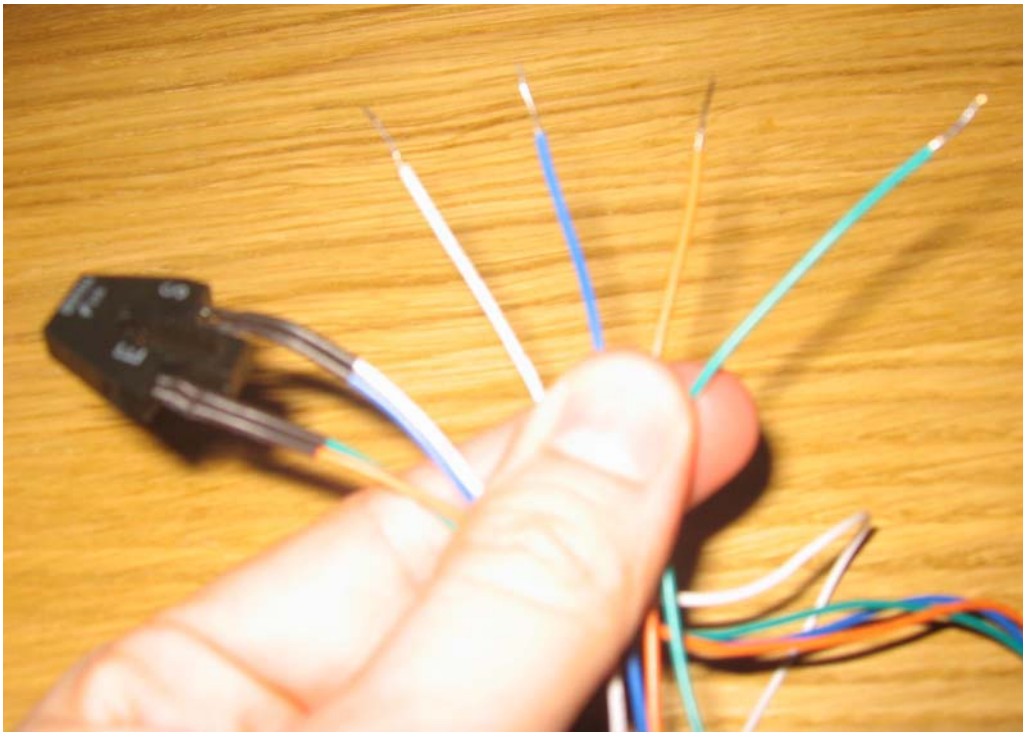


Figure B1: Cut approximately 8 inches off of each wire (save the cut lengths) and strip the wires about ¾ of an inch.

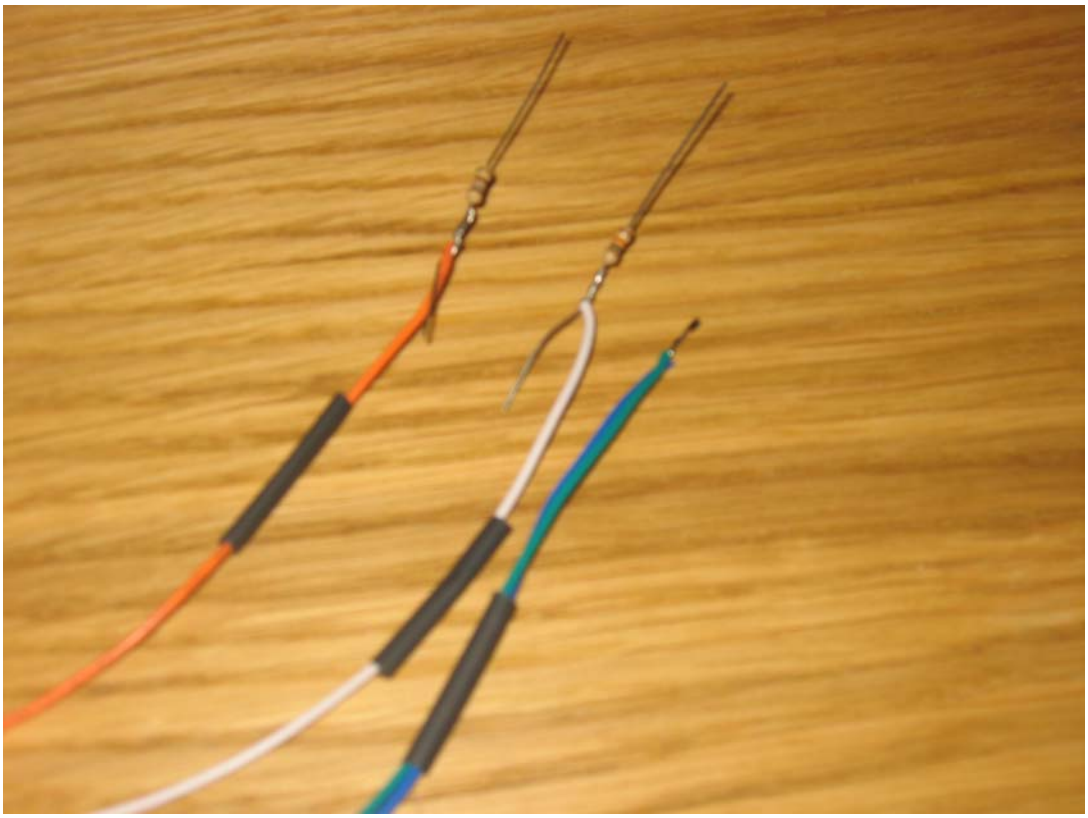


Figure B2: Hand twist the green and blue leads together. Slip 1" lengths of shrink tubing over each of the wires (the blue and green wires are considered one wire beyond this point). Hand twist the 150 Ohm resistor to the orange wire and the 10K Ohm resistor to the white wire.

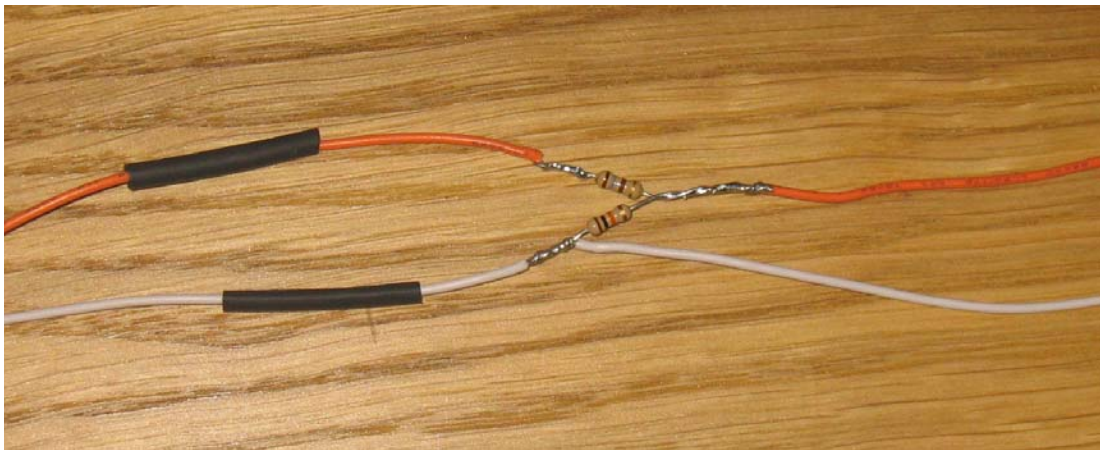


Figure B3: Hand twist the free resistor leads from the orange and white wires together. Twist the 8" cut length of orange wire to the intertwined resistors. Twist the 8" cut length of white wire to the <u>top</u> of the resistor on the white wire. The new length of white wire should be joined to the attached white wire on the side towards the sensor (not towards the free ends). At this point the excess resistor leads can be trimmed and the connections can be soldered.

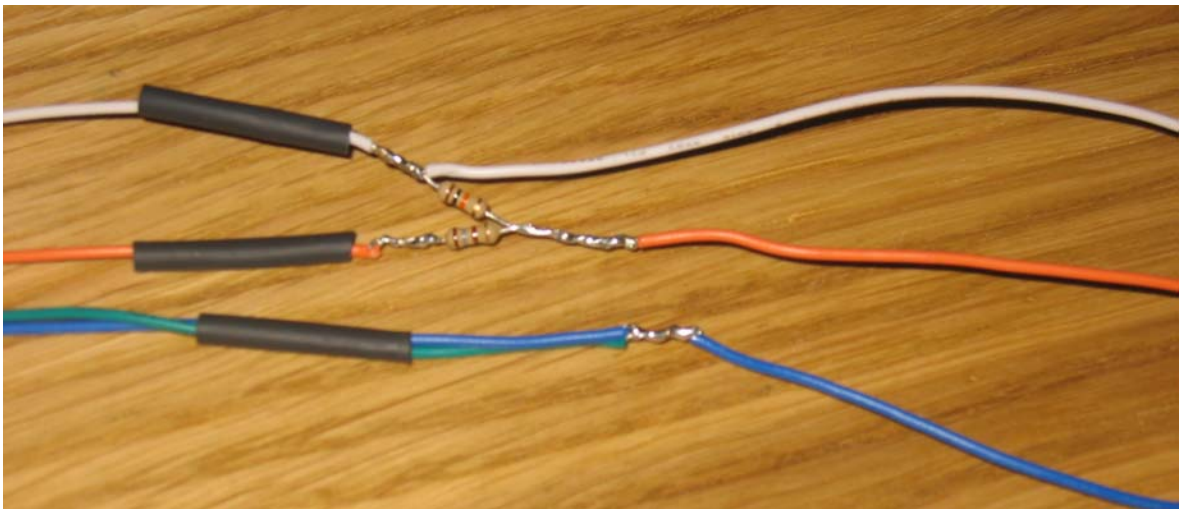


Figure B4: Twist and solder the 8" cut length of the blue wire to the intertwined blue/green wires.

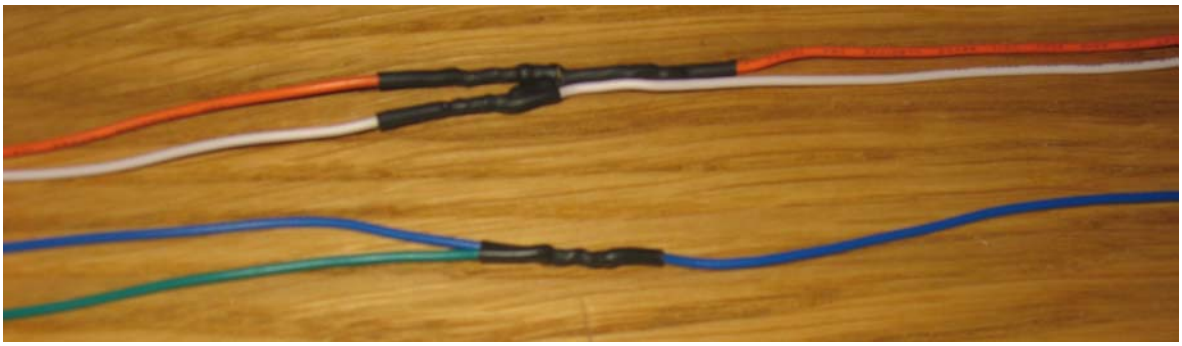


Figure B5: Position and heat the shrink tubing to protect the joints.

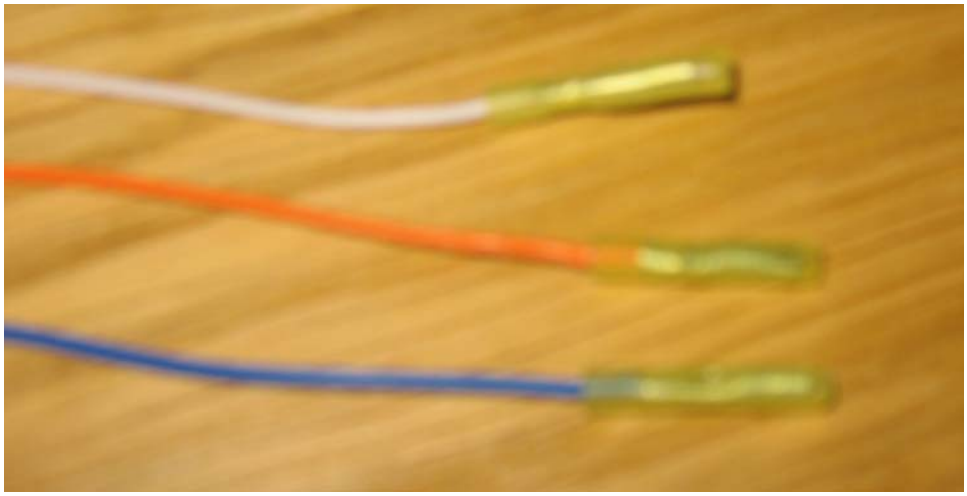


Figure B6: Crimp connectors on the free end of the wires.

Appendix C: Project Settings

The following XML can be pasted into an xxx.zdsproj file in order to reproduce the working project settings. The version of the Zilog IDE is the ZDS II – ZNEO 4.10.2

```
<project type="Executable" project-type="Standard" configuration="Debug"
created-by="b:4.10:06062301" modified-by="b:4.10:06062301">
<cpu>Z16F2811AL</cpu>

<!-- file information -->
<files>
<file filter-key="">src\timer.c</file>
<file filter-key="">src\button.c</file>
<file filter-key="">src\led.c</file>
<file filter-key="">src\clock.c</file>
<file filter-key="">src\adc.c</file>
<file filter-key="">src\tilt_sensor_data_collect.c</file>
<file filter-key="">src\tilt_sensor_calibrate.c</file>
<file filter-key="">src\motor.c</file>
<file filter-key="">src\selfbalancebot_main.c</file>
<file filter-key="">src\sensor.c</file>
<file filter-key="">src\menu.c</file>
<file filter-key="">src\pid.c</file>
</files>

<!-- configuration information -->
<configurations>
<configuration name="Debug" >
<tools>
<tool name="Assembler">
<options>
<option name="define" type="string" change-
action="assemble">_Z16F2811AL=1,_Z16F_SERIES=1</option>
<option name="include" type="string" change-action="assemble"></option>
<option name="list" type="boolean" change-action="none">true</option>
<option name="listmac" type="boolean" change-action="none">false</option>
<option name="name" type="boolean" change-action="none">true</option>
<option name="pagelen" type="integer" change-action="none">56</option>
<option name="pagewidth" type="integer" change-action="none">80</option>
<option name="quiet" type="boolean" change-action="none">true</option>
</options>
</tool>
<tool name="Compiler">
<options>
<option name="chartype" type="string" change-action="compile">U</option>
<option name="define" type="string" change-
action="compile">_Z16F2811AL,_Z16F_SERIES</option>
<option name="genprintf" type="boolean" change-action="compile">true</option>
<option name="keepasm" type="boolean" change-action="none">false</option>
<option name="keeplst" type="boolean" change-action="none">true</option>
<option name="list" type="boolean" change-action="none">false</option>
<option name="listinc" type="boolean" change-action="none">false</option>
<option name="model" type="string" change-action="compile">S</option>
<option name="modsect" type="boolean" change-action="compile">false</option>
<option name="stdinc" type="string" change-action="compile"></option>
<option name="usrinc" type="string" change-action="compile">hdr</option>
<option name="regvar" type="boolean" change-action="compile">false</option>
<option name="regvarcache" type="boolean" change-action="none">false</option>
<option name="reduceopt" type="boolean" change-action="compile">true</option>
```

```
<option name="watch" type="boolean" change-action="none">false</option>
</options>
</tool>
<tool name="Debugger">
<options>
<option name="target" type="string" change-
action="rebuild">Z16F2811AL</option>
<option name="debugtool" type="string" change-
action="none">USBSmartCable</option>
</options>
</tool>
<tool name="FlashProgrammer">
<options>
<option name="erasebeforeburn" type="boolean" change-
action="none">false</option>
<option name="eraseinfopage" type="boolean" change-
action="none">false</option>
<option name="enableinfopage" type="boolean" change-
action="none">false</option>
<option name="includeserial" type="boolean" change-
action="none">false</option>
<option name="offset" type="integer" change-action="none">0</option>
<option name="snenable" type="boolean" change-action="none">false</option>
<option name="sn" type="string" change-action="none">0</option>
<option name="snsize" type="integer" change-action="none">0</option>
<option name="snstep" type="integer" change-action="none">0</option>
<option name="snstepformat" type="integer" change-action="none">0</option>
<option name="snaddress" type="string" change-action="none">0</option>
<option name="snformat" type="integer" change-action="none">0</option>
<option name="snbigendian" type="boolean" change-action="none">true</option>
<option name="singleval" type="string" change-action="none">0</option>
<option name="singlevalformat" type="integer" change-action="none">0</option>
</options>
</tool>
<tool name="General">
<options>
<option name="warn" type="boolean" change-action="none">true</option>
<option name="debug" type="boolean" change-action="assemble">true</option>
<option name="debugcache" type="boolean" change-action="none">true</option>
<option name="igcase" type="boolean" change-action="assemble">false</option>
<option name="outputdir" type="string" change-
action="compile">Debug\</option>
</options>
</tool>
<tool name="Librarian">
<options>
<option name="outfile" type="string" change-
action="build">.\Debug\counter.lib</option>
<option name="warn" type="boolean" change-action="none">false</option>
</options>
</tool>
<tool name="Linker">
<options>
<option name="directives" type="string" change-action="build"></option>
<option name="createnew" type="boolean" change-action="build">true</option>
<option name="eram" type="string" change-action="build">800000-
87FFFF</option>
```

```
<option name="erom" type="string" change-action="build">008000-
01FFFF</option>
<option name="exeform" type="string" change-
action="build">OMF695,INTEL32</option>
<option name="fplib" type="string" change-action="build">Real</option>
<option name="iodata" type="string" change-action="build">FFC000-
FFFFFF</option>
<option name="linkctlfile" type="string" change-action="build"></option>
<option name="map" type="boolean" change-action="none">true</option>
<option name="maxhexlen" type="integer" change-action="build">64</option>
<option name="objlibmods" type="string" change-action="build"></option>
<option name="of" type="string" change-action="build">Debug\counter</option>
<option name="padhex" type="boolean" change-action="build">false</option>
<option name="quiet" type="boolean" change-action="none">true</option>
<option name="ram" type="string" change-action="build">FFB000-FFBFFF</option>
<option name="relist" type="boolean" change-action="build">false</option>
<option name="rom" type="string" change-action="build">000000-007FFF</option>
<option name="sort" type="string" change-action="none">NAME</option>
<option name="startuplnkcmds" type="boolean" change-
action="build">true</option>
<option name="startuptype" type="string" change-
action="build">Standard</option>
<option name="undefisfatal" type="boolean" change-action="none">true</option>
<option name="useadddirectives" type="boolean" change-
action="build">false</option>
<option name="usecrun" type="boolean" change-action="build">true</option>
<option name="warnoverlap" type="boolean" change-action="none">true</option>
<option name="warnisfatal" type="boolean" change-action="none">false</option>
<option name="xref" type="boolean" change-action="none">false</option>
</options>
</tool>
</tools>
</configuration>
<configuration name="Release" >
<tools>
<tool name="Assembler">
<options>
<option name="define" type="string" change-
action="assemble">_Z16F2811AL=1,_Z16F_SERIES=1</option>
<option name="include" type="string" change-action="assemble"></option>
<option name="list" type="boolean" change-action="none">true</option>
<option name="listmac" type="boolean" change-action="none">false</option>
<option name="name" type="boolean" change-action="none">true</option>
<option name="pagelen" type="integer" change-action="none">56</option>
<option name="pagewidth" type="integer" change-action="none">80</option>
<option name="quiet" type="boolean" change-action="none">true</option>
</options>
</tool>
<tool name="Compiler">
<options>
<option name="chartype" type="string" change-action="compile">U</option>
<option name="define" type="string" change-
action="compile">_Z16F2811AL,_Z16F_SERIES</option>
<option name="genprintf" type="boolean" change-action="compile">true</option>
<option name="keepasm" type="boolean" change-action="none">false</option>
<option name="keeplst" type="boolean" change-action="none">true</option>
<option name="list" type="boolean" change-action="none">false</option>
<option name="listinc" type="boolean" change-action="none">false</option>
```

```
<option name="model" type="string" change-action="compile">S</option>
<option name="modsect" type="boolean" change-action="compile">false</option>
<option name="stdinc" type="string" change-action="compile"></option>
<option name="usrinc" type="string" change-action="compile">hdr</option>
<option name="regvar" type="boolean" change-action="compile">true</option>
<option name="regvarcache" type="boolean" change-action="none">false</option>
<option name="reduceopt" type="boolean" change-
action="compile">false</option>
<option name="watch" type="boolean" change-action="none">false</option>
</options>
</tool>
<tool name="Debugger">
<options>
<option name="target" type="string" change-
action="rebuild">Z16F2811AL</option>
<option name="debugtool" type="string" change-
action="none">USBSmartCable</option>
</options>
</tool>
<tool name="FlashProgrammer">
<options>
<option name="erasebeforeburn" type="boolean" change-
action="none">false</option>
<option name="eraseinfopage" type="boolean" change-
action="none">false</option>
<option name="enableinfopage" type="boolean" change-
action="none">false</option>
<option name="includeserial" type="boolean" change-
action="none">false</option>
<option name="offset" type="integer" change-action="none">0</option>
<option name="snenable" type="boolean" change-action="none">false</option>
<option name="sn" type="string" change-action="none">0</option>
<option name="snsize" type="integer" change-action="none">0</option>
<option name="snstep" type="integer" change-action="none">0</option>
<option name="snstepformat" type="integer" change-action="none">0</option>
<option name="snaddress" type="string" change-action="none">0</option>
<option name="snformat" type="integer" change-action="none">0</option>
<option name="snbigendian" type="boolean" change-action="none">true</option>
<option name="singleval" type="string" change-action="none">0</option>
<option name="singlevalformat" type="integer" change-action="none">0</option>
</options>
</tool>
<tool name="General">
<options>
<option name="warn" type="boolean" change-action="none">true</option>
<option name="debug" type="boolean" change-action="assemble">false</option>
<option name="debugcache" type="boolean" change-action="none">false</option>
<option name="igcase" type="boolean" change-action="assemble">false</option>
<option name="outputdir" type="string" change-
action="compile">Release\</option>
</options>
</tool>
<tool name="Librarian">
<options>
<option name="outfile" type="string" change-
action="build">.\Release\counter.lib</option>
<option name="warn" type="boolean" change-action="none">false</option>
</options>
```

```
</tool>
<tool name="Linker">
<options>
<option name="directives" type="string" change-action="build"></option>
<option name="createnew" type="boolean" change-action="build">true</option>
<option name="eram" type="string" change-action="build">800000-
87FFFF</option>
<option name="erom" type="string" change-action="build">008000-
01FFFF</option>
<option name="exeform" type="string" change-
action="build">OMF695,INTEL32</option>
<option name="fplib" type="string" change-action="build">Real</option>
<option name="iodata" type="string" change-action="build">FFC000-
FFFFFF</option>
<option name="linkctlfile" type="string" change-action="build"></option>
<option name="map" type="boolean" change-action="none">true</option>
<option name="maxhexlen" type="integer" change-action="build">64</option>
<option name="objlibmods" type="string" change-action="build"></option>
<option name="of" type="string" change-
action="build">Release\counter</option>
<option name="padhex" type="boolean" change-action="build">false</option>
<option name="quiet" type="boolean" change-action="none">true</option>
<option name="ram" type="string" change-action="build">FFB000-FFBFFF</option>
<option name="relist" type="boolean" change-action="build">false</option>
<option name="rom" type="string" change-action="build">000000-007FFF</option>
<option name="sort" type="string" change-action="none">NAME</option>
<option name="startuplnkcmds" type="boolean" change-
action="build">true</option>
<option name="startuptype" type="string" change-
action="build">Standard</option>
<option name="undefisfatal" type="boolean" change-action="none">true</option>
<option name="useadddirectives" type="boolean" change-
action="build">false</option>
<option name="usecrun" type="boolean" change-action="build">true</option>
<option name="warnoverlap" type="boolean" change-action="none">true</option>
<option name="warnisfatal" type="boolean" change-action="none">false</option>
<option name="xref" type="boolean" change-action="none">false</option>
</options>
</tool>
</tools>
</configuration>
</configurations>

<!-- watch information -->
<watch-elements>
<watch-element expression="motors" />
<watch-element expression="sensors" />
<watch-element expression="pid" />
</watch-elements>

<!-- breakpoint information -->
<breakpoints>
</breakpoints>

</project>
```