Electronic Control of a Two-Dimensional, Knee-less, Bipedal Robot

Final Report for T&AM 492
under Professor Andy Ruina
August 2, 2005

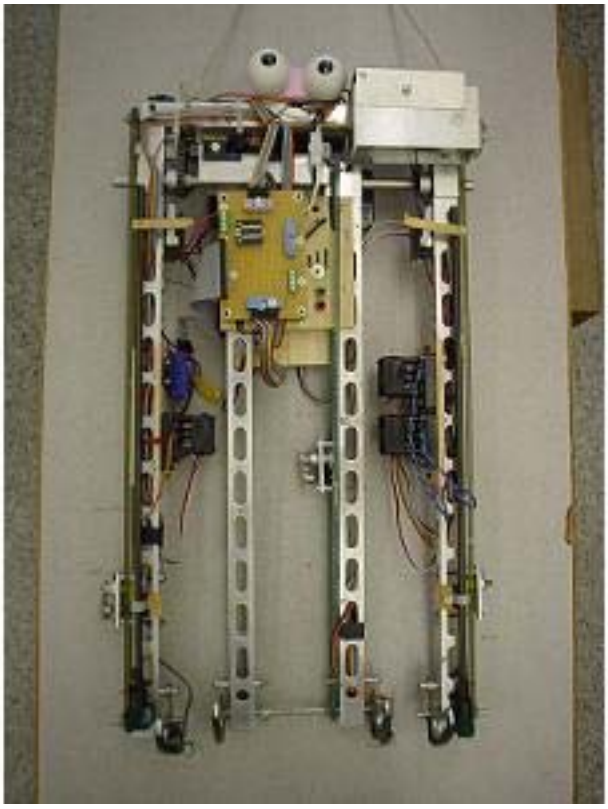
Matt Haberland
1291 Shaker Woods Rd.
Herndon, VA 20170
(703) 904-8566

Abstract

This paper details the implementation of a new autonomous control system for an existing two dimensional, knee-less, bipedal walking robot. After removing one of several unsuccessful control systems, sensors and motor controllers were installed and wired through custom circuits to a central robot controller, which guides the robot through a state machine algorithm implemented in the C programming language. Several failure modes have been encountered but remedied by the adjustment of parameters in the code such that the robot successfully demonstrates a walking cycle. Proposed mechanical, structural, and electrical improvements will eliminate these failures and yield an efficient and robust mechanical walker.

Introduction

There are several approaches toward creating a walking machine. One of the most popular is to model the geometry of the machine after that of the human body and precisely control joint angles in time to mimic the human walking motions. While these machines are versatile and robust, they are extremely inefficient, resulting in short runtimes. The passive dynamic approach, on the other hand, models dynamic properties of the machine after those of the human body but offers no control or power other than that of gravity. These machines are extremely efficient, but not at all versatile or robust, which eliminates practical use. An approach in between these extremes seeks to apply passive dynamic principles for efficiency, but also provide powered control when necessary to enhance versatility and robustness. The robotic subject of the following paper achieves a walking cycle through this approach and thus has the potential to achieve new levels of efficiency and reliability.



Methods

The mechanical structure of the two dimensional, knee-less, bipedal walker was originally created by Gosse [1] in 1998. Energy was to be added to the system by actuation of the ankles alone. A control system was implemented in 2000 by Yevmenenko [2], and again in 2003 by Yeshua [3], but the robot never achieved a walking cycle. As part of the Marathon Walking Robot project in 2005, a motor was added for hip actuation by Seidel and Strasberg [4] and the feet were replaced by Harry [5]. In addition, the entire electronic control system, including all hardware and software, was replaced, hence the need for documentation provided in this paper.

Hardware

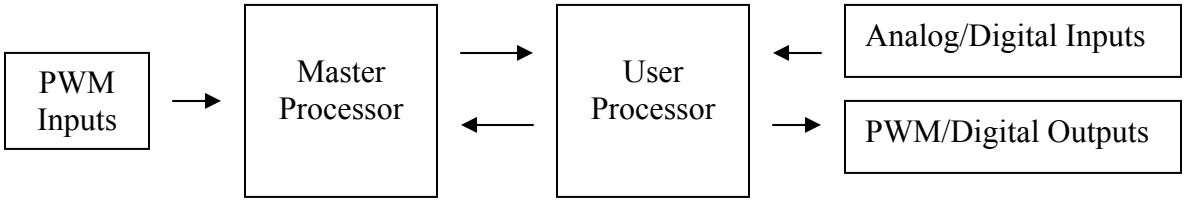
Robot Controller

The Innovation FIRST Mini Robot Controller was chosen to control the robot because it provides a user-friendly interface between sensors and a microprocessor. Eliminating the need to order and prepare dozens of discrete components, the Robot Controller combines a programmable chip, programming port, power terminals, and a wide variety of pins for signal input and output in a single, convenient package. Prewritten functions, instructive documentation, and useful software facilitate generation of custom programs.

Feature:	**Details:**	**Use:**
Speed	10MIPS	Sufficiently fast processor means precise and accurate robot control
Power	Approx. 1W	Low power consumption means greater robot efficiency and longer walking time
Size	3.4” X 4.6” X ¾”	Small size means low weight and more space for other robot parts
Language	C	Powerful, familiar language means flexibility and easy use
Digital Inputs	22 Max	Used to take input from limit switches
Analog Inputs	16 max, 10-bit	Used to take input from potentiometers
PWM Outputs	8	Used to control motors

Internal Processors

The robot controller actually contains two Microchip 18F8520 PICmicro microcontrollers – one programmable User processor that runs the code for the robot, and one Master processor that relays radio control signals to and monitors the operations of the User processor. The Master processor and User processor only communicate once



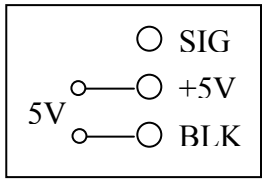
every 17ms, which limits response to radio input. However, all digital/analog in and PWM ports are wired directly to the User processor, providing for immediate communication between the computer and onboard electronics. Other than this, the workings of the Master processor are irrelevant to the user of the system and thus are not disclosed by Innovation FIRST; in practice it is only necessary to understand the operations of the User processor.

Ports

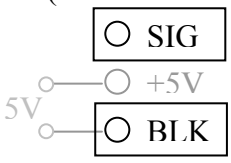
Each of the 16 Digital In/Out – Analog In ports can be configured in the code as either a digital in, digital out, or analog in. Each of the ports labeled "Interrupt" is permanently configured as a digital in, but when triggered can be used to fire interrupt conditions in the code. The PWM outputs can be used to send PWM signals to motor controllers using prewritten functions. The robot controller also features solenoid outputs and PWM inputs, but they are not used on the robot.

Digital In/Out – Analog In

The Digital In/Out – Analog In ports have three pins each – one "SIG" or signal pin, one "+5V" or "high" pin, and one "BLK", black, or "low" pin. A 5V potential difference is always maintained between the "+5V" pin and "BLK" pin; if these pins are shorted the mini robot controller automatically shuts down. Depending on the state of the digital/analog input device, however, the potential difference between the "SIG" pin and the "BLK" pin may vary. The voltage between these pins is measured by the robot controller, converted to either a 1-bit or 10-bit value, and can be referenced in the code as input from the sensors.

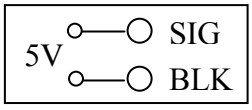


(These Pins Used)

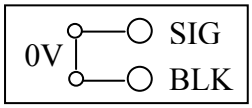


Digital Input

When configured as a digital input, only the "SIG" and "BLK" pins are used. The robot controller measures the voltage between the pins as either high (5V) when the circuit between them is open or low (0V) when the circuit is closed. The high measurement is converted to a logical 1 and the low measurement is converted to a logical 0 for use in robot code. It has been found experimentally that the controller can only detect a change in the digital input value a few hundred times per second.



(High Log. 1)

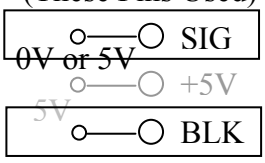


(Low Log. 0)

Digital Output

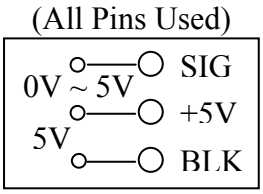
When configured as a digital output, only the "SIG" and "BLK" pins are used. The robot controller outputs either a logical "high" (5V) voltage between the pins or a logical "low" (0V) voltage between the pins as specified in the code. The robot can change the signal at a maximum rate of about 1 kHz.

(These Pins Used)



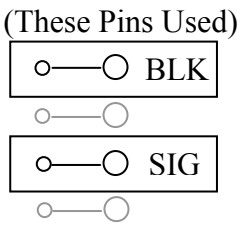
Analog Input

When configured as an analog input, all three pins are used. The "+5V" and "BLK" pins provide a constant reference voltage for the sensor, while the voltage between "SIG" and "BLK" varies depending on the state of the sensor. The robot controller measures the voltage between these pins (0V-5V) and converts this to a 10-bit value (an integer from 0-1023) for use in the code.



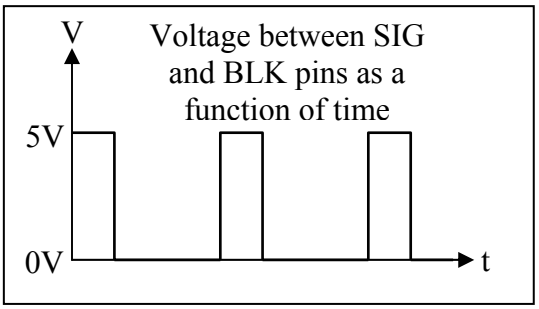
Interrupt Digital Inputs

The "interrupts" are ports that are permanently configured as digital inputs, but can be used to fire an "interrupt" condition in the code when they are triggered. When the logical state of these ports changes, the processor immediately jumps to a specific portion of code, the "interrupt handler". Currently, the ports are being used as regular digital inputs; the interrupt feature is not being used.



PWM Output

A PWM (Pulse Width Modulation) output sends signals to a voltage controller which in turn sends a specified voltage to a motor. A PWM signal is a series of digital pulses which vary in length depending on the desired motor voltage. The voltage controller (which can be a simple H-Bridge or hobby speed controller) amplifies the voltage of these signals and provides current to the device. This pulsing voltage simulates a constant voltage proportional to the percentage of digital "high" time, the "duty cycle". The PWM ports have four pins – one BLK pin, two supply voltage pins, and one SIG pin. Only the SIG pin and BLK pin are used in this robot, in much the same way as a simple digital output. The purpose of using a PWM port instead of a regular digital out is that routines are already programmed into the microprocessor for generating proper PWM signals corresponding to an 8-bit integer specified in the code.

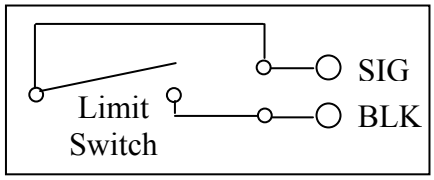


Radio PWM Input

Radio control is not critical for this robot; the only control an operator has is turning. The current radio control system completely bypasses the IFI controller, so the radio PWM inputs are not being used. However, the IFI Radio PWM inputs can receive signals from a hobby radio receiver through standard PWM cables. When the radio receiver is plugged into this port and a signal is received from the radio transmitter, the Master processor converts the signal into an 8-bit value which is relayed to the User processor every 17ms for use in the code.

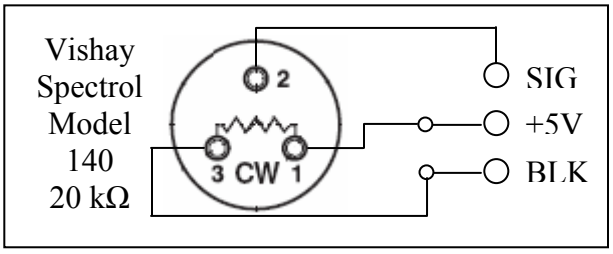
Sensors

Limit Switches



There are two limit switches, one at the heel of each foot. Each limit switch is simply a contact switch that is pressed when its respective heel is supported by the ground. There are two wires leading into the limit switch – one "signal" wire, and one "black" wire. The signal wire plugs into the "SIG" pin of a digital in on the robot controller, the black wire plugs into the "BLK" pin of the digital input. When the limit switch is not pressed ("open"), the voltage between the two pins is 5V and the robot controller reads this as a single bit – "high" or "1". When the limit switch is pressed ("closed"), an electrical connection is made between the two wires inside the switch, creating a short circuit. This forces the voltage between the two pins to jump to zero, and the robot controller reads this as "low" or "0". So when the heel of the robot is in the air, the limit switch is not pressed and there is a 5V potential difference between its lead wires, and thus the robot controller reads a "1" for that digital input. When the heel of the robot is in contact with the ground, the limit switch is pressed, shorting the lead wires, so there is 0V between them and thus the robot controller reads a "0". In this way the limit switches tell the robot how it is supported by the ground.

Potentiometers



There are three potentiometers, one at each ankle and one at the hip. Potentiometers are variable resistors with three pins. The right "reference" pin is connected to the "5V" pin of the robot controller, the left "black" pin is connected to the "BLK" pin, and the center "signal" pin is connected to the "SIG" pin. There is always a fixed 20 kΩ between the "reference" pin and the "black" pin, but turning the knob of the potentiometer changes the voltage between the "signal" pin and the "black" pin. When connected properly, the robot controller measures the voltage between the "signal" pin and the "black" pin and converts the measurement to a 10-bit value (an integer between 0 and 1023). For example, when the foot is in the air and is rotated to a fully retracted position, the voltage between the signal and black pin is .5V, corresponding to a value of 100. In this way the potentiometer tells the robot the position of the ankle.

Encoder

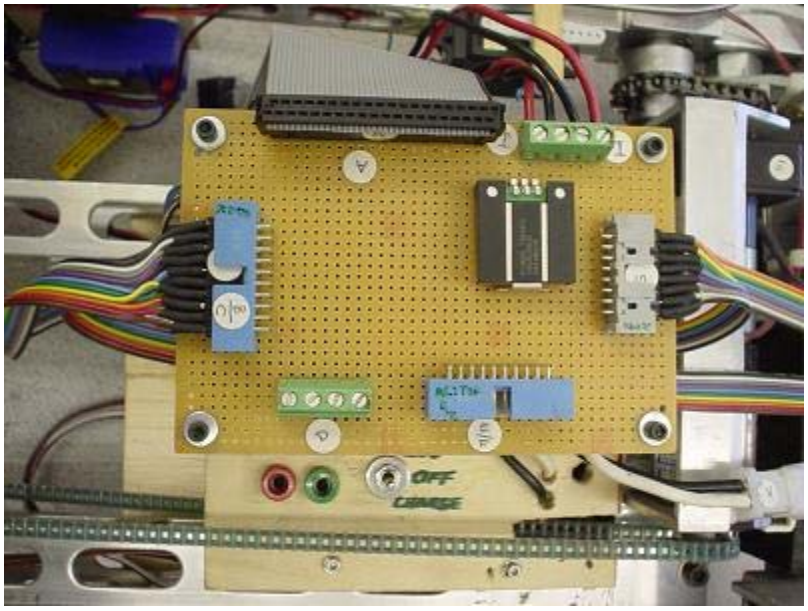
Encoders are mounted on each of the motors for measuring angular changes, but none of them are in use. An encoder uses a beam of light and a slotted wheel coupled to the motor shaft to detect changes in angular displacement. When the beam of light passes through a slot in the wheel, the encoder sends a pulse to the robot controller indicating a change in motor shaft angle. In practice the pulses come too rapidly to be processed by the robot controller. Possible solutions will be addressed later in the paper.

Voltage Controller

The PWM output was originally connected to an H-Bridge Voltage Controller created by former lab consultant Mike Sherback. A PWM output sent square pulses with variable lengths corresponding to an 8-bit value (an integer between 0 and 255) at a fixed frequency of 2kHz. The H-Bridge boosted the voltage of the signal to 12V and provided sufficient current to power the motor. By switching the power on an off in pulses, the H-Bridge simulated a continuously variable, constant voltage between 0V and 12V due to the inductive properties of the motor. The simulated voltage was proportional to the length of a pulse over the period of the signal – the "duty cycle" percentage. A separate two-bit signal (generated using digital outputs) controlled the function of the motor – forward, reverse, coast, or brake. The H-bridge was replaced, however, because of its inability to switch rapidly between forward and reverse. The most recent voltage controllers are Innovation First Victor 884 Speed Controllers. A function written by Innovation First generates a PWM signal at a frequency of 120 Hz with a duty cycle measured to vary between 10% and 20%. The speed controller then converts this signal to a simulated voltage between –12V and 12V.

Interface Board

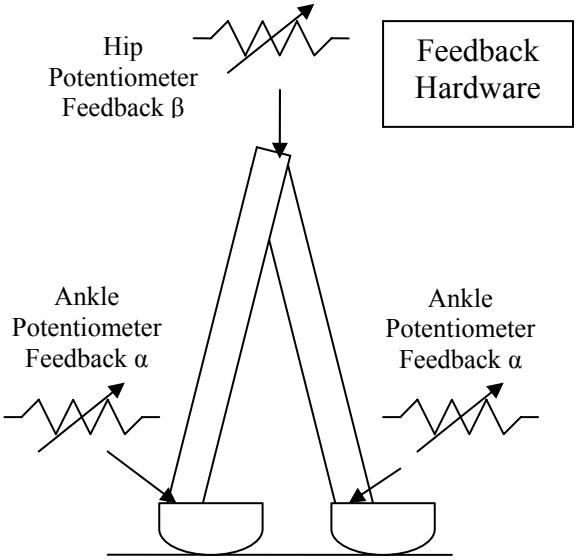
The interface board carries all signals to and from the robot controller via a 40-pin IDE cable and routes them to the proper sensors via smaller ribbon cables. Also included is an IC DC/DC converter that regulates voltage supplied from the battery down to 6V for the robot controller and radio receiver. Although the function of the interface board is simple, the detailed design is rather complicated and its fabrication using a breadboard, wire, headers, and solder was very time-consuming, so complete technical drawings are included as an appendix.



Software

Algorithm

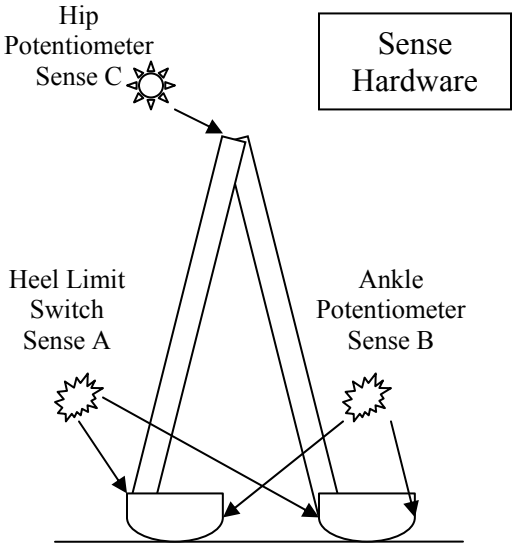
The most difficult part of programming the software was finding a simple but flexible and efficient algorithm. The robot is modeled as a state machine – at any given point in time the robot is in a particular state in which it only performs a certain subset of its possible actions based on the relevant feedbacks, and only switches to the next state when given the cue from one of its senses.



Feedbacks:

Feedback α: Ankle Angle
A potentiometer senses the angle of the foot relative to the leg.

Feedback β: Hip Angle
A potentiometer senses the angle between the legs.



Senses:

Sense A – Heel Strike
The robot senses front heel strike with a contact limit switch on the heel. When the robot's front heel strikes the ground at the end of a step, the limit switch is pressed and the robot registers heel strike.

Sense B – Toe Lift
The robot senses rear toe lift with a potentiometer on the robot's ankle. When the ankle angle (Feedback α) reaches an experimentally optimized value, the robot registers toe lift.

Sense C – Hip Switch
The robot senses hip switch with a potentiometer at the hips. When the hip angle (Feedback β) reaches a value corresponding with the inner and outer legs being in line, the robot registers hip switch.

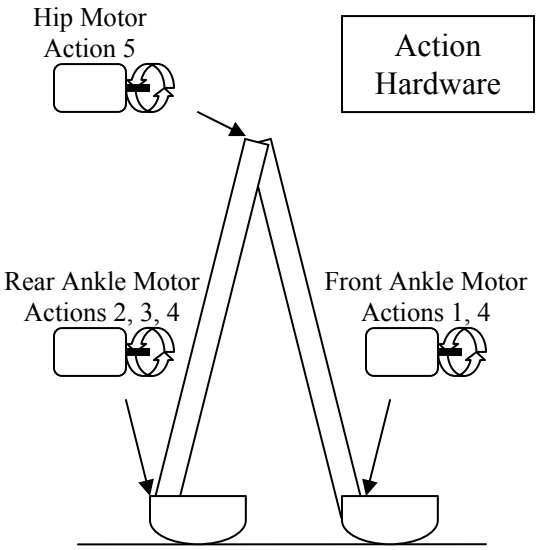
Actions:

Action 1: Ankle Preparation
The robot performs front ankle preparation, rotating foot to the proper position for heel strike (Sense A). Position is controlled with a "P-loop" using information about current ankle angle (Feedback α).

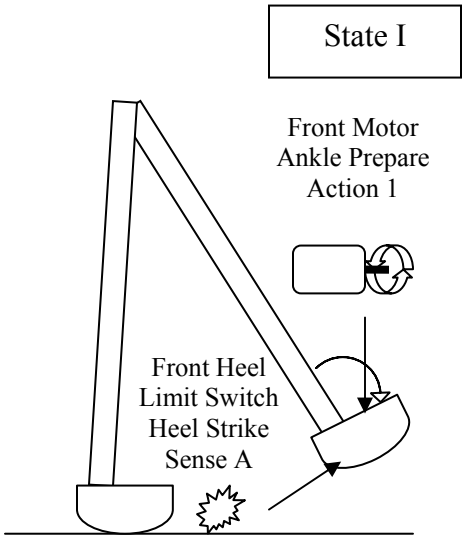
Action 2: Ankle Push
Robot performs rear ankle push. The voltage applied to the ankle motor is constant, as specified in the code.

Action 3: Ankle Retraction
Robot performs rear ankle retraction, rotating foot to prescribed position to provide clearance with the ground at hip switch (Sense C). Position is controlled with a "P-loop" using information about current ankle angle (Feedback α).

Action 4: Ankle Lock
Robot performs ankle lock, holding the foot at a prescribed position. Position is controlled with a "P-loop" using information about current ankle angle (Feedback α).



Action 5: Hip Swing
Robot performs hip swing. The voltage applied to the hip motor is a function of hip angle (Feedback β).



States

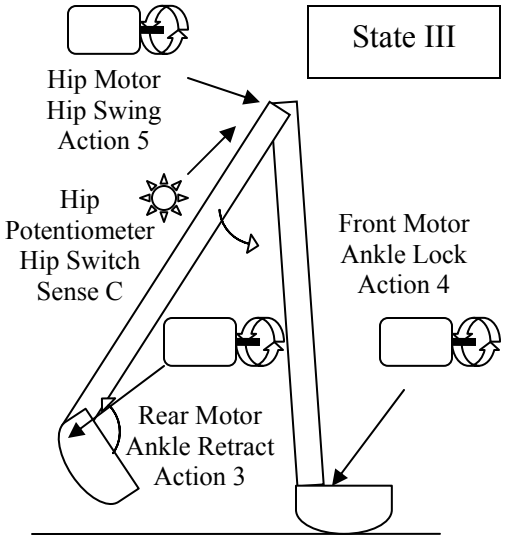
Initial State

State I
The robot has been set up manually by the user with its rear foot on the ground and front heel in the air. The robot prepares front foot for heel strike (Action 1). When the operator releases the robot, the robot passively falls as a double pendulum with a rolling contact pivot. When heel strike (Sense A) occurs, the robot enters State II.

Walking Cycle

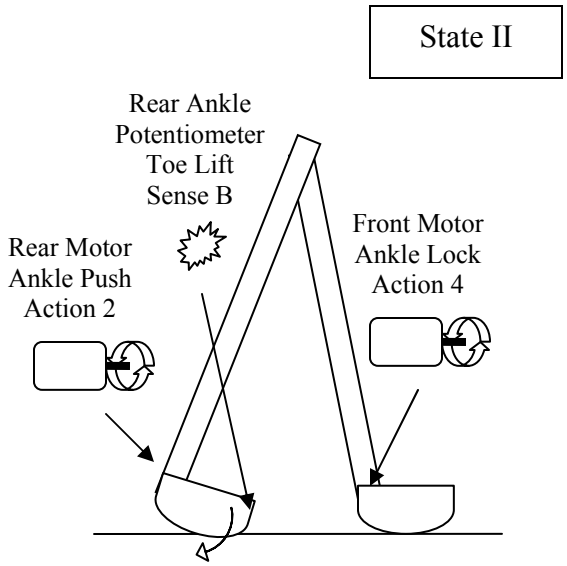
State II

The robot performs rear ankle push (Action 2) and front ankle lock (Action 4). When rear toe lift (Sense B) occurs, the robot enters State III.



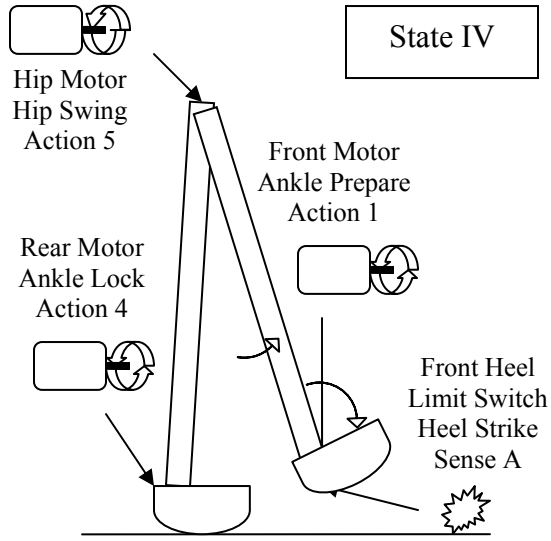
State IV

The robot performs front ankle preparation (Action 1), rear ankle lock (Action 4), and hip swing (Action 5). When front heel strike (Sense A) occurs, robot returns to State II.



State III

The robot performs rear ankle retraction (Action 3), front ankle lock (Action 4), and hip swing (Action 5). When hip switch (Sense C) occurs, robot enters State IV.



Code

The software is written in C with MPLAB IDE, compiled into a single .hex file by the MPLAB C18 Compiler, and downloaded to the mini robot controller via serial cable using IFI Loader. Three files of the default code included with the robot controller were heavily modified to achieve desired performance.

user_routines_fast.c

The algorithm detailed above is implemented using a single variable to keep track of the robot state, one function for each sense to return the logical state of the sense, one function for each robot action to cause the hardware to perform the action, a single function to return continuous feedback from a specified potentiometer, and various processor functions to assist the rest of the functions with their tasks. The Process_Data_from_Local_IO() function contains the state machine itself. A "switch-case" statement tests the "State" variable - for each state, certain action function are called and a conditional statement calls a sense function. If the sense function in the conditional statement returns true, the state variable changes to its next value. When the code reaches the end of the switch statement, the loop repeats. All the functions are documented by comments in the code itself, but one requiring brief explanation here is the "Position2PWM" function. This function implements the P-Loop that controls the position of a specified motor by applying a voltage proportional to the difference between its current position and the desired position plus a specified offset voltage.

user_routines.c

The User_Initialization() routine is used to set up ports on the robot controller for use in the code.

user_routines.h

This is the header file where custom functions are prototyped and user-defined macros are declared. Once the robot began executing the four states successfully, this file became the most important in debugging because it is where the values of all experimentally determined program constants could be changed. For example, this file contains the gain and offset constants used in calls to the "Position2PWM" function mentioned above, which had to be tuned experimentally as described in the following section.

Adjustments

After all hardware was wired and code was loaded on the robot, a tremendous amount of both mechanical and electrical "tweaking" was necessary before a stable walking cycle was achieved. In general, the scientific method was followed – the robot would be set in its initial state and allowed to follow its instructions. When the robot failed, the cause of the problem would be determined and a single parameter would be adjusted based on experience with passive walkers, insight from simulations, and physical intuition in an attempt to remedy the problem. This basic cycle of testing and tweaking was executed

until a walking cycle was realized. Listed below are failure modes and solutions categorized by the state in which the failure occurred. Problems that were particularly difficult to resolve are marked with an asterisk(*).

State I:

Few failures occurred in the initial state as the robot is supported by the operator. However, sometimes the robot failed to continue into State II upon heel strike, which was often due to incorrect code upload or a malfunctioning heel limit switch.

State II:

Toe lift does not occur: One of the first problems encountered was an inability to advance to State III because toe lift was defined to occur when a limit switch mounted on the front of the foot was closed. This was often impossible because the motors did not have enough torque to push the ankle into that position. The solution was to remove the limit switches and re-define toe lift to occur when the ankle had rotated to a specified position a few degrees from the "prepared" angle.

State III:

Foot Scuff: The rear legs would hit the ground instead of swinging through. This failure had one of three causes:

1. Failure of the rear feet to fully retract: This could be fixed by adjusting the "retracted" position parameter in the user_routines.h file of the code.
2. Failure of the stance feet to maintain position: The front ankles would yield under the weight of robot, causing the rear feet to hit the ground. This was fixed by increasing both the gain and the offset in the P-loop controlling the angle of the rear feet.
3. Robot tipped too far forward*: This meant that more energy was added to the system than necessary to maintain the robot walking speed. This problem was solved either by reducing the angle over which ankle push occurred (reducing energy input), or by increasing the speed of hip swing (increasing walking speed).

Fall Backward: The robot would fall backward upon toe lift due to insufficient ankle push. This happened whenever the sprocket turning the potentiometer skipped a link in the chain. This could be fixed either by re-calibrating the potentiometer position in the code or by slipping the chain back into its proper position. Sometimes it was also necessary to increase the "toe lift" parameter so that the robot would push off further before retracting the feet. If neither of these strategies solved the problem, hip swing gain was lowered to prevent the robot from "kicking" itself over backwards.

State IV:

State Skip: Sometimes the robot skipped from State IV back to State II before heel strike occurred. This was usually due to a stuck heel limit switch.

Foot Scuff*: The feet would scuff because they were instructed to prepare for heel strike as soon as State IV began. This was fixed by delaying ankle preparation until the hip had rotated enough for the feet to clear the ground.

Front Legs Fall*: In an ideal walking cycle, heel strike would occur almost immediately after the front legs reach the desired stride angle, about 30 degrees, just as the feet begin retrograde motion. In reality, however, the legs occasionally reach the desired angle and then fall back excessively before heel strike, resulting in a high velocity impact and an unstably short step. This causes the robot to scuff its rear foot and fall on the following step. To reduce the severity of the problem, a rubber mechanical stop was installed at the hip to create a plastic collision at the prescribed angle. In addition, the hip swing was set to a higher gain and offset voltage to lock the hip motor when it reached the desired position. Locking the legs in this manner wasted a tremendous amount of energy, so over time the gain on the hip motor has been relaxed to make the robot more efficient. The current setting is much more efficient than it has been, and it permits the robot to execute many successive steps. Inevitably, however, slight environmental perturbations cause heel strike to be delayed until long after hip swing is complete and this failure ensues. This is the current mode of failure; it has not been resolved but possible solutions will be discussed.

Code Asymmetry:

It became evident that some failures were almost always associated with a certain pair of legs but not with the other. This was due to asymmetry in the mass distributions of the two pairs of legs, which was partially corrected by the relocation of the control system from the top of the robot to the inner legs. However, problems persisted so the code was changed such that separate constants could be specified for the inner and outer pairs of legs. For example, it was noted that the inner feet rarely scuffed the ground, but the outer legs persistently had this problem. The cause was excessive ankle push at the outer feet during State II, so the outer leg toe lift constant was lowered while that of the inner legs was preserved.

Adjusting a parameter to alleviate one problem would often introduce another problem. For example, the need for the end of hip swing to coincide with heel strike in State IV prompted a decrease in the hip swing gain parameter. However, this decrease caused the rear foot to scuff during State III. Therefore the code was adjusted to allow hip swing gain to be increased in State III but decreased in State IV.

Additional Mechanical Adjustments

Many routine adjustments and error repairs were made to the robot in the debugging process, but the volume of material that would be required to completely document each repair is prohibitive. Some examples of this work include complete replacement of the hip potentiometer mounting bracket, replacement of the coupling between the hip potentiometer and hip motor, and replacement of the mis-cut carbon fiber tube on the chain tensioner. The hip motor bracket was straightened and notched to prevent collision with battery terminals, unnecessarily tapped frame holes were drilled out, and ankle drive gear set screw holes were added.

Results

The robot successfully executes a walking cycle, recovers from minor conditional imperfections, and fails when faced with significant adverse effects. It first walked on June 6, 2005, when it took 10 unsupported steps before being stopped by the operator. On average, the robot has a step length of about .3m, a step frequency of 1.5 Hz, and thus walks at approximately .45m/s. In its best performance to date, the robot walked 39 steps in 26.5 seconds and covered a distance of 12m.

To start the robot, the operator must provide it with acceptable initial conditions in State I, usually by setting the outer legs on the floor and bringing the inner legs to the front rubber stop with the feet approximately .1 m above the floor. Alternatively, the operator can guide the robot into a walking cycle by applying forces as necessary – which requires more human control but in the long run is easier to replicate and is more reliable.

The most effective method for diagnosing failures was filming robot operation and watching the footage in slow motion to determine the precise reason for failure. The best tool for determining solutions for errors was the application of natural physical intuition and insight gained by experimenting with the simulation. Another procedure for identifying and fixing problems was to help the robot walk by giving corrective tugs as necessary. Often experimenting with intuitive impulses in this manner led to important information. For example, before the robot could walk on its own, it would walk well if given slight, periodic pulls backwards to prevent it from falling forwards. From this, it was deduced that decreasing ankle push would have the same effect as periodic tugs, and making this correction resulted in free walking.

It is clear that the robot is much less efficient than desired as the motors become hot and the battery is significantly drained after only minutes of use. The robot requires a fully charged battery to function properly; currently it can only be tested for about five minutes at a time before showing symptoms of fatigue, most noticeably sluggishness in the hip swing.

Discussion

The initial goal of this project was accomplished in full – the existing robot has been modified such that it is capable of executing a walking cycle. The next goal of the project, robust walking, has been partially satisfied – the robot has shown the ability to recover from slight perturbations, frequently for 20 steps or more. Inevitably, however, the robot encounters challenges posed by the environment and internal mechanical difficulties that it cannot overcome. The final, long term goal of this project - efficient, long distance walking – is only beginning to be tackled. For example, the current walking cycle of the robot wastes a tremendous amount of energy locking the feet in specified positions and moving the hip to an angle it should ultimately swing to on its own. However, physical improvements must be made before the control strategy can begin to address this issue. The following is a list of proposed modifications that will make the robot more efficient and robust, organized by category in order of priority.

Recommendations

Mechanical

First, the robot must be improved mechanically. The feet must be aligned properly so that they provide symmetric support on the ground, as the robot often wobbles and turns sideways unexpectedly. The leg chain tensioner slots should be lengthened to provide for further tensioning as the chains are rather loose and unexpectedly skip links, causing the potentiometer calibration problems. There have been problems with the limit switches sticking, breaking, and not triggering, so new limit switches should be mounted robustly such that they are triggered whenever any part of the heel is in contact with the ground, and not triggered when the heel is off the ground. A new hip motor chain tensioning system incorporating a rotating idler sprocket rather than a fixed bolt should be created, as the current setup introduces high levels of friction. A motor locking mechanism should be devised to eliminate the inefficiencies associated with stalling the motors, and a clutch could be used to disengage the hip from its motor for more efficient passive motion. The current feet should be replaced with versions that are slightly shorter to compensate for insufficiencies in motor torque mentioned previously. The hip motor mounting bracket should be replaced so that it does not require a spacing shim; this will make future adjustments and repairs easier. Although the new "homemade" hip potentiometer-motor coupling is sufficient, a standard coupling should be purchased and installed as a final solution to the frustrating, long-standing problem of inaccurate hip potentiometer readings. These changes will not only improve the functionality of the robot, but will also reduce the frequency of emergency repairs, which have slowed down the debugging process considerably in the past. Also, many aspects of the current code were written to reflect imperfections in the robot's mechanical functionality, and relaxing this constraint will offer more freedom for success.

Structural

The next task would be to make the robot more symmetric by providing for easy repositioning of control system components to adjust weight distribution. A battery box should be made and an adjustable mounting system should be devised for easy battery removal, installation, and repositioning. The robot controller, interface board, and speed controllers should be remounted on thin aluminum plates for durability, as they will need to be repositioned frequently until desired symmetry is achieved.

Electrical

Hardware

Once the robot is mechanically reliable and structurally symmetric, joint angular velocity data should be generated by an analog differentiator circuit and fed into an analog input. This would enable an upgrade of the position control algorithm to a PD or PID loop, which would eliminate the "bouncing" evident in free ankle positioning and would more effectively lock the stance feet. Later, a method for using the encoders already mounted on the motors should be devised – perhaps by using a quadrature counter chip or by replacing the robot controller with a faster model more suitable for

interpreting the quadrature signal, as encoders would provide much more precise positioning information and angular velocity data. The speed controllers should be replaced with more efficient and precisely controlled H-bridges, and a torque control system should be devised to allow for a more efficient control strategy. Data from the accelerometer already installed on the robot should be used in the code to determine the overall angle of the robot with respect to the vertical. Information from a gyro could be used to help distinguish between the different components of the overall acceleration vector. Once more advanced control over actuators and better feedback data are available, the code can be changed to reflect the fact that robot performance can more closely match simulation, which will improve the robot's performance considerably.

Software

The current asymmetric state machine architecture is necessary for the robot in its present state, but it should be replaced by a more elegant structure equivalent to that in simulation code. This would allow for most of the performance optimization to occur in simulation rather than by physical testing and tweaking. If developed, such an agreement between simulation and the actual machine would be a true milestone in robotic walking not only because it would become relatively easy to optimize this system for efficiency and long distance walking, but also because it would aid in the generation of new and improved physical robot designs.

Conclusion

The Marathon Walking Robot project is in the process of succeeding in its goals. The robot has already achieved a walking cycle and has the ability to recover from mild disturbances. Clearly the robot must become even more stable and much more efficient for it to cover distances on the order of kilometers, but efforts to the present have provided a firm foundation and great insight on what must be done to reach this goal. It is within realistic hope that further time and effort invested in this machine will yield significant results.

Acknowledgements

My experience has given me a unique view of the "university research" process. The professor organizes a graduate student project. Graduate students lead a team of undergraduates. As an undergraduate, I have high school students and experiences to thank for helping me get started. I would like to thanks Lucas Waye of FIRST Team #639 at Ithaca High School for helping me get comfortable with robot controller programming, and Kyle Witte of FIRST Team #116 at Herndon High School for providing recommendations about electronics. I am thankful for my own experience with the FIRST Robotics program and to Dave Lavery for inspiring my initial interest in robotics. Thanks to Kevin Watson for the code repository. Thanks to Walt Haberland for proofreading the report. Thank you to Gregg Stiesberg for assistance with virtually every aspect of the project. Finally, thanks to Professor Andy Ruina for providing many of the pieces for the robot control algorithm and for support and guidance of this project.

Citations

[1] Gosse, L. 1998. *Minimally Power Walker Based on Passive Models*. Master of Engineering Project Report.
[2] Yevmenenko, Y. 2000. *Powered Straight Legged Walker with Circular Feet*. Master of Engineering Thesis.
[3] Yeshua, O. 2003. *Power & Control of a 2D Passive-Dynamic Walking Robot*. MAE 490 Final Paper.
[4] Seidel, W. and Strasberg, M. 2005. *Mechanical Hip Actuation of a 2-D Passive Dynamics Based Walker*. T&AM 492 Final Paper.
[5] Harry, Z. 2005. *2-Dimensional Bipedal Passive-Dynamics Based Walker*. M&AE 491 Senior Design Project Paper.
[6] Innovation FIRST. *2004 EDU Robot Controller Reference Guide*. Available at: http://www.ifirobotics.com/docs/legacy/edu-rc-2004_ref_guide_2004-jan-14a.pdf
[7] Innovation FIRST. *2004 Programming Reference Guide*. Available at: http://www.ifirobotics.com/docs/legacy/2004-programming-reference-guide-12-apr-2004.pdf
[8] Innovation FIRST. *EDU Default Software Reference Guide*. Available at: http://www.ifirobotics.com/docs/legacy/edu-default-software-guide_10-15-2003.pdf
[9] Innovation FIRST. *12V Victor 884 Users Manual*. Available at: http://www.ifirobotics.com/docs/ifi-v884-users-manual-1-26-05.pdf
[10] Vishay Spectrol. *Vishay Spectrol Model 140*. Available at: http://www.vishay.com/docs/57039/140142.pdf
[11] Watson, Kevin. *Kevin's FRC Code Repository*. Available at: http://kevin.org/frc/

Appendices

user_routines.c

```
/*******************************************************************************
* FILE NAME: user_routines.c <EDU VERSION>
*
* DESCRIPTION:
*  This file contains the default mappings of inputs
*  (like switches, joysticks, and buttons) to outputs on the EDU RC.
*
* USAGE:
*  You can either modify this file to fit your needs, or remove it from your
*  project and replace it with a modified copy.
*
*******************************************************************************/

#include "ifi_aliases.h"   // Macros/Aliases used to refer to ports
#include "ifi_default.h"   // Prototypes for default functions
#include "ifi_utilities.h" // Prototypes for utility functions
#include "user_routines.h" // Prototypes for user defined

/*** DEFINE USER VARIABLES AND INITIALIZE THEM HERE ***/
// No variables needed

/*******************************************************************************
* FUNCTION NAME: Setup_Who_Controls_Pwms
* PURPOSE:       Each parameter specifies what processor will control the pwm.
*
* CALLED FROM:   User_Initialization
*     Argument             Type    IO   Description
*     --------             ----    --   -----------
*     pwmSpec1              int     I   USER/MASTER (defined in ifi_aliases.h)
*     pwmSpec2              int     I   USER/MASTER
*     pwmSpec3              int     I   USER/MASTER
*     pwmSpec4              int     I   USER/MASTER
*     pwmSpec5              int     I   USER/MASTER
*     pwmSpec6              int     I   USER/MASTER
*     pwmSpec7              int     I   USER/MASTER
*     pwmSpec8              int     I   USER/MASTER
* RETURNS:       void
*******************************************************************************/
static void Setup_Who_Controls_Pwms(int pwmSpec1,int pwmSpec2,int pwmSpec3,int pwmSpec4,
                                    int pwmSpec5,int pwmSpec6,int pwmSpec7,int pwmSpec8)
{
  txdata.pwm_mask = 0xFF;         /* Default to master controlling all PWMs. */
  if (pwmSpec1 == USER)           /* If User controls PWM1 then clear bit0. */
    txdata.pwm_mask &= 0xFE;      /* same as txdata.pwm_mask = txdata.pwm_mask & 0xFE; */
  if (pwmSpec2 == USER)           /* If User controls PWM2 then clear bit1. */
    txdata.pwm_mask &= 0xFD;
  if (pwmSpec3 == USER)           /* If User controls PWM3 then clear bit2. */
    txdata.pwm_mask &= 0xFB;
  if (pwmSpec4 == USER)           /* If User controls PWM4 then clear bit3. */
    txdata.pwm_mask &= 0xF7;
  if (pwmSpec5 == USER)           /* If User controls PWM5 then clear bit4. */
    txdata.pwm_mask &= 0xEF;
  if (pwmSpec6 == USER)           /* If User controls PWM6 then clear bit5. */
    txdata.pwm_mask &= 0xDF;
  if (pwmSpec7 == USER)           /* If User controls PWM7 then clear bit6. */
    txdata.pwm_mask &= 0xBF;
  if (pwmSpec8 == USER)           /* If User controls PWM8 then clear bit7. */
    txdata.pwm_mask &= 0x7F;
}


/*******************************************************************************
* FUNCTION NAME: User_Initialization
* PURPOSE:       This routine is called first (and only once) in the Main function.
*                You may modify and add to this function.
```

```
*               The primary purpose is to set up the DIGITAL IN/OUT - ANALOG IN
*               pins as analog inputs, digital inputs, and digital outputs.
* CALLED FROM:   main.c
* ARGUMENTS:     none
* RETURNS:       void
*******************************************************************************/
void User_Initialization (void)
{
  rom const char *strptr = "IFI User Processor Initialized ...";

/* FIRST: Set up the pins you want to use as analog INPUTs. */
  IO1 = IO2 = IO3 = IO4 = IO5 = IO6 = IO7 = IO8 = INPUT;
/*
Port 1:  Inner Potentiometer
Port 2:  Outer Potentiometer
Port 3:  Hip Potentiometer
Ports 4-8: Unused/To Be Used with Current Sensor/Accelerometer
*/

/* SECOND: Configure the number of analog channels. */
  Set_Number_of_Analog_Channels(EIGHT_ANALOG);     /* See ifi_aliases.h */
// There are eight analog channels corresponding to ports 1-8

/* THIRD: Set up any extra digital inputs. */
  IO9 = IO10 = INPUT;
// Ports 9 and 10: Unused

/* FOURTH: Set up the pins you want to use as digital OUTPUTs. */
  IO11 = IO12 = IO13 = IO14 = IO15 = IO16 = OUTPUT;
// Ports 11-16: Unused/For Use with Mike Sherback's H-Bridge

/* FIFTH: Initialize the values on the digital outputs. */
  rc_dig_out11 = rc_dig_out12 = rc_dig_out13 = 0;
  rc_dig_out14 = rc_dig_out15 = rc_dig_out16 = 0;
// Initialize Outputs to LOW - BRAKE setting on H-Bridge

/* SIXTH: Set your initial PWM values. */
  pwm01 = pwm02 = pwm03 = pwm04 = pwm05 = pwm06 = pwm07 = pwm08 = 131;
// Set all the PWMs to neutral

/* SEVENTH: Choose which processor will control which PWM outputs. */
  Setup_Who_Controls_Pwms(USER, USER, USER, MASTER, MASTER, MASTER, MASTER, MASTER);
// All relevant PWMs controlled by user so that they can be generated every program loop

/* EIGHTH: Set your PWM output type.  Only applies if USER controls PWM 1, 2, 3, or 4. */
  Setup_PWM_Output_Type(IFI_PWM,IFI_PWM,IFI_PWM,IFI_PWM);
// Use IFI PWM signal for Victor 884

/* Add any other user initialization code here. */

  Initialize_Serial_Comms();

  Putdata(&txdata);             /* DO NOT CHANGE! */

  User_Proc_Is_Ready();         /* DO NOT CHANGE! - last line of User_Initialization */
}


/*******************************************************************************
* FUNCTION NAME: Process_Data_From_Master_uP
* PURPOSE:       Executes every 17ms when it gets new data from the master
*                microprocessor.
* CALLED FROM:   main.c
* ARGUMENTS:     none
* RETURNS:       void
*******************************************************************************/
void Process_Data_From_Master_uP(void)
{
  Getdata(&rxdata);   /* Get fresh data from the master microprocessor. */
   Putdata(&txdata);             /* DO NOT CHANGE! */
}
```

user_routines_fast.c

```
/*******************************************************************************
* FILE NAME: user_routines_fast.c <EDU VERSION>
*
* DESCRIPTION:
*  This file is where the user can add their custom code within the framework
*  of the routines below.
*
* USAGE:
*  You can either modify this file to fit your needs, or remove it from your
*  project and replace it with a modified copy.
*
* OPTIONS:  Interrupts are disabled and not used by default.
*
*******************************************************************************/

#include "ifi_aliases.h"   // Macros/Aliases used to refer to ports
#include "ifi_default.h"   // Prototypes for default functions
#include "ifi_utilities.h" // Prototypes for utility functions
#include "user_routines.h" // Prototypes for user defined

/*** DEFINE USER VARIABLES AND INITIALIZE THEM HERE ***/
/* Variable "State" remembers state of the robot.  "State" is initialized to
   I (corresponding with the integer 0), and cycles II-IV (corresponding with integers
   1-3. */
unsigned char State = StateI;
// Variable "infront" remembers which pair of legs is in front
unsigned char infront = OUT;
// Variable "infrontlast" remembers which pair of legs was last in front
unsigned char infrontlast = OUT;
// Variable "inback" remembers which pair of legs is in back
unsigned char inback = IN;

/*******************************************************************************
* FUNCTION NAME: InterruptVectorLow
* PURPOSE:       Low priority interrupt vector
* CALLED FROM:   nowhere by default
* ARGUMENTS:     none
* RETURNS:       void
* DO NOT MODIFY OR DELETE THIS FUNCTION
*******************************************************************************/
#pragma code InterruptVectorLow = LOW_INT_VECTOR
void InterruptVectorLow (void)
{
  _asm
    goto InterruptHandlerLow  /*jump to interrupt routine*/
  _endasm
}

/*******************************************************************************
* FUNCTION NAME: InterruptHandlerLow
* PURPOSE:       Low priority interrupt handler
* If you want to use these external low priority interrupts or any of the
* peripheral interrupts then you must enable them in your initialization
* routine.  Innovation First, Inc. will not provide support for using these
* interrupts, so be careful.  There is great potential for glitchy code if good
* interrupt programming practices are not followed.  Especially read p. 28 of
* the "MPLAB(R) C18 C Compiler User's Guide" for information on context saving.
* CALLED FROM:   this file, InterruptVectorLow routine
* ARGUMENTS:     none
* RETURNS:       void
*******************************************************************************/
#pragma code

/* You may want to save additional symbols. */
#pragma interruptlow InterruptHandlerLow save=PROD

void InterruptHandlerLow ()
{
```

```
  unsigned char int_byte;
  if (INTCON3bits.INT2IF)         /* The INT2 pin is RB2/INTERRUPTS 1. */
  {
    INTCON3bits.INT2IF = 0;
  }
  else if (INTCON3bits.INT3IF)    /* The INT3 pin is RB3/INTERRUPTS 2. */
  {
    INTCON3bits.INT3IF = 0;
  }
  else if (INTCONbits.RBIF)       /* INTERRUPTS 3-4 (RB4, RB5, RB6, or RB7) changed. */
  {
    int_byte = PORTB;             /* You must read or write to PORTB   */
    INTCONbits.RBIF = 0;          /* and clear the interrupt flag      */
  }                               /* to clear the interrupt condition. */
}

/*****************************************************************************
* Processing Functions
*****************************************************************************/

/*****************************************************************************
* FUNCTION NAME: Potentiometer
* PURPOSE:       Read specified potentiometer (Continuous Feedbacks Alpha and Beta)
* CALLED FROM:   Sense Functions
* ARGUMENTS:     motor (IN, OUT, or HIP)
* RETURNS:       potentiometer (reading of specified potentiometer)
*****************************************************************************/

int Potentiometer(char motor)
{
  unsigned int potentiometer;
  switch(motor)
  {
    case IN:
    {
      potentiometer = Get_Analog_Value(PotIn);
      break;
    }
    case OUT:
    {
      potentiometer =  Get_Analog_Value(PotOut);
      break;
    }
    case HIP:
    {
      potentiometer = Get_Analog_Value(PotHip);
      break;
    }
  }
  return potentiometer;
}

/*****************************************************************************
* FUNCTION NAME: SendPWM
* PURPOSE:       Sets pwm of specified motor to specified value
* CALLED FROM:   Action Functions
* ARGUMENTS:     motor (IN, OUT, or HIP); PWM (signed 8-bit)
* RETURNS:       void
*****************************************************************************/
void SendPWM(char motor, int pwm)
{
  pwm = 131 + pwm; // Convert signed 8-bit number to unsigned 8-bit number
  if (pwm >255)        // If value is greater than allowed maximum
  {
   pwm = 255;      // Set to allowed maximum
  }
  else if (pwm<0)  // If value is less than allowed minimum
  {
   pwm = 0;        // Set to allowed minimum
  }
  switch(motor)
```

```
  {
    case IN:
    {
      MotorInPWM = pwm;
      break;
    }
    case OUT:
    {
      MotorOutPWM = pwm;
      break;
    }
    case HIP:
    {
      MotorHipPWM = pwm;
      break;
    }
  }
}

/*******************************************************************************
* FUNCTION NAME: InFrontInBack
* PURPOSE:       Determine which pair of legs is in front and which is in back
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       void
*******************************************************************************/
void InFrontInBack(void)
{
  if (Potentiometer(HIP) > HIPSWITCH)
  {
    infront = IN;
    inback = OUT;
  }
  if (Potentiometer(HIP) <= HIPSWITCH)
  {
    infront = OUT;
    inback = IN;
  }
}

/*******************************************************************************
* FUNCTION NAME: Position2PWM
* PURPOSE:       For position control, convert current position to recommended
*                pwm signal
* CALLED FROM:   Sense Functions
* ARGUMENTS:     Ptarget, motor, tolerance, scale, offset
* RETURNS:       pwm
*******************************************************************************/
int Position2PWM(int Ptarget, char motor, char tolerance, char scale, char offset)
{
  int position = Potentiometer(motor);
  int pwm;

  if (position > (Ptarget - tolerance)       // if motor is within Tolerance
    && position < (Ptarget + tolerance))
  {
    pwm = NEUTRAL;                           // turn motor off
  }
// Otherwise, Scale voltage proportional to position error, bit shift, and add offset
  else if (Ptarget-position < 0)
  {
    pwm = ((int)scale*(Ptarget-position))/80 - (int)offset;
  }
  else
  {
    pwm = ((int)scale*(Ptarget-position))/80 + (int)offset;
  }
  return pwm;
}
```

```
/*******************************************************************************
* Sense Functions
*******************************************************************************/

/*******************************************************************************
* FUNCTION NAME: HeelStrike
* PURPOSE:       Determine if heel strike (Sense A) occurs
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       IN, OUT, or NONE
*******************************************************************************/
char HeelStrike (void)
{
  char heelstrike = NONE;
  if (infront == IN & LimitHeelIn == CLOSED)
  {
    heelstrike = IN;
  }
  else if (infront == OUT & LimitHeelOut == CLOSED)
  {
    heelstrike = OUT;
  }
  return heelstrike;
}

/*******************************************************************************
* FUNCTION NAME: ToeLift
* PURPOSE:       Determine if toe lift (Sense B) occurs
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       IN, OUT, or NONE
*******************************************************************************/
char ToeLift (void)
{
  char toeoff = NONE;
  int ankle;
  ankle = Potentiometer(inback);
  if (inback == IN & ankle >= ToeOffIn)
  {
    toeoff = IN;
  }
  else if (inback == OUT & ankle >= ToeOffOut)
  {
    toeoff = OUT;
  }
  return toeoff;
}

/*******************************************************************************
* FUNCTION NAME: HipSwitch
* PURPOSE:       Determine if hip switch (Sense C) occurs
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       TRUE or FALSE
*******************************************************************************/
char HipSwitch (void)
{
  char hipswitch = FALSE;
  int ana = Potentiometer(HIP);
  if (infrontlast != infront)
  {
    hipswitch = TRUE;
    infrontlast = infront;
  }
  else
  {
    hipswitch = FALSE;
    infrontlast = infront;
  }
  return hipswitch;
}
```

```
/*******************************************************************************
* Action Functions
*******************************************************************************/

/*******************************************************************************
* FUNCTION NAME: AnklePrepare
* PURPOSE:       Prepares Ankle (Action 1)
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       none
*******************************************************************************/
void AnklePrepare(void)
{
  if (infront == IN)
  {
    if (Potentiometer(HIP) > PREPAREIN)
    {
      SendPWM(infront, Position2PWM(PreparedIn, infront,
              PrepareTolerance, PrepareScale, PrepareOffset));
    }
    else
    {
      SendPWM(infront, NEUTRAL);
    }
  }
  if (infront == OUT)
  {
    if (Potentiometer(HIP) < PREPAREOUT)
    {
      SendPWM(infront, Position2PWM(PreparedOut, infront,
              PrepareTolerance, PrepareScale, PrepareOffset));
    }
    else
    {
      SendPWM(infront, NEUTRAL);
    }
  }
}

/*******************************************************************************
* FUNCTION NAME: AnklePush
* PURPOSE:       Pushes Ankle (Action 2)
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       none
*******************************************************************************/
void AnklePush (void)
{
  SendPWM(inback,FULL);
}

/*******************************************************************************
* FUNCTION NAME: AnkleRetract
* PURPOSE:       Retracts Ankle (Action 3)
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       none
*******************************************************************************/
void AnkleRetract(void)
{
  SendPWM(inback, Position2PWM(Retracted, inback,
          RetractTolerance, RetractScale, RetractOffset));
}
```

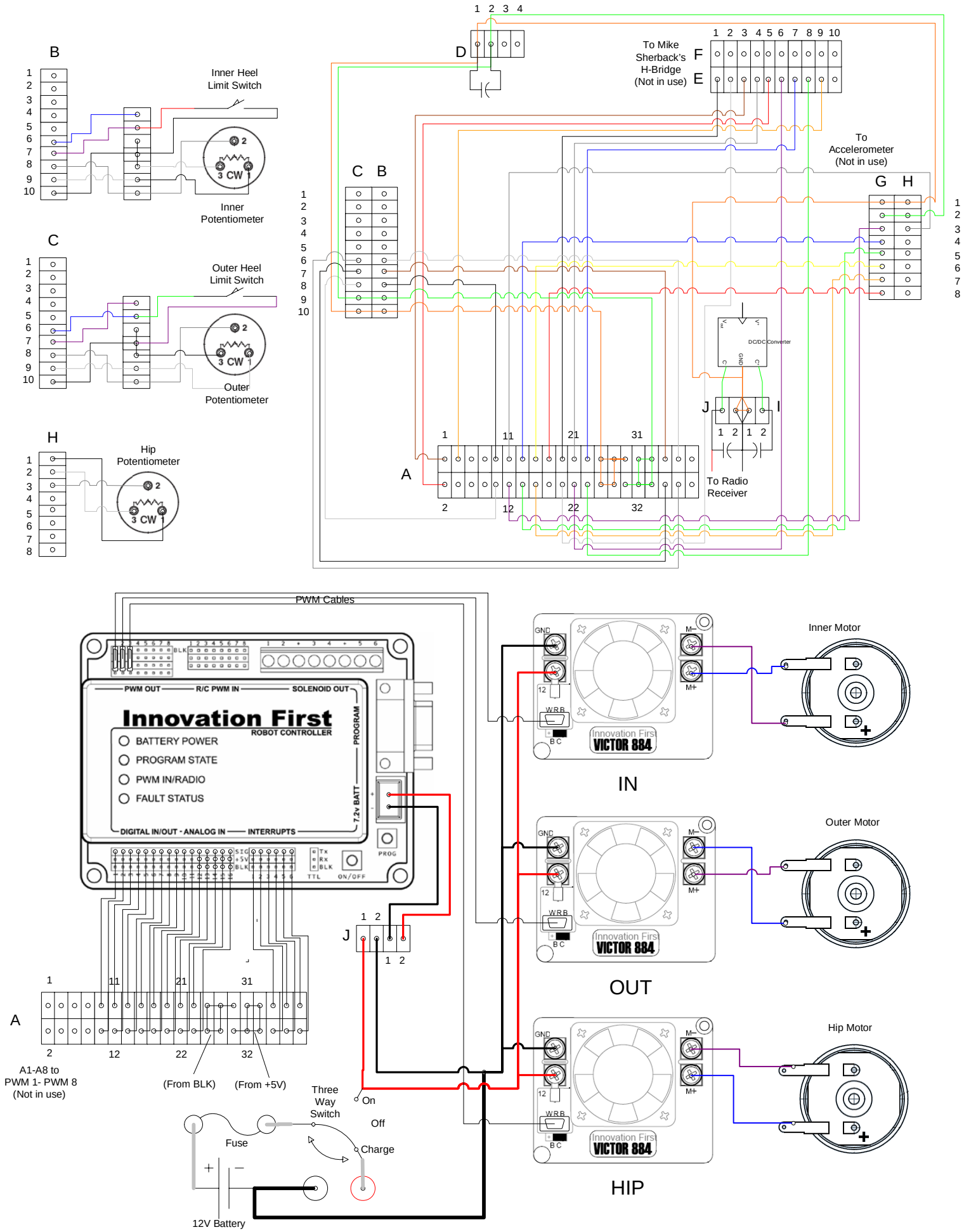
```
/*****************************************************************************
* FUNCTION NAME: AnkleLock
* PURPOSE:       Locks support ankle (Action 4)
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       none
*****************************************************************************/
void AnkleLock (void)
{
  if (State == StateII | State == StateIII)
  {
    SendPWM(infront, Position2PWM(PreparedIn, infront,
            AnkleLockTolerance, AnkleLockScale, AnkleLockOffset));
  }
  if (State == StateIV)
  {
    SendPWM(inback, Position2PWM(PreparedIn, inback,
            AnkleLockTolerance, AnkleLockScale, AnkleLockOffset));
  }
}

/*****************************************************************************
* FUNCTION NAME: HipSwing
* PURPOSE:       Swings Hip (Action 5)
* CALLED FROM:   State Machine Function
* ARGUMENTS:     none
* RETURNS:       none
*****************************************************************************/
void HipSwing (void)
{
  if (State == StateIII & Potentiometer(inback) < 300)
  {
    if (inback == IN)
    {
      SendPWM(HIP, Position2PWM(SwingIn, HIP, HipSwingTolerance,
              HipSwingScaleInIII, HipSwingOffsetInIII));
    }
    else
    {
      SendPWM(HIP, Position2PWM(SwingOut, HIP, HipSwingTolerance,
              HipSwingScaleOutIII, HipSwingOffsetOutIII));
    }
  }
  if (State == StateIV)
  {
    if (infront == IN)
    {
      SendPWM(HIP, Position2PWM(SwingIn, HIP, HipSwingTolerance,
              HipSwingScaleInIV, HipSwingOffsetInIV));

    }
    else
    {
      SendPWM(HIP, Position2PWM(SwingOut, HIP, HipSwingTolerance,
              HipSwingScaleOutIV, HipSwingOffsetOutIV));
    }
  }
}
```

```
/*****************************************************************************
* State Machine Function
*****************************************************************************/

/*****************************************************************************
* FUNCTION NAME: Process_Data_From_Local_IO
* PURPOSE:       Execute user's realtime code.
* You should modify this routine by adding code which you wish to run fast.
* It will be executed every program loop, and not wait for fresh data
* from the master processor.
* CALLED FROM:   main.c
* ARGUMENTS:     none
* RETURNS:       void
*****************************************************************************/
void Process_Data_From_Local_IO(void)
{
  // Begin Initial State

  InFrontInBack();                  // Determine which pair of legs is in front

  switch (State)
  {
    case StateI:                    // If in State I
    {
      AnklePrepare();               // Prepare Front Ankle (Action 1)
      SendPWM(inback, NEUTRAL);     // Send no power to rear ankle
      SendPWM(HIP, NEUTRAL);        // Send no power to hip
      if (HeelStrike() == infront)  // If Heel Strike (Sense A)
      {
        State = StateII;            // Assume State II
      }
      break;
    }
    // Begin Walking Cycle
    case StateII:                   // If in State II
    {
      SendPWM(HIP, NEUTRAL);        // Send no power to hip
      AnklePush();                  // Push Rear Ankle (Action 2)
      AnkleLock();                  // Lock Front Ankle (Action 4)
      if (ToeLift())                // If Toe Lift (Sense B)
      {
        State = StateIII;           // Assume State III
      }
      break;
    }
    case StateIII:                  // If in State III
    {
      AnkleRetract();               // Retract Rear Ankle (Action 3)
      AnkleLock();                  // Lock Front Ankle (Action 4)
      HipSwing();                   // Swing Hip (Action 5)
      if (HipSwitch())              // If Hip Switch (Sense C)
      {
        State = StateIV;            // Assume State IV
      }
      break;
    }
    case StateIV:                   // If in State IV
    {
      AnkleLock();                  // Lock Rear Ankle (Action 4)
      HipSwing();                   // Swing Hip (Action 5)
      AnklePrepare();               // Prepare Front Ankle (Action 1)
      if (HeelStrike())             // If Heel Strike (Sense A)
      {
        State = StateII;            // Assume State II
      }
      break;
    }
  }
  // Send PWM signals set by action functions
  Generate_Pwms(MotorInPWM, MotorOutPWM, MotorHipPWM, 131,131,131,131,131);
}
```

user_routines.h

```
/*******************************************************************************
* FILE NAME: user_routines.h
*
* DESCRIPTION:
*  This is the include file which corresponds to user_routines.c and
*  user_routines_fast.c
*  It contains some aliases and function prototypes used in those files.
*
* USAGE:
*  If you add your own routines to those files, this is a good place to add
*  your custom macros (aliases), type definitions, and function prototypes.
*******************************************************************************/

#ifndef __user_program_h_
#define __user_program_h_

/*******************************************************************************
                            MACRO DECLARATIONS
*******************************************************************************/
/* Add your macros (aliases and constants) here.                               */
/* Do not edit the ones in ifi_aliases.h                                       */
/* Macros are substituted in at compile time and make your code more readable */
/* as well as making it easy to change a constant value in one place, rather  */
/* than at every place it is used in your code.                                */
/*
 EXAMPLE CONSTANT:
#define PI_VAL          3.1415

 EXAMPLE ALIAS:
#define LIMIT_SWITCH_1  rc_dig_int1  (Points to another macro in ifi_aliases.h)
*/

/* Used in limit switch routines in user_routines.c */
#define OPEN        1     /* Limit switch is open (input is floating high). */
#define CLOSED      0     /* Limit switch is closed (input connected to ground). */

// PWM Constants
#define FULL                255           // Full PWM Signal
#define NEUTRAL             0             // Neutral PWM Signal

/* Motor Constants - Assign constants for referencing motors and corresponding
   potentiometers in code */
#define NONE                0             // No Motor
#define IN                  1             // Inner Motor
#define OUT                 2             // Outer Motor
#define HIP                 3             // Hip motor

// Logical Constants
#define TRUE                1
#define FALSE               0

// States
#define StateI              0             // Numeral N corresponds to int. n = |N|-1
#define StateII             1
#define StateIII            2
#define StateIV             3

// Positions
#define Retracted           40            // Ankle Pot. reading, retracted position
#define PreparedIn          370           // Inner Pot. reading, prepared position
#define PreparedOut         340           // Outer Pot. reading, prepared position
#define ToeOffIn            PreparedIn+70  // Inner Pot. reading, Toe Off position
#define ToeOffOut           PreparedOut+15 // Outer Pot. reading, Prepared position
#define HIPSWITCH           315           // Hip Pot. reading, Hip Switch
#define SwingIn             HIPSWITCH+105  // Hip Pot. reading, inner step
#define SwingOut            HIPSWITCH-105  // Hip Pot. reading, outer step
#define PREPAREIN           SwingIn-30     // Hip Pot. reading, inner Ankle Prepare
#define PREPAREOUT          SwingOut+30    // Hip Pot. reading, outer Ankle Prepare
```

```
// Tolerances
#define PrepareTolerance      3
#define RetractTolerance      3
#define AnkleLockTolerance    3
#define HipSwingTolerance     5

//  Scaling Factors
#define HipSwingScaleInIII    30            // Inner leg swing, State II
#define HipSwingScaleInIV     20            // Inner leg swing, State III
#define HipSwingScaleOutIII   40            // Outer leg swing, State II
#define HipSwingScaleOutIV    20            // Outer leg swing, State III
#define AnkleLockScale        9
#define PrepareScale          8
#define RetractScale          8

// Offsets
#define HipSwingOffsetInIII   30            // Inner leg swing, State II
#define HipSwingOffsetInIV    20            // Inner leg swing, State III
#define HipSwingOffsetOutIII  40            // Outer leg swing, State II
#define HipSwingOffsetOutIV   20            // Outer leg swing, State III
#define AnkleLockOffset       9
#define PrepareOffset         7
#define RetractOffset         7

// Input Referencing Macros
#define PotIn                 rc_ana_in01   // Inner Potentiometer
#define PotOut                rc_ana_in02   // Outer Potentiometer
#define PotHip                rc_ana_in03   // Hip Potentiometer

#define LimitHeelIn           rc_dig_int1   // Inner Heel Limit Switch
#define LimitHeelOut          rc_dig_int4   // Outer Heel Limit Switch

#define MotorInPWM            pwm01         // Inner Motor
#define MotorOutPWM           pwm02         // Outer Motor
#define MotorHipPWM           pwm03         // Hip Motor
 /*****************************************************************************
                           FUNCTION PROTOTYPES
*****************************************************************************/

/* These routines reside in user_routines.c */
void User_Initialization(void);
void Process_Data_From_Master_uP(void);

/* These routines reside in user_routines_fast.c */
void InterruptHandlerLow (void);  /* DO NOT CHANGE! */

// Processor Functions
int Potentiometer (char motor);   // Continuous Feedback Alpha and Beta
void SendPWM(char motor, int pwm);
void InFrontInBack(void);
int Position2PWM(int Ptarget, char motor, char tolerance, char scale, char offset);

// Sense Functions
char HeelStrike(void);                  // Sense A
char ToeLift(void);                     // Sense B
char HipSwitch(void);                   // Sense C

// Action Functions
void AnklePrepare(void);                // Action 1
void AnklePush(void);                   // Action 2
void AnkleRetract(void);                // Action 3
void AnkleLock (void);                  // Action 4
void HipSwing(void);                    // Action 5

// State Machine Function
void Process_Data_From_Local_IO(void);   // States I-IV

#endif
```


B
Inner Heel Limit Switch
Inner Potentiometer
C
Outer Heel Limit Switch
Outer Potentiometer
H
Hip Potentiometer
3 CW
D
To Mike Sherback's H-Bridge (Not in use)
F
E
To Accelerometer (Not in use)
G H
C B
DC/DC Converter
J I
To Radio Receiver
A
PWM Cables
Innovation First
ROBOT CONTROLLER
BATTERY POWER
PROGRAM STATE
PWM IN/RADIO
FAULT STATUS
PWM OUT
R/C PWM IN
SOLENOID OUT
DIGITAL IN/OUT - ANALOG IN
INTERRUPTS
PROGRAM
7.2v BATT
PROG
ON/OFF
Inner Motor
Outer Motor
Hip Motor
VICTOR 884
IN
OUT
HIP
A1-A8 to PWM 1- PWM 8 (Not in use)
(From BLK)
(From +5V)
Three Way Switch
On
Off
Charge
Fuse
12V Battery