

5-2014

Development of a Control System for a Power Wheelchair Trainer

Stewart James Hildebrand
Grand Valley State University

Follow this and additional works at: <https://scholarworks.gvsu.edu/theses>



Part of the [Engineering Commons](#)

ScholarWorks Citation

Hildebrand, Stewart James, "Development of a Control System for a Power Wheelchair Trainer" (2014).
Masters Theses. 717.

<https://scholarworks.gvsu.edu/theses/717>

This Thesis is brought to you for free and open access by the Graduate Research and Creative Practice at ScholarWorks@GVSU. It has been accepted for inclusion in Masters Theses by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.

Development of a Control System for a Power Wheelchair Trainer

Stewart James Hildebrand

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Master of Science in Engineering

Biomedical Engineering emphasis

Padnos College of Engineering and Computing

May 2014

Abstract

The development of a Power Wheelchair Trainer for use by individuals with severe motor, cognitive, and communication deficits is described. These individuals, who are limited in their ability to use self-initiated mobility, are typically not considered to be candidates for power wheelchair use. The Power Wheelchair Trainer provides a motorized platform that allows a manual wheelchair to be temporarily converted into a power wheelchair, thereby permitting these individuals to practice using powered mobility while optimally positioned in their own customized seating systems. To accommodate the needs of these individuals, the Power Wheelchair Trainer incorporates additional control features not available in commercial systems. These additional features include: 1) a platform that may be raised or lowered for loading and unloading of the manual wheelchair without lifting; 2) a wireless joystick and multiple types of switch inputs for versatile drive control; 3) a safety remote with an override joystick and emergency stop switch; and 4) a user interface permitting the therapist to configure programmable settings. Programmable settings included: 1) joystick center dead-zone; 2) joystick outer dead-zone with time-out; 3) joystick sensitivity, also known as tremor dampening; 4) joystick throw; 5) speed; 6) acceleration; 7) deceleration; and 8) invert mode. A prototype of the Power Wheelchair Trainer was developed for use with children and young adults at the Lincoln Development Center, an area public school in Grand Rapids, Michigan that provides special education services. Power mobility experts have expressed positive feedback about the Power Wheelchair Trainer and would like to see more rehabilitation centers and schools offer this unique training opportunity.

Table of Contents

Abstract.....	3
I. Introduction	10
II. Literature Review.....	12
II.1 Independent Mobility during Childhood Development.....	12
II.2 Current Research in Power Mobility Training.....	13
II.3 Drive Control.....	14
II.4 Prior Work – The Turtle Trainer	18
II.5 Previous Versions of the Power Wheelchair Trainer	20
II.6 Programmable Features Available in Commercial Power Wheelchair Control Systems	25
II.6.1 Invacare MK6 Programmable Features	25
II.6.2 PGDT R-Net Omni+ Programmable Features.....	26
II.7 PI Control.....	27
III. Approach and Methods	30
III.1 Requirements Gathering.....	30
III.2 Electrical Design	31
III.3 Software Design	31
IV. Design Input and Requirements	33
IV.1 List of Requirements.....	33
IV.2 Required Features Not Available in Commercially Available Control Systems	34
IV.2.1 Outer Dead Zone	34

IV.2.2 Wireless Joystick.....	35
IV.2.3 Remote Emergency Stop and Override	35
IV.2.4 Linear Actuator Control	36
IV.2.5 Named Profiles	36
IV.3 Input Requirements	37
IV.3.1 Emergency Stop.....	37
IV.4 Programming	38
IV.5 Improvements over Previous Versions.....	42
IV.5.1 Transportability	42
IV.5.2 Joystick Improvements.....	43
IV.5.3 Tray.....	44
IV.5.4 Veer Correction	44
IV.6 Quick Start Guide	45
IV.7 Operator Manual	45
V. Theory of Operation	46
V.1 Specifications Table	46
V.2 Functional Diagram.....	53
V.3 Main Receiver Circuit Board	55
V.3.1 Motor Controller.....	56
V.3.2 Platform Actuators	58

V.3.3 Control Panel	59
V.3.4 Emergency Stop Switches.....	62
V.3.5 Receiving Wireless Module	63
V.4 Remote Circuit Boards	63
V.4.1 Power Switch	64
V.4.2 LED	64
V.4.3 Transmitting Wireless Modules.....	64
V.4.4 Learner Joystick	64
V.4.5 Therapist Remote	66
V.5 Wireless Hardware.....	67
V.6 Software.....	68
V.6.1 Software Architecture and Main Decision Module	68
V.6.2 Emergency Stop.....	71
V.6.3 Wireless Protocol.....	71
V.6.4 User Interface Module.....	74
V.6.5 Driving Logic and Stored Parameters	77
V.6.6 Motor Controller Communication.....	82
V.6.7 LCD Driver	82
V.6.8 Debounce Module	86
V.6.9 Watchdog Timer	87

V.7 Enclosures and Connectors.....	87
V.8 Obstacles Overcome and Lessons Learned.....	89
V.8.1 Joystick Shearing/Skewed Axes	89
V.8.2 LCD Goes Blank	90
VI. Assessment	91
VI.1 Verification.....	91
VI.1.1 Specification Testing.....	91
VI.2 Validation.....	95
VII. Future Work.....	98
VII.1 Effects of Power Mobility during Childhood Development.....	98
VII.2 Unloading the Wheelchair in the event of a Power Failure	98
VII.3 Mid-wheel Drive	98
VII.4 Music and Vibration Feedback	99
VII.5 Low Battery Indicator	99
VII.6 EMI.....	99
VIII. Conclusion.....	100
Appendix A : Invacare MK6 Programmable Settings	101
Appendix B : PGDT R-Net Omni+ Programmable Settings.....	102
Appendix C : Schematics	105
Appendix D : Source Code (Main Board)	114
Appendix E : Source Code (Remote)	178
Appendix F : Bill of Materials	194

Appendix G : Quick Start Guide	197
Appendix H : Operator Manual.....	199
IX. Intended Use and Safety.....	200
IX.1 Intended Use.....	200
IX.2 Safety	200
X. Parts of the system	201
X.1 Power Unit	202
X.2 Control Panel.....	203
X.3 Learner Joystick.....	204
X.4 Therapist Remote.....	204
X.5 Tray	205
XI. Prior to Use	206
XI.1 Charging	206
XII. Operation	208
XII.1 Loading and Unloading a Learner	208
XII.2 Drive Control	209
XII.2.1 Tray Setup.....	209
XII.2.2 Learner Joystick.....	211
XII.2.3 Therapist Remote	212
XII.3 Driving Profiles.....	213
XII.3.1 Profile Settings.....	213

XIII. Disassembly for Transport	220
XIV. Troubleshooting.....	224
XV. Maintenance.....	227
XVI. References	228

I. Introduction

The Power Wheelchair Trainer is a device that aids with the training of powered mobility by converting a manual wheelchair into a power wheelchair. With the Power Wheelchair Trainer, children and adults with severe motor, cognitive, and communication deficits who are not typically considered candidates for power mobility are able practice driving a power wheelchair while optimally positioned in their own customized seating system. Power wheelchairs without customized seating may not provide sufficient support, and learners in such a power wheelchair may not be positioned to optimize training capabilities. With the Power Wheelchair Trainer, learners can practice using power mobility with the goal of demonstrating that they are able to safely operate the controls of a power wheelchair, a prerequisite to becoming qualified for insurance reimbursement. Often a child will already have a manual wheelchair with customized seating for optimal support, but a power wheelchair with customized seating might not be readily available or it would be impractical to set one up for ongoing training purposes. The Power Wheelchair Trainer is a device that addresses the need for a simple conversion of a manual wheelchair into a power wheelchair.

This thesis project involved developing and building a control system for version 5 of the Power Wheelchair Trainer. The control system is the part of the Power Wheelchair Trainer that makes the wheels turn in response to joystick movements or other inputs. The control system for the Power Wheelchair Trainer maximizes learner and therapist safety, and allows for loading and unloading a manual wheelchair without lifting. The Power Wheelchair Trainer project has been ongoing at Grand Valley State University since Fall 2008. Table 1 gives a brief overview of the prior versions of the Power Wheelchair Trainer.

Version	Year	Description
1	2008	Rear wheel support only.
2	2009	Rear wheel support only. Improved caster.
3	2010	Platform with wheels underneath.
4	2011	Platform that is able to be lowered. Drive wheels in rear.
5	2012	Control system built by author.

Table 1: Power Wheelchair Trainer History

The existing literature is discussed related to the need for self-initiated mobility during childhood development, as well as, prior work on powered mobility training devices and how this prior work relates to the Power Wheelchair Trainer. Then the required features of the control system are described, and the development of the prototype is documented. Finally, an assessment plan is detailed, carried out, and documented.

II. Literature Review

This section presents the context in which this thesis project exists. This includes a review of existing work, and its relevancy to the Power Wheelchair Trainer project. First an overview is given of why mobility is important to childhood development. Then a brief review of current power mobility training research is given. A description of various power mobility access methods is given, and then a prior power mobility training project, the Turtle Trainer, is reviewed. Previous versions of the Power Wheelchair Trainer are then described, and finally an overview of programmable features in two commercially available control systems is given.

II.1 Independent Mobility during Childhood Development

Children with cerebral palsy, arthrogryposis, spinal muscular atrophy, spinal cord injury, osteogenesis imperfecta, and other neuromuscular or musculoskeletal impairments have difficulty physically exploring and interacting with their environment (Jones, McEwen, & Neas, 2012; Hays, 1987; Tefft, Guerette, & Furumasa, 1999). Children with severe motor impairments often do not have the ability to move independently making it difficult to explore and interact with their environment. This inability to move independently negatively influences the development of cognitive skills, spatial awareness skills, and social skills. The ability of children to move independently and explore their environment during childhood is important for the development of cognitive and psychosocial skills (Jones et al., 2012; Tefft et al., 1999). Tefft et al. (1999) identified which cognitive skills are influential in a young child's ability to operate a power wheelchair functionally. Basic problem solving skills and spatial relations are predictors of a child's ability to navigate a wheelchair through tight places, around stationary objects, and through a crowd of people since spatial relations help the child understand where objects are in relation to him or herself (Tefft et al., 1999). According to Tefft et al. (1999), self-initiated movement plays a crucial role in childhood cognitive and psychosocial development.

Additionally, independent mobility allows children with motor, cognitive, and communication impairments to integrate more fully into their educational programs (Tefft et al., 1999). The importance of independent mobility during childhood warrants the need for children with severe motor impairments to explore and interact with their environment, and power mobility may be a viable choice in many situations.

II.2 Current Research in Power Mobility Training

Jones et al. (2012) studied the effects of power wheelchairs on children with severe motor impairments and their development of psychosocial and cognitive skills. Jones et al. (2012) provided the families of the children involved in the study with training guidelines, and the children were provided with power wheelchairs with customized seating for home use. Jones et al. (2012) found that the use of power mobility enhances the development of young children with severe motor limitations. The study also found that the amount of time required for a group of children with motor impairments to become proficient in power wheelchair use ranged from 12 to 42 weeks, and that some children as young as 14 months of age may be suitable to begin using power mobility (Jones et al., 2012). As a result of this study, Jones et al. (2012) discussed that “more intense training in a structured and controlled environment” could help learners to attain wheelchair driving proficiency more quickly (p. 137). This study supports the incentive to have a power mobility training device available to children with motor impairments, and a viable location may be at their school or rehabilitation centers.

According to Durkin's (2009) responsive partner theory, the role of the power mobility teacher (therapist) should not be to train the child, but to set up a learning environment that meets the individual needs of the child (learner) while he or she plays in the powered mobility device (Livingstone, 2010; Durkin, 2006; Durkin, 2009). Learners with motor impairments often require customized seating, but it is impractical for the therapist to install customized seating on a power wheelchair that is used by

several learners. If a learner requiring customized seating were to be placed directly into an off-the-shelf power wheelchair with a standard seat, the learning environment would not be ideal since he or she would not be optimally positioned and may not have sufficient hand control without the extra support normally provided by customized seating.

Learners who are training with power mobility can be classified into *potential drivers* or *cause and effect learners*. Learners who are using power mobility to explore their environment are potential drivers. Learners who are trying to understand the concept of pushing a switch to make something happen are cause and effect learners. Nilsson (2007) introduced the concept of “driving to learn” instead of learning to drive. “Driving to learn” was a study about using power mobility intervention to learn tool use, where the tool is the joystick. Nilsson’s study found that learners participating in more than 30 training sessions for a period longer than 2 years, and training in more than one location were associated with reaching control of steering (Nilsson, 2007). As a result of the study, the theory of de-plateauing emerged. De-plateauing can be defined as skill improvement beyond preconceived expectations, which is made possible with assistance and training. The theory of de-plateauing is based on evidence that individuals with profound cognitive disabilities were capable of exceeding expectations and learning tool-use (Nilsson, 2007). Individuals who would not typically be considered candidates for power mobility due to cognitive disabilities or motor impairments may be able to use power mobility with sufficient training.

II.3 Drive Control

The industry de-facto standard method of power wheelchair drive control is the contactless inductive joystick. Pictured in Figure 1 is an inductive joystick from an Invacare MK IV controller. An inductive joystick is a proportional access method for power wheelchair control, meaning that driving speed depends on the angle of deflection of the joystick handle. An inductive joystick uses several

electromagnetic coils to measure the joystick handle position (U.S. Patent No. 6,043,806 A, 2000). One primary coil is mounted on the internal end of the joystick handle, and 4 secondary coils are mounted in fixed positions on the joystick base (2 coils for each axis). The primary coil induces a current which is picked up in the secondary coils. As the joystick handle moves the induced currents will change depending on the joystick handle position. An inductive joystick is distinct from a resistive joystick, another proportional control method. A resistive joystick uses a pair of potentiometers to measure the joystick handle position, one for each axis. The potentiometers in resistive joysticks are prone to dust buildup and may wear down with excessive use, whereas inductive joysticks are less prone to wear and tear, last much longer, and do not require maintenance. A third type of joystick is also available, known as a switch joystick or digital joystick. A switch joystick is essentially a handle that activates one of four switches. It is either on or off with no proportionality. Joysticks require grading of force, and people with abnormal muscle tone may not be able to effectively use a joystick (Lange, 2010a).



Figure 1: Inductive Joystick

Alternative methods of power wheelchair control include various types of switches mounted on the wheelchair or a tray. Between one and four switches may be set up to drive the wheelchair forward, right turn, left turn, or reverse. Each switch can simply be assigned to a driving direction, or some systems can be set up to allow the driver to activate both right turn and left turn switches at the same

time in order to drive forward eliminating the need for a dedicated forward switch. Other systems can be set up with a switch that can be used to drive either forward or reverse, alongside a toggle switch to switch between forward and reverse. Pictured in Figure 2 is a switch used for power wheelchair control. Learners can activate switches using their hands, upper arms, lower arms, tongue, legs, head movements. Most commonly these switches have a standard 3.5mm mono jack for connecting to a power wheelchair control system. AbleNet®, Inc (Roseville, Minnesota) offers a variety of switches of different sizes and mounting options (AbleNet, 2012). Switches with low or high activation force are available depending on the learner's ability to push on the switch.



Figure 2: Round Switch

Pictured in Figure 3 is a gooseneck switch. A gooseneck switch is a switch with a different mounting system. It has an adjustable arm with a lever that activates a single switch when the lever is moved or pushed, and it may be clamped on to a wheelchair. If a learner is unable to extend his or her arm to reach a switch mounted on a tray, then the learner could use his or her arm or head to activate a gooseneck switch mounted on the wheelchair.



Figure 3: Gooseneck Switch

A switch that does not require any force to activate is known as a fiber optic switch. A fiber optic switch detects the presence of an object by the reflection of light. Fiber optic switches are typically configured in an array (Lange, 2010b). Another type of switch that does not require any force to activate is a capacitive switch, similar to a computer touchpad. Capacitive switches have a larger area of activation than fiber optic switches.

If a user is only able to activate a single switch, a scanning technique may be employed to make it possible to fully operate the wheelchair with only one switch (Lange, 2010b). A display highlights the different drive directions one by one, and the driver activates the switch when the desired direction is highlighted. This method requires basic hand-eye coordination, and can be slow and laborious, but for some drivers it may be one of the few options available to enable power wheelchair access.

Access using head movements can be made possible with the head array (Lange, 2010b). A head array is a non-proportional access method utilizing proximity sensors to detect head movement, and allows the driver to control a power wheelchair with head movements. Typically three proximity sensors are mounted in a headrest. One sensor for driving left, a second sensor for driving right, and a third sensor mounted directly behind the head for driving forward or reverse. No force is required to activate the sensors – the head simply needs to get close to the proximity sensor to activate movement.

A head array is typically used in conjunction with a switch to toggle forward or reverse, which can be mounted at the distal end of a side proximity sensor. Some head arrays are based on mechanical switches instead of proximity sensors but offer the same basic functionality.

Another alternative method of power wheelchair control is sip-n-puff (Lange, 2010b). A sip-n-puff system allows the wheelchair driver to control the power wheelchair by mouth. An air tube is connected to the control system which measures the pressure of the air in the tube. The wheelchair driver then sips or puffs on the tube to control the power wheelchair. For example, a light puff to drive forward, a light sip to drive reverse, a hard puff to turn right, and a hard sip to turn left.

Several other control methods exist that are not commercially available due to lack of demand, safety concerns, or they only exist in research settings. Examples include a head controlled proportional joystick, chin joystick, and tongue joystick. A head controlled joystick consists of a joystick mounted in the headrest, and the driver pushed his or her head back against the joystick to activate movement. The head controlled joystick is a safety concern because applying sustained pressure against a rear pad may lead to increased muscle tone (Lange, 2012a). A chin cup joystick allows the driver to use his or her chin to drive the power wheelchair, but bumps in the terrain may lead to extraneous joystick movement and erratic driving. A tongue joystick allows the driver to use their tongue to control a power wheelchair. A tongue joystick can be mounted externally, or inside the mouth. The tongue joystick currently only exists in research settings.

II.4 Prior Work – The Turtle Trainer

The Turtle Trainer concept was first presented at a Rehabilitation Engineering and Assistive Technology Society of North America (RESNA) conference (Bresler, 1990). The Turtle Trainer is a device that converts a manual wheelchair into a power wheelchair, and it was developed to help evaluate an individual's ability to use power mobility. Figure 4 is an artist's rendition of the Turtle Trainer. Bresler

(1990) describes the device as “a motorized cart with wheelchair tiedowns,” (p. 399) and it has a ramp in the front for loading and unloading a manual wheelchair. The Turtle Trainer is a no-lift device, meaning that no lifting is required to load and unload a manual wheelchair. The Turtle Trainer also supports several types of input controls including a joystick and switches, and it features a wired emergency stop switch. An artist’s rendition of the Turtle Trainer is shown in Figure 4.

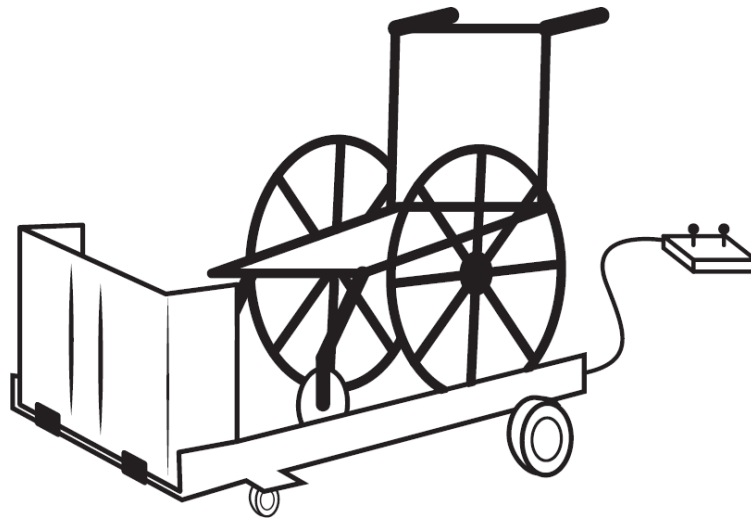


Figure 4: Artist’s Rendition of the Turtle Trainer

Before the Turtle Trainer individuals unable to self-propel a manual wheelchair who would like to train using a power wheelchair were limited to using a demo power wheelchair without customized seating, having an evaluator take the time to customize the seating on a demo power wheelchair, or use a computer simulation. The Turtle Trainer gave power mobility evaluators a new option to evaluate power mobility readiness. Bresler (1990) reported that individuals whom evaluators were sure could not use power mobility surprised the researcher and the school staff when using the Turtle Trainer.

The Turtle Trainer did ultimately not succeed commercially because it was focused exclusively on teaching students to drive instead of the benefits of mobility, and it did not have any clinical data to prove the effectiveness of the product. Additionally, the design was large and unsophisticated, and the large front ramp created a blind spot directly in front of the Turtle Trainer making it difficult to navigate

effectively. The design of the Power Wheelchair Trainer ensures that there is no blind spot in the front and enable no-lift wheelchair loading.

II.5 Previous Versions of the Power Wheelchair Trainer

The Power Wheelchair Trainer project has been ongoing at Grand Valley State University since Fall 2008. Version 1 (shown in Figure 5) was built to solve the problem of converting a manual wheelchair into a power wheelchair. A manual wheelchair's rear wheels would be mounted on version 1 of the Power Wheelchair Trainer while the manual wheelchair's own front casters were utilized for driving. Version 1 of the Power Wheelchair Trainer proved the concept of converting a manual wheelchair into a power wheelchair, but it was difficult and cumbersome for a therapist to load the manual wheelchair.



Figure 5: Version 1 (2008)

Version 2, shown in Figure 6, was based on the same general concept as version 1, except that it had a wider wheelbase to allow for easier turns. Version 2 was focused on trying to make a better caster and ease of loading. Ramps were added to make loading easier, but versions 1 and 2 were ultimately not successful due to the reliance on the manual wheelchair's own front casters.

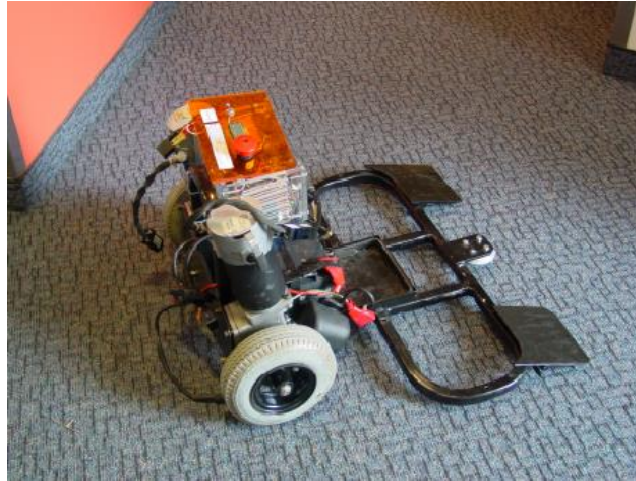


Figure 6: Version 2 (2009)

Version 3 of the Power Wheelchair Trainer is shown in Figure 7 and Figure 8. The concept for version 3 was to have all the electronics, including motors, batteries, and control system contained underneath a platform. A ramp and a winch were used to assist the therapist in loading the manual wheelchair on the platform. With this concept, however, it proved to be rather cumbersome and time consuming to load and unload a manual wheelchair, and there was a risk of injury to therapist due to sharp corners. The winch was removed due to its difficulty of operation. Version 3 was also heavy and not practical to transport in a vehicle. Version 3 of the Power Wheelchair Trainer was dismantled and parts used for version 4.

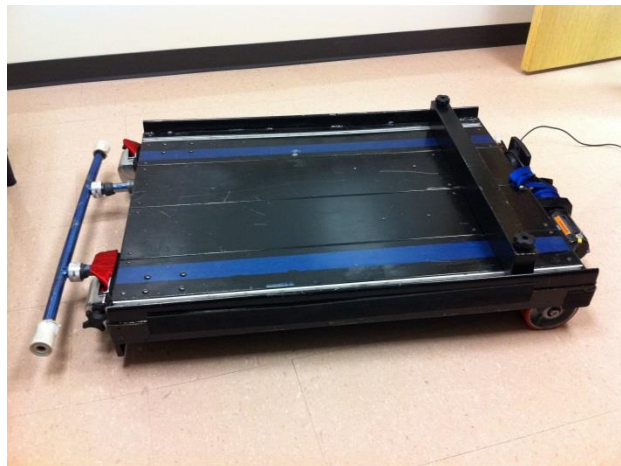


Figure 7: Version 3 (2010) Photo



Figure 8: Version 3 (2010) Rendering

Version 4 of the Power Wheelchair Trainer was built by engineering students at Grand Valley State University during the winter semester of 2011, and is pictured in Figure 9 and Figure 10. Unlike prior versions of the Power Wheelchair Trainer, version 4 was built around the concept of a platform that could be lowered for easy loading and unloading of the manual wheelchair with no lifting required. To load a wheelchair, the platform was lowered by pushing a switch that controls 4 linear actuators. As the linear actuators extended, the platform lowered. The front gate was removed, allowing the wheelchair to be easily wheeled onto the lowered platform and strapped down using industry standard tiedowns. The design for version 4 has had the most positive feedback from the therapists using it. Version 4 of the Power Wheelchair Trainer can be classified as versions 4a and 4b due to modifications to the design. “Version 4” will be used to refer to both version 4a and 4b. Version 4a had a square front gate that did not leave enough leg room, and the corners of the gate could damage the walls. Version 4a also had bumpers that would tear easily, and the drive wheels were small and caused damage due to spinning. Version 4b was upgraded to a rounded front gate, sturdier bumpers, and larger drive wheels. Version 4 utilized a PGDT R-Net Omni+ (PG Drives Technology, United Kingdom) power wheelchair control system together with a custom input switching and platform control system. The system was powered by two 12V lead acid batteries in series, and used industry standard power wheelchair DC brushed motors. For loading and unloading of a manual wheelchair, the main platform could be

lowered and raised, and the lowering and raising mechanism was provided by four linear actuators. There was a safety remote with an emergency stop switch that shuts down the PGDT R-Net Omni+ system. It was capable of being driven with switches, and a digital (4-direction switch) joystick. Switches could be mounted anywhere on the manual wheelchair to meet the needs of the learner, making it possible for him or her to use head movements to control the power wheelchair. The 4-direction joystick mounts in a customized tray that is mounted on the manual wheelchair. Version 3 of the Power Wheelchair Trainer contained the batteries, motors, and control system underneath the platform, in version 4 these are all contained in a chassis in the rear that is detachable from the main platform to make transportation in a vehicle easier. As of December 2013 version 4b of the Power Wheelchair Trainer was still in use. There have been important lessons learned from the perspective of supporting and maintaining version 4 of the Power Wheelchair Trainer:

- While commercial power wheelchairs typically use proportional inductive joysticks, the digital joystick is limited in the ability to finely control the drive speed.
- The small rear wheels on version 4a would spin when the Power Wheelchair Trainer was driven into walls or other obstacles resulting in damage to the training environment. These wheels have since been replaced with larger rear wheels on version 4b to reduce wheel spin and reduce damage to the training environment.
- The batteries are not removable resulting in an extremely heavy rear chassis that is difficult to lift into a vehicle for transport (yet still easier than version 3).
- The square front gate on version 4a (pictured in Figure 9) did not accommodate learners with long legs and large wheelchairs very well. The corners could also cause damage to the training environment and navigation through obstacles was difficult. The square front gate was replaced with a rounded front gate on version 4b (pictured in Figure 10) to alleviate these problems.

- Version 4a of the Power Wheelchair Trainer originally had sensing bumpers that would disable the motors on impact. However, these sensing bumpers were not sturdy enough and eventually tore apart. The sensing bumpers also did not allow for the children to learn from bumping into objects. The sensing bumper system was replaced with non-metallic electrical conduit on version 4b. The conduit bumpers also worked well to maneuver through doorways and obstacles because it was smooth and had some flexibility.
- When driving straight forward, version 4 of the Power Wheelchair Trainer would sometimes unintentionally veer to the right or left (drift). The issue happened more often after a turn where the front casters did not line up much like a common shopping cart. It was speculated that it could be caused by poor quality front casters, manufacturing differences in the motors, or uneven weight distribution. The front casters were replaced and this helped, but veering continued to be a problem.
- The sensor that detected when the platform was up was unreliable, but this was since fixed.
- There was exposed wiring underneath the switch panel that was not covered up very well. Loose connections could sometimes lead to the device being non-operational.



Figure 9: Version 4a (2011)



Figure 10: Version 4b (2011)

II.6 Programmable Features Available in Commercial Power Wheelchair

Control Systems

A goal while developing the control system for version 5 of the Power Wheelchair Trainer was to provide a feature set comparable to those of commercial power wheelchairs while simplifying the programming aspects and removing unnecessary features. Programming is defined as modifying the configurable parameters (e.g. speed, acceleration, etc.) of the control system to meet a learner's needs. The programmable features of two commercially available power wheelchair control systems were evaluated, 1) the PG Drives Technology (PGDT) R-Net Omni+ power wheelchair controller (this controller was used in version 4 of the Power Wheelchair Trainer) and 2) the Invacare MK6 power wheelchair. Only the settings applicable to driving and joysticks were taken into account.

II.6.1 Invacare MK6 Programmable Features

Appendix A contains a list of the relevant programmable settings available in an Invacare MK6 controller (Invacare, 2011). The system is programmed with an external programming dongle. The system has a fixed number of 4 driving profiles; a feature termed "Drive Mode." The system allows for the maximum speeds, acceleration, and deceleration (braking) to be configured in the forward, reverse, and turning directions. There is an adjustable tremor dampening feature that accommodates tremor (ataxia) in the hand and upper extremities. There is a maximum power level setting that limits the

current available to the motors, in addition to a torque setting and a proprietary G-Trac feature, which uses gyroscope technology to ensure that the wheelchair drives in a straighter path. A traction setting lowers the speed when turning. A programmable joystick throw setting determines at what point full speed is reached in relation to joystick deflection. An axes select setting can assign joystick commands to a desired driving direction. An input type setting chooses between proportional joystick, switches, sip-n-puff, and other drive control inputs. Lastly, a momentary/latch setting can be set to command the wheelchair to drive forward after the joystick has been released, until the joystick is pulled reverse or emergency stop is activated.

II.6.2 PGDT R-Net Omni+ Programmable Features

Appendix B contains a list of the relevant programmable settings available in a PGDT R-Net Omni+ controller (PG Drives Technology, 2011). The system can be programmed on board, or with a separate programming module. The system features up to 8 named driving profiles. Most of the programmable settings use percentages, not real units like miles per hour. The system has an easily accessible speed setting, ranging from 1 to 5. Programmable settings like speed, acceleration, and other settings that affect the driving speed have two separate values for the minimum speed (1) and maximum speed (5). The maximum speed, acceleration, and deceleration are configurable in the forward, reverse, and turning directions. There is a setting for tremor dampening, which can be used for smoothing the effects of a learner's hand tremor. The system allows for programmable torque and power. A "fast brake rate" setting specifies the deceleration when pulling the joystick handle backwards to stop faster. Joystick throw is configurable in all four directions. There is a configurable center dead zone, or neutral area. The joystick can be inverted forward/reverse, and left/right. There is a latched setting that enables the wheelchair to keep driving forward after releasing the joystick. The system also features a boost drive current setting, and a current foldback threshold to protect the motors from overheating. There are settings to invert either motor's direction, along with a setting to swap the

right/left motor outputs. Finally, there is a setting for steer correction to compensate for differences in motor speeds.

II.7 PI Control

Version 4 of the Power Wheelchair Trainer occasionally experienced an issue called drifting, or veering. When the joystick handle was pushed straight forward, the Power Wheelchair Trainer would drive forward but veer to the right or left. This problem can arise for a number of different reasons, including manufacturing differences in the motors, uneven weight distribution, and poor quality front casters. The problem is amplified as the batteries discharge. This is a classic control problem that can be solved with a Proportional-Integral (PI) controller algorithm (Åström & Hägglund, 1995). The PI controller algorithm requires motor speed feedback, which can be provided by optical sensors. Power wheelchair motors do not typically have optical sensors built in, so the motors require modification for the Power Wheelchair Trainer. A slotted disk is mounted on the drive shaft of the motor, and a photo-interrupter is installed to read the movement of the drive shaft. This concept is illustrated in Figure 11.



Figure 11: Slotted Disc and Photo-interrupter

Given the complexity of modifying motors in this manner, the significance of the veering issue will be evaluated and the solution will only be implemented if deemed necessary. The signals from the

photo-interrupters attached to the right and left motors will be used in software to correct for veering while driving straight forward or reverse. The optical sensors allow the PI controller algorithm to properly control the speed of the motors using a feedback system. The feedback system for one motor can be modeled as the process in Figure 12.

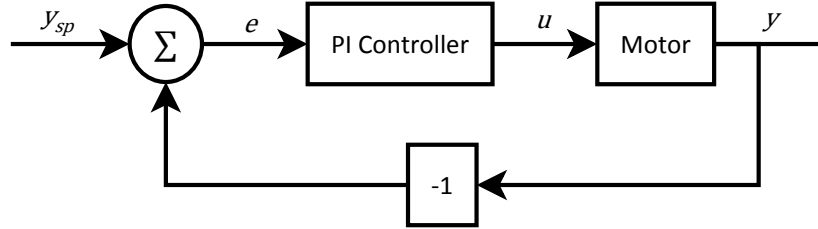


Figure 12: Motor Feedback Process

In Figure 12, y_{sp} is the setpoint, or the desired motor speed as determined by a joystick input. y is the process variable, or the motor speed as measured by an optical sensor. e is the control error, and is given by equation 1.

$$e = y_{sp} - y \quad (1)$$

u is the control variable, or the PI controller, and is given by equation 2.

$$u(t) = K \left(e(t) + \frac{1}{T} \int_0^t e(\tau) d\tau \right) \quad (2)$$

The PI controller is the sum of the proportional and integral terms. The controller parameters are K , the proportional gain, and T , integral time. These parameters can be tuned empirically to suit the application. For digital implementation purposes, it is helpful to write the PI controller equation in discrete form, shown in equation 3.

$$u(t_k) = Ke(t_k) + I(t_k) \quad (3)$$

$Ke(t_k)$ is the proportional term, and $I(T_k)$ is the integral term given by the recursive equation shown in equation 4, using Tustin's approximation.

$$I(t_k) = I(t_{k-1}) + \frac{Kh}{T} \times \frac{e(t_k) + e(t_{k-1})}{2} \quad (4)$$

In equation 4, h is the discrete sampling period.

III. Approach and Methods

The purpose of Power Wheelchair Trainer is to assist in the training of children and adults with severe motor impairments to use a power wheelchair. To facilitate this, a prototype of the Power Wheelchair Trainer was developed and produced. This prototype was dubbed version 5. The overall mechanical design of version 5 of the Power Wheelchair Trainer was based on the version 4 design because that design has received positive feedback from therapists concerning ease of loading and unloading manual wheelchairs (C. Ripmaster, personal communication, 2012). The control system, including circuit boards, cables, connectors, joystick, remote, and software were produced by the author. The remaining mechanical components of the Power Wheelchair Trainer, including frame, platform, mounting systems, housing for the joystick, and tray were produced by graduate students at Grand Valley State University other than the author and the development of these components is outside the scope of this thesis.

III.1 Requirements Gathering

The requirements for the control system of the Power Wheelchair Trainer were derived in collaboration with power mobility experts. During prototype development, weekly meetings were held with the “customer”. The prototype of the Power Wheelchair Trainer version 5 was built in close collaboration with power mobility experts, and feedback was gathered and implemented on a design-build-iterate basis. A minimum viable product approach was taken, meaning that if a feature request was deemed as non-critical to the application it was not implemented in favor of time constraints. The requirements were documented in section IV. From the user requirements, engineering specifications were derived in section V.

III.2 Electrical Design

The most significant change between version 4 and version 5 of the Power Wheelchair Trainer was to replace the Omni+ control system with a custom control system. The schematics, circuit board layouts, and software from version 4 of the Power Wheelchair Trainer were used as a starting point for the electrical design. The design software used for capturing schematics and laying out circuit boards was Eagle version 5.11 by CadSoft (Germany). The design needed to be reproducible in the future, so it is important to have the design files readily available. To accomplish that goal, and to assist with version control management, the schematics and circuit board layout files have been open sourced and hosted at the website <http://pwct.googlecode.com/> in addition being included in Appendix C, Appendix D, and Appendix E. A complete bill of materials is hosted at <http://pwct.googlecode.com/> and in Appendix F.

Component selection and circuit board layouts were performed by the author in order to meet the requirements, and a review was conducted with an electrical engineering professor at Grand Valley State University. Board layout files were then submitted to a circuit board manufacturing company and the components were ordered. Approximately 20% extra components were ordered for debugging purposes and in case of component failure. After the parts arrived, the circuit boards were assembled and debugged by the author.

III.3 Software Design

After the circuit boards were assembled, software development began. For the same reasons mentioned in the previous section, the source code was open sourced and hosted with version control at <http://pwct.googlecode.com/>, in addition to being included in Appendix D and Appendix E. The software was written in the C programming language, and the development tool used was the Atmel Studio development environment for the AVR processor. The first task was to remove obsolete code, such as code for the obsolete bumper sensor system in version 4 of the Power Wheelchair Trainer. All

required software functionality is divided into modules and sub-modules, and each was implemented in phases consisting of implementing a (sub-) module, testing, committing to version control, and iterating.

IV. Design Input and Requirements

This section describes the design requirements of the control system of the Power Wheelchair Trainer needed to fulfill the intended purpose. Control system requirements are listed and described from a user perspective. The motive for building a custom control system is 1) cost reduction compared to commercial systems, 2) convenience of a wireless joystick, 3) remote emergency stop and override, 4) drift correction, 5) and outer dead zone timeout for some students who get frustrated and push full forward (see section IV.2.1).

IV.1 List of Requirements

All user requirements for the control system are listed in Table 2, and are explained in detail in the following sections.

#	Requirement	Described in section
1	Wireless joystick	IV.2.2, IV.3, IV.5.2
2	Safety remote override	IV.2.3
3	Safety remote emergency stop	IV.2.3
4	Linear actuator control	IV.2.4
5	Switch inputs	IV.3
6	Hardwired emergency stops	IV.3
7	Charging terminal	IV.3
8	Named profiles ≥ 20	IV.2.5, IV.4
9	Joystick outer dead zone with programmable time-out	IV.2.1, IV.4
10	Programmable throw	IV.4
11	Programmable speed	IV.4
12	Programmable sensitivity	IV.4
13	Programmable acceleration	IV.4

14	Programmable deceleration	IV.4
15	Programmable center dead zone	IV.4, IV.5.2.1
16	Invert mode	IV.4
17	Proportional as switch	IV.4
18	Transportability	IV.5.1
19	Modular tray	IV.5.3
20	Minimal veering	IV.5.4

Table 2: User Requirements

IV.2 Required Features Not Available in Commercially Available Control

Systems

The control system in version 5 of the Power Wheelchair Trainer provides additional features compared to those available in a commercially available power wheelchair control system. These features were identified in collaboration with physical therapists. These novel features warrant the need to build a custom control system for the Power Wheelchair Trainer. Although version 4 of the Power Wheelchair Trainer utilized a commercial control system in combination with custom circuitry to accommodate the features not provided by the commercial system, it was a complex task to maintain the two separate electronic systems. The control system in version 5 of the Power Wheelchair Trainer is simplified by reducing the number of components with a fully custom control system and a simpler motor controller. In this section, the features that require custom electronics and software in version 5 of the Power Wheelchair Trainer are described.

IV.2.1 Outer Dead Zone

Power mobility experts expressed the need for a timeout option for some students who would push the joystick full forward due to frustration, resulting in their elbow locking into extension and being unable to voluntarily release (C. Ripmaster, personal communication, 2012). In other words, a way was

needed to keep the Power Wheelchair Trainer from going at maximum speed in this situation. A joystick outer dead zone with an adjustable time-out was implemented as a solution. If the joystick handle is fully deflected (in other words, the joystick handle is held all the way to the edge) for a specified time, the motors stop. When the joystick handle is fully deflected, the amount of time until the motors stop is programmable. Operation resumes after the joystick handle has been returned to center.

IV.2.2 Wireless Joystick

For convenience and ease of setup purposes, the learner joystick is wireless. This eliminates one step in the setup process and enables the learner to spend more time driving the Power Wheelchair Trainer. If the wireless connection to the learner joystick is lost, the Power Wheelchair Trainer stops moving. The joystick housing was designed to fit in a modular tray. The electronics and software for the joystick were produced by the author. The development process of the tray and housing for the joystick are out of scope for this thesis.

IV.2.3 Remote Emergency Stop and Override

The Power Wheelchair Trainer has a remote emergency stop, known as the therapist remote. When the emergency stop switch on the therapist remote is activated, the Power Wheelchair Trainer stops operation immediately. The therapist remote also has an override thumb joystick enabling the supervising therapist to take control of the Power Wheelchair Trainer if needed. The override joystick on the therapist remote takes priority over all other inputs. The therapist remote is required to be powered on and in proximity of the Power Wheelchair Trainer in order for the system to function. If the wireless connection to the therapist remote is lost, the Power Wheelchair Trainer stops moving. The therapist remote is capable of driving the Power Wheelchair Trainer independently, even when the learner joystick is powered off. The learner joystick cannot drive the Power Wheelchair Trainer without

the therapist remote in range. The therapist remote has a light indicator that lights up when the wireless connection to the receiver is lost.

IV.2.4 Linear Actuator Control

The main platform of the Power Wheelchair Trainer has the ability to be lowered and raised for no-lift loading and unloading of the manual wheelchair. Platform lowering and raising is controlled by a switch on the control panel. The action of lowering and raising the platform is carried out by linear actuators with built-in limit switches. When the platform is down, the Power Wheelchair Trainer is not operational and the LCD screen displays the message “Platform down” as seen in Figure 13.

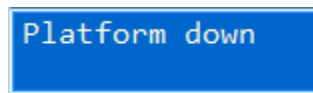


Figure 13: LCD Platform Down

As a safety precaution the Power Wheelchair Trainer does not drive while the platform is lowered.

IV.2.5 Named Profiles

Since the Power Wheelchair Trainer is used in a school setting with many different learners, each student may require different speed, acceleration, etc. settings. Commercially available control systems only allow for a small number of driving profiles to be specified. The Power Wheelchair Trainer control system provides at least 20 driving profiles with a target of 50. The control system provides the ability for the therapist to choose between several named driving profiles. Profile settings and names are specified by the therapist and individualized to each learner. Profile names make it easy for the therapist to choose the profile when a new learner is ready to drive.

IV.3 Input Requirements

To accommodate various learner capabilities, the Power Wheelchair Trainer has the capability to be driven using switches or joysticks:

- Up to four switches may optionally be plugged in, and each switch causes the Power Wheelchair Trainer to drive in one of the four directions: forward, reverse, right, or left. An activated switch is fully on, meaning that it is equivalent to pushing the joystick handle as far as it goes.
- A wireless learner joystick provides proportional control and is comparable to an industry standard inductive joystick.
- A thumb joystick on the therapist remote allows the supervising therapist to override movement of the Power Wheelchair Trainer.

The Power Wheelchair Trainer also has several non-drive-control inputs:

- As a safety feature the Power Wheelchair Trainer has hard-wired emergency stop inputs. The emergency stop switches use normally closed circuitry (see section V.3.4), so if the wiring fails it will trigger an emergency stop condition.
- An emergency stop switch on the therapist remote.
- A power switch on the main control panel.
- A platform raise/lower switch on the main control panel.
- A charging terminal.

IV.3.1 Emergency Stop

As a safety feature, pushing any of the emergency stop switches cause the motors to stop immediately. The source of the emergency stop is shown on the LCD screen. The message shown in Figure 14 will be displayed on the LCD when the hardwired emergency stop switches trigger the emergency stop condition.

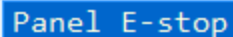
A blue rectangular box with a thin white border containing the text "Panel E-stop" in a white, monospaced font.

Figure 14: Panel Emergency Stop Display

The message shown in Figure 15 will be displayed on the LCD when the therapist remote triggers the emergency stop condition.

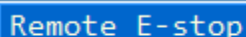
A blue rectangular box with a thin white border containing the text "Remote E-stop" in a white, monospaced font.

Figure 15: Remote Emergency Stop LCD Message

To clear the emergency stop condition and make the system usable again, the main power switch must be toggled off and on.

IV.4 Programming

A separate programming dongle is not necessary to adjust the configurable settings of the Power Wheelchair Trainer. Programming is done on the controller via a two-line character display and four arrow key menu buttons. The therapist interacts with this user interface to change settings. Detailed operation of the user interface is described in section XII.3 of the Operator Manual in Appendix H. The LCD shows the active profile and selected setting such as top speed, acceleration, etc. Additionally the LCD shows when there is an emergency stop condition, or when the platform is lowered. Figure 16 shows a mockup of what the user interface looks like. The arrow keys shown in the figure are pushbuttons.

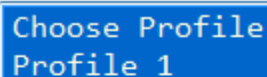
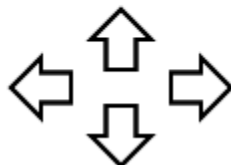
A blue rectangular box with a thin white border containing the text "Choose Profile" on the first line and "Profile 1" on the second line, both in a white, monospaced font.

Figure 16: User Interface - Character Display and Arrow Keys

The user interface and driving profiles are new in version 5 of the Power Wheelchair Trainer. The user interface enables the therapist to choose a driving profile and view or modify stored parameters. The right and left arrow keys on the control panel allow the therapist to cycle through the settings for viewing or changing, and the up and down keys allow the therapist to change the value. The LCD screen shows the current setting and its value on the first line, except for the choose profile setting where “Choose Profile” will be displayed on the first line, and the name of the selected profile on the second line. Modified parameters are remembered even when the system is powered off. Figure 17 shows an example of a setting and its value.

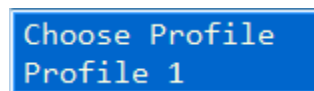


Figure 17: Example Setting and Value

The required programmable settings include a subset of the settings described in section II.6, in addition to a configurable outer dead zone described in section IV.2.1 and named profiles as described in section IV.2.5. The required programmable items and their descriptions are listed as follows:

1. Driving profile: This setting is used for easily switching between driving profiles customized for different learners. A profile name can be specified through the user interface to make the driving profile easier to remember. The values for the subsequent programmable settings are stored separately for each driving profile. When a driving profile is selected, the programmable settings for that driving profile become active. Available characters for specifying profile names are uppercase A-Z, lowercase a-z, numbers 0-9, and a blank or space. A mechanism is provided for editing profile names through the user interface.
2. Throw: The amount of joystick handle deflection that is required to reach maximum speed. There will be separate throw settings for forward, reverse, and turn.

3. Maximum speed: The maximum driving speed. There will be separate maximum speed settings for forward, reverse, and turn.
4. Sensitivity: How quickly the Power Wheelchair Trainer responds to joystick movement or activation of a switch. Sensitivity is also known as tremor dampening. This feature will be implemented as a low-pass filter with configurable cut-off frequency.
5. Acceleration and Deceleration: The maximum change in speed over time. This setting will determine how quickly the Power Wheelchair Trainer speeds up or slows down.
6. Center dead zone: A circular dead zone in the center of the joystick where the joystick is considered to be centered. This setting will determine how far the joystick handle has to travel from the center for the motors to start moving. Figure 18 shows the “shape” of the joystick with the gray area in the middle corresponding to the center dead zone, where the size of the gray circle (the dead zone) is configurable.

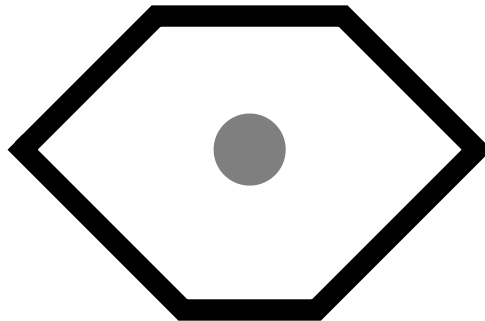


Figure 18: Center Dead Zone

7. Outer dead band: When the joystick handle is fully deflected, the motors will optionally shut off immediately or after a programmable time-out. The joystick handle is considered to be fully deflected when it is in the outer dead zone, indicated in Figure 19 with the gray area corresponding to the outer dead zone. The joystick must return to center before the Power Wheelchair Trainer will start driving again.

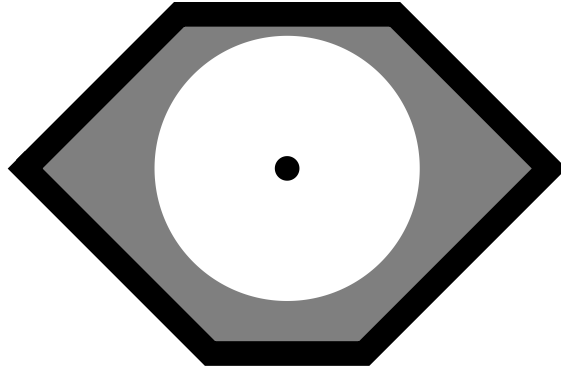


Figure 19: Outer Dead Band

8. Invert: When this setting is enabled, pushing the joystick handle forward commands to drive reverse, and pushing the joystick handle in reverse commands to drive forward.
9. Proportional as switch: Converts a proportional joystick into a switch joystick (digital joystick) with only four possible directions and no proportionality.

Profile name editing functionality is provided to easily identify each profile (see Figure 20). To enter name edit mode hold the left arrow key for 2 seconds while the profile setting is active.

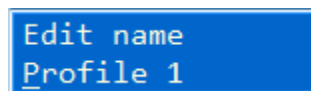


Figure 20: Name Edit Mode

Choose the character to modify using the right and left arrow keys (Figure 21).

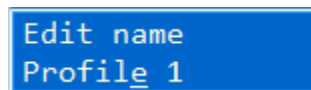


Figure 21: Character Selection

Modify the character with the up and down arrow keys (Figure 22). Allowed characters are A-Z, a-z, 0-9, and blank.

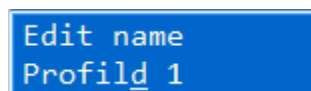


Figure 22: Character Modification

Exit name edit mode by holding the left arrow key for 2 seconds.

IV.5 Improvements over Previous Versions

This sub-section details the improvements that the control system for version 5 will provide over previous versions of the control system for the Power Wheelchair Trainer. Version 4 of the Power Wheelchair Trainer used the PGDT R-Net Omni+ system in addition to custom circuitry resulting in excessive exposed wiring. The control system for version 5 aims to be simpler than the previous version by replacing the Omni+ controller with a simpler motor controller. The control system for version 5 will be simpler with fewer components, resulting in less clutter and fewer exposed wires. Further improvements are detailed in the following sub-sections.

The improvements of version 5 over version 4 are summarized in Table 3, and are explained in the following sections.

Description	Detailed in section
Easier to transport in a vehicle	IV.5.1
Proportional joystick to allow for fine movements	IV.5.2
Improved joystick mounting mechanism	IV.5.2
Configurable joystick center dead zone	IV.5.2.1
Tray easier to set up	IV.5.3
Minimal veering	IV.5.4

Table 3: Summary of Improvements

IV.5.1 Transportability

Version 4 of the Power Wheelchair Trainer was heavy and difficult to load into a vehicle for transport, even though the power unit was detachable from the rest of the frame. The rear power unit contained 2 heavy batteries which made it difficult to load into a vehicle for transport. Version 5 of the Power Wheelchair Trainer has removable batteries to make loading into a transport vehicle easier. The batteries are able to be disconnected using appropriate high-amperage connectors. The batteries have handles to make them easier to pick up.

IV.5.2 Joystick Improvements

Version 4 of the Power Wheelchair Trainer used a wired 4-direction switch joystick, also known as a digital joystick. This joystick was bulky, and had an unsecure mounting system for mounting on the tray. Additionally, there was a lack of fine control and it was not intuitive to drive forward and turn simultaneously. The joystick in version 5 of the Power Wheelchair Trainer uses an industry standard inductive joystick with proportional control instead of a digital joystick. The joystick housing was developed to fit in a custom tray. For convenience and ease of setup, the joystick features a wireless connection to the Power Wheelchair Trainer with a minimum range of 25 feet and no required maximum range.

The learner joystick enables the learner to control the power wheelchair trainer with a hand joystick. The joystick perimeter has the irregular hexagonal shape shown in Figure 23 as viewed from the top, where the dot indicates the joystick center.

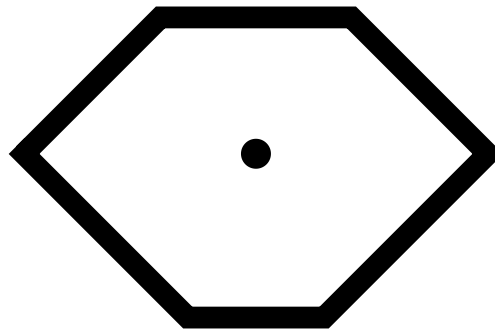


Figure 23: Joystick Shape

IV.5.2.1 Configurable Center Dead Zone

Some learners are not able to exert much force on the joystick handle and are only able to deflect the joystick handle a small distance, so a way is needed to make the Power Wheelchair Trainer move with little movement of the joystick. The learner joystick features a configurable center dead zone (described in section IV.4). The center dead zone, combined with configurable throw and sensitivity

(also described in section IV.4), allows the learner to drive the Power Wheelchair Trainer by deflecting the joystick handle a small distance. These settings are also useful in a cause and effect scenario when a joystick is preferred over switches.

IV.5.3 Tray

A tray was developed with a modular placement system. The tray is able to be mounted on the manual wheelchair, and the learner joystick is able to be mounted in the tray at an appropriate location for each learner without the use of tools. The tray has forearm support and learners are able to activate the joystick and/or switches with minimal movement of their upper extremities. The tray was also designed so the joystick is flush to the surface of the tray. Learners do not need to lift their hand up to make contact with the joystick, and this gives learners with minimal upper extremity function ability to use the joystick. The development process of the tray is out of scope for this thesis.

IV.5.4 Veer Correction

While veering was not anticipated to be an issue with a proportional joystick, veering could be a problem while using switches as input or when the joystick is configured to proportional as switch mode. The significance of the drift issue was evaluated in version 5 of the Power Wheelchair Trainer by driving straight forward 10 feet, then measuring the amount of veering to the right or left. If the Power Wheelchair Trainer veered significantly to the right or left, defined as 2 feet right or left for every 10 feet driven forward, then the plan for drift correction would be implemented described in section II.7. The test was performed and veering was measured to be 19 inches (1.58 feet) to the left, so the plan for veer correction was not implemented. The test result is recorded in Table 19 in section VI.1.1.

IV.6 Quick Start Guide

A quick start guide was developed to aid the therapist in quickly setting up and start using the Power Wheelchair Trainer. The quick start guide is a pamphlet with visual steps describing basic use and loading and unloading of a manual wheelchair, and can be found in Appendix G.

IV.7 Operator Manual

An operator manual was developed as a user reference guide. The operator manual contains detailed reference information about safety, use, setup, troubleshooting, transportation, and storage of the Power Wheelchair Trainer. The operator manual also contains details about configuration of the driving profiles on the control system. The operator manual can be found in Appendix H.

V. Theory of Operation

This section describes the functional operation and technical design of the Power Wheelchair Trainer control system. The functional components of the system are explained in detail along with their inputs and outputs. Then the technical implementation details are explained, which intends to serve as a reference for future maintenance or manufacturing. The implementation details include hardware schematics, printed circuit board layouts, component selection, connectors, bill of materials, budget, software development process, wireless communication protocol, and tools used. All source code, schematics, and circuit board files are available online from <http://pwct.googlecode.com> and in Appendix C, Appendix D, and Appendix E. The mechanical specifications and build details are not included in the scope of this thesis document.

V.1 Specifications Table

This sub-section contains engineering specifications for the Power Wheelchair Trainer control system. The engineering specifications have been derived from the user requirements. The specifications were tested after the Power Wheelchair Trainer was built. See section VI.1.1 for further testing details. Table 4 below lists the engineering specifications and delivered values.

#	Specification Name	Metric Description	Unit	Required Value	Target Value	Reasoning	Delivered Value
1	Power Wheelchair Trainer Battery Life	The amount of time the Power Wheelchair Trainer can drive on a single full charge.	Hours	3 or more	8 or more	Full-day operation.	5.25

#	Specification Name	Metric Description	Unit	Required Value	Target Value	Reasoning	Delivered Value
2	Learner Joystick Battery Life	The amount of time the learner joystick operate on fresh batteries	Hours	8 or more	10 or more	Full-day operation.	12+
3	Therapist Remote Battery Life	The amount of time the therapist remote shall operate on fresh batteries	Hours	8 or more	10 or more	Full-day operation.	24+
4	Right/Left Drift	How far the Power Wheelchair Trainer is allowed to veer to the right or left when driven 10 feet straight forward on a flat surface	Feet	2 or less	1 or less	Less veer than version 4. Some veer is acceptable for indoor usage.	1.583 to the left
5	Number of profiles	How many named driving profiles are available	Count	20 or more	50 or more	A number of learners at a school	20
6	Number of characters in profile name	The maximum number of characters that can be used to specify a profile name	Characters	16 or more	16 or more	As many characters as the LCD can show on 1 line	16
7	Number of lines on character display	How much information can be displayed	Lines	2 or more	6 or more	As much information as possible	2

#	Specification Name	Metric Description	Unit	Required Value	Target Value	Reasoning	Delivered Value
8	Wireless range – learner joystick	Learner joystick operational at least this many feet away from the receiver. No maximum is specified because if the therapist remote goes out of range the learner joystick will not be operational.	Feet	25 or more	50 or more	Indoor distances	23
9	Wireless range – therapist remote	Therapist remote operational at least this many feet away from the receiver, but at some point will go out of range.	Feet	25 to 200	50 to 100	Indoor distances	59

#	Specification Name	Metric Description	Unit	Required Value	Target Value	Reasoning	Delivered Value
10	Time to stop when therapist remote out of range	When the therapist remote goes out of wireless range (indicated by the light on the remote turning on) how much time is allowed to pass before the Power Wheelchair Trainer halts operation and stops the motors.	Seconds	1 or less	0.75 or less	Safety factor	0.824
11	Wireless Emergency stop	Number of emergency stop switches on the therapist remote	Count	1 or more	1 or more	Safety factor	1

#	Specification Name	Metric Description	Unit	Required Value	Target Value	Reasoning	Delivered Value
12	Time to stop from low speed when remote emergency stop pushed	When the emergency stop switch on the therapist remote is pushed, how much time is allowed to pass before the Power Wheelchair Trainer halts operation, stops the motors, and comes to a complete stop from from a forward speed of 35/125	Seconds	0.75 or less	0.5 or less	Safety factor	0.198
13	Emergency Stop	Number of hardwired emergency stop switches on the controller	Count	2 or more	3 or more	Safety factor	2

#	Specification Name	Metric Description	Unit	Required Value	Target Value	Reasoning	Delivered Value
14	Time to stop from low speed when hard wired panel emergency stop pushed	When one of the hard wired emergency stop switches on the back of the Power Wheelchair Trainer is pushed, how much time is allowed to pass before the Power Wheelchair Trainer halts operation, stops the motors, and comes to a complete stop from a forward speed of 35/125	Seconds	0.75 or less	0.5 or less	Safety factor	0.198
15	Learner Switch Drive Control	The learner joystick shall have four 3.5mm mono jacks for use with input switches.	Count	4 exact	4 exact	One for each of the directions	4
16	Platform raising and lowering	Time to raise or lower the platform	Seconds	3 to 45	3 to 20	A pleasant speed to travel 2 inches	3.8

#	Specification Name	Metric Description	Unit	Required Value	Target Value	Reasoning	Delivered Value
17	Maximum current supplied to motors	An upper limit on the continuous current supplied to the motors for protection	Amperes	30 or less	30 or less	Motor protection	25
18	Charging time	The time required to fully charge the main batteries of the Power Wheelchair Trainer from empty	Hours	8 or less	4 or less	Overnight charging	4.1
19	Voltage 3.3V rail	Main controller board	Volts	$3.3V \pm 0.25V$	$3.3V \pm 0.1V$	Operating voltage of microcontroller	3.29
20	Voltage 5V rail	Main controller board	Volts	$5V \pm 0.45V$	$5V \pm 0.15V$	Operating voltage of LCD screen	5.01
21	Voltage 5V rail	Learner joystick	Volts	$5V \pm 0.2V$	$5V \pm 0.1V$	Operating voltage of inductive joystick	4.99
22	Voltage 2.7V rail	Learner joystick and therapist remote	Volts	$2.7V \pm 0.1$	$2.7V \pm 0.05V$	Operating voltage of microcontroller	2.71
23	Linear actuator voltage – raising	Value will be the charge level of the first battery in series	Volts	$12V \pm 3V$	$12V \pm 2V$	Operating voltage of linear actuators	13.38

#	Specification Name	Metric Description	Unit	Required Value	Target Value	Reasoning	Delivered Value
24	Linear actuator voltage - lowering	Value will be the charge level of the second battery in series	Volts	12V ± 3V	12V ± 2V	Operating voltage of linear actuators	12.95
25	Outer Dead Zone	When configured to be off, do not stop driving the motors	Pass/fail	Pass	Pass	Feature defined	Pass
26	Outer Dead Zone	When configured to be on, how far from the center the joystick handle must be to be considered in the outer dead zone	Degrees	15 ± 6	15 ± 3	Joystick handle travel distance	15.6
27	Center Dead Zone	Minimum configurable deflection angle of the dead zone	Degrees	5 or less	0	Joystick handle travel distance	2.9
28	Center Dead Zone	Maximum configurable deflection angle of the dead zone	Degrees	15 ± 6	15 ± 3	Joystick handle travel distance	19.4

Table 4: Control System Specifications

V.2 Functional Diagram

A functional diagram of the Power Wheelchair Trainer control system is shown in Figure 24. This block diagram depicts the inputs and outputs of the control system hardware, and the arrows between

the blocks show the direction of flow of information. The abbreviations used in the figure include liquid crystal display (LCD) and light emitting diode (LED).

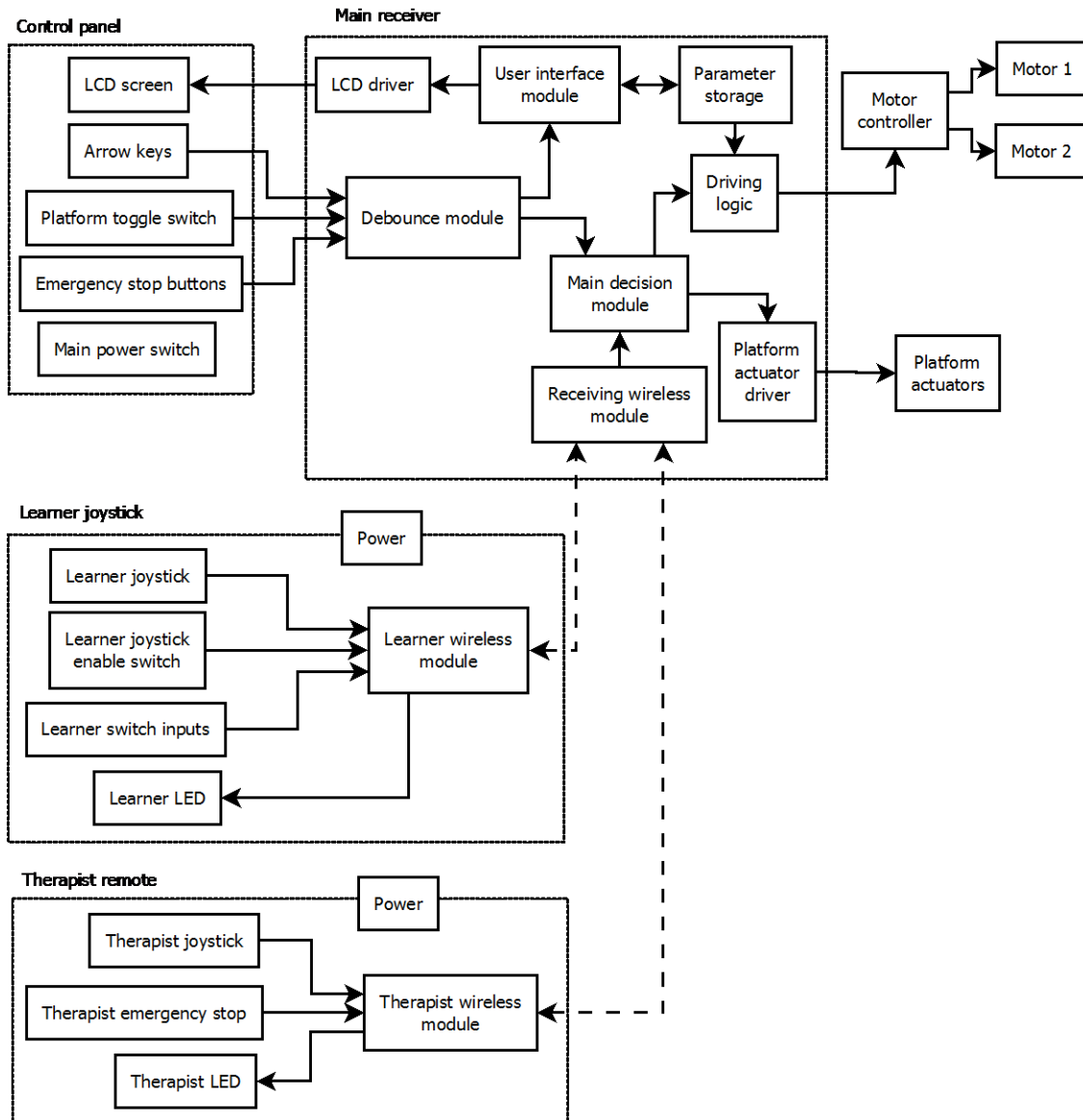


Figure 24: Functional Block Diagram of the Control System

The major components of the control system in version 5 include the therapist remote discussed in section IV.2.3, the learner joystick discussed in section, IV.2.2, IV.5.2, the main control panel, and remaining control system components. Each block of the function diagram in Figure 24 is described in the following sub-sections.

V.3 Main Receiver Circuit Board

There are two separate custom circuit boards in the system, (1) the main receiver on the Power Wheelchair Trainer that handles the drive logic, user interface, and linear actuator control, and (2) the wireless remote board used for both the learner joystick and the therapist remote (described in section V.4). This section describes the main receiver circuit board. Many components were carried over from the version 4 design, and new components were chosen to meet the requirements and specifications. The parts of the main receiver schematic (see Appendix C) that were reused as-is from version 4 include the linear actuator control, voltage regulators, microcontroller, and wireless module. The main receiver schematic was modified to add support for a liquid crystal display (LCD), user interface buttons, and motor driver interface.

The major components for the main receiver are summarized in Table 5. The complete bill of materials can be found in Appendix F.

Part	Description	Legacy or New?
Motor Controller	Sabertooth 2x60	New
Microcontroller	Atmel ATxmega64A1	Legacy
Wireless Transceiver	Nordic nRF24L01+	Legacy
LCD	Newhaven Display NHD-0216K1Z-NSW-BBW-L	New
Motors	30A DC Brushed motor	Legacy
Batteries	12V Lead Acid	Legacy
Linear Actuators	Hiwin LAS-1-1-50-12G	Legacy
Charger	Standard power wheelchair charger	Legacy

Table 5: Major Components – Receiver

All legacy components are acceptable since they have been proven in the version 4 design. The microcontroller is acceptable because it has plenty of I/O pins and enough processing power for the task of filtering joystick signals. Current draw of the microcontroller is negligible given two 12V gel lead acid

batteries. Reduced development time was a large contributing factor in choosing to keep many of the legacy components in addition to choosing new components.

V.3.1 Motor Controller

A new motor controller needed to be chosen that was capable of driving two industry-standard 30 ampere DC brushed motors commonly used in power wheelchairs and used for the Power Wheelchair Trainer. The Dimension Engineering (Akron, Ohio) Sabertooth 2x60 motor driver is capable of supplying up to 60 amperes on two channels, for a total of 120 amperes, and takes input voltages between 6V and 30V (Dimension Engineering, 2011). The Power Wheelchair Trainer uses two gel lead acid batteries connected in series for a nominal operating voltage of 24 volts. The Sabertooth 2x60 is also lower cost than the previous Omni+ system.

The motor controller consists of three parts: a software module, a voltage level shifter, and the Sabertooth 2x60. All parts are new in version 5 of the Power Wheelchair Trainer. The software module is described in section V.6.6. The microcontroller interfaces with the Sabertooth 2x60 motor controller using two input terminals, S1 and S2. The microcontroller operates at 3.3V, and the motor controller's interface uses 5V signaling, so a voltage level shifter was utilized to convert the voltage levels. The voltage level shifter schematic is shown in Figure 25.

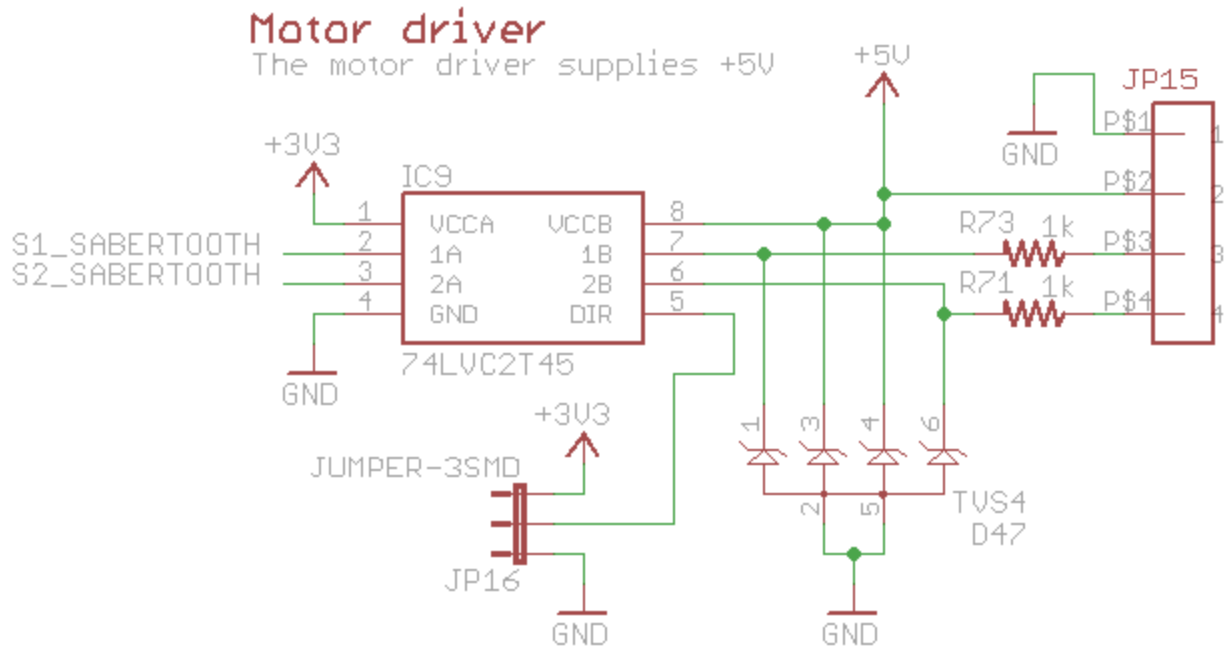


Figure 25: Motor Controller Level Shifter Schematic Excerpt

The interface pinout for the motor controller is shown in Table 6.

Pin	Signal Name	Signal Direction	Description
1	GND		Ground
2	5V	Sabertooth 2x60 → main board	5 volts supplied by the motor controller (up to 1 ampere)
3	S1	Microcontroller → Sabertooth 2x60	TTL level RS-232 serial 9600 baud 8-N-1
4	S2	Microcontroller → Sabertooth 2x60	Active-low emergency stop

Table 6: Motor Controller Interface

The S1 signal is an RS-232 compatible serial line with 9600 baud, 8 data bits, no parity, and 1 stop bit. The Sabertooth 2x60 converts speed and direction commands into right motor speed and left motor speed and provides the high current needed to make the motors run. The Sabertooth 2x60 also has a 5V voltage regulator that is used to supply the 5V rail on the main receiver.

The Sabertooth 2x60 is powered directly from the batteries. Each motor is connected to the Sabertooth 2x60 through a 25 ampere resettable fuse (TE Connectivity part # BD280-1927-25/16) for overcurrent protection. See Table 7 for a description of the battery and motor terminals.

Terminal Name	Description
B+	24V
B-	0V GND
M1A	Motor 1 positive terminal
M1B	Motor 1 negative terminal
M2A	Motor 2 positive terminal
M2B	Motor 2 negative terminal

Table 7: Motor Controller Power Terminals

V.3.2 Platform Actuators

The platform actuator module consists of both a legacy hardware module and software module. Four platform actuators take care of lowering and raising the main platform of the power wheelchair trainer. The linear actuators have built-in limit switches to prevent the platform from being raised or lowered too far. When the platform down switch is pressed on the control panel, a stored parameter is set to indicate to the driving module to stop driving. When the platform up switch is pressed the stored parameter is cleared to indicate to the driving module that it is safe to drive.

The linear actuators run on $\pm 12V$ and draw up to 2 amperes. To lower the platform, +12V nominal is applied, and to raise the platform up -12V is applied. The linear actuators are driven using MOSFETs in a half-bridge configuration. Current is drawn from one battery while lowering the platform, and from the other battery while raising the platform. A half-bridge MOSFET driver drives the MOSFETs.

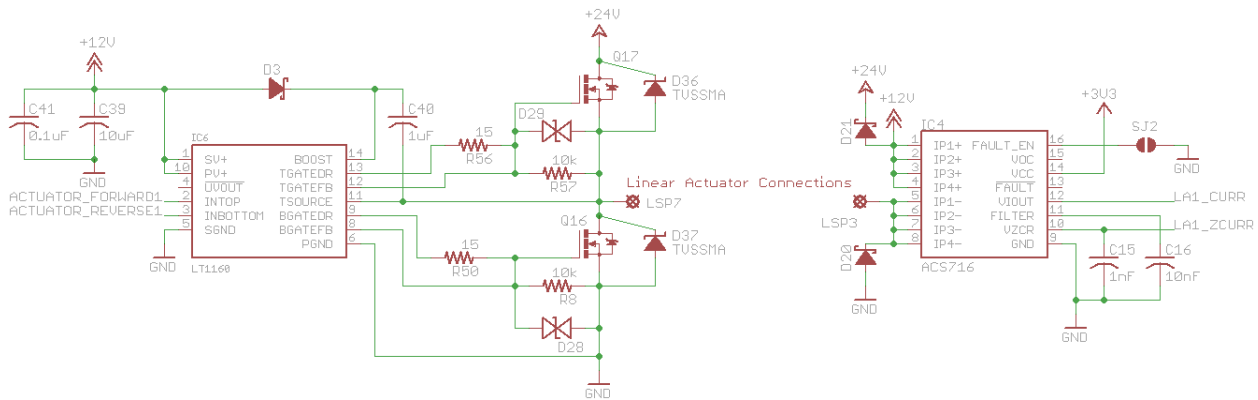


Figure 26: Linear Actuator Driver Schematic Excerpt

The current sensor design was carried over from the version 4 design and is unused in the version 5 design. An analog voltage reference (see Figure 43) is also unused since the main board is consequently not doing any analog to digital conversions.

V.3.3 Control Panel

The control panel is a group of user interface components consisting of an LCD screen, menu navigation arrow keys, platform raise/lower switch, power switch, and a pair of emergency stop switches. The control panel elements are pictured in Figure 27 and Figure 28. The control panel is located on the top face of the rear power unit, and the emergency stop faces are located on the rear face of the rear power unit next to the rear wheels.



Figure 27: Control Panel



Figure 28: Emergency Stop Switches

The following sub-sections describe the components of the control panel.

V.3.3.1 LCD Screen

The liquid crystal display (LCD) is a hardware module that can display 2 rows of 16 characters. The LCD receives input from the LCD driver software module. The LCD is manufactured by Newhaven Display (Elgin, Illinois) and the part number is NHD-0216K1Z-NSW-BBW-L. This LCD display was chosen because it can display two lines with 16 alpha-numeric characters on each line, and because of its low cost. Time constraint made it infeasible to upgrade to a 6-line LCD. The LCD operates on 5 volts and requires 5 volt signaling, so level shifters are necessary to convert the 3.3 volt signals from the microcontroller to 5 volts. The data interface is a 4-bit parallel bus with three control lines. The schematic for the voltage level shifters are shown in Figure 29.

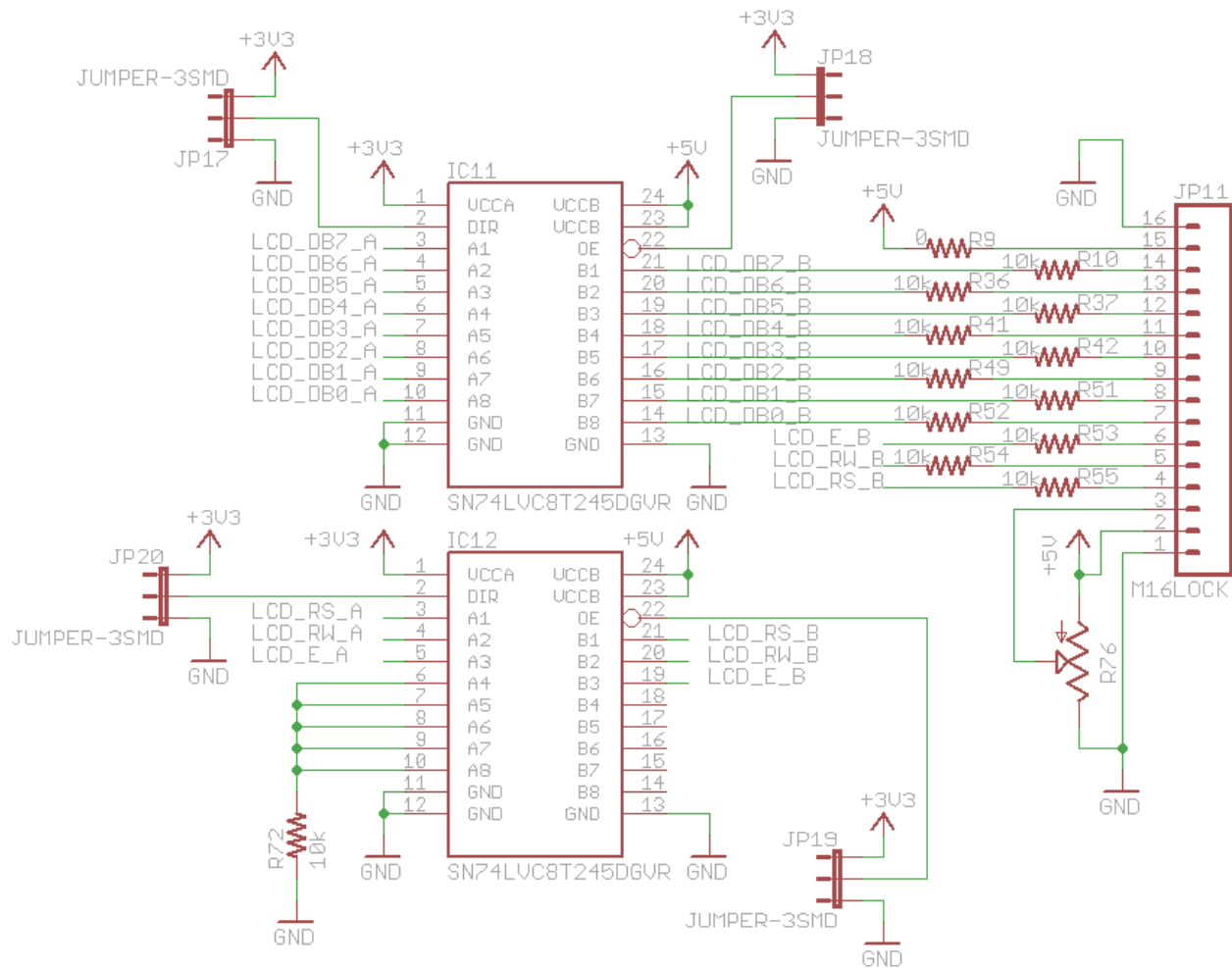


Figure 29: LCD Interface Schematic Excerpt

The interface pin description is shown in Table 8.

Pin Name	Signal Direction	Description
RS	Microcontroller → LCD	Register select
RW	Microcontroller → LCD	Read or write (always set to write)
E	Microcontroller → LCD	Data latch
DB7:DB4	Microcontroller → LCD	4-bit parallel data bus
DB3:DB0		Unused

Table 8: LCD Interface Pin Descriptions

Further LCD interface details are in the LCD software module section V.6.7.

V.3.3.2 Control Panel Arrow Keys

There are four arrow keys on the control panel: up, down, left, and right. The arrow keys are momentary pushbuttons connected to the microcontroller. External pull-up resistors are used on the signal lines. When pushed, the microcontroller pin is shorted to ground for active-low switch activation.

V.3.3.3 Platform Switch

A momentary rocker switch on the control panel provides control of the linear actuators. Lowering and rising of the platform is performed by the linear actuators. External pull-up resistors are used on the signal lines. When pushed, the microcontroller pin is shorted to ground for active-low switch activation. When the platform down switch on the control panel has been pressed, an internal variable is set to indicate to the main decision module to halt operation of the motors.

V.3.3.4 Power Switch

A simple switch powers the control system on or off. The switch is a double-pole, single-throw configuration which enables it to switch both 12V and 24V simultaneously. Since the current requirements of the motor controller are so high, the power switch only toggles power to the main board, not the motor controller. The motor controller remains on when the system is powered down, but without the microcontroller commands to the motor controller the serial link times out and the motors stop.

V.3.4 Emergency Stop Switches

The hardwired emergency stop switches utilize normally closed circuitry, meaning that any unexpected wiring failure will also trigger an emergency stop condition. In other words, the switch is set up in a “push-to-break” fashion. The emergency stop signal line also uses external pull-up resistors, and as a consequence of the normally closed switch, the emergency stop signal is active-high. The emergency stop switches are comprised of Apem (Haverhill, Massachusetts) part numbers A0150B and

A01ESSP3 (see Appendix F). The emergency stop input to the microcontroller is sampled in a polled fashion (see Figure 32), and when an emergency stop condition is detected an emergency stop subroutine is called. The emergency stop subroutine signals to the motor controller to shut down immediately and cut power to the motors. The emergency stop routine prevents any further operation until a power cycle is performed.

V.3.5 Receiving Wireless Module

The main receiver circuit board has a Nordic nRF24L01+ wireless transceiver. Wireless is described further in section V.5.

V.4 Remote Circuit Boards

The learner joystick and therapist remote use the same circuit board but not all components are populated on each board, and the two boards are programmed with different software. The parts of the wireless remote schematic (see Appendix C) that were reused as-is from the version 4 design are the microcontroller and wireless interface. The wireless remote schematic was modified to handle a resistive joystick or inductive joystick as input, and was additionally upgraded with on-board voltage regulators. The major components for the remote circuit boards are summarized in Table 9. The complete bill of materials can be found in Appendix F.

Part	Description	Legacy or New?	Notes
Microcontroller	ATtiny461	Legacy	
Wireless Transceiver	Nordic nRF24L01+	Legacy	
Voltage regulator 2.7V	Texas Instruments TPS61200DRCT	New	
Voltage regulator 5V	Texas Instruments TPS61202DSCR	New	Learner joystick only
Inductive Joystick	Invacare MK IV	New	Learner joystick only

Table 9: Major Components – Remote Controls

The microcontroller and wireless module were chosen because they have been proven in the version 4 design. The voltage regulators were chosen because of their capability to operate on a wide range of input voltages since the battery topography was not known at the time of the design. The voltage regulators automatically operate in either a switching boost regulator mode or linear regulator mode, and are able to handle low input voltages. The inductive joystick was chosen because it is the industry standard method of power wheelchair control.

V.4.1 Power Switch

A simple rocker switch powers the remote circuit board on or off. Power is provided by two AA batteries connected in series.

V.4.2 LED

A red LED turns on when there is no wireless connection to the receiver. The LED is connected in series with a current limiting resistor and is powered by a microcontroller pin.

V.4.3 Transmitting Wireless Modules

Each remote circuit board has a Nordic nRF24L01+ wireless transceiver. Wireless is described further in section V.5.

V.4.4 Learner Joystick

An industry-standard inductive hand joystick is used. The inductive joystick was taken from an Invacare MKIV controller. The joystick has a notch indicating the reverse direction as shown in Figure 30.

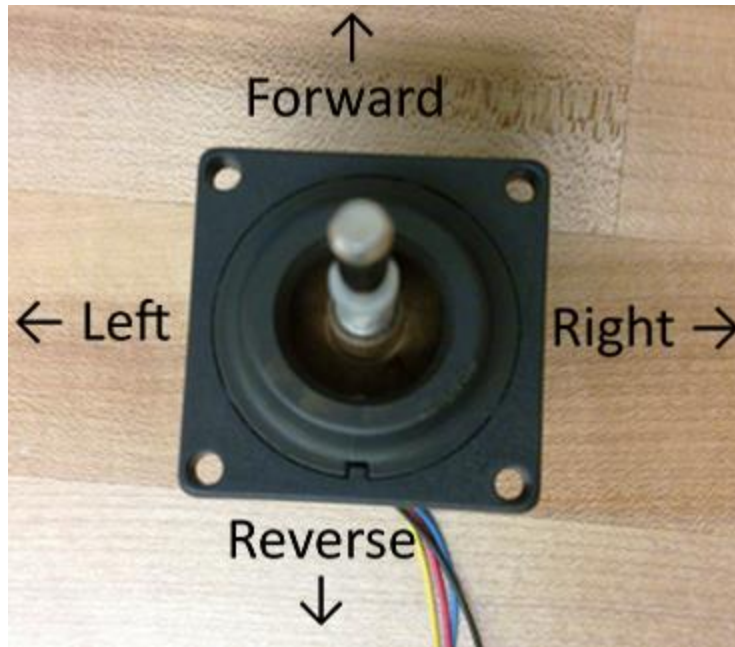


Figure 30: Invacare MKIV Inductive Joystick

The electrical interface of the inductive joystick was determined by the author, and results are shown in Table 10.

Wire Color	Signal Description	Signal Direction
Red	+5V supply voltage	Voltage regulator → joystick
Black	0V ground	
Yellow	Y-axis, Speed	Joystick → microcontroller
Blue	X-axis, Direction	Joystick → microcontroller

Table 10: Invacare MKIV Inductive Joystick Interface

Additional electrical characteristics of the inductive joystick were measured by the author and are shown in Table 11.

Measurement	Value
Start-up delay	150 microseconds
Current draw	9 milliamps

Table 11: Invacare MKIV Inductive Joystick Electrical Characteristics

The 5V supply for the inductive joystick is provided by a boost regulator. When the joystick is centered, the speed and direction outputs are nominally 2.5 volts. The voltage on the yellow wire increases as the joystick handle is pushed forward, and the voltage on the blue wire increases as the joystick handle is pushed to the right. These speed and direction signals are above the operating voltage of the microcontroller, so they are routed through voltage divider resistors (see Figure 40). The output of each voltage divider goes to a microcontroller pin capable of performing analog to digital conversions.

V.4.4.1 Learner Joystick Enable Switch

The joystick enable switch enables or disables the joystick, which is useful for using it purely for switch inputs. The switch is connected to a microcontroller pin with an external pull-up resistor, and when the switch is activated the pin is shorted to ground to indicate switch activation.

V.4.4.2 Learner Switch Inputs

The learner joystick provides 4 switch inputs for switch drive control. The switch inputs are for use with external momentary switches with a 3.5mm mono barrel jack connector. The switch inputs are enabled anytime the learner joystick is powered on. The learner joystick takes priority over the switch inputs, except when the learner joystick enable switch is in the off position. Each switch is connected to a separate microcontroller pin with an external pull-up resistor.

V.4.5 Therapist Remote

The 5V voltage regulator is not populated on the therapist remote circuit board since it does not need to power an inductive joystick.

V.4.5.1 Therapist Joystick

A thumb joystick overrides the learner joystick. The therapist remote has its own special driving profile with the full range of customizable settings. The special profile comes after the last learner driving profile on the Power Wheelchair Trainer. The thumb joystick uses two potentiometers to

measure the joystick position. The potentiometers are powered from the same operating voltage as the microcontroller, and outputs from the potentiometers are routed directly to microcontroller pins capable of performing analog to digital conversions.

V.4.5.2 Therapist Emergency Stop

The emergency stop switch on the therapist remote is a red momentary pushbutton. The microcontroller's internal pull-up resistor is used on the emergency stop signal line. When the switch is activated, the microcontroller pin is shorted to ground for active-low switch activation and the microcontroller signals the main decision module to call the emergency stop routine.

V.5 Wireless Hardware

The wireless module is legacy, carried over from version 4 of the Power Wheelchair Trainer. Wireless functionality for the Power Wheelchair Trainer and peripherals is provided by 3 Nordic nRF24L01+ transceivers. One of these is set up as a receiver on the main receiver circuit board, and the other 2 are set up as transmitters in the learner joystick and therapist remote. The wireless IC operates in the 2.4GHz band, with a specific RF channel frequency of 2.524GHz. This channel frequency is higher than common 802.11 frequencies, reducing the possibility of interference with Wi-Fi signals.

The schematic and board layout for the wireless IC, including the antenna trace (see Figure 31), was carried over from the proven version 4 design, which in turn was based on the Sparkfun Nordic FOB example project (Nordic, 2009).



Figure 31: Nordic nRF24L01+ PCB Antenna Trace

The wireless IC uses serial peripheral interface (SPI) to communicate to microcontroller. A pin diagram for the SPI interface is shown in Table 12.

Pin Name	Description	Direction
CE	Chip enable	Microcontroller → wireless IC
CSN	SPI Chip select	Microcontroller → wireless IC
SCK	SPI Clock	Microcontroller → wireless IC
MOSI	SPI Master out, slave in	Microcontroller → wireless IC
MISO	SPI Master in, slave out	Wireless IC → microcontroller
IRQ	Interrupt request	Wireless IC → microcontroller

Table 12: Wireless IC Interface Description

Additional details about wireless can be found in section V.5.

V.6 Software

The software architecture and modules are described in the following sub-sections.

V.6.1 Software Architecture and Main Decision Module

The main decision module initializes the system, determines what state of the system is in, and controls the overall flow of information. The overall software architecture consists of a main loop and interrupt-driven routines. Initialization routines are run before entering the main loop. During the initialization process, the initialization routines for the following modules are run:

- Driving logic
- Motor controller
- Debounce module
- Wireless module
- LCD driver

These initialization routines are discussed in their respective sections. Additionally, general microcontroller input/output (I/O) pin directions are initialized. After initialization, the main decision module enters the main loop. A software decision flowchart describing the polled portion of the software is shown on the next page in Figure 32, which includes a high-level overview of the main software loop for the learner joystick, therapist remote, and wheelchair.

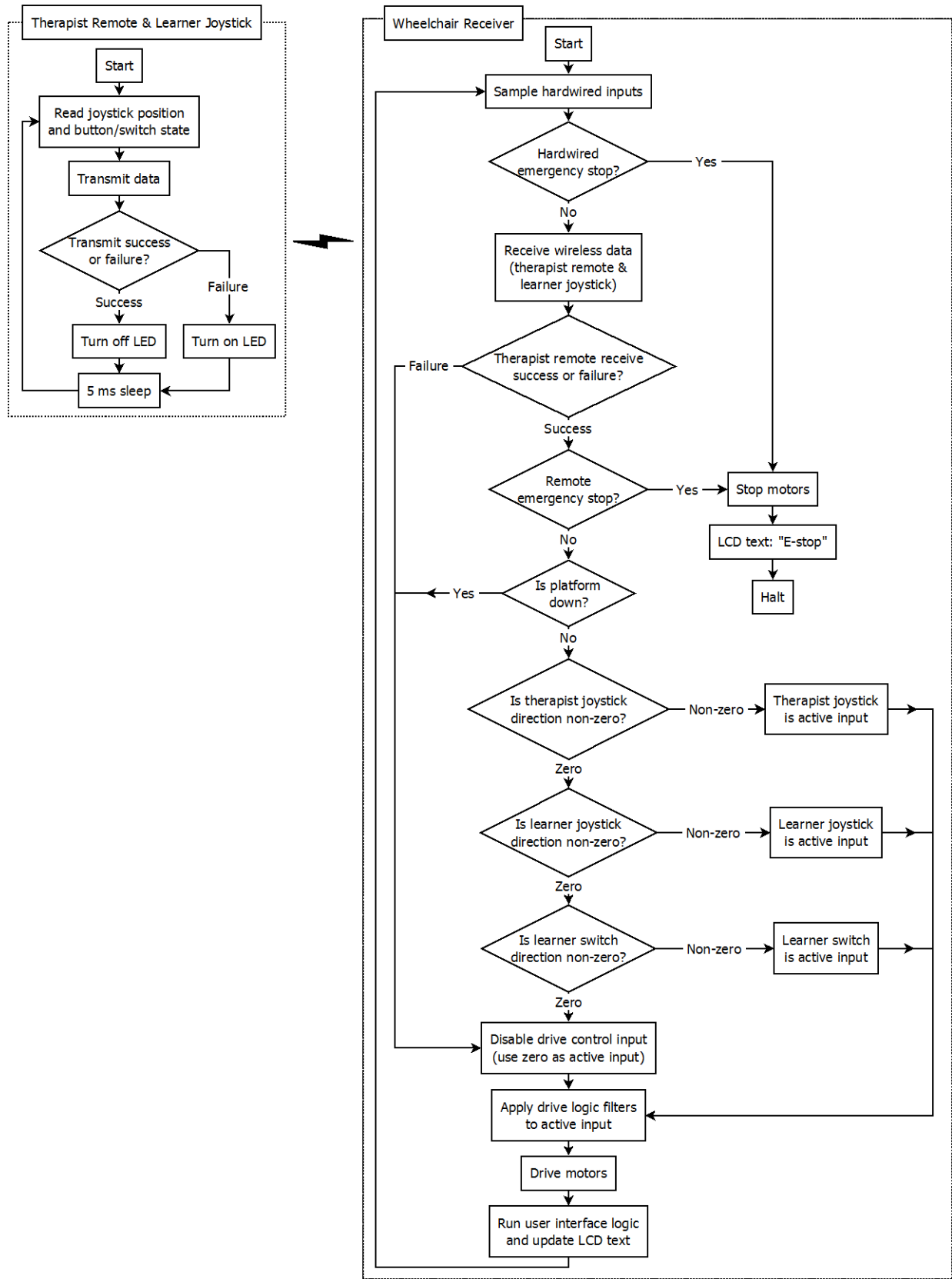


Figure 32: Software Decision Flowchart

Interrupt-driven portions and polled subroutines of the software are specific to certain modules, and are explained in their appropriate sub-section.

V.6.2 Emergency Stop

When the main decision module detects an emergency stop condition, an emergency stop subroutine is called. The emergency stop subroutine signals to the motor controller to shut down immediately and cut power to the motors. The emergency stop subroutine also signals to the user interface module to display an emergency stop message on the LCD screen. The emergency stop routine prevents any further operation until a power cycle is performed.

V.6.3 Wireless Protocol

Wireless functionality for the Power Wheelchair Trainer and peripherals is provided by 3 Nordic nRF24L01+ transceivers. As mentioned previously, one of these is set up as a receiver in the power unit, and the other 2 are set up as transmitters in the learner joystick and therapist remote. The software initializes the wireless IC and ensures reliable communication between the learner joystick, therapist remote, and main receiver. The nRF24L01+ supports up to 6 data pipes, or data streams. For the Power Wheelchair Trainer, 2 data pipes are as a way to distinguish between the learner joystick and therapist remote. To ensure reliable data transmission, automatic acknowledge packets (ACK), cyclic redundancy checksum (CRC) with two CRC bytes, and retry logic have been set up as recommended by the nRF24L01+ data sheet (Nordic Semiconductor, 2008). If the receiver receives a packet and successfully verifies the CRC, an ACK is sent as a reply. In the event of a CRC error, the receiver responds with a not acknowledged (NACK) packet, and the transmitter retries the transmission. The transmitters will continually retry transmissions until an ACK packet is received. When the 2 transmitters are transmitting at the same time they effectively block each other, however, retransmit delays are different for the 2 transmitters to prevent blocking more than once. If the therapist remote does not receive an

ACK packet, it retries up to 5 times with a delay of 500 microseconds until an ACK packet is received. If the learner joystick does not receive an ACK packet, it retries once with a delay of 2500 microseconds until an ACK packet is received. The 2 transmitters are differentiated at the receiver by having different data pipe addresses. The ACK, CRC, retransmit, and data pipe address functionality are all provided by the nRF24L01+ wireless transceiver. The nRF24L01+ can operate on a selection of channels within the 2.4GHz band. The chosen RF channel frequency in this application is 2.524GHz.

Two timers are set up on the main receiver for time-out purposes, behaving in a similar fashion to a watchdog timer with a time-out period 0.25 seconds. Two timers are required since it is necessary to determine the integrity of both the therapist remote data pipe and the learner joystick data pipe. Upon successful receipt of a data packet from the therapist remote or learner joystick, the corresponding timer is reset. If the timer is allowed to run out, then the joystick variables associated with the device are zeroed in the timer's interrupt service routine (ISR). In other words, if a wireless packet has not been received at the wheelchair in the last 0.25 seconds, or if the transmitter has not received an ACK packet in the last 0.25 seconds, the wireless connection is assumed to be lost due to the transmitter and receiver being out of range or turned off, and in this case the joystick variables are zeroed.

The receiving software module operates in an interrupt-driven manner. The receive interrupt routine is based on the flow diagram shown in Figure 33.

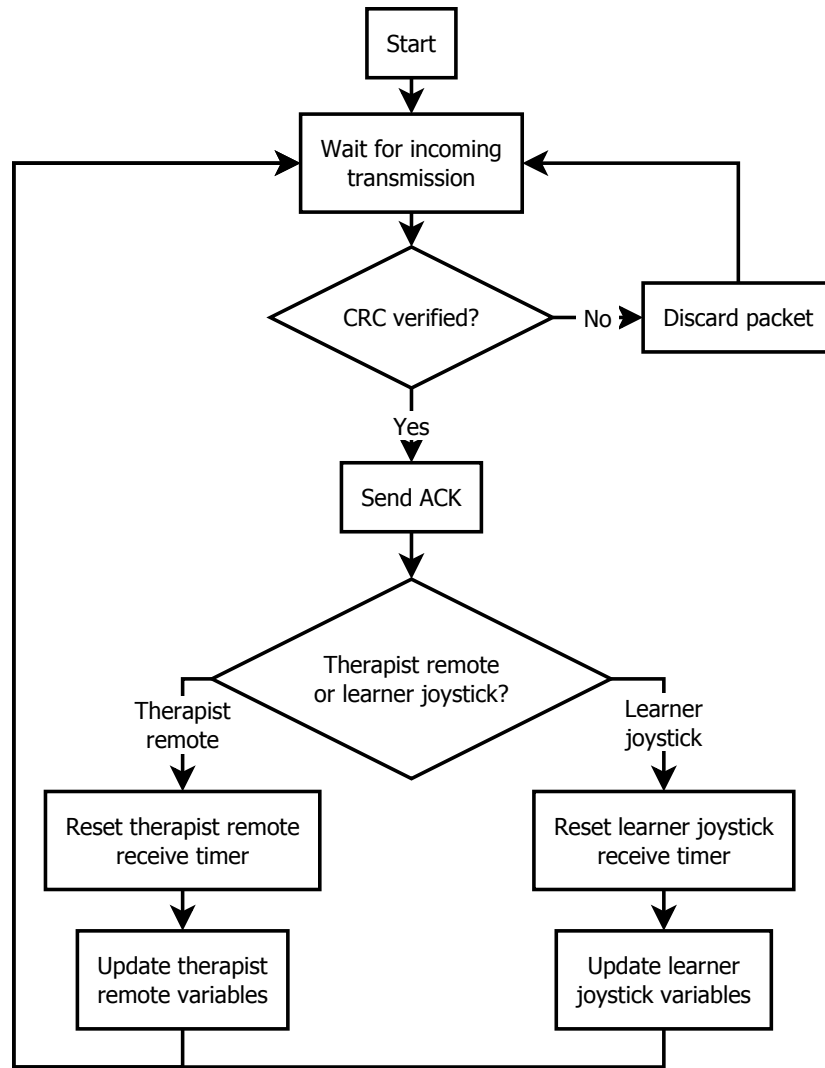


Figure 33: Wireless Receiving Flowchart

Timers are also set up at the transmitting end. If an ACK has not been registered in the last 0.25 seconds, the transmitting device turns on an LED to indicate a lost wireless connection.

Data is transmitted in a packet format consisting of a preamble, pipe address, packet control field, payload, and CRC bytes. The payload is the only field specified by the user, and the payload format is described in Table 13.

Byte 0	Switch joystick and emergency stop – see Table 14
Byte 1	Direction, X, horizontal axis, right/left
Byte 2	Speed, Y, vertical axis, forward/reverse

Byte 3	Reserved, set to 0x00
--------	-----------------------

Table 13: Payload Contents

A description of the bit meanings of byte 0 can be seen in Table 14 (tables are not necessary to describe the other bytes). Bytes 1 and 2 are signed 8-bit integers representing the joystick position in the speed and direction notation.

Bit	7	6	5	4	3	2	1	0
Meaning	<i>Reserved</i>	Right	Left	Reverse	Forward	<i>Reserved</i>	<i>Reserved</i>	Emergency stop

Table 14: Payload Byte 0

Bits 3-6 represent the state of the switch inputs and are valid only for the learner joystick, though the same packet format is used for both the learner joystick and therapist remote. Bit 0 represents the state of the emergency stop switch on the therapist remote. A logical '1' means that the switch is activated and a logical '0' means that the switch is inactivated.

The nRF24L01+ has selectable data rates of 250 kilobits per second (kbps), 1 megabit per second (Mbps), or 2 Mbps. 250 kbps is chosen since that data rate allows for lower receive sensitivity and has farther range. A transmit period must be chosen such that the available data bandwidth is not saturated and can allow for re-transmits to occur. The length of a packet is given by: 8 preamble bits + 40 address bits + 9 packet control field bits + 32 payload bits + 16 CRC bits = 105 bits total. An ACK packet has a zero payload, so an ACK packet amounts to 73 bits. Under ideal conditions, one transmission utilizes 178 bits. Since there are two transmitters, two transmissions need to be counted, so 356 bits. The theoretical maximum packet transmission period is 1.424 milliseconds, but to allow for ACK delays, processing delays, and re-transmits, and packet transmission period of 5 milliseconds was chosen.

V.6.4 User Interface Module

The user interface allows the user to view and change stored parameters. The arrow key inputs are debounced in the debounce module (section V.6.8), and the debounce module ensures that a press

of an arrowkey only returns true once for a call to the function since the user interface module is a polled software module. There are 3 non-volatile variables are needed for the current menu state:

- Active profile: indicates the currently selected profile.
- Active setting: indicates the active programmable setting.
- Platform down: indicates whether the platform is currently lowered or raised.

One additional variable is stored in random access memory (RAM):

- Name edit mode: indicates whether the user interface is in name edit mode.

The current setting and value is displayed on line 1 of the LCD screen. The current profile name is always displayed on line 2 of the LCD screen. The user interface also allows for editing profile names.

A flow diagram modeling the user interface subroutine is shown in Figure 34.

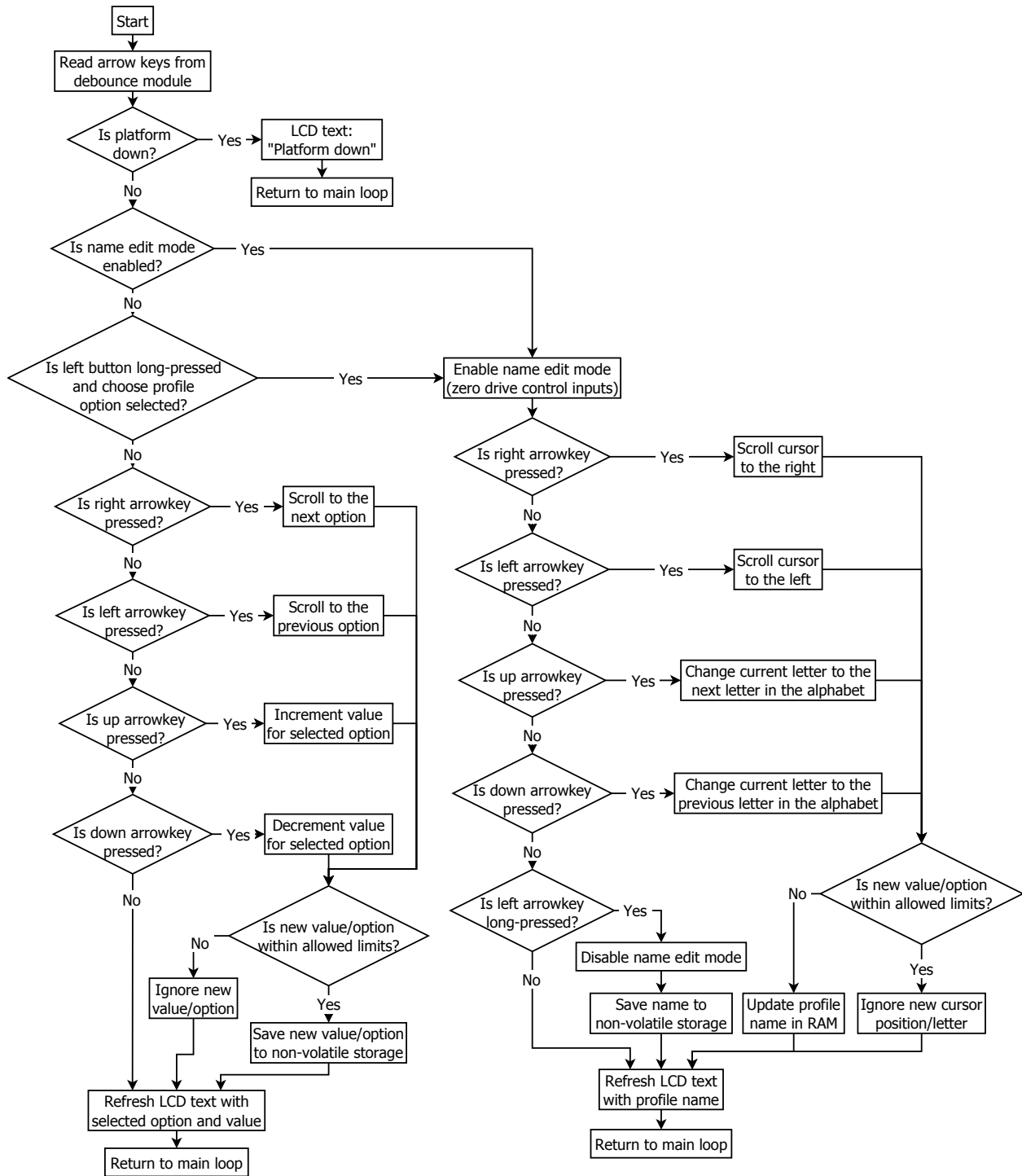


Figure 34: User Interface Flowchart

V.6.5 Driving Logic and Stored Parameters

The driving logic is run in a polled fashion since it involves lots of mathematical operations and it is undesirable to put heavy-weight code in an interrupt service routine (ISR). If the code gets stuck, the watch dog timer (section V.6.9) will ensure a safe reset.

The driving logic and stored parameter modules work together to prioritize drive inputs and apply filters to joystick movements according to programmable values. Multiple driving profiles can be specified, and each driving profile contains an array of programmable values. Non-volatile storage is utilized so the customized profile settings are not lost when the system is turned off. The non-volatile storage can be categorized into general parameters, and profile-specific parameters. The general parameters in non-volatile storage determine the current menu state of the user interface, and were described in the previous section.

The profile-specific parameters stored in non-volatile storage are listed and described in Table 15. The driving logic module functions in a polled fashion. The initialization sequence initializes a timer for use with sensitivity, acceleration, and deceleration (described in Table 15). The driving logic module continuously reads the following inputs:

- Therapist remote joystick position
- Learner joystick position
- Switch inputs in all four directions

The stored parameters for the active profile are used in the process of translating the joystick position or switch direction into a final driving speed and direction that will get passed on to the motor controller. The driving logic module also takes care of input priority: the learner joystick overrides the switch inputs, and the therapist joystick overrides all. This is also visualized in Figure 32 above. The

joystick position is described by speed and direction. This notation is similar to y and x coordinates in a Cartesian coordinate system. The speed and direction are defined as follows:

- Speed: the forward/reverse axis of the joystick, where zero is center, positive values are forward, and negative values are reverse. Speed is also known as the vertical axis, y-axis, or yellow wire.
- Direction: the right/left axis of the joystick, where zero is center, positive values are right, and negative values are left. Direction is also known as the horizontal axis, x-axis, or blue wire.

Table 15 describes all the stored parameters contained in a driving profile. Each driving profile contains a unique array of stored parameters. With the exception of the profile name, each stored parameter is part of a filter, transform, or limit that is applied to the joystick speed and direction. Each item (except profile name) is applied to the joystick variables in the order it is listed in the table.

Stored Parameter Name	Function
Profile name	16 characters are used to specify the name of the driving profile.

Stored Parameter Name	Function
Outer dead zone	<p>If the outer dead zone stored parameter is zero, the outer dead zone is disabled. Otherwise, given the joystick position in Cartesian coordinates the distance from center is calculated using the Pythagorean theorem, given by equation 5.</p> $d = \sqrt{x^2 + y^2} \quad (5)$ <p>If the distance is greater than an empirically determined preset value, the joystick is considered to be in the outer dead zone and a timer is started.</p> <p>After this, one of the following scenarios occur:</p> <ul style="list-style-type: none"> • If the joystick remains in the outer dead zone for the duration of the time specified by the stored parameter, the motors stop. The joystick must return to center before the motors start again. • If the joystick goes out of the dead zone before the time is up, the timer is stopped and reset.
Invert	<p>If invert is enabled (specified by the stored parameter) then forward and reverse are swapped. This setting does not apply to the switch inputs.</p>
Proportional as switch	<p>When this setting is enabled (specified by the stored parameter) the learner joystick emulates a switch joystick with only 4 possible directions (forward, reverse, left, and right) and no diagonal. The top speed stored parameter determines the speed and direction values. This setting does not apply to the switch inputs.</p>

Stored Parameter Name	Function
Center dead zone	<p>The speed and direction notation is defined in the Cartesian coordinate system. Conversion from Cartesian to polar coordinates is accomplished with equations 6 and 7.</p> $r = \sqrt{x^2 + y^2} \tag{6}$ $\theta = atan2(y, x) \tag{7}$ <p>(x, y) are the direction and speed, respectively, in Cartesian coordinates, and (r, θ) are the radius and angle in polar coordinates. <i>atan2</i> is the C standard arctangent function which takes into account the sign of both arguments in order to determine quadrant (<i>atan2</i>, n.d.).</p> <p>The dead zone is applied by subtracting the stored parameter from the radius. If the resulting radius is negative, it is set to zero (indicating a centered joystick; the joystick is in the dead zone). Finally, the radius and angle are converted back to Cartesian coordinates using equations 8 and 9.</p> $x = r \times \cos(\theta) \tag{8}$ $y = r \times \sin(\theta) \tag{9}$ <p>This setting does not apply to the switch inputs.</p>
Forward, reverse, and turn throw	<p>Joystick throw determines how far to deflect the joystick to attain a certain speed. This is implemented by multiplying the joystick position by the stored parameter. There is a separate stored parameter for forward, reverse, and turn, thus throw is applied separately for forward, reverse, and turn. This setting does not apply to the switch inputs.</p>

Stored Parameter Name	Function
Forward, reverse, and turn maximum speed	If speed or direction is greater than the stored parameter, then set the speed or direction to the stored parameter. Top speed is applied separately for forward, reverse, and turn.
Sensitivity	<p>Sensitivity is implemented as a first order low-pass filter, given by equation 10.</p> $x_n = x_n + \alpha(x_{n-1} - x_n) \tag{10}$ <p>The stored parameter specifies α, the cut-off frequency of the filter. The filter iterates every 1 millisecond, and it is only applied while speed is increasing. The filter is applied to speed and direction in Cartesian coordinates.</p>
Acceleration and Deceleration	The stored parameter specifies how many milliseconds to wait before speed/direction is allowed to increase or decrease by one. If speed/direction is increasing, use the acceleration stored parameter. If decreasing, use the deceleration stored parameter.

Table 15: Stored Parameter Functions

Simple error checking is performed in case of non-volatile storage failure. During initialization, the settings are checked that they are within the range of allowed values. Additionally, every time a value is written to non-volatile storage, the value is read back and compared to the expected value. In the event of a failed check, the message show in Figure 35 is displayed on the LCD. This message indicates that the non-volatile storage is corrupted and the microcontroller should be replaced.

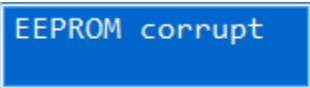


Figure 35: EEPROM Corrupt LCD Message

V.6.6 Motor Controller Communication

The motor controller software module initializes the serial communication peripheral of the microcontroller required for sending speed and direction commands to the hardware motor controller. The motor controller software uses the Universal Serial Interface (USI) peripheral of the microcontroller to output a RS-232 compatible 9600 baud, 8 data bits, no parity, 1 stop bit signal on the S1 pin of the Sabertooth 2x60 motor controller. The microcontroller must continually send speed and direction commands to the Sabertooth 2x60. If the Sabertooth 2x60 has not received a command in 500 milliseconds, a time-out occurs and the motors stop. The S2 pin of the Sabertooth 2x60 is an active-low emergency stop. A motor controller command packet is shown in Table 16.

Byte Number	Description
1	Address
2	Command
3	Data
4	7-bit checksum

Table 16: Motor Controller Packet Description

The Sabertooth 2x60 has a hardcoded address of 128. The command byte tells whether the packet is specifying the speed or the direction. The data byte specifies how fast to drive the motors. The Sabertooth 2x60 converts speed and direction values into left motor and right motor speeds. The checksum byte is calculated as the lower 7 bits of equation 11:

$$checksum = (address + command + data) \tag{11}$$

V.6.7 LCD Driver

The LCD driver is an interrupt-driven software module that converts text for display into electrical signals and commands to interface with the LCD application programming interface (API). The LCD module is new in version 5 of the Power Wheelchair Trainer. The module handles signal timing and

sends data according to the signaling protocol described in detail in the LCD controller application notes (Sitronix, 2006; Sunplus Technology, 2003). The LCD driver software is written in an interrupt-driven manner since it involves waiting periods and it is undesirable to block other more important parts of the software while waiting for an LCD character to be sent.

During the initialization routine, a timer peripheral is set up to trigger an interrupt service routine (ISR) for the LCD driver. To initiate sending a text string to the LCD, the main code calls a `lcdText()` subroutine that copies the text string to local memory for the LCD driver (see `lcd_driver.c` in Appendix D). Then the function sends the start command to the LCD by setting the appropriate data bits and a timer is started. At this point, the function returns and the main code can continue. After the timer reaches a delay value specified by LCD API, the interrupt service routine (ISR) for the LCD driver executes and sends the next command to the LCD, and sets a new timer value for the next state. The variables and registers used as part of the LCD driver and API are explained in Table 17.

Variable	Description
RS	RS: LCD API control variable. Register select.
E	E: LCD API control variable. Clock.
PORT	PORT: The microcontroller pins used to send commands to the LCD.
ADDR	ADDR: The memory address corresponding to the first line of the LCD.
ADDR2	ADDR2: The memory address corresponding to the second line of the LCD.
CurrLine	CurrLine: An LCD driver internal variable used to keep track of which line is being processed.
Line1	Line1: The text to send for line 1.
Line2	Line2: The text to send for line 2.
Delay	Delay: The amount of delay between sending commands.

CharPos	CharPos: An LCD driver internal variable used to keep track of which character is being processed.
---------	--

Table 17: LCD Variables

Figure 36 is a flow diagram of the LCD driver algorithm. In Figure 36, each box contains the commands to be executed during the ISR and the delay timer value until the next box is executed. It starts by sending the address of the first character of the first line of the LCD, and then the subsequent 16 characters are transmitted. Then it sends the address of the second line of the LCD screen, and the remaining 16 characters are transmitted.

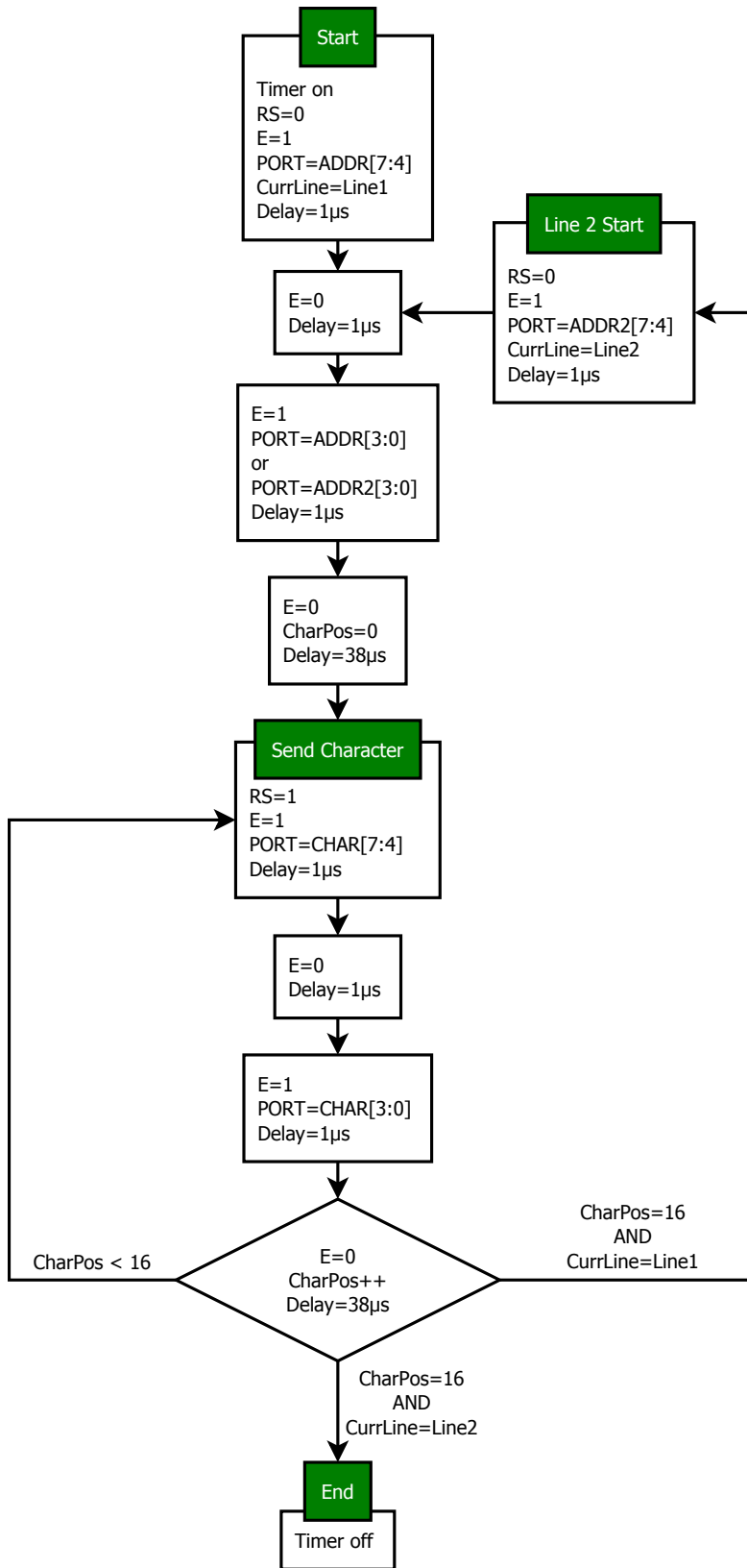


Figure 36: LCD Signaling Flow Diagram

V.6.8 Debounce Module

The debounce module corresponds to the “Sample hardwired inputs” box of the flowchart in Figure 32. The debounce module is an interrupt-driven software module that filters out false keypresses. When a keypress is recorded by a fast microprocessor, a phenomenon known as switch bounce or contact bounce can introduce extraneous unintended keypresses. Switch contacts are usually made of springy metals, and when the contacts strike together their momentum and elasticity cause the contacts to bounce apart. The result is that electrical contact is unstable for a short period of time before steady state is reached. To solve this problem, the debounce module monitors the button input and waits for a period of approximately 40 milliseconds before considering a button as being pushed. The debounce module filters the keypresses from the control panel, including the arrow keys, platform switch, and emergency stop switches. The debounce module was rewritten for version 5 of the Power Wheelchair Trainer to be more modular.

During the initialization routine, a timer peripheral is initialized for debouncing purposes. The debouncing algorithm works by remembering the previous 8 keypress input samples, each measured 5 milliseconds apart. The debounced output is only set or cleared when all 8 previous samples are in agreement. An interrupt service routine (ISR) runs every 5 milliseconds, triggered by the timer peripheral. The pseudo-code for the ISR is listed in Table 18 below. InputValues is an 8-bit unsigned integer.

1	InputValues = (InputValues << 1) (ReadInput() ? 1 : 0);
2	if (InputValues == UINT8_MAX)
3	DebouncedValue = 1;
4	if (InputValues == 0)
5	DebouncedValue = 0;

Table 18: Debounce Code Listing

Additionally, the debounce module detects when a button has been pressed for a longer period of 2 seconds, known as a long-press. The debounce module ensures that a call to an input function only

returns true once per keypress/falling edge to allow proper operation of polled software routines. The first call to an input function during a switch activation returns true, and subsequent calls return false. The full source code can be found in PWCT_io.c in Appendix D.

V.6.9 Watchdog Timer

To ensure software reliability and safety, the microcontroller's watchdog timer peripheral has been enabled with a time-out period of 125 milliseconds. If the watchdog timer is allowed to run out, a microcontroller reset is initiated. The main loop resets the watchdog timer on every iteration.

V.7 Enclosures and Connectors

All electrical components were enclosed in order to reduce the amount of visible wiring. The main receiver circuit board and motor controller were enclosed in a custom enclosure located between the two batteries in the rear power frame of the Power Wheelchair Trainer. The LCD, user interface (UI) buttons, power switch, and linear actuator switch were mounted on the top panel of the enclosure. This concept is illustrated in Figure 37.

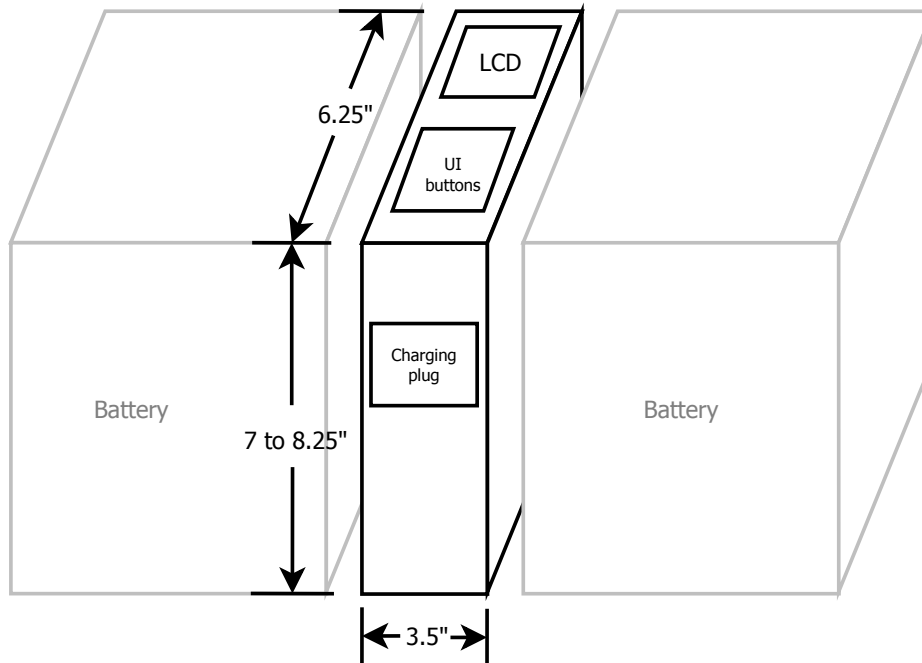


Figure 37: Electrical Enclosure

As few connectors as possible were located on the outside of the electrical enclosure. These connectors are for the batteries, motors, charging, emergency stop switches, and linear actuators. Modular connectors and wiring were chosen to provide sufficient amperage. An overview of the external connectors is shown in Figure 38, with the number of required conductors and current rating shown by the wires. Where current rating is not shown, it can be assumed to be negligible.

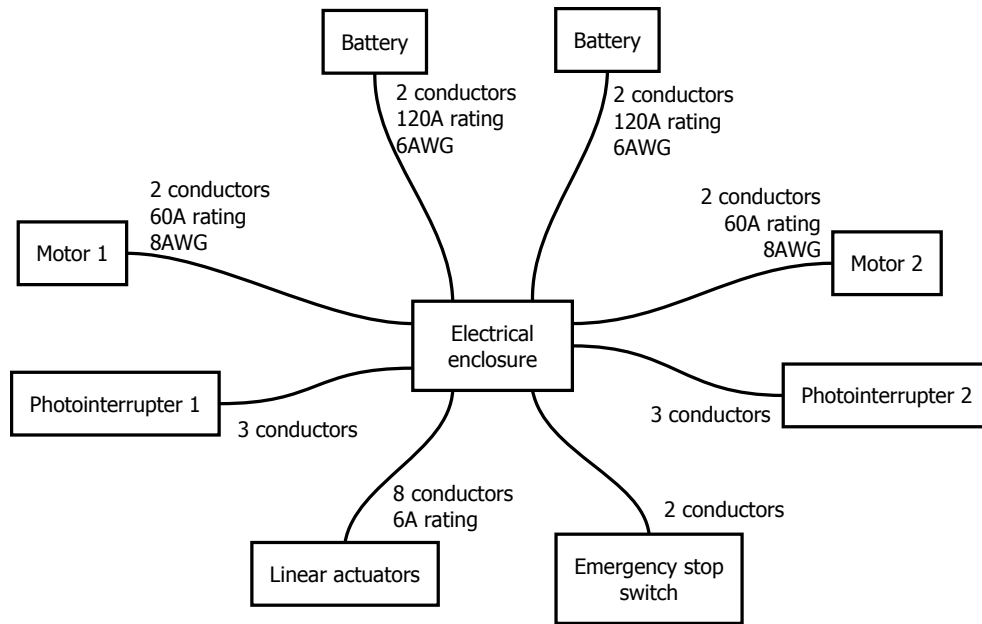


Figure 38: Overview of Connectors

Wire gauges were chosen to safely accommodate amperage ratings. The Sabertooth 2x60 datasheet recommends 8 AWG wire for motor connections (Dimension Engineering, 2011). All connectors are keyed such that they only fit one way in order to avoid a reverse polarity condition. The complete bill of materials for connectors is shown in Appendix F.

V.8 Obstacles Overcome and Lessons Learned

V.8.1 Joystick Shearing/Skewed Axes

For unknown reasons the inductive joystick started having a problem where pushing the joystick straight forward would cause the Power Wheelchair Trainer to turn slightly. The joystick axes could be roughly represented by the skewed axes shown in Figure 39.

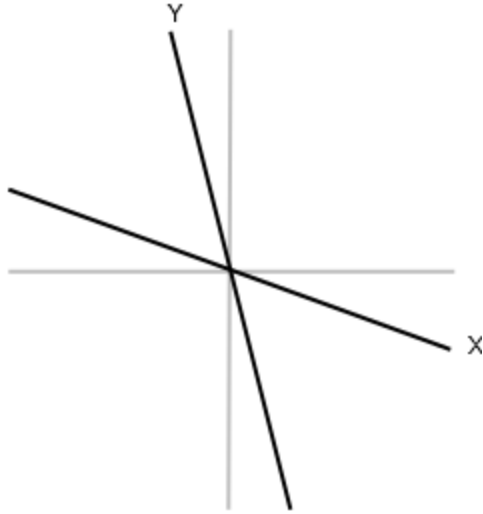


Figure 39: Skewed Axes

The symptom was temporarily compensated for in software using a shear mapping algorithm with empirically determined parameters. However, the joystick skew changed over time and it became obvious that the software fix was not a permanent one. Ultimately the inductive joystick was replaced and the problem has not re-appeared as of this writing.

V.8.2 LCD Goes Blank

Occasionally, the LCD screen would go blank while using the linear actuators to lower/raise the platform. It was determined that this was due to electromagnetic interference (EMI). The MOSFET half-bridge driver requires a PWM signal in order to turn on the top MOSFETs, resulting in high-current switching causing lots of EMI. This EMI was picked up by the LCD wiring, occasionally causing the LCD to go blank. The issue was resolved by routing the LCD wires through a ferrite bead.

VI. Assessment

The Food & Drug Administration (FDA) gives guidance for medical device manufacturers on how to apply verification and validation processes to the design of the medical device (Food & Drug Administration, 1997). These guidelines were taken into account when evaluating the design and performance of the Power Wheelchair Trainer control system.

VI.1 Verification

The purpose of verification is to ensure that the Power Wheelchair Trainer meets all engineering specification and assures the quality of the design. FDA guidelines say that medical device manufacturers should perform verification activities and document them, where the verification activity confirms that the design meets the requirements (Food & Drug Administration, 1997). For the Power Wheelchair Trainer controller design, verification was carried out by testing each of the requirements specified in Table 4 in section V.1 using the final prototype.

VI.1.1 Specification Testing

Verification testing was performed and results recorded in Table 19. The table describes the testing method for each requirement, and the test results were recorded and verified that they meet the acceptable range of values. All engineering specifications from Table 4 in section V.1 were tested. Voltage levels were measured with a Fluke 189 True RMS Multimeter.

#	Procedure	Unit	Required Value	Target Value	Result	Pass /fail
1	From a full charge, prop up the power unit, start driving forward, and start a timer. Stop the timer when the battery voltage reaches 19V.	Hours	3 or more	8 or more	5.25	Pass

#	Procedure	Unit	Required Value	Target Value	Result	Pass /fail
2	Turn off the Power Wheelchair Trainer. From a fresh set of AA batteries, turn on the learner joystick, observe that the red LED is on, and start a timer. Stop the timer when the LED turns off.	Hours	8 or more	10 or more	12+	Pass
3	Turn off the Power Wheelchair Trainer. From a fresh set of AA batteries, turn on the therapist remote, observe that the red LED is on, and start a timer. Stop the timer when the LED turns off.	Hours	8 or more	10 or more	24+	Pass
4	Set "prop as switch" on, drive forward 10 ft, measure veer.	Feet	2 or less	1 or less	1.583 to the left	Pass
5	Count the number of available driving profiles.	Count	20 or more	50 or more	20	Pass
6	Count the number of available characters in a profile name.	Characters	16 or more	16 or more	16	Pass
7	Count the number of lines on the LCD	Lines	2 or more	6 or more	2	Pass
8	Turn on the Power Wheelchair Trainer and the learner joystick. Move the learner joystick away from the Power Wheelchair Trainer until the red LED turns on, and measure the distance.	Feet	25 or more	50 or more	23	Fail
9	Turn on the Power Wheelchair Trainer and the therapist remote. Move the therapist remote away from the Power Wheelchair Trainer until the red LED turns on, and measure the distance.	Feet	25 to 200	50 to 100	59	Pass

#	Procedure	Unit	Required Value	Target Value	Result	Pass /fail
10	Set forward speed to 35, start recording video, drive PWCT forward, turn off remote, measure time from switch turned off to the Power Wheelchair Trainer comes to a complete stop.	Seconds	1 or less	0.75 or less	0.824	Pass
11	Count the number of emergency stop switches on the therapist remote.	Count	1 or more	1 or more	1	Pass
12	Set forward speed to 35, start recording video, drive the Power Wheelchair Trainer forward, push remote emergency stop switches, and measure the time from button press to the Power Wheelchair Trainer comes to a complete stop.	Seconds	0.75 or less	0.5 or less	0.198	Pass
13	Count the number of hardwired emergency stop switches on the Power Wheelchair Trainer.	Count	2 or more	3 or more	2	Pass
14	Set forward speed to 35, start recording video, drive PWCT forward, push hardwired emergency stop switches, measure time from button press to the Power Wheelchair Trainer comes to a complete stop.	Seconds	0.75 or less	0.5 or less	0.198	Pass
15	Count the number of switch input jacks on the learner joystick.	Count	4	4	4	Pass
16	Start with platform up, start recording video, push platform down switch, measure time it takes the platform to lower.	Seconds	3 to 45	3 to 20	3.89	Pass

#	Procedure	Unit	Required Value	Target Value	Result	Pass /fail
16	Start with platform down, start recording video, push platform up switch, measure time it takes the platform to raise.	Seconds	3 to 45	3 to 20	3.759	Pass
17	Observe the presence of a 30A (or less) fuse or resettable fuse on both of the motor connections inside the control unit enclosure.	Amperes	30 or less	30 or less	2 25A fuses	Pass
18	Start with the batteries at 19V, connect the charger, and measure the time until the charger light turns green.	Hours	8 or less	4 or less	4.1	Pass
19	3.3V main	Volts	$3.3V \pm 0.25V$	$3.3V \pm 0.1V$	3.2953	Pass
20	5V main	Volts	$5V \pm 0.45V$	$5V \pm 0.15V$	5.0091	Pass
21	5V learner joystick	Volts	$5V \pm 0.2V$	$5V \pm 0.1V$	4.9984	Pass
22	2.7V therapist remote	Volts	$2.7V \pm 0.1$	$2.7V \pm 0.05V$	2.7124	Pass
22	2.7V learner joystick	Volts	$2.7V \pm 0.1$	$2.7V \pm 0.05V$	2.6926	Pass
23	Linear actuator voltage – raising	Volts	$12V \pm 3V$	$12V \pm 2V$	13.38	Pass
24	Linear actuator voltage – lowering	Volts	$12V \pm 3V$	$12V \pm 2V$	12.95	Pass
25	Set the Outer Dead Zone to be off. Hold the joystick handle all the way to the edge and observe that the Power Wheelchair Trainer keeps driving.	Pass/fail	Pass	Pass	Pass	Pass
26	Set the Outer Dead Zone to be on, and start recording video of the joystick. Start moving the joystick handle to the edge, and hold it still when the Power Wheelchair Trainer stops. Measure the joystick deflection angle.	Degrees	15 ± 6	15 ± 3	15.6°	Pass

#	Procedure	Unit	Required Value	Target Value	Result	Pass /fail
27	Set the Center Dead Zone to 1. Start moving the joystick handle, and hold it still when the Power Wheelchair Trainer starts moving. Measure the deflection angle.	Degrees	5 or less	0	2.9°	Pass
28	Set the Center Dead Zone to 10. Start moving the joystick handle, and hold it still when the Power Wheelchair Trainer starts moving. Measure the deflection angle.	Degrees	15 ± 6	15 ± 3	19.4°	Pass

Table 19: Specification Tests

One test did not pass verification testing. The failed test was test #8, learner joystick wireless range. The required value was 25 feet of wireless range, but the test yielded only 23 feet until the wireless connection was lost. Wireless range depends on environmental factors such as building structure, interference, and the joystick's relation to the human body. Since the learner joystick is less than 10 feet away from the receiver in the typical use case, it was deemed unnecessary to resolve the issue.

VI.2 Validation

The purpose of validation is to ensure that the Power Wheelchair Trainer meets the user needs and fulfills the intended uses. FDA guidelines state that medical devices need to undergo clinical evaluation and should be tested in the actual or simulated use environment as a part of validation (Food & Drug Administration, 1997). In some circumstances, a comparison with a predicate device may serve as a form of validation (Teixeira & Bradley, 2002). The most likely predicate device for the Power Wheelchair Trainer would be a power wheelchair. While there are similarities between a power wheelchair and the Power Wheelchair Trainer, they ultimately solve two different problems and have different intended uses. To validate that the design of the Power Wheelchair Trainer meets the user

needs and fulfills the intended uses, the Power Wheelchair Trainer was demonstrated to experts in the field of power mobility and informal feedback gathered. The Power Wheelchair Trainer was demonstrated to the following power mobility and rehabilitation experts:

- Karen Koch, Occupational Therapist, Registered (OTR), of Blossomland Learning Center in Berrien County, Michigan
- Beth McCarty, Occupational Therapist, Registered, Licensed (OTR/L), Assistive Technology Professional (ATP), and the staff at the Aaron W. Perlman Center at Cincinnati Children's Hospital, Cincinnati, Ohio
- Linda Bidabe, Creator of MOVE International, Bakersfield, California
- Edward Hurvitz, Doctor of Medicine (MD), of the University of Michigan, Ann Arbor, Michigan

The Power Wheelchair Trainer was trialed at Blossomland Learning Center under supervision of Karen Koch, OTR. Several children trialed the Power Wheelchair Trainer, and Karen Koch, OTR expressed positive feedback.

The Power Wheelchair Trainer was also trialed with several children at the Aaron W. Perlman center at Cincinnati Children's Hospital under the supervision of Beth McCarty, OTR/L, ATP, a power mobility expert. Here several children also tried the Power Wheelchair Trainer with great success. Beth McCarty suggested that a power mobility training device for smaller children could also be useful. We had to take it apart to fit in the elevator.

Linda Bidabe of California came to Lincoln Development Center and observed several training sessions. Linda Bidabe expressed positive feedback about the Power Wheelchair Trainer.

In June 2013, the Power Wheelchair Trainer was demonstrated to Edward Hurvitz and a team of power mobility staff of Ann Arbor. The author found that due to the large size it was difficult to navigate the Power Wheelchair Trainer through indoor hallways. The Power Wheelchair Trainer

required partial disassembly to fit in an elevator, and there were two doorways that were too narrow for the trainer, also requiring partial disassembly. Hurvitz's team also noted that mid-wheel drive is preferred to rear-wheel drive, and they hardly ever sell anything that is not mid-wheel drive anymore. The team also noted that fully-custom electronics is cumbersome to maintain when compared to a commercially available control system such as those from Invacare (Elyria, Ohio). If the electronics were to fail, replacement parts for the custom system are not readily available, and repairs are often time consuming and expensive. The issue of FDA approval also came up, and some of the team noticed that there was no disengage lever on the motors which may be a barrier to FDA approval. The wireless joystick may also be a barrier to FDA approval.

The Power Wheelchair Trainer has been in use since September, 2012, and as of this writing is still in use, at the Lincoln Development Center, an area public school in Grand Rapids, Michigan that provides special education services for individuals aged 5 to 26. Students with various motor impairments who lack independent mobility skills are selected to participate in training using the Power Wheelchair Trainer based on their interest in the environment and their apparent desire to move. Individual 30-minute practice sessions with the Power Wheelchair Trainer are supervised by a physical therapist and conducted 1-2 times per week during the school year. The training sessions consist of both structured movement tasks and unstructured self-directed mobility exploration. Some individuals are working on developing basic cause and effect skills while others are learning to drive safely within their environment.

VII. Future Work

VII.1 Effects of Power Mobility during Childhood Development

Current research shows that the ability of children to move independently and explore their environment during childhood is important for the development of cognitive and psychosocial skills (Jones et al., 2012; Tefft et al., 1999). Research is needed to assess the effects of power mobility training on the cognitive and psychosocial skills of children with severe motor impairments, or children who lack independent mobility skills during childhood development.

VII.2 Unloading the Wheelchair in the event of a Power Failure

One notable unsolved safety issue is that if the battery drains and platform will not go down, there is no safe and easy way to unload the learner and his/her wheelchair without lifting. Possible ideas to work around this issue would be to mount the front casters on the frame instead of the gate so that the front gate can be removed. Another possibility is to revert back to the version 4a front gate where the front gate swings open. A design change involving a U-Haul® style ramp is possibly a viable option. The ramp would slide out from underneath the platform. A ramp like the one on the original Turtle Trainer (Bresler, 1990) would block the view and is therefore not viable.

VII.3 Mid-wheel Drive

Feedback from power mobility experts noted that the rear-wheel drive design makes it difficult to navigate around poles and other stationary objects. A mid-wheel drive design would make navigation easier for the user.

VII.4 Music and Vibration Feedback

Suggest from therapists suggested that to enhance the learning experience through positive reinforcement, the Power Wheelchair Trainer could provide vibration and/or musical feedback while driving.

VII.5 Low Battery Indicator

The Power Wheelchair Trainer currently does not provide a way to tell the current charge of the batteries unless the charger is plugged in. A low battery indicator on the control panel would be useful to the supervising therapist.

VII.6 EMI

A problem with the LCD going blank was resolved with a ferrite bead (see section V.8.2), but such a fix is considered to be temporary. The root cause of the problem is that the linear actuator driver performs high power switching, resulting in electromagnetic interference (EMI). Future work would involve reducing that EMI. One method of doing so is to route unrelated traces on the circuit board perpendicularly to reduce crosstalk. Another method is use p-channel MOSFETs as the top MOSFET in the half-bridge configuration of the linear actuator drivers.

VIII. Conclusion

This thesis documented the development of a control system for the Power Wheelchair Trainer, a device that converts a manual wheelchair into a power wheelchair. A literature review was conducted with a focus on the need for power mobility at a young age, power mobility training, and technical aspects relating to the project including a review of wheelchair drive control input types, prior work and previous versions of the Power Wheelchair Trainer, and common programmable settings in power wheelchair control systems. The required features of the control system have been gathered and documented, along with how version 5 of the Power Wheelchair Trainer control system was different from prior versions. The Power Wheelchair Trainer control system was built successfully and provided the required features. A fully custom control system has potential barriers to FDA approval because it is unknown if the FDA will approve a wireless joystick. A future version would make use of a commercially available control system in order to increase likelihood of FDA approval. The design and build process was documented, and the prototype Power Wheelchair Trainer was tested and evaluated to ensure that it meets engineering requirements and fulfills the intended use. Physical therapists and powered mobility experts have expressed positive feedback about the Power Wheelchair Trainer and would like to see more rehabilitation centers and schools offer this unique powered mobility training opportunity. The Power Wheelchair Trainer can aid with further research on the effects of power mobility on cognitive and psychosocial skills in children with severe motor impairments.

Appendix A: Invacare MK6 Programmable Settings

The following table is copied from the Invacare MK6i Electronics Programming Manual

(Invacare, 2011).

Setting	Description
Forward Speed	Sets maximum forward speed
Forward Acceleration	Time taken to reach maximum forward speed
Forward Braking	Maximum braking force available to Stop or Slow the wheelchair
Reverse Speed	Sets the maximum reverse speed, independent of turning and forward speed
Reverse Acceleration	Time taken to reach maximum Reverse speed
Reverse Braking	Maximum braking force available to Stop or Slow the wheelchair in Reverse
Turning Speed	Sets Maximum Turning Speed – Independent of Forward Speed
Turning Acceleration	How quickly the wheelchair reaches the programmed turning speed
Turning Deceleration	How quickly the wheelchair “brakes” out of a turn when returning joystick to neutral
Tremor Dampening	Accommodates Upper Extremity Tremors / Ataxia
Power Level	Sets the Max power (current) available to the motors / drive wheels, or the point at which the wheelchair will stall at an obstacle or under a load
G-Trac	Proprietary electronic gyroscope technology to ensure the wheelchair drives in a straighter path
Torque	A function of Time and Power. How quickly programmed Power Level is reached
Traction	A reduction of the speed when going into and coming out of turns
Joystick Throw	Used to calibrate joystick throw. Sets the point for reaching full speed in relation to joystick displacement. Used with individuals having reduced range of motion available for joystick operation.
Axes Select	Assigns / Re-Assigns joystick commands to a desired direction. Each of the four input axes can be redirected to any output axis, or turned off.
Input Type	Selection between proportional joystick, digital (4-direction) joystick, sip-n-puff, and other inputs
Color Theme	Sets the background color of the liquid crystal display (LCD)
Momentary/Latch	Determines the mode for FORWARD driving commands. Momentary commands are only active while the command is being given. Latched commands remain active after release of the driver control – until 2 reverse commands or emergency stop switch is activated.

Appendix B: PGDT R-Net Omni+ Programmable Settings

The following table is copied directly from the R-Net Technical Manual (PG Drives Technology, 2011).

Setting	Description
Speed	A user-adjustable speed setting with easily accessible speed increase and decrease buttons. Setting ranges from 1 to 5.
Maximum Forward Speed	Sets the forward driving speed of the wheelchair when the joystick is deflected full ahead and the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.
Minimum Forward Speed	Sets the forward driving speed of the wheelchair when the joystick is deflected full ahead and the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Forward Speed in steps of 1%.
Maximum Reverse Speed	Sets the reverse driving speed of the wheelchair when the joystick is deflected to full reverse and the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.
Minimum Reverse Speed	Sets the reverse driving speed of the wheelchair when the joystick is deflected to full reverse and the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Reverse Speed in steps of 1%.
Maximum Turning Speed	Sets the turning speed of the wheelchair when the joystick is deflected fully left or right and the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.
Minimum Turning Speed	Sets the turning speed of the wheelchair when the joystick is deflected full left or right and the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Turning Speed in steps of 1%.
Maximum Forward Acceleration	Sets the acceleration rate of the wheelchair in the forward direction when the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.
Minimum Forward Acceleration	Sets the acceleration rate of the wheelchair in the forward direction when the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Forward Acceleration in steps of 1%.
Maximum Forward Deceleration	Sets the deceleration rate of the wheelchair in the forward direction when the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.
Minimum Forward Deceleration	Sets the deceleration rate of the wheelchair in the forward direction when the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Forward Deceleration in steps of 1%.
Maximum Reverse Acceleration	Sets the acceleration rate of the wheelchair in the reverse direction when the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.

Setting	Description
Minimum Reverse Acceleration	Sets the acceleration rate of the wheelchair in the reverse direction when the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Reverse Acceleration in steps of 1%.
Maximum Reverse Deceleration	Sets the deceleration rate of the wheelchair in the reverse direction when the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.
Minimum Reverse Deceleration	Sets the deceleration rate of the wheelchair in the reverse direction when the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Reverse Deceleration in steps of 1%.
Maximum Turn Acceleration	Sets the acceleration rate of the wheelchair into a turn when the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.
Minimum Turn Acceleration	Sets the acceleration rate of the wheelchair into a turn when the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Turn Acceleration in steps of 1%.
Maximum Turn Deceleration	Sets the deceleration rate of the wheelchair while turning when the speed setting is at maximum (all 5 bars illuminated). Programmable range is 0 to 100% in steps of 1%.
Minimum Turn Deceleration	Sets the deceleration rate of the wheelchair while turning when the speed setting is at minimum (just 1 bar illuminated). Programmable range is 0 to Maximum Turn Deceleration in steps of 1%.
Power	This reduces the power of the wheelchair. Power is the ability of a wheelchair to climb a hill or overcome an obstacle. If it is set to 100% then the wheelchair will provide full power.
Torque	This parameter can be used to boost the power to the motors at low drive speeds. This is useful for overcoming obstacles such as door thresholds or thick pile carpets and for countering Rollback. Programmable range is 0% to 100% in steps of 1%.
Tremor Dampening	This parameter can be used to reduce the effects of a user's hand tremor. The programmable range is 0% to 100%.
Fast Brake Rate	This parameter sets the deceleration rate that is used while fast braking. Fast braking is when the joystick is pulled to the reverse position to make a faster stop. The programmable range is 0 to 200.
Joystick Forward Throw	This sets the amount of forward movement of the joystick that is required to reach full forward speed. This is particularly useful for users with limited hand movement. The programmable range is 25% to 100% in steps of 1%.
Joystick Reverse Throw	This sets the amount of reverse movement of the joystick that is required to reach full reverse speed. This is particularly useful for users with limited hand movement. The programmable range is 25% to 100% in steps of 1%.
Joystick Left Throw	This sets the amount of left movement of the joystick that is required to reach full left turning speed. This is particularly useful for users with limited hand movement. The programmable range is 25% to 100% in steps of 1%.
Joystick Right Throw	This sets the amount of right movement of the joystick that is required to reach full right turning speed. This is particularly useful for users with limited hand movement. The programmable range is 25% to 100% in steps of 1%.

Setting	Description
Joystick Deadband	This sets the size of the joystick's neutral area. In other words, how far the joystick has to be deflected before the brakes are released and drive commences. The programmable range is 10% to 50% in steps of 1%.
Invert Fwd Rev JS Axis	If it is required that the joystick has to be pulled in reverse to initiate forward driving, enable this setting
Invert Left Right JS Axis	If it is required that the joystick has to be pushed left to initiate a right turn, enable this setting
Latched Drive	This sets the type of latched drive of the wheelchair. Step and Cruise modes are available.
Latched Timeout	This sets the time-out period for latched drive and actuator control. The programmable range is 0 to 250 Seconds in steps of 1 Second.
Maximum Current Limit	Sets the long-term maximum current output of the Power Module.
Boost Drive Current	Sets the short-term maximum current output of the Power Module.
Boost Drive Time	Sets the period of time that the level of current set by Boost Drive Current is available.
Current Foldback Threshold, Time, and Level	Three parameters can be used to protect the wheelchair motors from overheating.
Invert M1 Direction	This inverts the direction of rotation of motor channel M1 on the Power Module.
Invert M2 Direction	This inverts the direction of rotation of motor channel M2 on the Power Module.
Motor Swap	This swaps the motor output channels, M1 and M2, of the Power Module.
Steer Correct	This parameter compensates for any mis-match in motor speeds, thereby ensuring the wheelchair drives in a straight line when the joystick is being pushed directly forward. This is particularly useful for switch type Input Devices. The programmable range is -9 to 9 in steps of 1.

Appendix C: Schematics

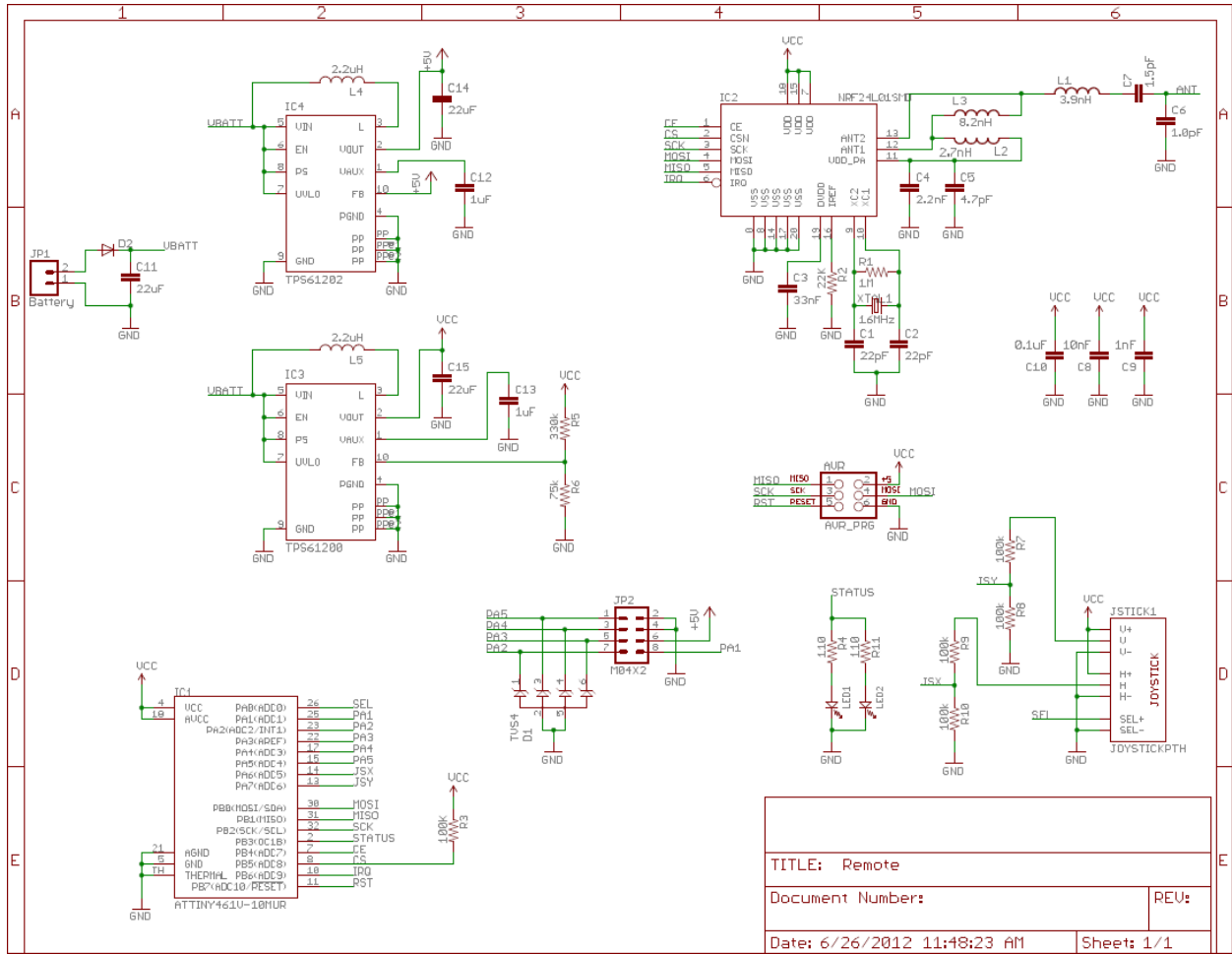


Figure 40: Learner Joystick and Therapist Remote Schematic

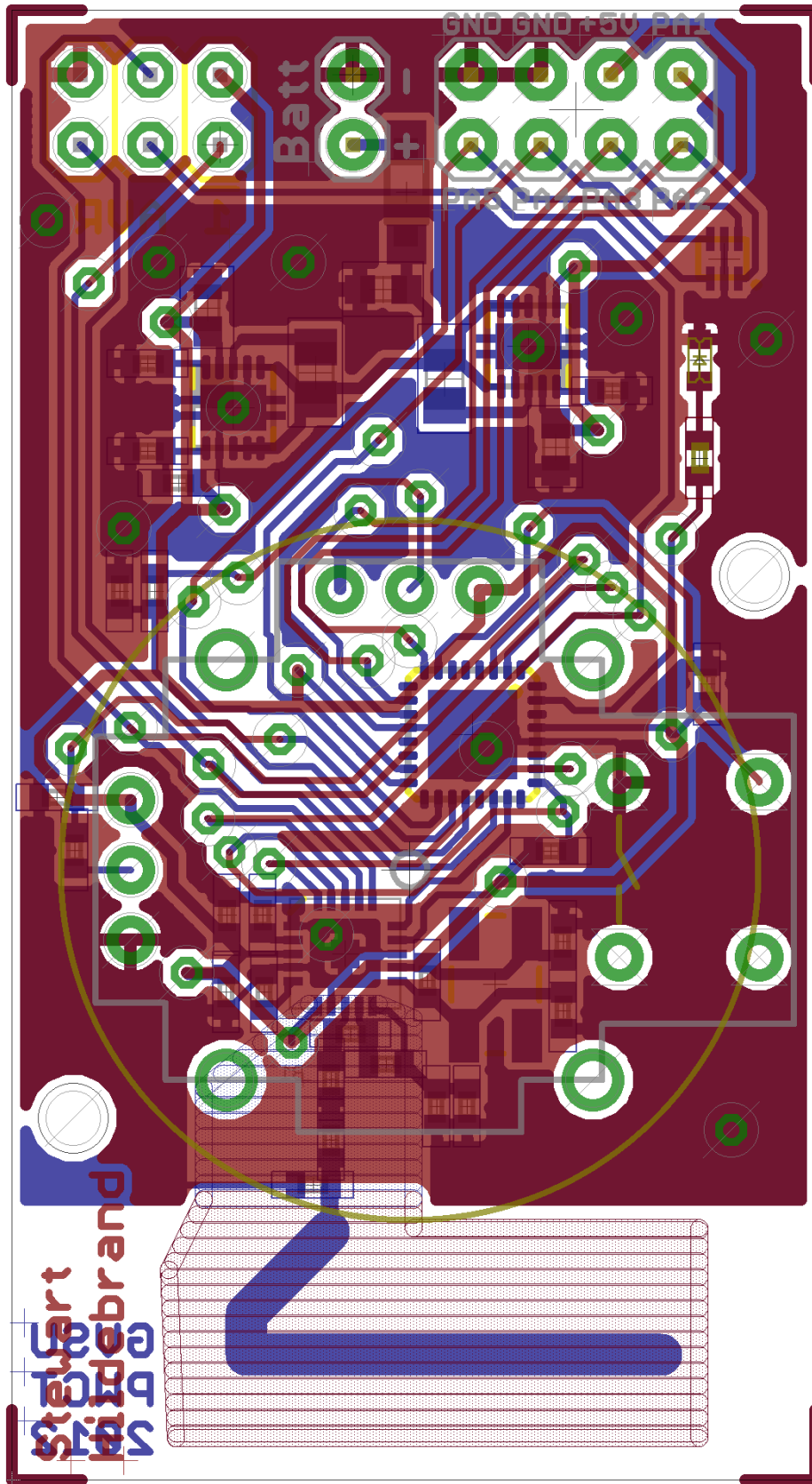


Figure 41: Learner Joystick and Therapist Remote Board Layout

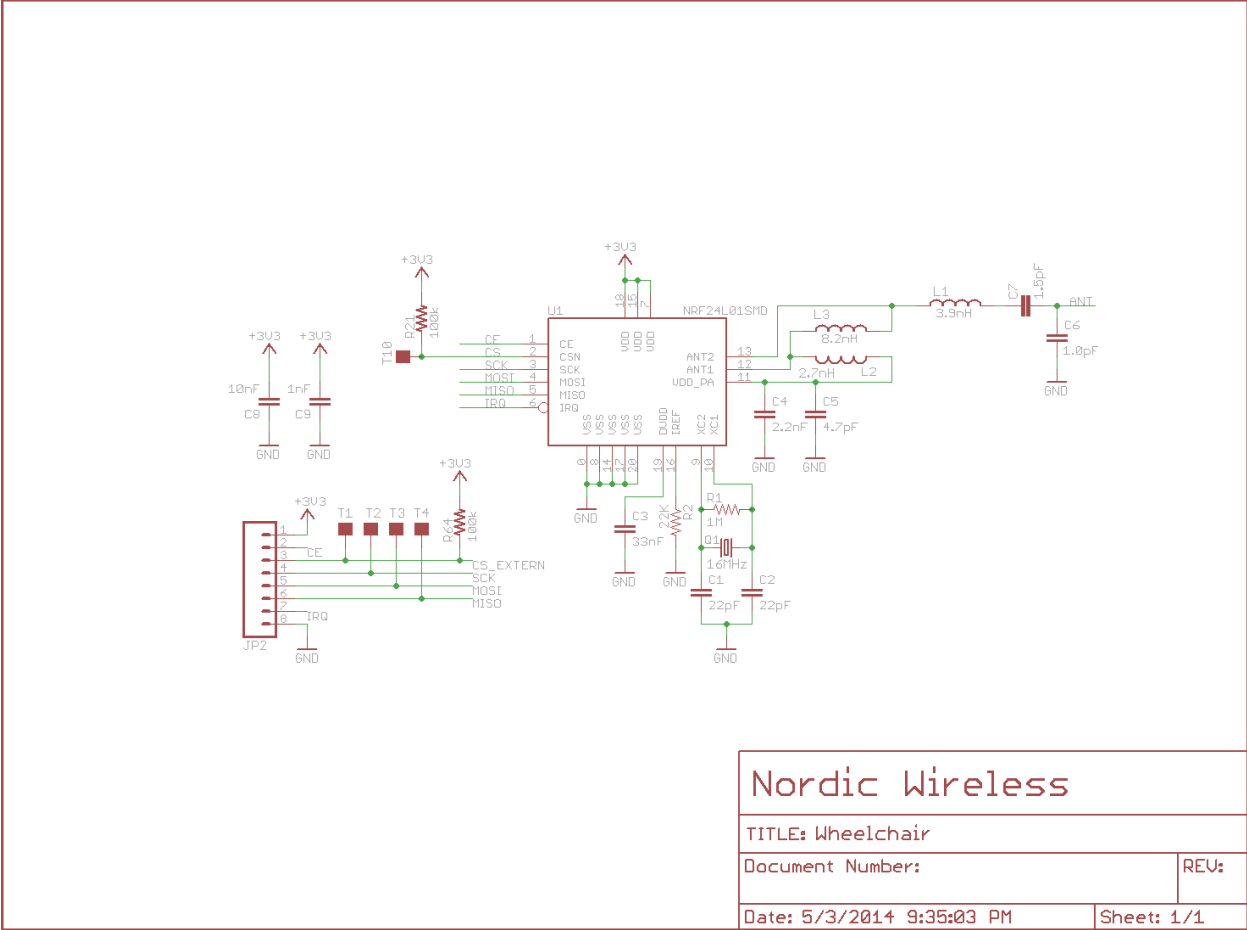


Figure 42: Wheelchair Schematic 1 of 6

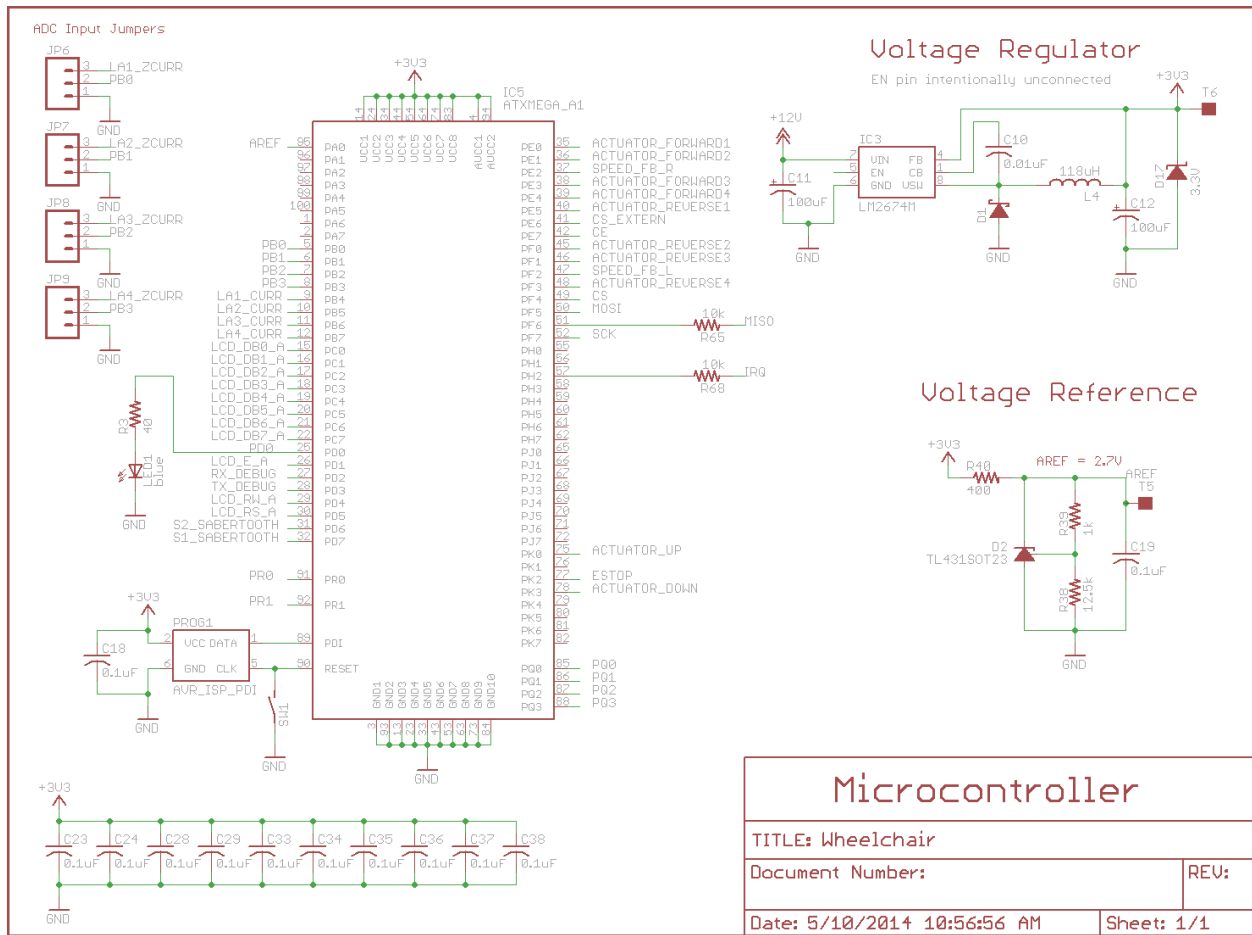


Figure 43: Wheelchair Schematic 2 of 6

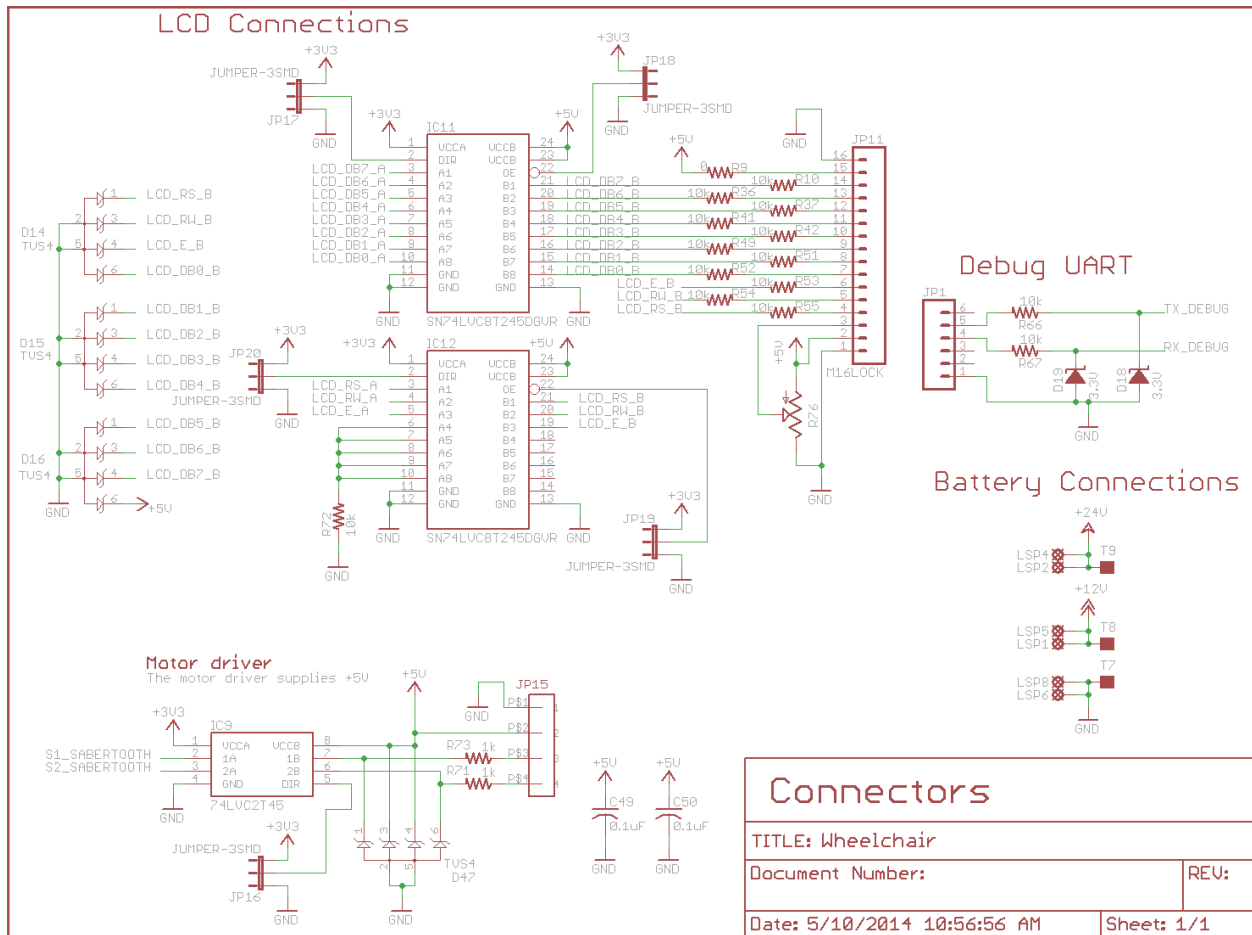
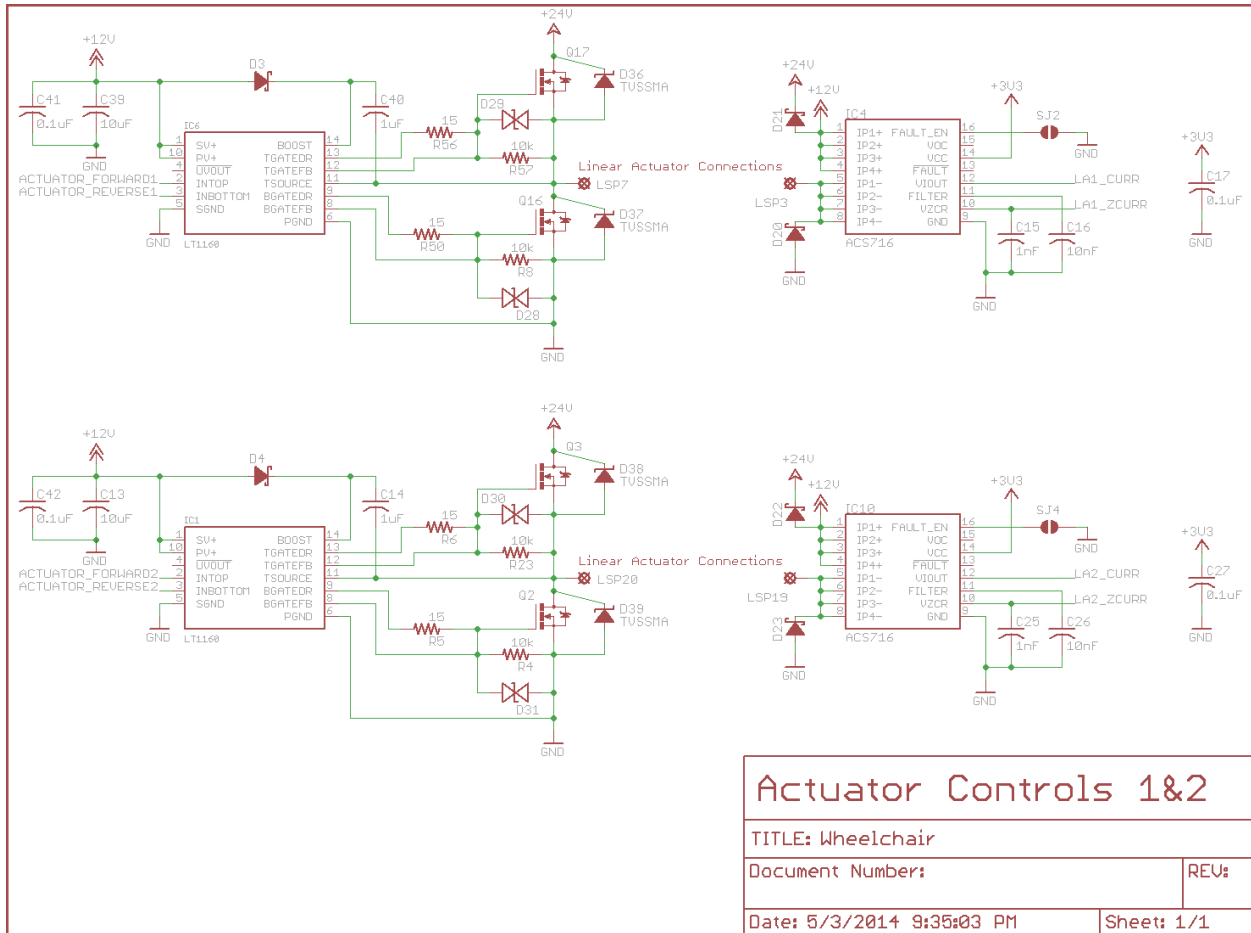
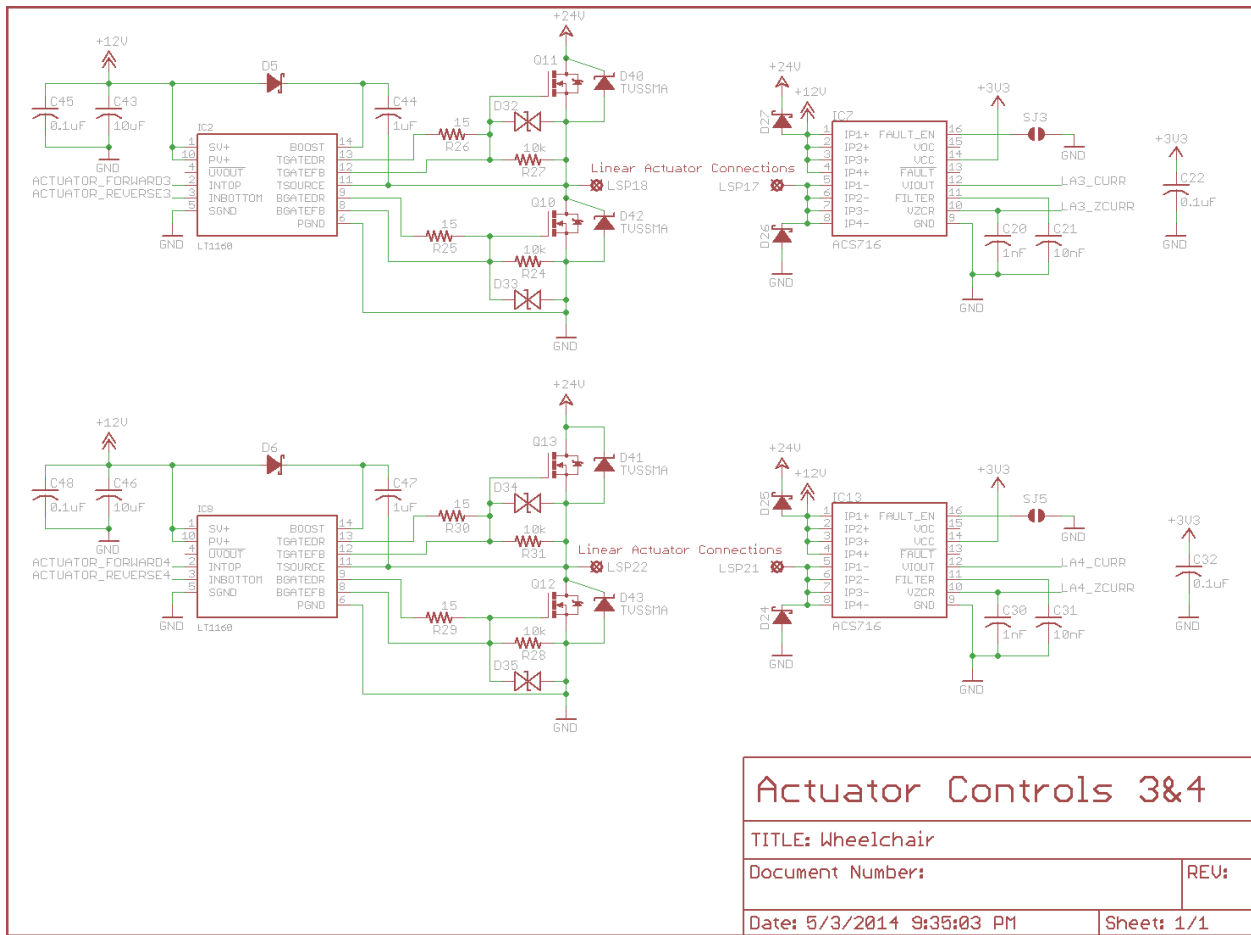


Figure 44: Wheelchair Schematic 3 of 6



Actuator Controls 1&2	
TITLE: Wheelchair	
Document Number:	REV:
Date: 5/3/2014 9:35:03 PM	Sheet: 1/1

Figure 45: Wheelchair Schematic 4 of 6



Actuator Controls 3&4	
TITLE: Wheelchair	
Document Number:	REV:
Date: 5/3/2014 9:35:03 PM	Sheet: 1/1

Figure 46: Wheelchair Schematic 5 of 6

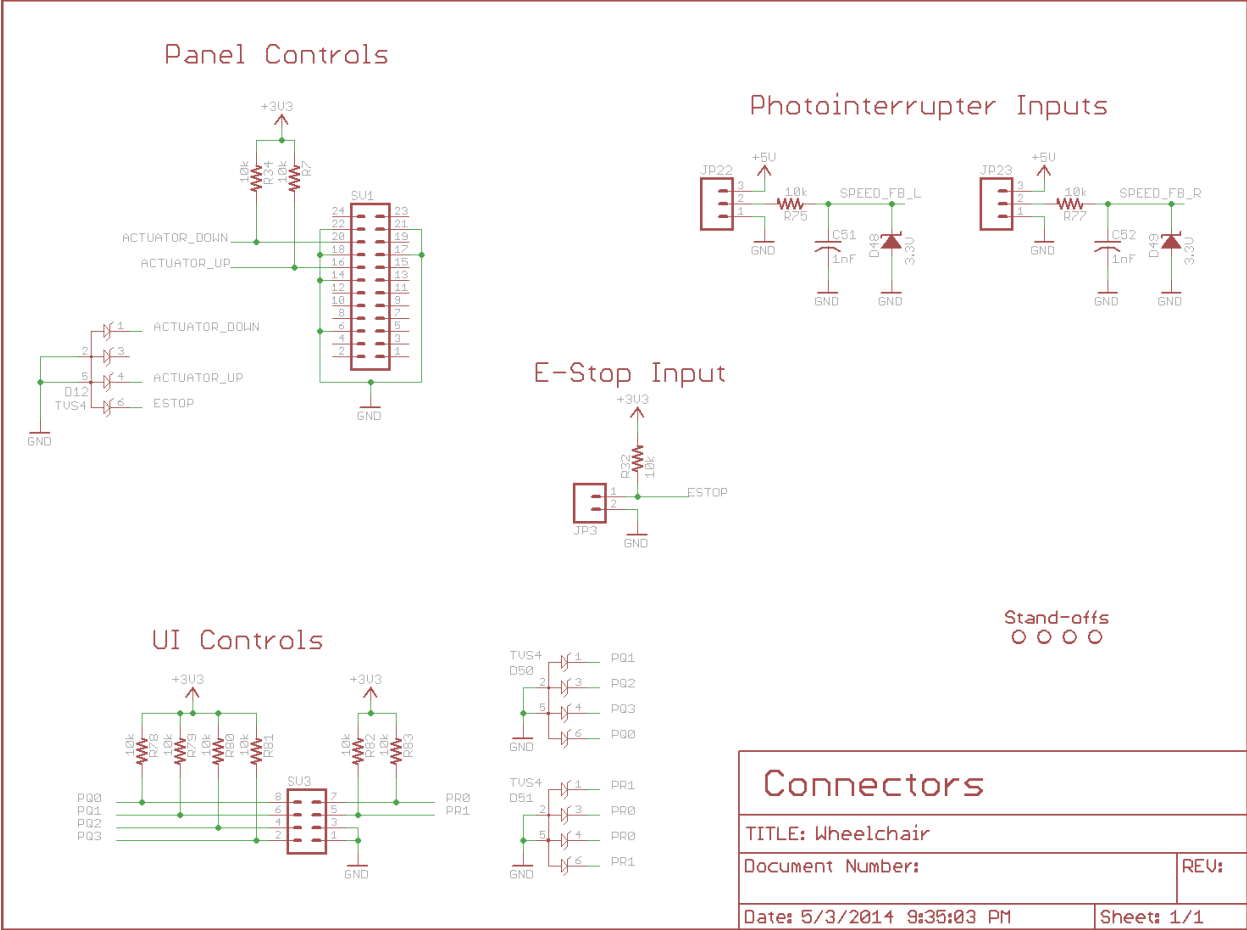


Figure 47: Wheelchair Schematic 6 of 6

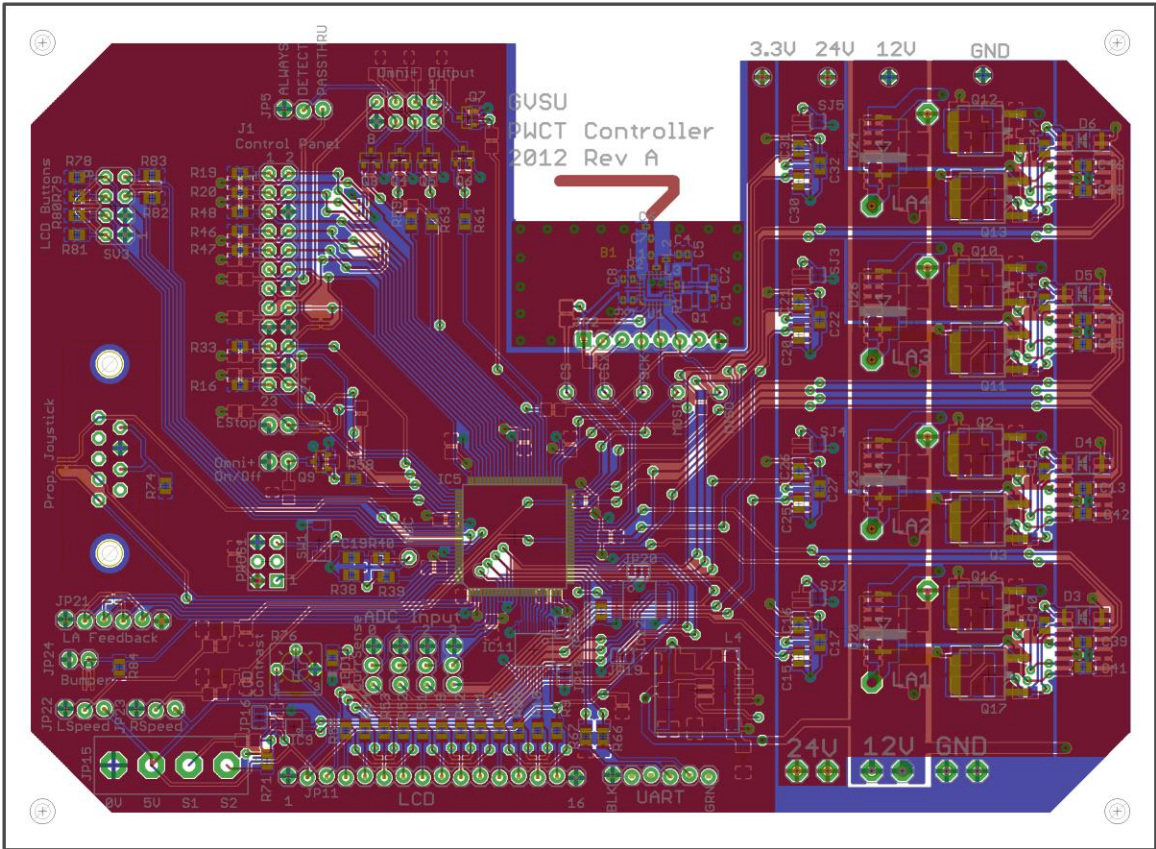


Figure 48: Wheelchair Board Layout

Appendix D: Source Code (Main Board)

main.c

```
/* This file has been prepared for Doxygen automatic documentation generation... kind
of.*/
/*! \file *****
*
* \brief PWCT main function source file
*
* This file contains the main function of the Power Wheelchair Trainer
*
* \par Target note:
* This code is written for an XMEGA 64 A1 device
*
* \author
* Stew Hildebrand, Jeff VanOss, Anderson Peck, Paul Shields
*
* $Revision: 1 $
* $Date: 03-28-2011$ \n
*
*****/

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h> // strtol
#include "../atmel/clksys_driver.h"
#include "../atmel/wdt_driver.h"
#include "util.h"
#include "nordic_driver.h"
#include "linear_actuator.h"
#include "PWCT_io.h"
#include "motor_driver.h"
#include "lcd_driver.h"
#include "menu.h"
#include "joystick_algorithm.h"
//#include "test.h"

static void eStop(const char *estopText)
{
    motorEStop();
    while (1)
    {
        lcdText(estopText, "Ver. " __DATE__, 0);
        WDT_Reset();
    }
}

static void displayResetReason(double delayTime_ms)
{
    uint8_t status = RST.STATUS;
    char lcdLine[LCD_NUM_CHARACTERS+1];
    lcdLine[LCD_NUM_CHARACTERS] = '\0';
```

main.c

```
    sprintf(lcdLine, "RST.STATUS=0x%02x", status);

    RST.STATUS = status & 0x3F;

    if (status & RST_SDRF_bm) {
        lcdText("Spike Detect Rst", lcdLine, 1);
        while (1) {
        }
    }
    if (status & RST_SRF_bm) {
        lcdText("Software Reset", lcdLine, 1);
        while (1) {
        }
    }
    if (status & RST_PDIRF_bm) {
        lcdText("PDI Reset", lcdLine, 1);
        _delay_ms(delayTime_ms);
    }
    if (status & RST_WDRF_bm) {
        lcdText("Watchdog Reset", lcdLine, 1);
        while (1) {
        }
    }
    if (status & RST_BORF_bm) {
        lcdText("Brown-out Reset", lcdLine, 1);
        while (1) {
        }
    }
    if (status & RST_EXTRF_bm) {
        lcdText("External Reset", lcdLine, 1);
        _delay_ms(delayTime_ms);
    }
    if (status & RST_PORF_bm) {
        lcdText("Power-on Reset", lcdLine, 1);
        _delay_ms(delayTime_ms);
    }
    if (status == 0) {
        lcdText("RST.STATUS == 0", lcdLine, 1);
        while (1) {
        }
    }
}

/*! \brief Main function
 *
 * This function initializes the hardware, starts monitoring input signals.
 */
int main( void )
{
    uint8_t actuatorSwitchState = 0;
    states state = IDLE;
    //states previousState = IDLE;
    int16_t speed;
    int16_t dir;

    //Setup the 32MHz Clock
```

main.c

```
//start 32MHz oscillator
CLKSYS_Enable( OSC_RC32MEN_bm );
//wait for 32MHz oscillator to stabilize
while ( CLKSYS_IsReady( OSC_RC32MRDY_bm ) == 0 );
//set clock as internal 32MHz RC oscillator
CLKSYS_Main_ClockSource_Select( CLK_SCLKSEL_RC32M_gc );

// Enable global interrupts.
sei();
joystickAlgorithmInit();
initMotorDriver();

dbgLEDinit();
dbgUSARTinit();

initLinearActuators();

initPWCTio();

nordic_Initialize();

initLCDDriver();

displayResetReason(500);

menuInit();

WDT_EnableAndSetTimeout(WDT_PER_128CLK_gc); //set watchdog timer for 0.125s
period

printf("\nReset\n");

//testJoystickDriveMotors();

//testPropJoy();

//testNordicWireless();

//testInputs();

//testMotorDriver();

//Run Operational State Machine
while(1) {
    WDT_Reset();

    getProportionalMoveDirection(&speed, &dir);
    menuUpdate(speed, dir);

    //check inputs for state changes
    SampleInputs();

    actuatorSwitchState = ActuatorSwitchPressed();

    if (nordic_getInstructorEStop()) {
        eStop("Remote E-stop");
    }
}
```

main.c

```
    } else if (PanelEStopPressed()) {
        eStop("Panel E-stop");
    } else if (menuGetIsPlatformDown() || ((state == IDLE || state == LOAD) &&
actuatorSwitchState)) {
        state = LOAD;
    } else if (!menuGetIsPlatformDown() && (speed != 0 || dir != 0)) {
        state = MOVE;
    } else {
        state = IDLE;
    }
}

/*
if (previousState != state)
{
    switch (state) {
        case IDLE:
            printf("Idle\n");
            break;
        case MOVE:
            printf("Move\n");
            break;
        case LOAD:
            printf("Load\n");
            break;
    }
    previousState = state;
}
*/

//set state output
switch(state) {
case IDLE:
    StopPlatform();
    OmniStopMove();
    //turn off platform down LED
    PORTK.OUTCLR = PIN5_bm;
    break;
case LOAD:
    OmniStopMove();
    switch(actuatorSwitchState) {
case 0: //actuator switch not pressed, stop platform
        StopPlatform();
        break;
case 1: //actuator switch down, lower platform
        LowerPlatform();
        menuPlatformDownPushed();
        break;
case 2: // actuator switch up, raise platform
        RaisePlatform();
        menuPlatformUpPushed();
        break;
    }
    //turn on platform down LED
    PORTK.OUTSET = PIN5_bm;
    break;
case MOVE:
```

main.c

```
        StopPlatform();
        setMotors(speed, dir);
        //turn off platform down LED
        PORTK.OUTCLR = PIN5_bm;
        break;
    }
}
return 1;
}
```

joystick_algorithm.h

```
/*
 * joystick_algorithm.h
 *
 * Created: 9/27/2012 9:52:55 PM
 * Author: Stew
 */

#ifndef JOYSTICK_ALGORITHM_H_
#define JOYSTICK_ALGORITHM_H_

void joystickAlgorithmInit();
void getProportionalMoveDirection(int16_t *speed, int16_t *dir);

#endif /* JOYSTICK_ALGORITHM_H_ */
```

joystick_algorithm.c

```
/*
 * joystick_algorithm.c
 *
 * Created: 9/27/2012 9:52:10 PM
 * Author: Stew
 */

#include <stdint.h>
#include <math.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "../atmel/avr_compiler.h"
#include "nordic_driver.h"
#include "menu.h"

// Input to time-based filters (acceleration, tremor dampening)
static volatile int16_t gSpeedPreFilter = 0;
static volatile int16_t gDirPreFilter = 0;

static volatile double gSpeedBetweenFilters = 0;
static volatile double gDirBetweenFilters = 0;
```

joystick_algorithm.c

```
// Output from time-based filters
static volatile int16_t gSpeedPostFilter = 0;
static volatile int16_t gDirPostFilter = 0;

static volatile uint8_t gIsOuterDeadBand = 0;
static volatile uint8_t gIsOuterDeadBandTimeout = 0;

static volatile uint8_t gOverridden = 0;

void joystickAlgorithmInit()
{
    // TCD1 is the timer acceleration and tremor filtering
    TCD1.CTRLA = TC_CLKSEL_DIV1_gc; // 1 tick = 31.25 nanoseconds
    TCD1.CTRLB = TC_WGMODE_FRQ_gc;
    TCD1.INTCTRLB = TC_CCAINTLVL_MED_gc;
    TCD1.CCA = 32000; // Goal: interrupt every 1 millisecond

    PMIC.CTRL |= PMIC_MEDLVLEN_bm;
}

static void outerDeadBandLogic(int16_t speed, int16_t dir)
{
    // Calculate joystick distance from center using pythagorean theorem
    double r = sqrt(speed * speed + dir * dir);

    AVR_ENTER_CRITICAL_REGION();
    if (menuGetOuterDeadBand(gOverridden) && (
        (r > 60) ||
        (gIsOuterDeadBand && gIsOuterDeadBandTimeout && (r >
menuGetCenterDeadBand(gOverridden) + 5))
    )) {
        gIsOuterDeadBand = 1;
    } else {
        gIsOuterDeadBand = 0;
    }
    AVR_LEAVE_CRITICAL_REGION();
}

static void centerDeadBand(int16_t *x, int16_t *y, uint8_t deadBand)
{
    double theta;
    double r;

    // Convert to polar coordinates
    r = sqrt((*x) * (*x) + (*y) * (*y));
    theta = atan2(*y, *x);

    // Apply the deadband
    r = r - (double)deadBand;
    if (r < 0) {
        r = 0;
    }

    // Convert back to cartesian coordinates
    *x = (r * cos(theta));
    *y = (r * sin(theta));
}
```

joystick_algorithm.c

```
}  
  
void getProportionalMoveDirection(int16_t *returnSpeed, int16_t *returnDir)  
{  
    int16_t speed = nordic_getInstructorSpeed();  
    int16_t dir = nordic_getInstructorDirection();  
  
    if (speed >= -1 && speed <= 1 &&  
        dir >= -1 && dir <= 1) {  
        gOverridden = 0;  
        speed = nordic_getWirelessPropJoySpeed();  
        dir = nordic_getWirelessPropJoyDirection();  
    } else {  
        gOverridden = 1;  
    }  
  
    if (menuGetMotorsDisabled()) {  
        speed = 0;  
        dir = 0;  
    }  
  
    //TODO: we should still be able to override with the therapist remote when the  
    outer dead zone shut-off is in effect  
    //TODO: also, check the logic for switch inputs and propasswitch mode  
    outerDeadBandLogic(speed, dir);  
    if (gIsOuterDeadBand && gIsOuterDeadBandTimeout) {  
        speed = 0;  
        dir = 0;  
    }  
  
    if (menuGetInvert(gOverridden)) {  
        speed = -speed;  
    }  
  
    // Proportional joystick as switch joystick  
    if (menuGetPropAsSwitch(gOverridden))  
    {  
        uint8_t threshold = 50;  
        if (speed > threshold) {  
            speed = menuGetTopFwdSpeed(gOverridden);  
        } else if (speed < -threshold) {  
            speed = -menuGetTopRevSpeed(gOverridden);  
        } else {  
            speed = 0;  
        }  
  
        if (dir > threshold) {  
            dir = menuGetTopTurnSpeed(gOverridden);  
        } else if (dir < -threshold) {  
            dir = -menuGetTopTurnSpeed(gOverridden);  
        } else {  
            dir = 0;  
        }  
    }  
    else  
    {
```


joystick_algorithm.c

```
// apply center dead band
centerDeadBand(&dir, &speed, menuGetCenterDeadBand(gOverridden));

// fwd/rev throw
if (speed > 0) {
    speed *= menuGetFwdThrow(gOverridden);
}
if (speed < 0) {
    speed *= menuGetRevThrow(gOverridden);
}
// turn throw
dir *= menuGetTurnThrow(gOverridden);

// Top speeds
if (speed > menuGetTopFwdSpeed(gOverridden)) {
    // max forward speed
    speed = menuGetTopFwdSpeed(gOverridden);
} else if (speed < -menuGetTopRevSpeed(gOverridden)) {
    // max reverse speed
    speed = -menuGetTopRevSpeed(gOverridden);
}

// max turn speed
if (dir > menuGetTopTurnSpeed(gOverridden)) {
    dir = menuGetTopTurnSpeed(gOverridden);
} else if (dir < -menuGetTopTurnSpeed(gOverridden)) {
    dir = -menuGetTopTurnSpeed(gOverridden);
}
}

// We don't want to get interrupted while accessing shared variables
AVR_ENTER_CRITICAL_REGION();

// todo: will the outer dead zone stop signal that the joystick is not active,
then switches take precedence even if the joystick is at full blast?
if (speed >= -1 && speed <= 1 &&
    dir >= -1 && dir <= 1 && !gOverridden) {
    // Buddy buttons only active if joystick not active
    if (nordic_getStudentForward() != nordic_getStudentReverse()) {
        if (nordic_getStudentForward()) {
            speed = menuGetTopFwdSpeed(gOverridden);
        } else if (nordic_getStudentReverse()) {
            speed = -menuGetTopRevSpeed(gOverridden);
        }
    }
    if (nordic_getStudentRight() != nordic_getStudentLeft()) {
        if (nordic_getStudentRight()) {
            dir = menuGetTopTurnSpeed(gOverridden);
        } else if (nordic_getStudentLeft()) {
            dir = -menuGetTopTurnSpeed(gOverridden);
        }
    }
}

// We've applied the direct-mapped logic, now it's time to hand it off to the
filters
```

joystick_algorithm.c

```
gSpeedPreFilter = speed;
gDirPreFilter = dir;

speed = gSpeedPostFilter;
dir = gDirPostFilter;
AVR_LEAVE_CRITICAL_REGION();

*returnSpeed = speed;
*returnDir = dir;
}

// Filter topography and variable naming:
// preFilter --> [low-pass] --> betweenFilters --> [accel/decel] --> postFilter
ISR(TCD1_CCA_vect)
{
    static uint8_t accelerationCount = 0;
    static uint8_t decelerationCount = 0;
    static uint16_t outerDeadBandMillisecondCount = 0;
    static uint8_t outerDeadBandTime = 0;

    if (gIsOuterDeadBand) {
        if (!gIsOuterDeadBandTimeout) {
            uint8_t timeoutTime = menuGetOuterDeadBand(gOverridden) - 1;
            if (timeoutTime == 0) {
                gIsOuterDeadBandTimeout = 1;
            } else if (timeoutTime > 0) {
                outerDeadBandMillisecondCount++;
                if (outerDeadBandMillisecondCount >= 500) {
                    outerDeadBandMillisecondCount = 0;
                    outerDeadBandTime++;
                    if (outerDeadBandTime >= timeoutTime) {
                        gIsOuterDeadBandTimeout = 1;
                    }
                }
            }
        }
    }
} else {
    outerDeadBandMillisecondCount = 0;
    outerDeadBandTime = 0;
    gIsOuterDeadBandTimeout = 0;
}

// Low-pass filter (aka Tremor Dampening aka Tremor Suppression aka Sensitivity)
if ((gSpeedPreFilter > 0 && gSpeedPreFilter > gSpeedBetweenFilters) ||
    (gSpeedPreFilter < 0 && gSpeedPreFilter < gSpeedBetweenFilters)) {
    gSpeedBetweenFilters = gSpeedPreFilter +
menuGetSensitivity(gOverridden) * (gSpeedPreFilter - gSpeedBetweenFilters);
} else {
    gSpeedBetweenFilters = gSpeedPreFilter;
}
if ((gDirPreFilter > 0 && gDirPreFilter > gDirBetweenFilters) ||
    (gDirPreFilter < 0 && gDirPreFilter < gDirBetweenFilters)) {
    gDirBetweenFilters = gDirBetweenFilters + menuGetSensitivity(gOverridden) *
(gDirPreFilter - gDirBetweenFilters);
} else {
    gDirBetweenFilters = gDirPreFilter;
}
```

joystick_algorithm.c

```
    }  
  
    // Acceleration/deceleration: must wait X milliseconds before speed/dir is changed  
by 1  
    accelerationCount++;  
    decelerationCount++;  
    if (accelerationCount >= menuGetAcceleration(gOverridden))  
    {  
        accelerationCount = 0;  
  
        if (gSpeedPostFilter >= 0 && gSpeedBetweenFilters > gSpeedPostFilter) {  
            gSpeedPostFilter++;  
        }  
        if (gSpeedPostFilter <= 0 && gSpeedBetweenFilters < gSpeedPostFilter) {  
            gSpeedPostFilter--;  
        }  
        if (gDirPostFilter >= 0 && gDirBetweenFilters > gDirPostFilter) {  
            gDirPostFilter++;  
        }  
        if (gDirPostFilter <= 0 && gDirBetweenFilters < gDirPostFilter) {  
            gDirPostFilter--;  
        }  
    }  
  
    if (decelerationCount >= menuGetDeceleration(gOverridden))  
    {  
        decelerationCount = 0;  
  
        if (gSpeedPostFilter > 0 && gSpeedBetweenFilters < gSpeedPostFilter) {  
            gSpeedPostFilter--;  
            if (gSpeedBetweenFilters < 0) {  
                gSpeedPostFilter--;  
            }  
        }  
        if (gSpeedPostFilter < 0 && gSpeedBetweenFilters > gSpeedPostFilter) {  
            gSpeedPostFilter++;  
            if (gSpeedBetweenFilters > 0) {  
                gSpeedPostFilter++;  
            }  
        }  
  
        if (gDirPostFilter > 0 && gDirBetweenFilters < gDirPostFilter) {  
            gDirPostFilter--;  
            if (gDirBetweenFilters < 0) {  
                gDirPostFilter--;  
            }  
        }  
        if (gDirPostFilter < 0 && gDirBetweenFilters > gDirPostFilter) {  
            gDirPostFilter++;  
            if (gDirBetweenFilters > 0) {  
                gDirPostFilter++;  
            }  
        }  
    }  
}
```

lcd_driver.h

```
/*
 * lcd_driver.h
 *
 * Created: 6/4/2012 4:22:33 PM
 * Author: Stew
 */

#ifndef LCD_DRIVER_H_
#define LCD_DRIVER_H_

// BEGIN HARDWARE SPECIFIC

#define LCD_NUM_CHARACTERS 16

// DB7 = PC7
#define LCD_DB7_PORT PORTC
#define LCD_DB7_PIN_bm PIN7_bm

// DB6 = PC6
#define LCD_DB6_PORT PORTC
#define LCD_DB6_PIN_bm PIN6_bm

// DB5 = PC5
#define LCD_DB5_PORT PORTC
#define LCD_DB5_PIN_bm PIN5_bm

// DB4 = PC4
#define LCD_DB4_PORT PORTC
#define LCD_DB4_PIN_bm PIN4_bm

// Operation Enable, E = PD1
#define LCD_E_PORT PORTD
#define LCD_E_PIN_bm PIN1_bm

// Read/Write, RW = PD4
#define LCD_RW_PORT PORTD
#define LCD_RW_PIN_bm PIN4_bm
//RW=0: write
//RW=1: read WARNING READ NOT SUPPORTED BY HARDWARE

// Register Select, RS = PD5
#define LCD_RS_PORT PORTD
#define LCD_RS_PIN_bm PIN5_bm
//RS=0: instruction
//RS=1: data

// END HARDWARE SPECIFIC

// Instructions/Commands

#define LCD_CMD_CLEAR_DISPLAY 0x01
    // Execution time = 1.52msec
```

lcd_driver.h

```
// Also returns home

#define LCD_CMD_RETURN_HOME 0x02
// Execution time = 1.52msec

#define LCD_CMD_ENTRY_MODE_SET 0x04
// Execution time = 38usec
#define LCD_CMD_ENTRY_MODE_I_D_bm 0x02
// I/D=1: cursor moves right / DDRAM address increments / display shift left
// I/D=0: cursor moves left / DDRAM address decrements / display shift right
#define LCD_CMD_ENTRY_MODE_S_bm 0x01
// S=0: don't shift display
// S=1: during DDRAM write operation, shift display

#define LCD_CMD_DISPLAY_ON_OFF 0x08
// Execution time = 38usec
#define LCD_CMD_DISPLAY_ON_OFF_D_bm 0x04
// D=1: display on
// D=0: display off
#define LCD_CMD_DISPLAY_ON_OFF_C_bm 0x02
// C=1: cursor on
// C=0: cursor off
#define LCD_CMD_DISPLAY_ON_OFF_B_bm 0x01
// B=1: cursor blink on
// B=0: cursor blink off

#define LCD_CMD_CURSOR_OR_DISPLAY_SHIFT 0x10
// Execution time = 38usec
#define LCD_CMD_CURSOR_OR_DISPLAY_SHIFT_S_C_bm 0x08
// S/C=0: shift cursor
// S/C=1: shift display
#define LCD_CMD_CURSOR_OR_DISPLAY_SHIFT_R_L_bm 0x04
// R/L=0: shift left
// R/L=1: shift right

#define LCD_CMD_FUNCTION_SET 0x20
// Execution time = 38usec
#define LCD_CMD_FUNCTION_DL_bm 0x10
// DL=1: 8 bit bus
// DL=0: 4 bit bus
#define LCD_CMD_FUNCTION_N_bm 0x08
// N=0: 1-line display
// N=1: 2-line display
#define LCD_CMD_FUNCTION_F_bm 0x04
// F=0: 5x8
// F=1: 5x11

#define LCD_CMD_SET_CGRAM_ADDR 0x40
// Execution time = 38usec
#define LCD_CGRAM_ADDR_bm 0x3F

#define LCD_CMD_SET_DDRAM_ADDR 0x80
// Execution time = 38usec
#define LCD_DDRAM_ADDR_bm 0x7F

#define LCD_LINE_1_START_ADDR 0x00
```

lcd_driver.h

```
#define LCD_LINE_2_START_ADDR 0x40

void initLCDDriver(void);
void lcdText(const char *line1, const char *line2, uint8_t blocking);
void lcdCommandBlocking(uint8_t command);

#endif /* LCD_DRIVER_H_ */
```

lcd_driver.c

```
/*
 * lcd_driver.c
 *
 * Created: 6/4/2012 4:22:20 PM
 * Author: Stew
 *
 * Interrupt-driven 4-bit parallel LCD driver for a 2-line LCD
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdio.h>
#include <string.h>
#include "../atmel/avr_compiler.h"
#include "lcd_driver.h"

// 0.25 µs per tick
#define LCD_TIMER_CLKSEL TC_CLKSEL_DIV8_gc

static volatile uint8_t gLCDState;
static volatile uint8_t gLCDCharPosition;
static volatile char *gLCDCurrentLine;

static volatile char gLCDLine1[LCD_NUM_CHARACTERS + 1];
static volatile char gLCDLine2[LCD_NUM_CHARACTERS + 1];

static inline uint8_t lcdBusy(void)
{
    return gLCDState;
}

static void lcdBusyWait(void)
{
    do {} while (lcdBusy());
}

static inline void setTimerPeriod(uint16_t period)
{
    TCE1.CCA = period;
    TCE1.CTRLFSET = TC_CMD_RESTART_gc; // The Counter, direction, and all compare
    outputs are set to zero
    TCE1.INTFLAGS = TC0_CCAIF_bm; // Clear compare interrupt
}
```

lcd_driver.c

```
static inline void stopTimer()
{
    TCE1.CTRLA = TC_CLKSEL_OFF_gc;
    TCE1.CTRLFSET = TC_CMD_RESTART_gc; // The Counter, direction, and all compare
    outputs are set to zero
    TCE1.INTFLAGS = TC0_CCAIF_bm; // Clear compare interrupt
}

static inline void startTimer()
{
    TCE1.CTRLFSET = TC_CMD_RESTART_gc; // The Counter, direction, and all compare
    outputs are set to zero
    TCE1.INTFLAGS = TC0_CCAIF_bm; // Clear compare interrupt
    TCE1.CTRLA = LCD_TIMER_CLKSEL;
}

static inline void lcdESet(void)
{
    LCD_E_PORT.OUTSET = LCD_E_PIN_bm;
}

static inline void lcdEClr(void)
{
    LCD_E_PORT.OUTCLR = LCD_E_PIN_bm;
}

static inline void lcdRSSet(void)
{
    LCD_RS_PORT.OUTSET = LCD_RS_PIN_bm;
}

static inline void lcdRSClr(void)
{
    LCD_RS_PORT.OUTCLR = LCD_RS_PIN_bm;
}

static void lcdSetPortData(uint8_t data) {
    if (data & 0x8) {
        LCD_DB7_PORT.OUTSET = LCD_DB7_PIN_bm;
    } else {
        LCD_DB7_PORT.OUTCLR = LCD_DB7_PIN_bm;
    }
    if (data & 0x4) {
        LCD_DB6_PORT.OUTSET = LCD_DB6_PIN_bm;
    } else {
        LCD_DB6_PORT.OUTCLR = LCD_DB6_PIN_bm;
    }
    if (data & 0x2) {
        LCD_DB5_PORT.OUTSET = LCD_DB5_PIN_bm;
    } else {
        LCD_DB5_PORT.OUTCLR = LCD_DB5_PIN_bm;
    }
    if (data & 0x1) {
        LCD_DB4_PORT.OUTSET = LCD_DB4_PIN_bm;
    } else {

```

lcd_driver.c

```
        LCD_DB4_PORT.OUTCLR = LCD_DB4_PIN_bm;
    }
}

static void lcdNibble(uint8_t data)
{
    lcdSetPortData(data);
    lcdESet();
    _delay_us(1);
    lcdEClr();
}

void lcdCommandBlocking(uint8_t command)
{
    lcdBusyWait();
    lcdRSCLR();
    lcdNibble(command >> 4);
    _delay_us(1);
    lcdNibble(command & 0x0F);

    if ((command == LCD_CMD_CLEAR_DISPLAY) || ((command & 0xFE) ==
LCD_CMD_RETURN_HOME)) {
        _delay_ms(1.52);
    } else {
        _delay_us(38);
    }
}

static inline void lcdStartWrite(void)
{
    // case 0: Set E - ADDRLINE1[7:4]
    lcdRSCLR();
    lcdESet();
    lcdSetPortData((LCD_CMD_SET_DDRAM_ADDR | (LCD_DDRAM_ADDR_bm &
LCD_LINE_1_START_ADDR)) >> 4);
    gLCDState = 1;
    gLCDCurrentLine = gLCDLine1;
    setTimerPeriod(4);
    startTimer();
}

// Copies src into dest, max LCD_NUM_CHARACTERS chars, pads with ' ', null terminates
static void copyString(volatile char *dest, const char *src)
{
    size_t i;
    size_t srclen = strlen(src);
    if (srclen > LCD_NUM_CHARACTERS) {
        srclen = LCD_NUM_CHARACTERS;
    }
    for (i = 0; i < srclen; i++) {
        dest[i] = src[i];
    }
    for (; i < LCD_NUM_CHARACTERS; i++) {
        dest[i] = ' ';
    }
    dest[i] = '\0';
}
```


lcd_driver.c

```
}

void lcdText(const char *line1, const char *line2, uint8_t blocking)
{
    if (blocking) {
        while (lcdBusy()) {
            // twiddle thumbs
        }
    } else if (lcdBusy()) {
        return;
    }

    copyString(gLCDLine1, line1);
    copyString(gLCDLine2, line2);

    lcdStartWrite();
}

void initLCDDriver(void)
{
    // I/O pin setup
    LCD_DB7_PORT.DIRSET = LCD_DB7_PIN_bm;
    LCD_DB7_PORT.OUTCLR = LCD_DB7_PIN_bm;
    LCD_DB6_PORT.DIRSET = LCD_DB6_PIN_bm;
    LCD_DB6_PORT.OUTCLR = LCD_DB6_PIN_bm;
    LCD_DB5_PORT.DIRSET = LCD_DB5_PIN_bm;
    LCD_DB5_PORT.OUTCLR = LCD_DB5_PIN_bm;
    LCD_DB4_PORT.DIRSET = LCD_DB4_PIN_bm;
    LCD_DB4_PORT.OUTCLR = LCD_DB4_PIN_bm;
    LCD_RS_PORT.DIRSET = LCD_RS_PIN_bm;
    LCD_RS_PORT.OUTCLR = LCD_RS_PIN_bm;
    LCD_RW_PORT.DIRSET = LCD_RW_PIN_bm;
    LCD_RW_PORT.OUTCLR = LCD_RW_PIN_bm;
    LCD_E_PORT.DIRSET = LCD_E_PIN_bm;
    LCD_E_PORT.OUTCLR = LCD_E_PIN_bm;

    // Timer setup
    TCE1.CTRLB = TC_WGMODE_FRQ_gc;
    TCE1.INTCTRLB = TC_CCAINTLVL_LO_gc;
    gLCDState = 0;

    // LCD init sequence

    _delay_ms(40); //Wait >40 msec after power is applied

    // Start out as 8-bit bus

    lcdRSClr();
    lcdNibble((LCD_CMD_FUNCTION_SET | LCD_CMD_FUNCTION_DL_bm) >> 4); //Wake up #1
    _delay_ms(5);
    lcdNibble((LCD_CMD_FUNCTION_SET | LCD_CMD_FUNCTION_DL_bm) >> 4); //Wake up #2
    _delay_us(160);
    lcdNibble((LCD_CMD_FUNCTION_SET | LCD_CMD_FUNCTION_DL_bm) >> 4); //Wake up #3
    _delay_us(160);
    lcdNibble((LCD_CMD_FUNCTION_SET) >> 4); // Tell the LCD to switch to 4-bit bus
    (this command is still 8-bit)
}
```

lcd_driver.c

```
_delay_us(38);

//Now it's a 4-bit bus

lcdCommandBlocking(LCD_CMD_FUNCTION_SET | LCD_CMD_FUNCTION_N_bm); // 2-line LCD,
5x8
lcdCommandBlocking(LCD_CMD_DISPLAY_ON_OFF | LCD_CMD_DISPLAY_ON_OFF_D_bm);
//Display ON
lcdCommandBlocking(LCD_CMD_ENTRY_MODE_SET | LCD_CMD_ENTRY_MODE_I_D_bm); //Cursor
moves right
lcdCommandBlocking(LCD_CMD_CLEAR_DISPLAY); // Last thing to do before writing text
}

ISR(TCE1_CCA_vect)
{
    AVR_ENTER_CRITICAL_REGION();
    switch (gLCDState)
    {
    default:
        break;
    case 1: // Clear E - ADDRLINEx[7:4]
        lcdEClr();
        gLCDState = 2;
        break;
    case 2: // Set E - ADDRLINEx[3:0]
        lcdESet();
        if (gLCDCurrentLine == gLCDLine1) {
            lcdSetPortData((LCD_CMD_SET_DDRAM_ADDR | (LCD_DDRAM_ADDR_bm &
LCD_LINE_1_START_ADDR)) & 0x0F);
        } else {
            lcdSetPortData((LCD_CMD_SET_DDRAM_ADDR | (LCD_DDRAM_ADDR_bm &
LCD_LINE_2_START_ADDR)) & 0x0F);
        }
        gLCDState = 3;
        break;
    case 3: // Clear E - ADDRLINEx[3:0]
        lcdEClr();
        gLCDCharPosition = 0;
        gLCDState = 4;
        setTimerPeriod(152);
        break;
    case 4: // Set E - CHAR[7:4]
        lcdRSSet();
        lcdESet();
        lcdSetPortData(gLCDCurrentLine[gLCDCharPosition] >> 4);
        gLCDState = 5;
        setTimerPeriod(4);
        break;
    case 5: // Clear E - CHAR[7:4]
        lcdEClr();
        gLCDState = 6;
        break;
    case 6: // Set E - CHAR[3:0]
        lcdESet();
        lcdSetPortData(gLCDCurrentLine[gLCDCharPosition] & 0x0F);
        gLCDState = 7;
    }
}
```

lcd_driver.c

```
        break;
    case 7: // Clear E - CHAR[3:0]
        lcdEClr();
        gLCDCharPosition++;
        if (gLCDCharPosition >= LCD_NUM_CHARACTERS) {
            if (gLCDCurrentLine == gLCDLine1) {
                gLCDState = 8;
            } else {
                gLCDState = 9;
            }
        } else {
            gLCDState = 4;
        }
        setTimerPeriod(152);
        break;
    case 8: // Set E - ADDRLINE2[7:4]
        lcdRSClr();
        lcdESet();
        lcdSetPortData((LCD_CMD_SET_DDRAM_ADDR | (LCD_DDRAM_ADDR_bm &
LCD_LINE_2_START_ADDR)) >> 4);
        gLCDState = 1;
        gLCDCurrentLine = gLCDLine2;
        setTimerPeriod(4);
        break;
    case 9: // End
        stopTimer();
        gLCDState = 0;
        break;
    }
    AVR_LEAVE_CRITICAL_REGION();
}
```

linear_actuator.h

```
/*
 * linear_actuator.h
 *
 * Created on: Mar 26, 2011
 * Author: grant
 */

#ifndef LINEAR_ACTUATOR_H_
#define LINEAR_ACTUATOR_H_

void initLinearActuators(void);
int8_t RaisePlatform(void);
int8_t LowerPlatform(void);
int8_t StopPlatform(void);

void PrintLACurrents(void);

//int8_t RaisePlatform(uint8_t ratePercent);
//int8_t LowerPlatform(uint8_t ratePercent);
```

linear_actuator.h

```
#endif /* LINEAR_ACTUATOR_H_ */
```

linear_actuator.c

```
/*
 * linear_actuator.c
 *
 * Created on: Mar 26, 2011
 * Author: grant
 */

#include <avr/io.h>
#include "linear_actuator.h"
#include "../atmel/TC_driver.h"
#include "../atmel/adc_driver.h"
#include "util.h"
#include "stdio.h"
#include "../atmel/pmic_driver.h"

//200 = 6.2us
#define TC_PERIOD 1000

//TODO set up current threshold
#define CURRENT_THRESHOLD_MAX 150
#define CURRENT_THRESHOLD_MIN -150

static int8_t adcb_offset0, adcb_offset1, adcb_offset2, adcb_offset3;
static int16_t adc_result0, adc_result1, adc_result2, adc_result3;
static uint8_t OVERCURRENT_FLAG;

static void setTop(void) {
    PORTE.OUTSET = PIN0_bm | PIN1_bm | PIN3_bm | PIN4_bm;
}

static void clrTop(void) {
    PORTE.OUTCLR = PIN0_bm | PIN1_bm | PIN3_bm | PIN4_bm;
}

static void setBottom(void) {
    PORTE.OUTSET = PIN5_bm;
    PORTF.OUTSET = PIN0_bm | PIN1_bm | PIN3_bm;
}

static void clrBottom(void) {
    PORTE.OUTCLR = PIN5_bm;
    PORTF.OUTCLR = PIN0_bm | PIN1_bm | PIN3_bm;
}

void initLinearActuators(void)
{
    //turn off timers
    TC0_ConfigClockSource( &TCE0, TC_CLKSEL_OFF_gc );

    //Enable output
```

linear_actuator.c

```
clrTop();
clrBottom();
PORTE.DIRSET = PIN0_bm | PIN1_bm | PIN3_bm | PIN4_bm | PIN5_bm;
PORTF.DIRSET = PIN0_bm | PIN1_bm | PIN3_bm;

/* Set the TC period.
 * 1000 at 32MHz is 32kHz, above human hearing range*/
TC_SetPeriod( &TCE0, TC_PERIOD ); // Timer/Counter E0

/* Configure the TC for single slope mode. */
TC0_ConfigWGM( &TCE0, TC_WGMODE_NORMAL_gc );

//set overflow interrupt
TC0_SetOverflowIntLevel( &TCE0, TC_OVFINTLVL_MED_gc);

// enable interrupt level
PMIC.CTRL |= PMIC_MEDLVLEN_bm;

/***** ADC CONFIG *****/
ADC_CalibrationValues_Load(&ADCB);

/* Set up ADC B to have signed conversion mode and 12 bit resolution. */
ADC_ConvMode_and_Resolution_Config(&ADCB, ADC_ConvMode_Signed,
ADC_RESOLUTION_12BIT_gc);

// The ADC has different voltage reference options, controlled by the REFSEL bits
in the
// REFCTRL register. Here an external reference is selected
ADC_Reference_Config(&ADCB, ADC_REFSEL_AREFA_gc);

// The clock into the ADC decide the maximum sample rate and the conversion time,
and
// this is controlled by the PRESCALER bits in the PRESCALER register. Here, the
// Peripheral Clock is divided by 512 ( gives 62.5 KSPS with 32Mhz clock )
ADC_Prescaler_Config(&ADCB, ADC_PRESCALER_DIV512_gc);

/* Setup channels*/
ADC_Ch_InputMode_and_Gain_Config(&ADCB.CH0, ADC_CH_INPUTMODE_DIFF_gc,
ADC_DRIVER_CH_GAIN_NONE);
ADC_Ch_InputMode_and_Gain_Config(&ADCB.CH1, ADC_CH_INPUTMODE_DIFF_gc,
ADC_DRIVER_CH_GAIN_NONE);
ADC_Ch_InputMode_and_Gain_Config(&ADCB.CH2, ADC_CH_INPUTMODE_DIFF_gc,
ADC_DRIVER_CH_GAIN_NONE);
ADC_Ch_InputMode_and_Gain_Config(&ADCB.CH3, ADC_CH_INPUTMODE_DIFF_gc,
ADC_DRIVER_CH_GAIN_NONE);

// Setting up the which pins to convert.
ADC_Ch_InputMux_Config(&ADCB.CH0, ADC_CH_MUXPOS_PIN4_gc, ADC_CH_MUXNEG_PIN0_gc);
ADC_Ch_InputMux_Config(&ADCB.CH1, ADC_CH_MUXPOS_PIN5_gc, ADC_CH_MUXNEG_PIN1_gc);
ADC_Ch_InputMux_Config(&ADCB.CH2, ADC_CH_MUXPOS_PIN6_gc, ADC_CH_MUXNEG_PIN2_gc);
ADC_Ch_InputMux_Config(&ADCB.CH3, ADC_CH_MUXPOS_PIN7_gc, ADC_CH_MUXNEG_PIN3_gc);

// Setup Interrupt Mode on complete
// ADC_Ch_Interrupts_Config(&ADCB.CH0, ADC_CH_INTMODE_COMPLETE_gc,
ADC_CH_INTLVL_MED_gc);
// ADC_Ch_Interrupts_Config(&ADCB.CH1, ADC_CH_INTMODE_COMPLETE_gc,
```

linear_actuator.c

```
ADC_CH_INTLVL_MED_gc);
// ADC_Ch_Interrupts_Config(&ADCB.CH2, ADC_CH_INTMODE_COMPLETE_gc,
ADC_CH_INTLVL_MED_gc);
// ADC_Ch_Interrupts_Config(&ADCB.CH3, ADC_CH_INTMODE_COMPLETE_gc,
ADC_CH_INTLVL_MED_gc);

// Enable PMIC interrupt level
PMIC_CTRL |= PMIC_MEDLVLEN_bm;

// Setup sweep of all 4 virtual channels.
ADC_SweepChannels_Config(&ADCB, ADC_SWEEP_0123_gc);

// Before the ADC can be used it must be enabled
ADC_Enable(&ADCB);

// Wait until the ADC is ready
ADC_Wait_32MHz(&ADCB);

/* Get offset value for ADC B. */
OVERCURRENT_FLAG = 0;
adc_result0 = adc_result1 = adc_result2 = adc_result3 = 0;
adcb_offset0 = adcb_offset1 = adcb_offset2 = adcb_offset3 = 0;
// adcb_offset0 = ADC_Offset_Get_Signed(&ADCB, &(ADCB.CH0), true);
// adcb_offset1 = ADC_Offset_Get_Signed(&ADCB, &(ADCB.CH1), true);
// adcb_offset2 = ADC_Offset_Get_Signed(&ADCB, &(ADCB.CH2), true);
// adcb_offset3 = ADC_Offset_Get_Signed(&ADCB, &(ADCB.CH3), true);
// printf("offsets:%d\t%d\t%d\t%d\n\r", adcb_offset0, adcb_offset1, adcb_offset2,
adcb_offset3);

//start single conversion
// ADC_Ch_Conversion_Start(&ADCB.CH0);

//enable free running mode
ADC_FreeRunning_Enable(&ADCB);
}

int8_t LowerPlatform(void)
{
    int8_t err = 0;

    if(OVERCURRENT_FLAG) {
        return -1;
    }

    //turn off bottom fets
    clrBottom();

    //turn on top fets and start timer to recharge bootstrap
    // TCE0.CNT = 0;
    setTop();
    TC0_ConfigClockSource( &TCE0, TC_CLKSEL_DIV1_gc );

    return err;
}

// Recharge Bootstrap Cap
```

linear_actuator.c

```
// necessary for the LT1160 to turn on the top mosfet
ISR(TCE0_OVF_vect)
{
    //turn off top fets
    clrTop();

    //turn on bottom fets
    setBottom();

    //wait a little bit for cap to charge
    asm volatile ("nop"); //31.25ns delay
    asm volatile ("nop"); //31.25ns delay

    //turn off bottom fets
    clrBottom();

    //turn on top fets
    setTop();
}

int8_t RaisePlatform(void)
{
    int8_t err = 0;

    if(OVERCURRENT_FLAG) {
        return -1;
    }

    //turn off raise platform pins and timer
    TC0_ConfigClockSource( &TCE0, TC_CLKSEL_OFF_gc );
    clrTop();

    //turn on lower platform pins
    setBottom();

    return err;
}

int8_t StopPlatform(void)
{
    int8_t err = 0;

    TC0_ConfigClockSource( &TCE0, TC_CLKSEL_OFF_gc );

    clrTop();
    clrBottom();

    return err;
}

//Returns 0 for no overcurrent, else bitwise mask for which linear actuator is over
//currenting
// 0x01, 0x02, 0x04, 0x08
/*
static int8_t isOverCurrent(void)
{

```

linear_actuator.c

```
        return OVERCURRENT_FLAG;
    }
    */

void PrintLACurrents(void)
{
    printf("1:%5d 2:%5d 3:%5d 4:%5d F:%d\n\r", adc_result0, adc_result1, adc_result2,
adc_result3, OVERCURRENT_FLAG);
}

static void checkForOverCurrent(int16_t adc_result, uint8_t LAnum)
{
    if(adc_result > CURRENT_THRESHOLD_MAX || adc_result < CURRENT_THRESHOLD_MIN) {
        StopPlatform();
        OVERCURRENT_FLAG = 1<<LAnum;
    } else {
        OVERCURRENT_FLAG = 0;
    }
}

ISR(ADCB_CH0_vect)
{
    //   dbgLEDTgl();
    //int16_t adc_result0;
    adc_result0 = ADC_ResultCh_GetWord_Signed(&ADCB.CH0, adcb_offset0);
    checkForOverCurrent(adc_result0, 0);
}

ISR(ADCB_CH1_vect)
{
    //   dbgLEDTgl();
    //int16_t adc_result1;
    adc_result1 = ADC_ResultCh_GetWord_Signed(&ADCB.CH1, adcb_offset1);
    checkForOverCurrent(adc_result1, 1);
}

ISR(ADCB_CH2_vect)
{
    //   dbgLEDTgl();
    //int16_t adc_result2;
    adc_result2 = ADC_ResultCh_GetWord_Signed(&ADCB.CH2, adcb_offset2);
    checkForOverCurrent(adc_result2, 2);
}

ISR(ADCB_CH3_vect)
{
    //   dbgLEDTgl();
    //int16_t adc_result3;
    adc_result3 = ADC_ResultCh_GetWord_Signed(&ADCB.CH3, adcb_offset3);
    checkForOverCurrent(adc_result3, 3);
}
```


menu.h

```
/*
 * menu.h
 *
 * Created: 8/16/2012 10:19:43 AM
 * Author: Stew
 */

#ifndef MENU_H_
#define MENU_H_

void menuInit();
uint8_t menuGetMotorsDisabled();
void menuUpdate(int16_t speed, int16_t dir);
void menuPlatformDownPushed();
void menuPlatformUpPushed();
uint8_t menuGetIsPlatformDown();
float menuGetFwdThrow(uint8_t overridden);
float menuGetRevThrow(uint8_t overridden);
float menuGetTurnThrow(uint8_t overridden);
uint8_t menuGetTopFwdSpeed(uint8_t overridden);
uint8_t menuGetTopRevSpeed(uint8_t overridden);
uint8_t menuGetTopTurnSpeed(uint8_t overridden);
double menuGetSensitivity(uint8_t overridden);
uint8_t menuGetAcceleration(uint8_t overridden);
uint8_t menuGetDeceleration(uint8_t overridden);
uint8_t menuGetOuterDeadBand(uint8_t overridden);
uint8_t menuGetCenterDeadBand(uint8_t overridden);
uint8_t menuGetPropAsSwitch(uint8_t overridden);
uint8_t menuGetInvert(uint8_t overridden);
void incrementWirelessTimeout();

#endif /* MENU_H_ */
```

menu.c

```
/*
 * menu.c
 *
 * Created: 8/16/2012 10:19:14 AM
 * Author: Stew
 */

#include <stdio.h>
#include <string.h>
#include <avr/eeprom.h>
#include "../atmel/avr_compiler.h"
#include "../atmel/wdt_driver.h"
#include "menu.h"
#include "PWCT_io.h"
#include "lcd_driver.h"
#include "motor_driver.h"

uint8_t gWirelessTimeoutCount = 0;
uint8_t gMotorsDisabled = 0;
```

menu.c

```
uint8_t gNameEditMode = 0;

// Define the order of the menu options, zero-based
#define MENU_OPTION_PROFILE 0
#define MENU_OPTION_FWD_THROW 1
#define MENU_OPTION_REV_THROW 2
#define MENU_OPTION_TURN_THROW 3
#define MENU_OPTION_TOP_FWD_SPEED 4
#define MENU_OPTION_TOP_REV_SPEED 5
#define MENU_OPTION_TOP_TURN_SPEED 6
#define MENU_OPTION_SENSITIVITY 7
#define MENU_OPTION_ACCELERATION 8
#define MENU_OPTION_DECELERATION 9
#define MENU_OPTION_CTR_DEAD_BAND 10
#define MENU_OPTION_OUTER_DEAD_BAND 11
#define MENU_OPTION_INVERT 12
#define MENU_OPTION_PROP_AS_SWITCH 13
// We must know how many menu options there are
#define LAST_MENU_OPTION MENU_OPTION_PROP_AS_SWITCH

// EEPROM variables, RAM shadow variables, and sane initial values
// Note: the initial values are only updated when programming the EEPROM memory
(wheelchair.eep)

#define PROFILE_COUNT 21

uint8_t EEMEM eepromIsPlatformDown = 0;
uint8_t eepromShadowIsPlatformDown = 0;

uint8_t EEMEM eepromCurrentProfile = 0;
uint8_t eepromShadowCurrentProfile = 0;

uint8_t EEMEM eepromMenuState = 0;
uint8_t eepromShadowMenuState = 0;

char EEMEM eepromProfileName[PROFILE_COUNT][LCD_NUM_CHARACTERS+1] = {"Profile 1",
"Profile 2", "Profile 3", "Profile 4", "Profile 5",
"Profile 6", "Profile 7", "Profile 8", "Profile 9", "Profile 10", "Profile 11",
"Profile 12", "Profile 13",
"Profile 14", "Profile 15", "Profile 16", "Profile 17", "Profile 18", "Profile
19", "Profile 20", "Remote override"};
char currentProfileName[LCD_NUM_CHARACTERS+1];

float EEMEM eepromFwdThrow[PROFILE_COUNT] = {1.0, 1.0, 2.0, 1.0, 1.0, 0.65, 0.75, 1.0,
1.05, 1.0, 0.8, 1.0, 1.05, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.5, 0.8};
float eepromShadowFwdThrow = 1.0;
float eepromShadowOverrideFwdThrow = 0.8;

float EEMEM eepromRevThrow[PROFILE_COUNT] = {0.75, 0.8, 0.8, 0.8, 0.8, 0.8, 0.75, 0.8,
0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 1.0, 0.6};
float eepromShadowRevThrow = 0.8;
float eepromShadowOverrideRevThrow = 0.6;

float EEMEM eepromTurnThrow[PROFILE_COUNT] = {0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.85, 0.6,
0.55, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.8, 0.4};
float eepromShadowTurnThrow = 0.6;
```

menu.c

```
float eepromShadowOverrideTurnThrow = 0.4;

uint8_t EEMEM eepromTopFwdSpeed[PROFILE_COUNT] = {40, 40, 35, 35, 45, 90, 15, 35, 45, 40,
45, 25, 50, 50, 50, 50, 50, 50, 125, 50};
uint8_t eepromShadowTopFwdSpeed = 50;
uint8_t eepromShadowOverrideTopFwdSpeed = 50;

uint8_t EEMEM eepromTopRevSpeed[PROFILE_COUNT] = {25, 35, 30, 20, 20, 35, 40, 35, 25, 35,
35, 35, 30, 35, 35, 35, 35, 35, 50, 35};
uint8_t eepromShadowTopRevSpeed = 35;
uint8_t eepromShadowOverrideTopRevSpeed = 35;

uint8_t EEMEM eepromTopTurnSpeed[PROFILE_COUNT] = {25, 25, 45, 30, 35, 45, 25, 35, 35,
30, 25, 35, 25, 30, 20, 20, 20, 20, 20, 35, 35};
uint8_t eepromShadowTopTurnSpeed = 20;
uint8_t eepromShadowOverrideTopTurnSpeed = 35;

uint8_t EEMEM eepromSensitivity[PROFILE_COUNT] = {9, 7, 4, 8, 5, 8, 8, 7, 6, 5, 6, 6, 7,
9, 3, 3, 3, 3, 3, 7, 7};
uint8_t eepromShadowSensitivity = 3;
uint8_t eepromShadowOverrideSensitivity = 7;
static const double gSensitivityMap[10] = {0.0001, 0.000167, 0.000278, 0.000463,
0.000772, 0.00129, 0.00214, 0.00357, 0.01, 0.5};

uint8_t EEMEM eepromAcceleration[PROFILE_COUNT] = {16, 8, 16, 8, 4, 4, 4, 8, 8, 16, 4,
16, 16, 16, 16, 16, 16, 16, 16, 16, 16};
uint8_t eepromShadowAcceleration = 16;
uint8_t eepromShadowOverrideAcceleration = 16;

uint8_t EEMEM eepromDeceleration[PROFILE_COUNT] = {12, 12, 12, 12, 12, 12, 12, 12, 8, 12,
12, 12, 12, 12, 12, 12, 12, 12, 12, 12};
uint8_t eepromShadowDeceleration = 12;
uint8_t eepromShadowOverrideDeceleration = 12;

uint8_t EEMEM eepromOuterDeadBand[PROFILE_COUNT] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0};
uint8_t eepromShadowOuterDeadBand = 0;
uint8_t eepromShadowOverrideOuterDeadBand = 0;

uint8_t EEMEM eepromCenterDeadBand[PROFILE_COUNT] = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2};
uint8_t eepromShadowCenterDeadBand = 2;
uint8_t eepromShadowOverrideCenterDeadBand = 2;

uint8_t EEMEM eepromPropAsSwitch[PROFILE_COUNT] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0};
uint8_t eepromShadowPropAsSwitch = 0;
uint8_t eepromShadowOverridePropAsSwitch = 0;

uint8_t EEMEM eepromInvert[PROFILE_COUNT] = {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0};
uint8_t eepromShadowInvert = 0;
uint8_t eepromShadowOverrideInvert = 0;

static void eepromCorrupt()
{
```

menu.c

```
motorEStop();
while (1)
{
    lcdText("EEPROM corrupt", "Ver. " __DATE__, 0);
    WDT_Reset();
}

void menuInit()
{
    // Initialize all shadow variables
    eepromShadowIsPlatformDown = eeprom_read_byte(&eepromIsPlatformDown);
    eepromShadowCurrentProfile = eeprom_read_byte(&eepromCurrentProfile);
    if (eepromShadowCurrentProfile >= PROFILE_COUNT) {
        eepromCorrupt();
    }
    eepromShadowMenuState = eeprom_read_byte(&eepromMenuState);
    eeprom_read_block(currentProfileName,
eepromProfileName[eepromShadowCurrentProfile], LCD_NUM_CHARACTERS+1);

    eepromShadowFwdThrow =
eeprom_read_float(&eepromFwdThrow[eepromShadowCurrentProfile]);
    eepromShadowOverrideFwdThrow = eeprom_read_float(&eepromFwdThrow[PROFILE_COUNT-
1]);

    eepromShadowRevThrow =
eeprom_read_float(&eepromRevThrow[eepromShadowCurrentProfile]);
    eepromShadowOverrideRevThrow = eeprom_read_float(&eepromRevThrow[PROFILE_COUNT-
1]);

    eepromShadowTurnThrow =
eeprom_read_float(&eepromTurnThrow[eepromShadowCurrentProfile]);
    eepromShadowOverrideTurnThrow = eeprom_read_float(&eepromTurnThrow[PROFILE_COUNT-
1]);

    eepromShadowTopFwdSpeed =
eeprom_read_byte(&eepromTopFwdSpeed[eepromShadowCurrentProfile]);
    eepromShadowOverrideTopFwdSpeed =
eeprom_read_byte(&eepromTopFwdSpeed[PROFILE_COUNT-1]);

    eepromShadowTopRevSpeed =
eeprom_read_byte(&eepromTopRevSpeed[eepromShadowCurrentProfile]);
    eepromShadowOverrideTopRevSpeed =
eeprom_read_byte(&eepromTopRevSpeed[PROFILE_COUNT-1]);

    eepromShadowTopTurnSpeed =
eeprom_read_byte(&eepromTopTurnSpeed[eepromShadowCurrentProfile]);
    eepromShadowOverrideTopTurnSpeed =
eeprom_read_byte(&eepromTopTurnSpeed[PROFILE_COUNT-1]);

    eepromShadowSensitivity =
eeprom_read_byte(&eepromSensitivity[eepromShadowCurrentProfile]);
    if (eepromShadowSensitivity >= 10) {
        eepromCorrupt();
    }
    eepromShadowOverrideSensitivity =
```

menu.c

```
    eeprom_read_byte(&eepromSensitivity[PROFILE_COUNT-1]);

    eepromShadowAcceleration =
eeprom_read_byte(&eepromAcceleration[eepromShadowCurrentProfile]);
    eepromShadowOverrideAcceleration =
eeprom_read_byte(&eepromAcceleration[PROFILE_COUNT-1]);

    eepromShadowDeceleration =
eeprom_read_byte(&eepromDeceleration[eepromShadowCurrentProfile]);
    eepromShadowOverrideDeceleration =
eeprom_read_byte(&eepromDeceleration[PROFILE_COUNT-1]);

    eepromShadowOuterDeadBand =
eeprom_read_byte(&eepromOuterDeadBand[eepromShadowCurrentProfile]);
    eepromShadowOverrideOuterDeadBand =
eeprom_read_byte(&eepromOuterDeadBand[PROFILE_COUNT-1]);

    eepromShadowCenterDeadBand =
eeprom_read_byte(&eepromCenterDeadBand[eepromShadowCurrentProfile]);
    eepromShadowOverrideCenterDeadBand =
eeprom_read_byte(&eepromCenterDeadBand[PROFILE_COUNT-1]);

    eepromShadowPropAsSwitch =
eeprom_read_byte(&eepromPropAsSwitch[eepromShadowCurrentProfile]);
    eepromShadowOverridePropAsSwitch =
eeprom_read_byte(&eepromPropAsSwitch[PROFILE_COUNT-1]);

    eepromShadowInvert = eeprom_read_byte(&eepromInvert[eepromShadowCurrentProfile]);
    eepromShadowOverrideInvert = eeprom_read_byte(&eepromInvert[PROFILE_COUNT-1]);
}

void incrementWirelessTimeout()
{
    gWirelessTimeoutCount++;
}

uint8_t menuGetMotorsDisabled()
{
    return gMotorsDisabled;
}

static void eepromUpdateByteSafe(uint8_t *eepromVariable, uint8_t *shadowVariable,
uint8_t newValue)
{
    uint8_t readValue;
    eeprom_busy_wait();
    eeprom_update_byte(eepromVariable, newValue);
    printf("EEPROM written\n");
    eeprom_busy_wait();
    readValue = eeprom_read_byte(eepromVariable);
    if (readValue != newValue) {
        eepromCorrupt();
    } else {
        *shadowVariable = newValue;
    }
}
}
```

menu.c

```
static void eepromUpdateFloatSafe(float *eepromVariable, float *shadowVariable, float
newValue)
{
    float readValue;
    eeprom_busy_wait();
    eeprom_update_float(eepromVariable, newValue);
    printf("EEPROM written\n");
    eeprom_busy_wait();
    readValue = eeprom_read_float(eepromVariable);
    if (readValue != newValue) {
        eepromCorrupt();
    } else {
        *shadowVariable = newValue;
    }
}

static void eepromUpdateStringSafe(char *src, char *eepromDst)
{
    char readString[LCD_NUM_CHARACTERS+1];
    eeprom_busy_wait();
    eeprom_update_block(src, eepromDst, LCD_NUM_CHARACTERS+1);
    printf("EEPROM written\n");
    eeprom_busy_wait();
    eeprom_read_block(readString, eepromDst, LCD_NUM_CHARACTERS+1);
    if (strcmp(readString, src)) {
        eepromCorrupt();
    }
}

void menuPlatformDownPushed() {
    if (!eepromShadowIsPlatformDown) {
        eepromUpdateByteSafe(&eepromIsPlatformDown, &eepromShadowIsPlatformDown,
1);
    }
}

void menuPlatformUpPushed() {
    if (eepromShadowIsPlatformDown) {
        eepromUpdateByteSafe(&eepromIsPlatformDown, &eepromShadowIsPlatformDown,
0);
    }
}

uint8_t menuGetIsPlatformDown() {
    return eepromShadowIsPlatformDown;
}

float menuGetFwdThrow(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideFwdThrow;
    else
        return eepromShadowFwdThrow;
}
```

menu.c

```
float menuGetRevThrow(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideRevThrow;
    else
        return eepromShadowRevThrow;
}

float menuGetTurnThrow(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideTurnThrow;
    else
        return eepromShadowTurnThrow;
}

uint8_t menuGetTopFwdSpeed(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideTopFwdSpeed;
    else
        return eepromShadowTopFwdSpeed;
}

uint8_t menuGetTopRevSpeed(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideTopRevSpeed;
    else
        return eepromShadowTopRevSpeed;
}

uint8_t menuGetTopTurnSpeed(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideTopTurnSpeed;
    else
        return eepromShadowTopTurnSpeed;
}

double menuGetSensitivity(uint8_t overridden)
{
    if (overridden)
        return gSensitivityMap[eepromShadowOverrideSensitivity];
    else
        return gSensitivityMap[eepromShadowSensitivity];
}

uint8_t menuGetAcceleration(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideAcceleration;
    else
        return eepromShadowAcceleration;
}
```

menu.c

```
uint8_t menuGetDeceleration(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideDeceleration;
    else
        return eepromShadowDeceleration;
}

uint8_t menuGetOuterDeadBand(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideOuterDeadBand;
    else
        return eepromShadowOuterDeadBand;
}

uint8_t menuGetCenterDeadBand(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideCenterDeadBand;
    else
        return eepromShadowCenterDeadBand;
}

uint8_t menuGetPropAsSwitch(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverridePropAsSwitch;
    else
        return eepromShadowPropAsSwitch;
}

uint8_t menuGetInvert(uint8_t overridden)
{
    if (overridden)
        return eepromShadowOverrideInvert;
    else
        return eepromShadowInvert;
}

uint8_t isValidChar(char c)
{
    uint8_t isValid = 0;
    if (c == ' ') {
        isValid = 1;
    }
    if (c >= '0' && c <= '9') {
        isValid = 2;
    }
    if (c >= 'A' && c <= 'Z') {
        isValid = 3;
    }
    if (c >= 'a' && c <= 'z') {
        isValid = 4;
    }
    return isValid;
}
```


menu.c

```
}

void menuUpdate(int16_t speed, int16_t dir)
{
    uint8_t up = lcdUpFallingEdge();
    uint8_t down = lcdDownFallingEdge();
    uint8_t right = lcdRightFallingEdge();
    uint8_t left = lcdLeftFallingEdge();
    uint8_t leftLongPress = lcdLeftLongPress();
    static uint8_t cursorPosition = 0;

    char lcdLine1[LCD_NUM_CHARACTERS+1];
    char lcdLine2[LCD_NUM_CHARACTERS+1];
    lcdLine1[LCD_NUM_CHARACTERS] = '\0';
    lcdLine2[LCD_NUM_CHARACTERS] = '\0';

    if (eepromShadowIsPlatformDown) {
        sprintf(lcdLine1, "Platform down");
        lcdLine2[0] = '\0';
        lcdText(lcdLine1, lcdLine2, 0);
        gMotorsDisabled = 1;
        return;
    }

    if (eepromShadowMenuState == MENU_OPTION_PROFILE && leftLongPress) {
        gNameEditMode = !gNameEditMode;

        if (gNameEditMode) {
            sprintf(lcdLine1, "Edit name");
            sprintf(lcdLine2, "%s", currentProfileName);
            lcdText(lcdLine1, lcdLine2, 1);

            // Display on, LCD cursor on, blink off
            lcdCommandBlocking(LCD_CMD_DISPLAY_ON_OFF |
LCD_CMD_DISPLAY_ON_OFF_D_bm | LCD_CMD_DISPLAY_ON_OFF_C_bm);
            lcdCommandBlocking(LCD_CMD_SET_DDRAM_ADDR | (LCD_DDRAM_ADDR_bm &
LCD_LINE_2_START_ADDR));
            cursorPosition = 0;
        } else {
            // Display on, LCD cursor off, blink off
            lcdCommandBlocking(LCD_CMD_DISPLAY_ON_OFF |
LCD_CMD_DISPLAY_ON_OFF_D_bm);

            // Write to eeprom
            eepromUpdateStringSafe(currentProfileName,
eepromProfileName[eepromShadowCurrentProfile]);
        }
    }

    if (gNameEditMode) {
        gMotorsDisabled = 1;
        if (!isValidChar(currentProfileName[cursorPosition])) {
            currentProfileName[cursorPosition] = ' ';
            currentProfileName[cursorPosition+1] = '\0';
        }
        if (left || right) {
```

menu.c

```
if (left && cursorPosition > 0) {
    cursorPosition--;
}
if (right && cursorPosition < LCD_NUM_CHARACTERS - 1) {
    cursorPosition++;
    if (!isValidChar(currentProfileName[cursorPosition])) {
        currentProfileName[cursorPosition] = ' ';
        currentProfileName[cursorPosition+1] = '\\0';
    }
}
// Change cursor position
lcdCommandBlocking((LCD_CMD_SET_DDRAM_ADDR | (LCD_DDRAM_ADDR_bm &
LCD_LINE_2_START_ADDR)) + cursorPosition);
}
if (up || down) {
    // Change letter
    if (up) {
        if (currentProfileName[cursorPosition] == ' ') {
            currentProfileName[cursorPosition] = 'A';
        } else if (currentProfileName[cursorPosition] == 'Z') {
            currentProfileName[cursorPosition] = 'a';
        } else if (currentProfileName[cursorPosition] == 'z') {
            currentProfileName[cursorPosition] = '0';
        } else if (currentProfileName[cursorPosition] == '9') {
            currentProfileName[cursorPosition] = ' ';
        } else {
            currentProfileName[cursorPosition]++;
        }
    }
    if (down) {
        if (currentProfileName[cursorPosition] == ' ') {
            currentProfileName[cursorPosition] = '9';
        } else if (currentProfileName[cursorPosition] == '0') {
            currentProfileName[cursorPosition] = 'z';
        } else if (currentProfileName[cursorPosition] == 'a') {
            currentProfileName[cursorPosition] = 'Z';
        } else if (currentProfileName[cursorPosition] == 'Z') {
            currentProfileName[cursorPosition] = 'A';
        } else {
            currentProfileName[cursorPosition]--;
        }
    }
}
sprintf(lcdLine1, "Edit name");
sprintf(lcdLine2, "%s", currentProfileName);
lcdText(lcdLine1, lcdLine2, 1);
lcdCommandBlocking((LCD_CMD_SET_DDRAM_ADDR | (LCD_DDRAM_ADDR_bm &
LCD_LINE_2_START_ADDR)) + cursorPosition);
}
return;
}

gMotorsDisabled = 0;

if (right)
{
    if (eepromShadowMenuState < LAST_MENU_OPTION) {
```

menu.c

```
        eepromUpdateByteSafe(&eepromMenuState, &eepromShadowMenuState,
eepromShadowMenuState + 1);
    }
    if (left)
    {
        if (eepromShadowMenuState > 0) {
            eepromUpdateByteSafe(&eepromMenuState, &eepromShadowMenuState,
eepromShadowMenuState - 1);
        }
    }

    switch (eepromShadowMenuState)
    {
    case MENU_OPTION_PROFILE:
        if (up && eepromShadowCurrentProfile < PROFILE_COUNT - 1) {
            eepromUpdateByteSafe(&eepromCurrentProfile,
&eepromShadowCurrentProfile, eepromShadowCurrentProfile + 1);
            menuInit();
        }
        if (down && eepromShadowCurrentProfile > 0) {
            eepromUpdateByteSafe(&eepromCurrentProfile,
&eepromShadowCurrentProfile, eepromShadowCurrentProfile - 1);
            menuInit();
        }
        sprintf(lcdLine1, "Choose Profile");
        sprintf(lcdLine2, "%s", currentProfileName);
        break;
    case MENU_OPTION_FWD_THROW:
        if (up && eepromShadowFwdThrow < 2.45) {
            eepromUpdateFloatSafe(&eepromFwdThrow[eepromShadowCurrentProfile],
&eepromShadowFwdThrow, eepromShadowFwdThrow + 0.05);
        }
        if (down && eepromShadowFwdThrow > 0.05) {
            eepromUpdateFloatSafe(&eepromFwdThrow[eepromShadowCurrentProfile],
&eepromShadowFwdThrow, eepromShadowFwdThrow - 0.05);
        }
        sprintf(lcdLine1, "Fwd Throw: %.2f", (double)eepromShadowFwdThrow);
        sprintf(lcdLine2, "%s", currentProfileName);
        break;
    case MENU_OPTION_REV_THROW:
        if (up && eepromShadowRevThrow < 2.45) {
            eepromUpdateFloatSafe(&eepromRevThrow[eepromShadowCurrentProfile],
&eepromShadowRevThrow, eepromShadowRevThrow + 0.05);
        }
        if (down && eepromShadowRevThrow > 0.05) {
            eepromUpdateFloatSafe(&eepromRevThrow[eepromShadowCurrentProfile],
&eepromShadowRevThrow, eepromShadowRevThrow - 0.05);
        }
        sprintf(lcdLine1, "Rev Throw: %.2f", (double)eepromShadowRevThrow);
        sprintf(lcdLine2, "%s", currentProfileName);
        break;
    case MENU_OPTION_TURN_THROW:
        if (up && eepromShadowTurnThrow < 2.45) {
            eepromUpdateFloatSafe(&eepromTurnThrow[eepromShadowCurrentProfile],
&eepromShadowTurnThrow, eepromShadowTurnThrow + 0.05);
```

menu.c

```
    }
    if (down && eepromShadowTurnThrow > 0.05) {
        eepromUpdateFloatSafe(&eepromTurnThrow[eepromShadowCurrentProfile],
&eepromShadowTurnThrow, eepromShadowTurnThrow - 0.05);
    }
    sprintf(lcdLine1, "Turn Throw: %.2f", (double)eepromShadowTurnThrow);
    sprintf(lcdLine2, "%s", currentProfileName);
    break;
case MENU_OPTION_TOP_FWD_SPEED:
    if (up && eepromShadowTopFwdSpeed < 125) {
        eepromUpdateByteSafe(&eepromTopFwdSpeed[eepromShadowCurrentProfile],
&eepromShadowTopFwdSpeed, eepromShadowTopFwdSpeed + 5);
    }
    if (down && eepromShadowTopFwdSpeed > 5) {
        eepromUpdateByteSafe(&eepromTopFwdSpeed[eepromShadowCurrentProfile],
&eepromShadowTopFwdSpeed, eepromShadowTopFwdSpeed - 5);
    }
    sprintf(lcdLine1, "Fwd Speed: %d", eepromShadowTopFwdSpeed);
    sprintf(lcdLine2, "%s", currentProfileName);
    break;
case MENU_OPTION_TOP_REV_SPEED:
    if (up && eepromShadowTopRevSpeed < 125) {
        eepromUpdateByteSafe(&eepromTopRevSpeed[eepromShadowCurrentProfile],
&eepromShadowTopRevSpeed, eepromShadowTopRevSpeed + 5);
    }
    if (down && eepromShadowTopRevSpeed > 5) {
        eepromUpdateByteSafe(&eepromTopRevSpeed[eepromShadowCurrentProfile],
&eepromShadowTopRevSpeed, eepromShadowTopRevSpeed - 5);
    }
    sprintf(lcdLine1, "Rev Speed: %d", eepromShadowTopRevSpeed);
    sprintf(lcdLine2, "%s", currentProfileName);
    break;
case MENU_OPTION_TOP_TURN_SPEED:
    if (up && eepromShadowTopTurnSpeed < 125) {

        eepromUpdateByteSafe(&eepromTopTurnSpeed[eepromShadowCurrentProfile],
&eepromShadowTopTurnSpeed, eepromShadowTopTurnSpeed + 5);
    }
    if (down && eepromShadowTopTurnSpeed > 5) {

        eepromUpdateByteSafe(&eepromTopTurnSpeed[eepromShadowCurrentProfile],
&eepromShadowTopTurnSpeed, eepromShadowTopTurnSpeed - 5);
    }
    sprintf(lcdLine1, "Turn Speed: %d", eepromShadowTopTurnSpeed);
    sprintf(lcdLine2, "%s", currentProfileName);
    break;
case MENU_OPTION_SENSITIVITY:
    if (up && eepromShadowSensitivity < 9) {
        eepromUpdateByteSafe(&eepromSensitivity[eepromShadowCurrentProfile],
&eepromShadowSensitivity, eepromShadowSensitivity + 1);
    }
    if (down && eepromShadowSensitivity > 0) {
        eepromUpdateByteSafe(&eepromSensitivity[eepromShadowCurrentProfile],
&eepromShadowSensitivity, eepromShadowSensitivity - 1);
    }
    sprintf(lcdLine1, "Sensitivity: %d", eepromShadowSensitivity + 1);
```

menu.c

```
    sprintf(lcdLine2, "%s", currentProfileName);
    break;
case MENU_OPTION_ACCELERATION:
    if (up && eepromShadowAcceleration > 4) {

        eepromUpdateByteSafe(&eepromAcceleration[eepromShadowCurrentProfile],
&eepromShadowAcceleration, eepromShadowAcceleration - 4);
        }
        if (down && eepromShadowAcceleration < 100) {

            eepromUpdateByteSafe(&eepromAcceleration[eepromShadowCurrentProfile],
&eepromShadowAcceleration, eepromShadowAcceleration + 4);
            }
            sprintf(lcdLine1, "Acceleration: %d", (104 - eepromShadowAcceleration) /
4);
            sprintf(lcdLine2, "%s", currentProfileName);
            break;
case MENU_OPTION_DECELERATION:
    if (up && eepromShadowDeceleration > 4) {

        eepromUpdateByteSafe(&eepromDeceleration[eepromShadowCurrentProfile],
&eepromShadowDeceleration, eepromShadowDeceleration - 4);
        }
        if (down && eepromShadowDeceleration < 100) {

            eepromUpdateByteSafe(&eepromDeceleration[eepromShadowCurrentProfile],
&eepromShadowDeceleration, eepromShadowDeceleration + 4);
            }
            sprintf(lcdLine1, "Deceleration: %d", (104 - eepromShadowDeceleration) /
4);
            sprintf(lcdLine2, "%s", currentProfileName);
            break;
case MENU_OPTION_OUTER_DEAD_BAND:
    if (up && eepromShadowOuterDeadBand < 20) {

        eepromUpdateByteSafe(&eepromOuterDeadBand[eepromShadowCurrentProfile],
&eepromShadowOuterDeadBand, eepromShadowOuterDeadBand + 1);
        }
        if (down && eepromShadowOuterDeadBand > 0) {

            eepromUpdateByteSafe(&eepromOuterDeadBand[eepromShadowCurrentProfile],
&eepromShadowOuterDeadBand, eepromShadowOuterDeadBand - 1);
            }
            if (eepromShadowOuterDeadBand == 0) {
                // 0: off
                sprintf(lcdLine1, "Outer DB: Off");
            } else if (eepromShadowOuterDeadBand == 1) {
                // 1: immediate
                sprintf(lcdLine1, "Outer DB: Immed.");
            } else {
                // 2: 0.5s, 3: 1.0s, 4: 1.5s, etc
                // Conversion: y=(x-1)/2
                sprintf(lcdLine1, "Outer DB: %d.%ds", (eepromShadowOuterDeadBand-
1)/2, (eepromShadowOuterDeadBand-1) % 2 ? 5 : 0);
            }
            sprintf(lcdLine2, "%s", currentProfileName);
```

menu.c

```
        break;
    case MENU_OPTION_CTR_DEAD_BAND:
        if (up && eepromShadowCenterDeadBand < 56) {

            eepromUpdateByteSafe(&eepromCenterDeadBand[eepromShadowCurrentProfile],
&eepromShadowCenterDeadBand, eepromShadowCenterDeadBand + 6);
        }
        if (down && eepromShadowCenterDeadBand > 2) {

            eepromUpdateByteSafe(&eepromCenterDeadBand[eepromShadowCurrentProfile],
&eepromShadowCenterDeadBand, eepromShadowCenterDeadBand - 6);
        }
        sprintf(lcdLine1, "Center DB: %d", (eepromShadowCenterDeadBand - 2) / 6 +
1);
        sprintf(lcdLine2, "%s", currentProfileName);
        break;
    case MENU_OPTION_PROP_AS_SWITCH:
        if (up || down) {
            if (eepromShadowPropAsSwitch) {

                eepromUpdateByteSafe(&eepromPropAsSwitch[eepromShadowCurrentProfile],
&eepromShadowPropAsSwitch, 0);
            } else {

                eepromUpdateByteSafe(&eepromPropAsSwitch[eepromShadowCurrentProfile],
&eepromShadowPropAsSwitch, 1);
            }
        }
        sprintf(lcdLine1, "PropAsSwitch:%s", eepromShadowPropAsSwitch ? " On" :
"Off");
        sprintf(lcdLine2, "%s", currentProfileName);
        break;
    case MENU_OPTION_INVERT:
        if (up || down) {
            if (eepromShadowInvert) {

                eepromUpdateByteSafe(&eepromInvert[eepromShadowCurrentProfile],
&eepromShadowInvert, 0);
            } else {

                eepromUpdateByteSafe(&eepromInvert[eepromShadowCurrentProfile],
&eepromShadowInvert, 1);
            }
        }
        sprintf(lcdLine1, "Invert: %s", eepromShadowInvert ? "On" : "Off");
        sprintf(lcdLine2, "%s", currentProfileName);
        break;
    default:
        lcdLine1[0] = '\0';
        lcdLine2[0] = '\0';
        break;
}

if ((up || down) && eepromShadowCurrentProfile == PROFILE_COUNT - 1) {
    menuInit();
}
```

menu.c

```
//if (eepromShadowMenuState != MENU_OPTION_PROFILE) {
    //sprintf(lcdLine2, "S=%4d T=%4d%3d", speed, dir, gWirelessTimeoutCount);
//}

    lcdText(lcdLine1, lcdLine2, 0);
}
```

motor_driver.h

```
/*
 * motor_driver.h
 *
 * Created: 6/4/2012 3:30:02 PM
 * Author: Stew
 */

#ifndef MOTOR_DRIVER_H_
#define MOTOR_DRIVER_H_

typedef union {
    uint8_t array[4];
    struct {
        uint8_t address;
        uint8_t command;
        uint8_t data;
        uint8_t checksum;
    } parts;
} sabertooth_packet;

#define SABERTOOTH_ADDRESS 128

// Sabertooth Commands
#define MOTOR_CMD_DRIVE_FORWARD_MOTOR_1 0
#define MOTOR_CMD_DRIVE_BACKWARDS_MOTOR_1 1
#define MOTOR_CMD_MIN_VOLTAGE 2
#define MOTOR_CMD_MAX_VOLTAGE 3
#define MOTOR_CMD_DRIVE_FORWARD_MOTOR_2 4
#define MOTOR_CMD_DRIVE_BACKWARDS_MOTOR_2 5
#define MOTOR_CMD_DRIVE_MOTOR_1_7_BIT 6
#define MOTOR_CMD_DRIVE_MOTOR_2_7_BIT 7
#define MOTOR_CMD_DRIVE_FORWARD_MIXED_MODE 8
#define MOTOR_CMD_DRIVE_BACKWARDS_MIXED_MODE 9
#define MOTOR_CMD_TURN_RIGHT_MIXED_MODE 10
#define MOTOR_CMD_TURN_LEFT_MIXED_MODE 11
#define MOTOR_CMD_DRIVE_FORWARDS_BACK_7_BIT 12
#define MOTOR_CMD_TURN_7_BIT 13
#define MOTOR_CMD_SERIAL_TIMEOUT 14
#define MOTOR_CMD_BAUD_RATE 15
#define MOTOR_CMD_RAMPING 16
#define MOTOR_CMD_DEADBAND 17

void initMotorDriver(void);
```

motor_driver.h

```
void motorEStop(void);
void resetMotorEStop(void);
void sendMotorCommand(uint8_t command, uint8_t data);
void setMotors(int16_t speed, int16_t dir);

#endif /* MOTOR_DRIVER_H_ */
```

motor_driver.c

```
/*
 * motor_driver.c
 *
 * Created: 6/4/2012 3:28:23 PM
 * Author: Stew
 *
 * The motor controller is a Sabertooth 2x60 from Dimension Engineering
 * See the Sabertooth's datasheet for some useful info
 * S1 = PD7/TXD1 (USARTD1)
 * S2 = PD6/RXD1 (USARTD1)
 *
 * DIP switch settings:
 * 1: down
 * 2: down
 * 3: up
 * 4: up
 * 5: up
 * 6: up
 *
 * Address = 128
 */

#include <avr/io.h>
#include "../atmel/usart_driver.h"
#include "motor_driver.h"
#include "menu.h"

USART_data_t USARTD1_data;

void initMotorDriver(void) {
    //Set PD6 and PD7 both as outputs
    PORTD.DIRSET = PIN6_bm | PIN7_bm;
    PORTD.OUTSET = PIN6_bm | PIN7_bm;

    // Use USARTD1 and initialize buffers.
    USART_InterruptDriver_Initialize(&USARTD1_data, &USARTD1, USART_DREINTLVL_HI_gc);

    // USARTD1, 8 Data bits, No Parity, 1 Stop bit.
    USART_Format_Set(USARTD1_data.usart, USART_CHSIZE_8BIT_gc,
    USART_PMODE_DISABLED_gc, false);

    // Disable RXC interrupt.
    USART_RxdInterruptLevel_Set(USARTD1_data.usart, USART_RXCINTLVL_OFF_gc);

    // Set Baudrate to 9600 bps
```


motor_driver.c

```
USART_Baudrate_Set(&USARTD1, 3317 , -4);

// Disable RX, enable TX
USART_Rx_Disable(USARTD1_data.usart);
USART_Tx_Enable(USARTD1_data.usart);

PMIC_CTRL |= PMIC_HILVLEN_bm;

// Let's hope that global interrupts are already enabled...

// Set minimum voltage to 18V
sendMotorCommand(MOTOR_CMD_MIN_VOLTAGE, 60);

// Enable serial timeout 500 ms
sendMotorCommand(MOTOR_CMD_SERIAL_TIMEOUT, 5);
}

void motorEStop(void)
{
    //S2 is an active-low E-stop when the motor driver is in packetized serial mode
    PORTD.OUTCLR = PIN6_bm;
}

void resetMotorEStop(void)
{
    PORTD.OUTSET = PIN6_bm;
}

static void sendMotorPacket(sabertooth_packet *packet)
{
    uint8_t i = 0;
    while (i < sizeof(packet->array))
    {
        if (USART_TXBuffer_PutByte(&USARTD1_data, packet->array[i]))
        {
            i++;
        }
    }
}

void sendMotorCommand(uint8_t command, uint8_t data)
{
    sabertooth_packet packet;
    packet.parts.address = SABERTOOTH_ADDRESS;
    packet.parts.command = command;
    packet.parts.data = data;
    packet.parts.checksum = (packet.parts.address + packet.parts.command +
packet.parts.data) & 0x7F;
    sendMotorPacket(&packet);
}

void setMotors(int16_t speed, int16_t dir)
{
    if (speed >= 0) {
        sendMotorCommand(MOTOR_CMD_DRIVE_FORWARD_MIXED_MODE, speed);
    } else {
```

motor_driver.c

```
        speed = -speed;
        sendMotorCommand(MOTOR_CMD_DRIVE_BACKWARDS_MIXED_MODE, speed);
    }
    if (dir >= 0) {
        sendMotorCommand(MOTOR_CMD_TURN_RIGHT_MIXED_MODE, dir);
    } else {
        dir = -dir;
        sendMotorCommand(MOTOR_CMD_TURN_LEFT_MIXED_MODE, dir);
    }
}

ISR(USARTD1_DRE_vect)
{
    USART_DataRegEmpty(&USARTD1_data);
}
```

nordic_driver.h

```
/*
 * nordic_driver.h
 *
 * Created on: Feb 22, 2011
 * Author: grant
 */

#ifndef NORDIC_DRIVER_H_
#define NORDIC_DRIVER_H_

// NORDIC COMMAND WORDS
// R_REGISTER 0b000XXXXX where XXXXX = 5 bit register map address
#define R_REGISTER_nCmd 0x00
// W_REGISTER 0b001XXXXX where XXXXX = 5 bit register map address
#define W_REGISTER_nCmd 0x20
#define R_RX_PAYLOAD_nCmd 0x61
#define W_TX_PAYLOAD_nCmd 0xA0
#define FLUSH_TX_nCmd 0xE1
#define FLUSH_RX_nCmd 0xE2
#define REUSE_TX_PL_nCmd 0xE3
#define R_RX_PL_WID_nCmd 0x60
// W_ACK_PAYLOAD 0b10101PPP where PPP = pipe number to write packet to
#define W_ACK_PAYLOAD_nCmd 0xA8
#define W_TX_PAYLOAD_NOACK_nCmd 0xB0
#define NOP_nCmd 0xFF

// NORDIC REGISTERS (5-bit addresses, 8-bit values)
#define CONFIG_nReg 0x00
#define EN_AA_nReg 0x01
#define EN_RXADDR_nReg 0x02
#define SETUP_AW_nReg 0x03
#define SETUP_RETR_nReg 0x04
#define RF_CH_nReg 0x05
#define RF_SETUP_nReg 0x06
#define STATUS_nReg 0x07
#define OBSERVE_TX_nReg 0x08
```

nordic_driver.h

```
#define RPD_nReg          0x09
#define RX_ADDR_P0_nReg  0x0A
#define RX_ADDR_P1_nReg  0x0B
#define RX_ADDR_P2_nReg  0x0C
#define RX_ADDR_P3_nReg  0x0D
#define RX_ADDR_P4_nReg  0x0E
#define RX_ADDR_P5_nReg  0x0F
#define TX_ADDR_nReg     0x10
#define RX_PW_P0_nReg    0x11
#define RX_PW_P1_nReg    0x12
#define RX_PW_P2_nReg    0x13
#define RX_PW_P3_nReg    0x14
#define RX_PW_P4_nReg    0x15
#define RX_PW_P5_nReg    0x16
#define FIFO_STATUS_nReg 0x17
#define DYNPD_nReg       0x1C
#define FEATURE_nReg     0x1D

typedef union {
    volatile uint8_t array[4];
    struct {
        volatile uint8_t SwitchState;
        volatile uint8_t JoyDirection;
        volatile uint8_t JoySpeed;
        volatile uint8_t Reserved;
    } parts;
} NORDIC_DATA_PACKET;

typedef struct {
    volatile NORDIC_DATA_PACKET data;
    volatile uint8_t rxpipe;
} NORDIC_PACKET;

void nordic_Initialize();

uint8_t nordic_getInstructorEStop();
uint8_t nordic_getInstructorLAUp();
uint8_t nordic_getInstructorLADown();
uint8_t nordic_getStudentForward();
uint8_t nordic_getStudentReverse();
uint8_t nordic_getStudentLeft();
uint8_t nordic_getStudentRight();

int8_t nordic_getWirelessPropJoySpeed();
int8_t nordic_getWirelessPropJoyDirection();

int8_t nordic_getInstructorSpeed();
int8_t nordic_getInstructorDirection();

#endif /* NORDIC_DRIVER_H_ */
```

nordic_driver.c

```
/*
```

nordic_driver.c

```
* nordic_driver.c
*
* Created on: Feb 22, 2011
* Author: grant
*/

#include <stdio.h>
#include <string.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "nordic_driver.h"
#include "PWCT_io.h"
#include "menu.h"
#include "../atmel/avr_compiler.h"

#define RXPIPE_INSTRUCTOR_REMOTE 0
#define RXPIPE_STUDENT_JOYSTICK 1

#define SPI_SS_bm PIN4_bm
#define SPI_MOSI_bm PIN5_bm
#define SPI_MISO_bm PIN6_bm
#define SPI_SCK_bm PIN7_bm

static volatile NORDIC_PACKET LAST_PACKET;

static uint8_t gStudentForward;
static uint8_t gStudentReverse;
static uint8_t gStudentLeft;
static uint8_t gStudentRight;
static int8_t gStudentSpeed;
static int8_t gStudentDirection;
static uint8_t gInstructorLAUp;
static uint8_t gInstructorLADown;
static uint8_t gInstructorEStop;
static int8_t gInstructorSpeed;
static int8_t gInstructorDirection;
static volatile uint8_t gIsInstructorTimeout;

static void timeoutTimerSetup(TC0_t * tc)
{
    // Packet time-out counter
    tc->CTRLA = TC_CLKSEL_OFF_gc;
    // 1 tick = 32us
    // 7812 ticks = 0.25s
    tc->PER = 7812;
    // Set timer to normal mode
    tc->CTRLB = TC_WGMODE_NORMAL_gc;
    // Set overflow interrupt (level med)
    tc->INTCTRLA = TC_OVFINTLVL_MED_gc;
    // reset packet watchdog timer
    tc->CNT = 0;
    // start timer
    tc->CTRLA = TC_CLKSEL_DIV1024_gc;
}
```

nordic_driver.c

```
static inline void hardwareSetup()
{
    // SPI prescaler = div128, enable, master
    SPIF.CTRL = SPI_PRESCALER_DIV128_gc | SPI_ENABLE_bm | SPI_MASTER_bm |
    SPI_MODE_0_gc;

    // No SPI interrupt
    SPIF.INTCTRL = SPI_INTLVL_OFF_gc;

    // MOSI and SCK as output
    PORTF.DIRSET = SPI_MOSI_bm | SPI_SCK_bm;

    // Set chip enable (CE) pin as output (not a part of SPI)
    PORTE.OUTCLR = PIN7_bm;
    PORTE.DIRSET = PIN7_bm;

    // Setup IRQ pin interrupt
    PORTH.DIRCLR = PIN2_bm;
    PORTH.PIN2CTRL = PORT_ISC_FALLING_gc;
    PORTH.INTCTRL = PORT_INT0LVL_MED_gc;
    PORTH.INT0MASK = PIN2_bm;

    // Enable med level interrupt
    PMIC.CTRL |= PMIC_MEDLVLEN_bm;

    timeoutTimerSetup(&TCD0);
    timeoutTimerSetup(&TCF0);
}

// CS line must be pulled low before calling this function and released when finished
static uint8_t SPI_TransceiveByte(uint8_t TXdata)
{
    // Send pattern
    SPIF.DATA = TXdata;

    // Wait for transmission complete
    while (!(SPIF.STATUS & SPI_IF_bm)) {

    }

    // Read received data
    uint8_t result = SPIF.DATA;

    return (result);
}

// pull CS low
static inline void chipSelect(void)
{
    PORTF.OUTCLR = PIN4_bm;
    PORTF.DIRSET = PIN4_bm;
}

// release CS
static inline void chipRelease(void)
{

```

nordic_driver.c

```
    PORTF.OUTSET = PIN4_bm;
    PORTF.DIRCLR = PIN4_bm;
}

//set CE high
static inline void activeMode(void)
{
    PORTE.OUTSET = PIN7_bm;
}

static int8_t nordic_SendCommand(uint8_t cmd, uint8_t *txdata, uint8_t *rxdata, uint8_t
dataSize, uint8_t *status)
{
    uint8_t i;
    uint8_t rx;
    uint8_t data;
    uint8_t statusFake;
    int8_t err = 0;

    //check to make sure parameters are valid
    if (status == NULL) {
        status = &statusFake;
    }

    chipSelect();

    //send command
    *status = SPI_TransceiveByte(cmd);

    //send/receive LSByte first
    if (dataSize != 0) {
        i = dataSize;
        do {
            i--;
            if (txdata == NULL) {
                data = 0;
            } else {
                data = txdata[i];
            }
            rx = SPI_TransceiveByte(data);
            if (rxdata != NULL) {
                rxdata[i] = rx;
            }
        } while(i != 0);
    }

    chipRelease();

    return err;
}

//Write a register that contains a single byte of data
static inline int8_t nordic_WriteRegister(uint8_t reg, uint8_t data, uint8_t *status)
{
    return nordic_SendCommand(W_REGISTER_nCmd | reg, &data, NULL, 1, status);
}
```

nordic_driver.c

```
//Writes a register with N bytes of data
static inline int8_t nordic_WriteRegisters(uint8_t reg, uint8_t *data, uint8_t size,
uint8_t *status)
{
    return nordic_SendCommand(W_REGISTER_nCmd | reg, data, NULL, size, status);
}

static void nordicSetup()
{
    uint8_t datas[10];
    uint8_t configRegValue;

    configRegValue = 0x0D; //RX_DR, TX_DS, MAX_RT on IRQ; CRC enabled, two CRC bytes;
RX mode
    nordic_WriteRegister(CONFIG_nReg, configRegValue, NULL);

    //enable auto acknowledge on pipe 0 and 1
    nordic_WriteRegister(EN_AA_nReg, 0x03, NULL);

    //enable auto retransmit, try 5 times with delay of 250us
    nordic_WriteRegister(SETUP_RETR_nReg, 0x05, NULL);

    //EN_RXADDR_nReg      Default data pipe 0 and 1 enabled
    //SETUP_AW_nReg       Default address width of 5 bytes

    //Set RF Channel as 0x7C
    nordic_WriteRegister(RF_CH_nReg, 0x7C, NULL);

    //Set output power 0dB, data rate of 250kbps
    nordic_WriteRegister(RF_SETUP_nReg, 0x27, NULL);

    //Rx Address data pipe 0
    datas[0] = 0xE7;
    datas[1] = 0xE7;
    datas[2] = 0xE7;
    datas[3] = 0xE7;
    datas[4] = 0xE7;
    nordic_WriteRegisters(RX_ADDR_P0_nReg, datas, 5, NULL);

    //Rx Address data pipe 1
    datas[0] = 0xC2;
    datas[1] = 0xC2;
    datas[2] = 0xC2;
    datas[3] = 0xC2;
    datas[4] = 0xC2;
    nordic_WriteRegisters(RX_ADDR_P1_nReg, datas, 5, NULL);

    //Set Payload width of 4 bytes
    nordic_WriteRegister(RX_PW_P0_nReg, sizeof(LAST_PACKET.data.array), NULL);
    nordic_WriteRegister(RX_PW_P1_nReg, sizeof(LAST_PACKET.data.array), NULL);

    //clear fifos (necessary for wdt/soft reset)
    nordic_SendCommand(FLUSH_RX_nCmd, NULL, NULL, 0, NULL);
    nordic_SendCommand(FLUSH_TX_nCmd, NULL, NULL, 0, NULL);
}
```

nordic_driver.c

```
//clear interrupts (necessary for wdt/soft reset)
nordic_WriteRegister(STATUS_nReg, 0x70, NULL);

configRegValue |= 0x02; //PWR_UP bit set
nordic_WriteRegister(CONFIG_nReg, configRegValue, NULL);

//wait for startup
_delay_us(1500);

activeMode(); //start receiving
}

void nordic_Initialize()
{
    hardwareSetup();
    nordicSetup();

    //set default values
    gStudentForward = 0;
    gStudentReverse = 0;
    gStudentLeft = 0;
    gStudentRight = 0;
    gStudentSpeed = 0;
    gStudentDirection = 0;
    gInstructorLAUp = 0;
    gInstructorLADown = 0;
    gInstructorEStop = 0;
    gInstructorSpeed = 0;
    gInstructorDirection = 0;
    gIsInstructorTimeout = 0;
}

uint8_t nordic_getInstructorEStop()
{
    return gInstructorEStop;
}

uint8_t nordic_getInstructorLAUp()
{
    return gInstructorLAUp;
}

uint8_t nordic_getInstructorLADown()
{
    return gInstructorLADown;
}

uint8_t nordic_getStudentForward()
{
    return gStudentForward;
}

uint8_t nordic_getStudentReverse()
{
    return gStudentReverse;
}
}
```


nordic_driver.c

```
uint8_t nordic_getStudentLeft()
{
    return gStudentLeft;
}

uint8_t nordic_getStudentRight()
{
    return gStudentRight;
}

int8_t nordic_getWirelessPropJoySpeed(void)
{
    return gStudentSpeed;
}

int8_t nordic_getWirelessPropJoyDirection(void)
{
    return gStudentDirection;
}

int8_t nordic_getInstructorSpeed()
{
    return gInstructorSpeed;
}

int8_t nordic_getInstructorDirection()
{
    return gInstructorDirection;
}

static void setVariables(uint8_t isTimeout)
{
    static uint8_t eStopCount = 0;
    if ((LAST_PACKET.data.array[0] & 0b00000001) >> 0) {
        if (eStopCount < 1) {
            eStopCount++;
        } else {
            gInstructorEStop = 1;
        }
    } else if (LAST_PACKET.rxfpipe == RXPIPE_INSTRUCTOR_REMOTE) {
        eStopCount = 0;
    }

    if (LAST_PACKET.rxfpipe == RXPIPE_INSTRUCTOR_REMOTE) {
        gInstructorLAUp = ( (LAST_PACKET.data.array[0] & 0b00000010) >> 1 );
        gInstructorLADown = ( (LAST_PACKET.data.array[0] & 0b00000100) >> 2 );
        gInstructorSpeed = LAST_PACKET.data.array[2];
        gInstructorDirection = LAST_PACKET.data.array[1];
    }

    if (LAST_PACKET.rxfpipe == RXPIPE_STUDENT_JOYSTICK) { // Student joystick
        gStudentForward = ( (LAST_PACKET.data.array[0] & 0b00001000) >> 3 );
        gStudentReverse = ( (LAST_PACKET.data.array[0] & 0b00010000) >> 4 );
        gStudentLeft = ( (LAST_PACKET.data.array[0] & 0b00100000) >> 5 );
        gStudentRight = ( (LAST_PACKET.data.array[0] & 0b01000000) >> 6 );
    }
}
```

nordic_driver.c

```
        gStudentSpeed = LAST_PACKET.data.array[2];
        gStudentDirection = LAST_PACKET.data.array[1];
    }
    if (isTimeout) {
        gStudentForward = 0;
        gStudentReverse = 0;
        gStudentLeft = 0;
        gStudentRight = 0;
        gStudentSpeed = 0;
        gStudentDirection = 0;
    }
}

ISR(PORTH_INT0_vect)
{
    uint8_t status;
    uint8_t size = 0;
    uint8_t data[4] = {0,0,0,0};

    //get status
    nordic_SendCommand(NOP_nCmd, NULL, NULL, 0, &status);

    if (status & 0x40) { // Data Ready RX FIFO

        //get payload size
        nordic_SendCommand(R_RX_PL_WID_nCmd, NULL, &size, 1, NULL);
        if (size == sizeof(data)) {
            //get latest packet
            nordic_SendCommand(R_RX_PAYLOAD_nCmd, NULL, data, size, NULL);
//get payload
            LAST_PACKET.data.array[0] = data[0];
            LAST_PACKET.data.array[1] = data[1];
            LAST_PACKET.data.array[2] = data[2];
            LAST_PACKET.data.array[3] = data[3];
            LAST_PACKET.rxpipeline = (status & 0x0E) >> 1;

            //reset packet receive time-out
            if (LAST_PACKET.rxpipeline == RXPIPE_INSTRUCTOR_REMOTE) {
                TCD0.CNT = 0;
                gIsInstructorTimeout = 0;
            }
            if (LAST_PACKET.rxpipeline == RXPIPE_STUDENT_JOYSTICK) {
                TCF0.CNT = 0;
            }
        }
        //clear fifo
        nordic_SendCommand(FLUSH_RX_nCmd, NULL, NULL, 0, NULL);

        //update remote variables
        if (!gIsInstructorTimeout) {
            setVariables(0);
        }
    }
    //} else {
    //    printf("Status=%d\n", status);
    //}
    if (status & 0x20) { // Data Sent TX FIFO
```

nordic_driver.c

```
        nordic_SendCommand(FLUSH_TX_nCmd, NULL, NULL, 0, NULL);
    }
    if (status & 0x10) { // Maximum number of TX retransmits
        nordic_SendCommand(FLUSH_TX_nCmd, NULL, NULL, 0, NULL);
    }

    //clear interrupts
    nordic_WriteRegister(STATUS_nReg, status & 0x70, NULL);
}

ISR(TCD0_OVF_vect) //packet receive time-out
{
    gIsInstructorTimeout = 1;
    LAST_PACKET.data.array[0] = 0;
    LAST_PACKET.data.array[1] = 0;
    LAST_PACKET.data.array[2] = 0;
    LAST_PACKET.data.array[3] = 0;
    LAST_PACKET.rxpipe = RXPIPE_INSTRUCTOR_REMOTE;
    setVariables(1);
    incrementWirelessTimeout();
}

ISR(TCF0_OVF_vect) //packet receive time-out
{
    LAST_PACKET.data.array[0] = 0;
    LAST_PACKET.data.array[1] = 0;
    LAST_PACKET.data.array[2] = 0;
    LAST_PACKET.data.array[3] = 0;
    LAST_PACKET.rxpipe = RXPIPE_STUDENT_JOYSTICK;
    setVariables(1);
    incrementWirelessTimeout();
}
```

PWCT_io.h

```
/*
 * PWCT_io.h
 *
 * Created on: Mar 28, 2011
 * Author: grant
 */

#ifndef PWCT_IO_H_
#define PWCT_IO_H_

#include <avr/io.h>
#include <stdint.h>
#include <stdbool.h>

typedef struct {
    PORT_t *port;
    uint8_t pin_bm;
    volatile uint8_t previous_values;
    volatile uint8_t debounced_value;
}
```

PWCT_io.h

```
    uint8_t previous_debounced_value;
    uint16_t pressedCount;
} debounced_input;

void initPWCTio(void);
void SampleInputs(void);
void OmniStopMove(void);
void OmniMove(uint8_t moveDir);
uint16_t getWiredPropJoySpeed(void);
uint16_t getWiredPropJoyDirection(void);
uint8_t lcdUpFallingEdge(void);
uint8_t lcdDownFallingEdge(void);
uint8_t lcdRightFallingEdge(void);
uint8_t lcdLeftFallingEdge(void);
uint8_t lcdLeftLongPress(void);

bool LimitSwitchPressed(void);
uint8_t ActuatorSwitchPressed(void);
bool PanelEStopPressed(void);
void PulsePGDTEstop(void);

#endif /* PWCT_IO_H_ */
```

PWCT_io.c

```
/*
 * PWCT_io.c
 *
 * Created on: Mar 28, 2011
 * Author: grant
 */

#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#include "../atmel/TC_driver.h"
#include "../atmel/port_driver.h"
#include "../atmel/pmic_driver.h"
#include "../atmel/adc_driver.h"
#include "nordic_driver.h"
#include "util.h"
#include "PWCT_io.h"

//pulse timer 1tick = 8us
//31250 ticks = 250ms
#define PGDT_ESTOP_PULSE_PERIOD 31250

#define ENABLE_INSTRUCTOR_REMOTE_LINEAR_ACTUATOR_CONTROL 0

/* Input/Output List
 * Item Pin In/Out
 * Note
 * Prop. Joy Detect PK7 Input
 * Prop. Joy Speed (Fwd/Rev) PA1 Input A/D
```

PWCT_io.c

```

* Prop. Joy Direction (L/R)          PA2          Input          A/D
* Bumper                            PA7          Input
* optional A/D
* Limit Switch 1                    PA3          Input
* A/D
* Limit Switch 2                    PA4          Input
* A/D
* Limit Switch 3                    PA5          Input
* A/D
* Limit Switch 4                    PA6          Input
* A/D
* Student Forward                   PJ3          Input
* Student Reverse                   PJ4          Input
* Student Left                      PJ5          Input
* Student Right                    PJ6          Input
* Student Fifth                    PJ7          Input
* Buddy Button Forward              PH6          Input
* Buddy Button Reverse              PH7          Input
* Buddy Button Left                 PJ0          Input
* Buddy Button Right                PJ1          Input
* Buddy Button Fifth                PJ2          Input
* Emergency Stop                    PK2          Input
* Normally closed, debounce
* Omni+ On/Off Switch               PK6          Output
* Panel LA Up                       PK0          Input
* Panel LA Down                     PK3          Input
* Panel LA LED                      PK5          Output
* Panel Bumper Override LED         PK4          Output
* Panel Bumper Override Switch      PK1          Input
* Omni+ Out Forward                 PH1          Output
* Omni+ Out Reverse                 PH0          Output
* Omni+ Out Left                   PH4          Output
* Omni+ Out Right                   PH3          Output
* Omni+ Out Fifth                   PH5          Output
* LCD Button 1                      PQ0          Input
* Debounce
* LCD Button 2                      PQ1          Input
* Debounce
* LCD Button 3                      PQ2          Input
* Debounce
* LCD Button 4                      PQ3          Input
* Debounce
* Extra I/O pin                    PR0          I/O
* Fwd/Rev Invert                   PR1          Input
*/

```

//these input flags are all active low

```

static uint8_t BB_FORWARD;
static uint8_t BB_REVERSE;
static uint8_t BB_LEFT;
static uint8_t BB_RIGHT;
static uint8_t BB_FIFTH;
static uint8_t STUDENT_FORWARD;
static uint8_t STUDENT_REVERSE;
static uint8_t STUDENT_LEFT;
static uint8_t STUDENT_RIGHT;

```

PWCT_io.c

```
static uint8_t STUDENT_FIFTH;
static uint8_t PANEL_LA_UP;
static uint8_t PANEL_LA_DOWN;
static uint8_t PANEL BUMPER_OVERRIDE;
static uint8_t PROP_JOY_DETECT;
static uint8_t INVERT_SWITCH;
static uint8_t LIMIT_SWITCH;

#define DEBOUNCED_INPUT_COUNT 5
static debounced_input gDebouncedInputs[DEBOUNCED_INPUT_COUNT];

static volatile uint16_t gWiredPropJoySpeed;
static volatile uint16_t gWiredPropJoyDirection;

static void joystickADCsetup(void)
{
    // ADC configuration for proportional joystick
    ADC_CalibrationValues_Load(&ADCA);

    ADC_ConvMode_and_Resolution_Config(&ADCA, ADC_ConvMode_Unsigned,
    ADC_RESOLUTION_12BIT_gc);

    // External reference on PA0/AREFA
    ADC_Reference_Config(&ADCA, ADC_REFSEL_AREFA_gc);
    ADC_Prescaler_Config(&ADCA, ADC_PRESCALER_DIV512_gc);

    // In Unsigned Single-ended mode, the conversion range is from ground to the
    reference voltage.
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH0, ADC_CH_INPUTMODE_SINGLEENDED_gc,
    ADC_DRIVER_CH_GAIN_NONE);
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH1, ADC_CH_INPUTMODE_SINGLEENDED_gc,
    ADC_DRIVER_CH_GAIN_NONE);

    ADC_Ch_InputMux_Config(&ADCA.CH0, ADC_CH_MUXPOS_PIN1_gc, 0);
    ADC_Ch_InputMux_Config(&ADCA.CH1, ADC_CH_MUXPOS_PIN2_gc, 0);

    ADC_FreeRunning_Enable(&ADCA);
    ADC_SweepChannels_Config(&ADCA, ADC_SWEEP_01_gc);

    //ADC_Events_Config(&ADCA, ADC_EVSEL_0123_gc, ADC_EVACT_SYNCHSWEEP_gc );
    ADC_Ch_Interrupts_Config(&ADCA.CH0, ADC_CH_INTMODE_COMPLETE_gc,
    ADC_CH_INTLVL_MED_gc);
    ADC_Ch_Interrupts_Config(&ADCA.CH1, ADC_CH_INTMODE_COMPLETE_gc,
    ADC_CH_INTLVL_MED_gc);

    PMIC_CTRL |= PMIC_MEDLVLEN_bm;

    ADC_Enable(&ADCA);
    ADC_Wait_32MHz(&ADCA);
}

static void setupEstopTimer()
{
    // TCF1 is the timer used for Omni E-stop pulse
    //turn off timers
    TC1_ConfigClockSource( &TCF1, TC_CLKSEL_OFF_gc );
}
```

PWCT_io.c

```
// Set the TC period.
TC_SetPeriod( &TCF1, 0xFFFF );

//Set timer in normal mode
TC1_ConfigWGM( &TCF1, TC_WGMODE_NORMAL_gc );

TC1_SetCCBIntLevel(&TCF1, TC_CCBINTLVL_MED_gc);

//start clocks
TC1_ConfigClockSource( &TCF1, TC_CLKSEL_DIV256_gc );

PMIC_CTRL |= PMIC_MEDLVLEN_bm;
}

static void setupDebouncedInputs()
{
    //LCD Button 1 PQ0
#define DEBOUNCE_INDEX_LCD1 0
    gDebouncedInputs[DEBOUNCE_INDEX_LCD1].port = &PORTQ;
    gDebouncedInputs[DEBOUNCE_INDEX_LCD1].pin_bm = PIN0_bm;

    //LCD Button 2 PQ1
#define DEBOUNCE_INDEX_LCD2 1
    gDebouncedInputs[DEBOUNCE_INDEX_LCD2].port = &PORTQ;
    gDebouncedInputs[DEBOUNCE_INDEX_LCD2].pin_bm = PIN1_bm;

    //LCD Button 3 PQ2
#define DEBOUNCE_INDEX_LCD3 2
    gDebouncedInputs[DEBOUNCE_INDEX_LCD3].port = &PORTQ;
    gDebouncedInputs[DEBOUNCE_INDEX_LCD3].pin_bm = PIN2_bm;

    //LCD Button 4 PQ3
#define DEBOUNCE_INDEX_LCD4 3
    gDebouncedInputs[DEBOUNCE_INDEX_LCD4].port = &PORTQ;
    gDebouncedInputs[DEBOUNCE_INDEX_LCD4].pin_bm = PIN3_bm;

    //Panel E-stop
#define DEBOUNCE_INDEX_ESTOP 4
    gDebouncedInputs[DEBOUNCE_INDEX_ESTOP].port = &PORTK;
    gDebouncedInputs[DEBOUNCE_INDEX_ESTOP].pin_bm = PIN2_bm;

    int i;
    for (i = 0; i < DEBOUNCED_INPUT_COUNT; i++)
    {
        // Set as input
        gDebouncedInputs[i].port->DIRCLR = gDebouncedInputs[i].pin_bm;

        // Enable pull-up
        PORTCFG.MPCMASK = gDebouncedInputs[i].pin_bm;
        gDebouncedInputs[i].port->PIN0CTRL = PORT_OPC_PULLUP_gc;

        // Default values
        gDebouncedInputs[i].previous_values = UINT8_MAX;
        gDebouncedInputs[i].debounced_value = 1;
        gDebouncedInputs[i].previous_debounced_value = 1;
    }
}
```

PWCT_io.c

```
        gDebouncedInputs[i].pressedCount = 0;
    }

    // E-stop needs an initial value of 0 since it is a normally closed switch
    gDebouncedInputs[DEBOUNCE_INDEX_ESTOP].previous_values = 0;
    gDebouncedInputs[DEBOUNCE_INDEX_ESTOP].debounced_value = 0;
    gDebouncedInputs[DEBOUNCE_INDEX_ESTOP].previous_debounced_value = 0;
}

static void setupDebounceTimer()
{
    // TCC1 is the timer used for debouncing
    TCC1.CTRLA = TC_CLKSEL_DIV4_gc;
    TCC1.CTRLB = TC_WGMODE_FRQ_gc;
    TCC1.INTCTRLB = TC_CCAINTLVL_LO_gc;
    TCC1.CCA = 40000; // Goal: interrupt every 5 milliseconds

    PMIC.CTRL |= PMIC_LOLVLEN_bm;
}

void initPWCTio(void)
{
    setupEstopTimer();

    setupDebouncedInputs();

    setupDebounceTimer();

    PORT_ConfigurePins( &PORTH, PIN6_bm | PIN7_bm, false, false,
PORT_OPC_PULLUP_gc, PORT_ISC_BOTHEDGES_gc ); // BB fwd, BB rev
    PORT_ConfigurePins( &PORTJ, 0xFF, false, false,
PORT_OPC_PULLUP_gc, PORT_ISC_BOTHEDGES_gc ); // remaining BB, switch joystick in
    PORT_ConfigurePins( &PORTK, PIN0_bm | PIN1_bm | PIN3_bm | PIN7_bm, false, false,
PORT_OPC_PULLUP_gc, PORT_ISC_BOTHEDGES_gc ); // LA up, bumper override, LA down, Prop.
Joy Detect
    PORT_ConfigurePins( &PORTK, PIN2_bm, false, false,
PORT_OPC_PULLUP_gc, PORT_ISC_BOTHEDGES_gc ); // e-stop button
    PORT_ConfigurePins( &PORTQ, PIN0_bm | PIN1_bm | PIN2_bm | PIN3_bm, false, false,
PORT_OPC_PULLUP_gc, PORT_ISC_BOTHEDGES_gc ); // LCD buttons
    PORT_ConfigurePins( &PORTR, PIN0_bm | PIN1_bm, false, false,
PORT_OPC_PULLUP_gc, PORT_ISC_BOTHEDGES_gc ); // LCD button, invert switch

    PORT_SetPinsAsInput( &PORTH, PIN6_bm | PIN7_bm);
    PORT_SetPinsAsInput( &PORTJ, PIN0_bm | PIN1_bm | PIN2_bm | PIN3_bm | PIN4_bm |
PIN5_bm | PIN6_bm | PIN7_bm );
    PORT_SetPinsAsInput( &PORTK, PIN0_bm | PIN1_bm | PIN2_bm | PIN3_bm | PIN7_bm);
    PORT_SetPinsAsInput( &PORTQ, PIN0_bm | PIN1_bm | PIN2_bm | PIN3_bm);
    PORT_SetPinsAsInput( &PORTR, PIN0_bm | PIN1_bm);

    //set outputs
    PORTK.OUTCLR = PIN4_bm | PIN5_bm; //leds off
    PORTK.OUTCLR = PIN6_bm; //Omni+ on/off switch disabled
    PORTK.DIRSET = PIN4_bm | PIN5_bm | PIN6_bm;
    PORTH.OUTCLR = PIN0_bm | PIN1_bm | PIN3_bm | PIN4_bm | PIN5_bm; //switch
joystick disabled
    PORTH.DIRSET = PIN0_bm | PIN1_bm | PIN3_bm | PIN4_bm | PIN5_bm;
```


PWCT_io.c

```
joystickADCsetup();

//get default values
SampleInputs();

PMIC.CTRL |= PMIC_MEDLVLEN_bm;
}

void SampleInputs(void)
{
    BB_FORWARD          = PORTH.IN & PIN6_bm;
    BB_REVERSE          = PORTH.IN & PIN7_bm;
    BB_LEFT              = PORTJ.IN & PIN0_bm;
    BB_RIGHT             = PORTJ.IN & PIN1_bm;
    BB_FIFTH             = PORTJ.IN & PIN2_bm;
    STUDENT_FORWARD     = PORTJ.IN & PIN3_bm;
    STUDENT_REVERSE     = PORTJ.IN & PIN4_bm;
    STUDENT_LEFT        = PORTJ.IN & PIN5_bm;
    STUDENT_RIGHT       = PORTJ.IN & PIN6_bm;
    STUDENT_FIFTH       = PORTJ.IN & PIN7_bm;
    PANEL_LA_UP         = PORTK.IN & PIN3_bm;
    PANEL_LA_DOWN       = PORTK.IN & PIN0_bm;
    PANEL BUMPER_OVERRIDE = PORTK.IN & PIN1_bm;
    PROP_JOY_DETECT     = PORTK.IN & PIN7_bm;
    INVERT_SWITCH       = PORTR.IN & PIN1_bm;
    LIMIT_SWITCH        = 0;
}

void OmniStopMove(void)
{
    PORTH.OUTCLR = PIN0_bm | PIN1_bm | PIN3_bm | PIN4_bm | PIN5_bm;
}

//moveDir is a bit field of active directions 0b000SFVLR
//S = Select/Fifth button
//F = Forward
//V = Reverse
//L = Left
//R = Right
void OmniMove(uint8_t moveDir)
{
    /*      Out Forward      PH1
     *      Out Reverse      PH0
     *      Out Left         PH4
     *      Out Right        PH3
     *      Out Fifth        PH5
     */

    if(moveDir & 0x10) { //fifth button
        PORTH.OUTSET = PIN5_bm;
    }
    else {
        PORTH.OUTCLR = PIN5_bm;
    }
}
```

PWCT_io.c

```
if((moveDir & 0x0C) == 0x08) { //forward
    PORTH.OUTSET = PIN1_bm;
}
else {
    PORTH.OUTCLR = PIN1_bm;
}

if((moveDir & 0x0C) == 0x04) { //reverse
    PORTH.OUTSET = PIN0_bm;
}
else {
    PORTH.OUTCLR = PIN0_bm;
}

if((moveDir & 0x03) == 0x02) { //left
    PORTH.OUTSET = PIN4_bm;
}
else {
    PORTH.OUTCLR = PIN4_bm;
}

if((moveDir & 0x03) == 0x01) { //right
    PORTH.OUTSET = PIN3_bm;
}
else {
    PORTH.OUTCLR = PIN3_bm;
}
}

uint16_t getWiredPropJoySpeed(void)
{
    uint16_t returnValue;
    AVR_ENTER_CRITICAL_REGION();
    returnValue = gWiredPropJoySpeed;
    AVR_LEAVE_CRITICAL_REGION();
    return returnValue;
}

uint16_t getWiredPropJoyDirection(void)
{
    uint16_t returnValue;
    AVR_ENTER_CRITICAL_REGION();
    returnValue = gWiredPropJoyDirection;
    AVR_LEAVE_CRITICAL_REGION();
    return returnValue;
}

static uint8_t isInputFallingEdge(uint8_t i)
{
    uint8_t isFallingEdge = 0;
    AVR_ENTER_CRITICAL_REGION();
    if (gDebouncedInputs[i].debounced_value == 0 &&
gDebouncedInputs[i].debounced_value != gDebouncedInputs[i].previous_debounced_value)
    {
        isFallingEdge = 1;
    }
}
```

PWCT_io.c

```
    }
    gDebouncedInputs[i].previous_debounced_value =
gDebouncedInputs[i].debounced_value;
    AVR_LEAVE_CRITICAL_REGION();
    return isFallingEdge;
}

static uint8_t isInputLongPress(uint8_t i)
{
    uint8_t isLongPress = 0;
    AVR_ENTER_CRITICAL_REGION();
    if (gDebouncedInputs[i].pressedCount > 400 && gDebouncedInputs[i].pressedCount <
UINT16_MAX) {
        gDebouncedInputs[i].pressedCount = UINT16_MAX;
        isLongPress = 1;
    }
    AVR_LEAVE_CRITICAL_REGION();
    return isLongPress;
}

uint8_t lcdUpFallingEdge(void)
{
    return isInputFallingEdge(DEBOUNCE_INDEX_LCD4);
}

uint8_t lcdDownFallingEdge(void)
{
    return isInputFallingEdge(DEBOUNCE_INDEX_LCD3);
}

uint8_t lcdRightFallingEdge(void)
{
    return isInputFallingEdge(DEBOUNCE_INDEX_LCD1);
}

uint8_t lcdLeftFallingEdge(void)
{
    return isInputFallingEdge(DEBOUNCE_INDEX_LCD2);
}

uint8_t lcdLeftLongPress(void)
{
    return isInputLongPress(DEBOUNCE_INDEX_LCD2);
}

bool LimitSwitchPressed(void)
{
    return !LIMIT_SWITCH;
}

/* 0 = NO
 * 1 = DOWN
 * 2 = UP
 */
uint8_t ActuatorSwitchPressed(void)
{
```

PWCT_io.c

```
//panel controls take priority over instructor remote controls
if(!PANEL_LA_UP && PANEL_LA_DOWN) {
    return 2;    //up is pressed
}
else if(PANEL_LA_UP && !PANEL_LA_DOWN) {
    return 1;    //down is pressed
}
#if ENABLE_INSTRUCTOR_REMOTE_LINEAR_ACTUATOR_CONTROL
else if(INSTRUCTOR_LA_UP && !INSTRUCTOR_LA_DOWN) {
    return 2;    //up is pressed
}
else if(!INSTRUCTOR_LA_UP && INSTRUCTOR_LA_DOWN) {
    return 1;    //down is pressed
}
#endif
else {
    return 0;    //nothing is pressed or up and down are both pressed
}
}

bool PanelEStopPressed(void)
{
    // normally closed switch
    return gDebounceInputs[DEBOUNCE_INDEX_ESTOP].debounced_value;
}

void PulsePGDTEstop(void)
{
    PORTK.OUTSET = PIN6_bm;
    //start pulse timer
    TC_SetCompareB(&TCF1, TCF1.CNT + PGDT_ESTOP_PULSE_PERIOD);
    TC1_EnableCCChannels( &TCF1, TC1_CCBEN_bm);
}

ISR(TCF1_CCB_vect) //Omni e-stop
{
    if((TCF1.CTRLB & TC1_CCBEN_bm) == 0) {
        return;
    }
    TC1_DisableCCChannels( &TCF1, TC1_CCBEN_bm);
    PORTK.OUTCLR = PIN6_bm;
}

ISR(ADCA_CH0_vect)
{
    AVR_ENTER_CRITICAL_REGION();
    gWiredPropJoySpeed = ADC_ResultCh_GetWord(&ADCA.CH0);
    AVR_LEAVE_CRITICAL_REGION();
}

ISR(ADCA_CH1_vect)
{
    AVR_ENTER_CRITICAL_REGION();
    gWiredPropJoyDirection = ADC_ResultCh_GetWord(&ADCA.CH1);
    AVR_LEAVE_CRITICAL_REGION();
}
}
```

PWCT_io.c

```
// Debounce timer ISR
ISR(TCC1_CCA_vect)
{
    int i;
    for (i = 0; i < DEBOUNCED_INPUT_COUNT; i++)
    {
        gDebouncedInputs[i].previous_values = (gDebouncedInputs[i].previous_values
        << 1) | ((gDebouncedInputs[i].port->IN & gDebouncedInputs[i].pin_bm) ? 1 : 0);

        if (gDebouncedInputs[i].previous_values == UINT8_MAX)
        {
            gDebouncedInputs[i].debounced_value = 1;
            gDebouncedInputs[i].pressedCount = 0;
        }
        else if (gDebouncedInputs[i].previous_values == 0)
        {
            gDebouncedInputs[i].debounced_value = 0;
            if (gDebouncedInputs[i].pressedCount < UINT16_MAX - 1) {
                gDebouncedInputs[i].pressedCount++;
            }
        }
    }
}
```

util.h

```
/*
 * util.h
 *
 * Created on: Nov 1, 2010
 * Author: grant
 */

#ifndef UTIL_H_
#define UTIL_H_

typedef enum {
    IDLE, MOVE, LOAD
} states;

void dbgLEDinit(void);
void dbgLEDset(void);
void dbgLEDclr(void);
void dbgLEDtgl(void);
void dbgUSARTinit(void);
void dbgPutChar(char c);
void dbgPutStr(char *str);

#endif /* UTIL_H_ */
```

util.c

```
/*
 * util.c
 *
 * Created on: Nov 10, 2010
 * Author: grant
 */
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include "../atmel/usart_driver.h"
#include "util.h"

static int uart_putchar (char c, FILE *stream);

/*! USART data struct. */
USART_data_t USARTD0_data;
static FILE mystdout = FDEV_SETUP_STREAM (uart_putchar, NULL, _FDEV_SETUP_WRITE);

void dbgLEDinit(void)
{
    PORTD.OUTCLR = PIN0_bm;
    PORTD.DIRSET = PIN0_bm;
}

void inline dbgLEDset(void)
{
    PORTD.OUTSET = PIN0_bm;
}

void inline dbgLEDclr(void)
{
    PORTD.OUTCLR = PIN0_bm;
}

void inline dbgLEDtgl(void)
{
    PORTD.OUTTGL = PIN0_bm;
}

void dbgUSARTinit(void)
{
    stdout = &mystdout;

    /* PD3 (TXD0) as output. */
    PORTD.DIRSET = PIN3_bm;
    /* PD2 (RXD0) as input. */
    PORTD.DIRCLR = PIN2_bm;

    /* Use USARTD0 and initialize buffers. */
    USART_InterruptDriver_Initialize(&USARTD0_data, &USARTD0, USART_DREINTLVL_LO_gc);

    /* USARTD0, 8 Data bits, No Parity, 1 Stop bit. */
}
```

util.c

```
USART_Format_Set(USARTD0_data.usart, USART_CHSIZE_8BIT_gc,
                 USART_PMODE_DISABLED_gc, false);

/* Enable RXC interrupt. */
USART_RxdInterruptLevel_Set(USARTD0_data.usart, USART_RXCINTLVL_LO_gc);

/* Set Baudrate to 115200 bps */
//USART_Baudrate_Set(USARTD0_data.usart, 1047 , -6);

/* Set Baudrate to 38400 bps */
//USART_Baudrate_Set(USARTD0_data.usart, 3269 , -6);

/* Set Baudrate to 9600 bps */
USART_Baudrate_Set(USARTD0_data.usart, 3317 , -4);

/* Enable both RX and TX. */
USART_Rx_Enable(USARTD0_data.usart);
USART_Tx_Enable(USARTD0_data.usart);

/* Enable PMIC interrupt level low. */
PMIC_CTRL |= PMIC_LOLVLEN_bm;
}

void dbgPutChar(char c)
{
    bool byteBuffered = false;
    while (byteBuffered == false) {
        byteBuffered = USART_TXBuffer_PutByte(&USARTD0_data, c);
    }
}

void dbgPutStr(char *str)
{
    uint8_t i = 0;
    uint8_t len;
    bool byteToBuffer;
    len = strlen(str);

    while (i < len) {
        byteToBuffer = USART_TXBuffer_PutByte(&USARTD0_data, str[i] == '\n' ? '\r'
: str[i]);
        if(byteToBuffer){
            i++;
        }
    }
}

/*
//0 for success, -1 for no available bytes
static int8_t dbgGetCharNonblocking(char *c)
{
    int8_t err = 0;

    if (USART_RXBufferData_Available(&USARTD0_data)) {
        *c = USART_RXBuffer_GetByte(&USARTD0_data);
    }
}
*/
```

util.c

```
    else {
        err = -1;    //no bytes available
    }
    return err;
}

//0 for success, -1 for no available bytes
//max length is the max number of characters in string, so actually string length
//would be maxLength+1
//return 0 success, -1 didn't reach end of string
static int8_t dbgGetStrNonblocking(char *str, uint8_t maxLength)
{
    uint8_t i = 0, err = 0;

    do {
        if (USART_RXBufferData_Available(&USARTD0_data)) {
            str[i] = USART_RXBuffer_GetByte(&USARTD0_data);
            i++;
        }
        else {
            str[i] = 0;
            err = -1;
            break;
        }
    } while (str[i-1] != 0);

    return err;
}
*/

/*! \brief Receive complete interrupt service routine.
 *
 * Receive complete interrupt service routine.
 * Calls the common receive complete handler with pointer to the correct USART
 * as argument.
 */
ISR(USARTD0_RXC_vect)
{
    USART_RXComplete(&USARTD0_data);

    //echo
    if (USART_RXBufferData_Available(&USARTD0_data)) {
        dbgPutChar(USART_RXBuffer_GetByte(&USARTD0_data));
    }
}

/*! \brief Data register empty interrupt service routine.
 *
 * Data register empty interrupt service routine.
 * Calls the common data register empty complete handler with pointer to the
 * correct USART as argument.
 */
ISR(USARTD0_DRE_vect)
{
    USART_DataRegEmpty(&USARTD0_data);
}
```


util.c

```
}  
  
//-----  
//put a byte in the passed file stream  
static int uart_putchar (char c, FILE *stream)  
{  
    //replace new line with carriage return  
    if (c == '\n')  
        return uart_putchar('\r', stream);  
  
    while (!USART_TXBuffer_PutByte(&USARTD0_data, c))  
    {  
    }  
  
    return 0;  
}
```

Appendix E: Source Code (Remote)

main.c

```
/*
 * main.c
 *
 * Created on: Apr 11, 2011
 * Author: grant
 */
#include <string.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "remote_hardware.h"
#include "nordic_driver.h"
#include "nordic_hardware_specific.h"

#define sbi(var, mask) ((var) |= (uint8_t)(1 << mask))
#define cbi(var, mask) ((var) &= (uint8_t)~(1 << mask))

#define XOFFSET 0
#define YOFFSET 0

// Shear mapping: x' = x + m * y
static inline uint8_t shearMapX(uint8_t x, uint8_t y) {
    return ((int16_t)x - XOFFSET - ((int16_t)y - YOFFSET) / 4) + XOFFSET;
}

// Shear mapping: y' = y + m * x
static inline uint8_t shearMapY(uint8_t x, uint8_t y) {
    return ((int16_t)y - YOFFSET - ((int16_t)x - XOFFSET) / 4) + YOFFSET;
}

static void sendData(void)
{
    NORDIC_PACKET testPacket;
    int8_t x = getADC5(); // Direction
    int8_t y = getADC6(); // Speed
    memset(&testPacket, 0, sizeof(testPacket));

#ifdef STUDENT_JOYSTICK
    testPacket.data.array[0] = getBuddyButtons();
    if (isJoystickEnabled()) {
        testPacket.data.array[1] = x - 118; // shearMapX(x, y);
        testPacket.data.array[2] = y - 117; // shearMapY(x, y);
    } else {
        testPacket.data.array[1] = 0;
        testPacket.data.array[2] = 0;
    }
#else //INSTRUCTOR_REMOTE
    testPacket.data.array[0] = getEStop();
    if (x == 0 || x == 1) {
        x = 127;
    } else {
        x = -(x - 129);
    }
#endif
}
```

main.c

```
    }
    y = y - 128;
    testPacket.data.array[1] = x;
    testPacket.data.array[2] = y;
#endif

    nordic_TransmitData(&testPacket);
}

int main(void)
{
    //enable interrupts
    sei();

    initHardware();
    nordic_Initialize();

    while(1) {
        sendData();
        _delay_ms(5);
    }

    return 0;
}
```

nordic_driver.h

```
/*
 * nordic_driver.h
 *
 * Created on: Feb 22, 2011
 * Author: grant
 */

#ifndef NORDIC_DRIVER_H_
#define NORDIC_DRIVER_H_

#include <avr/io.h>

// NORDIC COMMAND WORDS
// R_REGISTER 0b000XXXXX where XXXXX = 5 bit register map address
#define R_REGISTER_nCmd 0x00
// W_REGISTER 0b001XXXXX where XXXXX = 5 bit register map address
#define W_REGISTER_nCmd 0x20
#define R_RX_PAYLOAD_nCmd 0x61
#define W_TX_PAYLOAD_nCmd 0xA0
#define FLUSH_TX_nCmd 0xE1
#define FLUSH_RX_nCmd 0xE2
#define REUSE_TX_PL_nCmd 0xE3
#define R_RX_PL_WID_nCmd 0x60
// W_ACK_PAYLOAD 0b10101PPP where PPP = pipe number to write packet to
#define W_ACK_PAYLOAD_nCmd 0xA8
#define W_TX_PAYLOAD_NOACK_nCmd 0xB0
#define NOP_nCmd 0xFF
```

nordic_driver.h

```
// NORDIC REGISTERS (5-bit addresses, 8-bit values)
#define CONFIG_nReg          0x00
#define EN_AA_nReg          0x01
#define EN_RXADDR_nReg      0x02
#define SETUP_AW_nReg       0x03
#define SETUP_RETR_nReg     0x04
#define RF_CH_nReg          0x05
#define RF_SETUP_nReg       0x06
#define STATUS_nReg         0x07
#define OBSERVE_TX_nReg     0x08
#define RPD_nReg            0x09
#define RX_ADDR_P0_nReg     0x0A
#define RX_ADDR_P1_nReg     0x0B
#define RX_ADDR_P2_nReg     0x0C
#define RX_ADDR_P3_nReg     0x0D
#define RX_ADDR_P4_nReg     0x0E
#define RX_ADDR_P5_nReg     0x0F
#define TX_ADDR_nReg        0x10
#define RX_PW_P0_nReg       0x11
#define RX_PW_P1_nReg       0x12
#define RX_PW_P2_nReg       0x13
#define RX_PW_P3_nReg       0x14
#define RX_PW_P4_nReg       0x15
#define RX_PW_P5_nReg       0x16
#define FIFO_STATUS_nReg    0x17
#define DYNPD_nReg          0x1C
#define FEATURE_nReg        0x1D

typedef union {
    uint8_t array[4];
    struct {
        uint8_t SwitchState;
        uint8_t JoyDirection;
        uint8_t JoySpeed;
        uint8_t Reserved;
    } parts;
} NORDIC_DATA_PACKET;

typedef struct {
    NORDIC_DATA_PACKET data;
    uint8_t rxpipe;
} NORDIC_PACKET;

int8_t nordic_Initialize();
void nordic_TransmitData(NORDIC_PACKET * packet);

//nordic_IRQ() is called from an ISR in nordic_hardware_specific.c
//triggered by the falling edge of the IRQ pin from the nordic chip
uint8_t nordic_IRQ(void);

#endif /* NORDIC_DRIVER_H */
```

nordic_driver.c

```
/*
 * nordic_driver.c
 *
 * Created on: Feb 22, 2011
 * Author: grant
 */
#include <stdio.h>
#include "nordic_driver.h"
#include "nordic_hardware_specific.h"
#include "remote_hardware.h"

static volatile NORDIC_PACKET LAST_PACKET;

//make sure txdata and rxdata are at least of length dataSize
static int8_t nordic_SendCommand(uint8_t cmd, uint8_t *txdata, uint8_t *rxdata, uint8_t
dataSize, uint8_t *status)
{
    uint8_t i;
    uint8_t rx;
    uint8_t data;
    uint8_t statusFake;
    int8_t err = 0;

    //check to make sure parameters are valid
    if(status == NULL) {
        status = &statusFake;
    }

    standbyMode();

    // _delay_us(4);
    chipSelect();

    //send command
    *status = SPI_TransceiveByte(cmd);

    //send/receive LSByte first
    if(dataSize != 0) {
        i = dataSize;
        do {
            i--;
            if(txdata == NULL) {
                data = 0;
            }
            else {
                data = txdata[i];
            }
            rx = SPI_TransceiveByte(data);
            if(rxdata != NULL) {
                rxdata[i] = rx;
            }
        } while(i != 0);
    }

    chipRelease();
}
```

nordic_driver.c

```
    return err;
}

//Write a register that contains a single byte of data
static inline int8_t nordic_WriteRegister(uint8_t reg, uint8_t data, uint8_t *status)
{
    return nordic_SendCommand(W_REGISTER_nCmd | reg, &data, NULL, 1, status);
}

//Writes a register with N bytes of data
static inline int8_t nordic_WriteRegisters(uint8_t reg, uint8_t *data, uint8_t size,
uint8_t *status)
{
    return nordic_SendCommand(W_REGISTER_nCmd | reg, data, NULL, size, status);
}

//Read a register that contains a single byte of data
static inline int8_t nordic_ReadRegister(uint8_t reg, uint8_t *data, uint8_t *status)
{
    return nordic_SendCommand(R_REGISTER_nCmd | reg, NULL, data, 1, status);
}

//Read a register with N bytes of data
static inline int8_t nordic_ReadRegisters(uint8_t reg, uint8_t *data, uint8_t size,
uint8_t *status)
{
    return nordic_SendCommand(R_REGISTER_nCmd | reg, NULL, data, size, status);
}

static inline void setDirTx(void)
{
    standbyMode();
    //RX_DR, TX_DS, MAX_RT on IRQ; CRC enabled, two CRC bytes; TX mode; PWR_UP bit set
    nordic_WriteRegister(CONFIG_nReg, 0x0E, NULL);
}

int8_t nordic_Initialize()
{
    uint8_t configRegValue;
    uint8_t datas[10];
    int8_t err = 0;

    initalizeHardwareForNordic();

    //Initialize Nordic nRF24L01+
    configRegValue = 0x0C;    //RX_DR, TX_DS, MAX_RT on IRQ; CRC enabled, two CRC
bytes; TX mode
    err = nordic_WriteRegister(CONFIG_nReg, configRegValue, NULL);

    //enable auto acknowledge on pipe 0 and 1
    err = nordic_WriteRegister(EN_AA_nReg, 0x03, NULL);

#ifdef INSTRUCTOR_REMOTE
    //enable auto retransmit, try 5 times with delay of 500us
    err = nordic_WriteRegister(SETUP_RETR_nReg, 0x15, NULL);
#else //STUDENT_JOYSTICK
```

nordic_driver.c

```
//enable auto retransmit, try 1 time with delay of 2500us
err = nordic_WriteRegister(SETUP_RETR_nReg, 0x91, NULL);
#endif

//EN_RXADDR_nReg    Default data pipe 0 and 1 enabled
//    todo: enable only 0 or 1 ?
//SETUP_AW_nReg      Default address width of 5 bytes

//Set RF Channel as 0x7C
err = nordic_WriteRegister(RF_CH_nReg, 0x7C, NULL);

//Set output power 0dB, data rate of 250kbps
err = nordic_WriteRegister(RF_SETUP_nReg, 0x27, NULL);

//Rx Address data pipe 0
//ACK comes in on RX data pipe 0
#ifdef INSTRUCTOR_REMOTE
datas[0] = 0xE7;
datas[1] = 0xE7;
datas[2] = 0xE7;
datas[3] = 0xE7;
datas[4] = 0xE7;
#else //STUDENT_JOYSTICK
datas[0] = 0xC2;
datas[1] = 0xC2;
datas[2] = 0xC2;
datas[3] = 0xC2;
datas[4] = 0xC2;
#endif

err = nordic_WriteRegisters(RX_ADDR_P0_nReg, datas, 5, NULL);

//Rx Address data pipe 1
datas[0] = 0xC2;
datas[1] = 0xC2;
datas[2] = 0xC2;
datas[3] = 0xC2;
datas[4] = 0xC2;
err = nordic_WriteRegisters(RX_ADDR_P1_nReg, datas, 5, NULL);

//Tx Address
#ifdef INSTRUCTOR_REMOTE
datas[0] = 0xE7;
datas[1] = 0xE7;
datas[2] = 0xE7;
datas[3] = 0xE7;
datas[4] = 0xE7;
#else //STUDENT_JOYSTICK
datas[0] = 0xC2;
datas[1] = 0xC2;
datas[2] = 0xC2;
datas[3] = 0xC2;
datas[4] = 0xC2;
#endif

err = nordic_WriteRegisters(TX_ADDR_nReg, datas, 5, NULL);
```

nordic_driver.c

```
//Set Payload width of 4 bytes
err = nordic_WriteRegister(RX_PW_P0_nReg, sizeof(LAST_PACKET.data.array), NULL);
err = nordic_WriteRegister(RX_PW_P1_nReg, sizeof(LAST_PACKET.data.array), NULL);

//clear fifos (necessary for wdt/soft reset)
nordic_SendCommand(FLUSH_RX_nCmd, NULL, NULL, 0, NULL);
nordic_SendCommand(FLUSH_TX_nCmd, NULL, NULL, 0, NULL);

//clear interrupts (necessary for wdt/soft reset)
nordic_WriteRegister(STATUS_nReg, 0x70, NULL);

configRegValue |= 0x02; //PWR_UP bit set
err = nordic_WriteRegister(CONFIG_nReg, configRegValue, NULL);

//wait for startup
_delay_us(1500);

return err;
}

//This sends out the data in txdata, leaves chip in standby tx mode
void nordic_TransmitData(NORDIC_PACKET * packet)
{
    nordic_WriteRegister(STATUS_nReg, 0x70, NULL); //Clear any interrupts

    setDirTx(); //set to Tx mode, powered up

    nordic_SendCommand(FLUSH_TX_nCmd, NULL, NULL, 0, NULL); //Clear TX FIFO

    // put dataSize bytes from txdata into the tx fifo
    nordic_SendCommand(W_TX_PAYLOAD_nCmd, packet->data.array, NULL, sizeof(packet->data.array), NULL);

    //Pulse CE to start transmission for at least 10us
    activeMode();
    _delay_us(50);
    standbyMode();
}

//Nordic IRQ pin interrupt
inline uint8_t nordic_IRQ(void)
{
    uint8_t status, previousMode, size = 0;
    uint8_t data[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    //get status
    previousMode = standbyMode();
    nordic_SendCommand(NOP_nCmd, NULL, NULL, 0, &status);

    if (status & 0x40) { // Data Ready RX FIFO
        //get latest packet
        nordic_SendCommand(R_RX_PL_WID_nCmd, NULL, &size, 1, NULL); //get payload
size
        if (size > sizeof(data)) {
            size = sizeof(data);
        }
    }
}
```


nordic_driver.c

```
    if (size != 0) {
        nordic_SendCommand(R_RX_PAYLOAD_nCmd, NULL, data, size, NULL); //get
payload
        LAST_PACKET.data.array[0] = data[0];
        LAST_PACKET.data.array[1] = data[1];
        LAST_PACKET.data.array[2] = data[2];
        LAST_PACKET.data.array[3] = data[3];
        LAST_PACKET.rxpipes = (status & 0x0E) >> 1;
    }
    //clear fifo
    nordic_SendCommand(FLUSH_RX_nCmd, NULL, NULL, 0, NULL);
}
if (status & 0x20) { // Data Sent TX FIFO
    nordic_SendCommand(FLUSH_TX_nCmd, NULL, NULL, 0, NULL);
}
if (status & 0x10) { // Maximum number of TX retransmits
    nordic_SendCommand(FLUSH_TX_nCmd, NULL, NULL, 0, NULL);
    setLEDDelay();
} else {
    clrLEDDelay();
}

//clear interrupts
nordic_WriteRegister(STATUS_nReg, status & 0x70, NULL);
setMode(previousMode);

return status;
}
```

nordic_hardware_specific.h

```
/*
 * nordic_hardware_specific.h
 *
 * Created on: Apr 12, 2011
 * Author: grant
 */

#ifndef NORDIC_HARDWARE_SPECIFIC_H_
#define NORDIC_HARDWARE_SPECIFIC_H_

#include <avr/io.h>
#include <avr/interrupt.h>
#include "util/delay.h"

//function to disable interrupts
#define AVR_ENTER_CRITICAL_REGION() cli()

//function to enable interrupts
#define AVR_LEAVE_CRITICAL_REGION() sei()

void chipSelect(void);
void chipRelease(void);
uint8_t standbyMode(void);
```

nordic_hardware_specific.h

```
void activeMode(void);
void setMode(uint8_t mode);
void initializeHardwareForNordic(void);
uint8_t SPI_TransceiveByte(uint8_t data);

#endif /* NORDIC_HARDWARE_SPECIFIC_H_ */
```

nordic_hardware_specific.c

```
/*
 * nordic_hardware_specific.c
 *
 * Created on: Apr 12, 2011
 * Author: grant
 */

#include "nordic_hardware_specific.h"
#include "nordic_driver.h"

#define BITBANG_SPI 1

#define sbi(var, mask) ((var) |= (uint8_t)(1 << mask))
#define cbi(var, mask) ((var) &= (uint8_t)~(1 << mask))

//pull CS low
inline void chipSelect(void)
{
    cbi(PORTB, PB5);
    sbi(DDRB, PB5);
}

//release CS
inline void chipRelease(void)
{
    sbi(PORTB, PB5);
    cbi(DDRB, PB5);
}

//set CE low
inline uint8_t standbyMode(void)
{
    uint8_t mode;
    mode = PORTB & _BV(PB4);
    cbi(PORTB, PB4);
    return mode;
}

//set CE high
inline void activeMode(void)
{
    sbi(PORTB, PB4);
}

//set CE low if mode == 0, else set CE high
```

nordic_hardware_specific.c

```
inline void setMode(uint8_t mode)
{
    if (mode) {
        activeMode();
    } else {
        standbyMode();
    }
}

//Initialize microcontroller pin directions, spi, interrupts
void initializeHardwareForNordic(void)
{
    //init SPI DDR
    //MOSI, CLK, CS are outputs
#ifdef BITBANG_SPI
    DDRB |= _BV(PB0) | _BV(PB2) | _BV(PB5);
#endif
    //set chip enable (CE) pin as output, set CE to low for standby mode
    cbi(PORTB, PB4);
    sbi(DDRB, PB4);

    //init IRQ interrupt, PB6, INT0
    sbi(MCUCR, ISC01);
    sbi(GIMSK, INT0);
}

uint8_t SPI_TransceiveByte(uint8_t data)
{
    //Bit Bang SPI
#ifdef BITBANG_SPI
#define TX_PORT          PORTB
#define TX_PORT_PIN     PINB
#define TX_PORT_DD      DDRB
#define TX_SCK          2 //Output
#define TX_MISO          1 //Input
#define TX_MOSI          0 //Output
//#define RF_DELAY      5
#define RF_DELAY        55

    //sample on rising edge, setup on falling edge
    //CPOL = 0, CPHA=0
    uint8_t i, incoming = 0;

    //Send outgoing byte
    for(i = 0 ; i < 8 ; i++)
    {
        if(data & 0b10000000)
            sbi(TX_PORT, TX_MOSI);
        else
            cbi(TX_PORT, TX_MOSI);

        cbi(TX_PORT, TX_SCK); //TX_SCK = 0;
        _delay_us(RF_DELAY);

        //MISO bit is valid after clock goes going low
        incoming <<= 1;
    }
#endif
}
```

```

                                nordic_hardware_specific.c
    if( TX_PORT_PIN & (1<<TX_MISO) ) incoming |= 0x01;

    sbi(TX_PORT, TX_SCK); //TX_SCK = 1;
    _delay_us(RF_DELAY);

    data <<= 1;
}
cbi(TX_PORT, TX_SCK); //TX_SCK = 0 after byte sent

return(incoming);
/* sample on falling edge, setup on rising edge
 * CPOL = 0, CPHA=1
  uint8_t i, incoming = 0;

  //Send outgoing byte
  for(i = 0 ; i < 8 ; i++)
  {
      if(data & 0b10000000)
          sbi(TX_PORT, TX_MOSI);
      else
          cbi(TX_PORT, TX_MOSI);

      sbi(TX_PORT, TX_SCK); //TX_SCK = 1;
      _delay_us(RF_DELAY);

      //MISO bit is valid after clock goes going high
      incoming <<= 1;
      if( TX_PORT_PIN & (1<<TX_MISO) ) incoming |= 0x01;

      cbi(TX_PORT, TX_SCK); //TX_SCK = 0;
      _delay_us(RF_DELAY);

      data <<= 1;
  }

  return(incoming);
*/

#undef TX_PORT
#undef TX_PORT_PIN
#undef TX_PORT_DD
#undef TX_SCK
#undef TX_MISO
#undef TX_MOSI
#undef RF_DELAY
#else
    //use hardware module
#endif
}

//need nordic IRQ ISR here, only respond to falling edge of nordic IRQ
ISR(INT0_vect)
{
    nordic_IRQ();
}

```

remote_hardware.h

```
/*
 * remote_hardware.h
 *
 * Created on: Apr 12, 2011
 * Author: grant
 */

#ifndef REMOTE_HARDWARE_H_
#define REMOTE_HARDWARE_H_

#include <avr/io.h>

// #define INSTRUCTOR_REMOTE
#define STUDENT_JOYSTICK

#ifndef INSTRUCTOR_REMOTE
#ifndef STUDENT_JOYSTICK
#error "Please #define either INSTRUCTOR_REMOTE or STUDENT_JOYSTICK in remote_hardware.h"
#endif
#endif

#ifdef INSTRUCTOR_REMOTE
#ifdef STUDENT_JOYSTICK
#error "Please only #define one of INSTRUCTOR_REMOTE or STUDENT_JOYSTICK in remote_hardware.h"
#endif
#endif

typedef struct {
    volatile uint8_t *pin;
    uint8_t pin_bm;
    volatile uint8_t previous_values;
    volatile uint8_t debounced_value;
} debounced_input;

uint8_t getADC5(void);
uint8_t getADC6(void);
#ifdef INSTRUCTOR_REMOTE
uint8_t getEStop(void);
#endif // INSTRUCTOR_REMOTE
void initHardware(void);
void setLED(void);
void clrLED(void);
void tglLED(void);
void setLEDDelay();
void clrLEDDelay();
#ifdef STUDENT_JOYSTICK
uint8_t isJoystickEnabled();
uint8_t getBuddyButtons();
#endif // STUDENT_JOYSTICK

#endif /* REMOTE_HARDWARE_H_ */
```

remote_hardware.c

```
/*
 * remote_hardware.c
 *
 * Created on: Apr 12, 2011
 * Author: grant
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include "remote_hardware.h"

static uint8_t valueADC5;
static uint8_t valueADC6;

#ifdef INSTRUCTOR_REMOTE
#define DEBOUNCED_INPUT_COUNT 1
static debounced_input gDebouncedInputs[DEBOUNCED_INPUT_COUNT];

static void setupDebouncedInputs()
{
    // Assumption: pins are already set as input and pull-ups enabled

    //E-stop button
    #define DEBOUNCE_INDEX_ESTOP 0
    gDebouncedInputs[DEBOUNCE_INDEX_ESTOP].pin = &PINA;
    gDebouncedInputs[DEBOUNCE_INDEX_ESTOP].pin_bm = _BV(PA1);

    int i;
    for (i = 0; i < DEBOUNCED_INPUT_COUNT; i++)
    {
        // Default values
        gDebouncedInputs[i].previous_values = UINT8_MAX;
        gDebouncedInputs[i].debounced_value = 1;
    }
}
#endif // INSTRUCTOR_REMOTE

static void setupDebounceTimer()
{
    //TCCR0A: TCW0 ICEN0 ICNC0 ICES0 ACIC0 - - CTC0
    TCCR0A = _BV(CTC0);

    // Prescaler = 64
    //TCCR0B: - - - TSM PSR0 CS02 CS01 CS00
    TCCR0B = _BV(CS01) | _BV(CS00);

    // Goal: interrupt every 5 milliseconds
    OCR0A = 78;

    //TIMSK: OCIE1D OCIE1A OCIE1B OCIE0A OCIE0B TOIE1 TOIE0 TICIE0
    TIMSK |= _BV(OCIE0A);
}
}
```

remote_hardware.c

```
void initHardware(void)
{
    //Init LED
    PORTB &= ~_BV(PB3);
    DDRB |= _BV(PB3);

    //Init ADC values
    valueADC5 = 125;
    valueADC6 = 125;

    //set buttons pins as inputs
    DDRA &= ~(_BV(PA0) | _BV(PA1) | _BV(PA2) | _BV(PA3) | _BV(PA4) | _BV(PA5));
    //enable pullups on button lines
    PORTA |= _BV(PA0) | _BV(PA1) | _BV(PA2) | _BV(PA3) | _BV(PA4) | _BV(PA5);

#ifdef INSTRUCTOR_REMOTE
    setupDebouncedInputs();
#endif // INSTRUCTOR_REMOTE
    setupDebounceTimer();

    //INIT ADC
    // ADMUX = REFS1:0 ADLAR MUX4:0
    // Select Vcc voltage reference, left adjust result
#define ADMUX_ADC5_bm (_BV(ADLAR) | 0x05) //ADC5 = PA6 = x-axis = horizontal axis =
right/left = joystick direction = blue wire
#define ADMUX_ADC6_bm (_BV(ADLAR) | 0x06) //ADC6 = PA7 = y-axis = vertical axis = fwd/rev
= joystick speed = yellow wire
    ADMUX = ADMUX_ADC5_bm;

    // ADCSRA = ADEN ADSC ADATE ADIF ADIE ADPS2:0
    ADCSRA = _BV(ADEN) | _BV(ADIE) | _BV(ADPS2); //Enable ADC, interrupt enabled,
clock divider 16 (62 KHz@1MHz)

    // ADCSRB = BIN GSEL - REFS2 MUX5 ADTS2:0

    DIDR0 = _BV(ADC5D) | _BV(ADC6D); //disable digital inputs on channels used for
analog

    ADCSRA |= _BV(ADSC); //Start conversion
}

inline void setLED(void)
{
    PORTB |= _BV(PB3);
}

inline void clrLED(void)
{
    PORTB &= ~_BV(PB3);
}

inline void tglLED(void)
{
    if(PORTB & _BV(PB3)) {
        PORTB &= ~_BV(PB3);
    } else {
```

remote_hardware.c

```
        PORTB |= _BV(PB3);
    }
}

// Turns the LED on only after a 0.25 delay
static volatile uint8_t gLEDDelayOn = 0;
void setLEDDelay()
{
    gLEDDelayOn = 1;
}

// Turns the LED off immediately and resets the LED delay
void clrLEDDelay()
{
    gLEDDelayOn = 0;
}

uint8_t getADC5(void)
{
    return valueADC5;    // Left/Right
}

uint8_t getADC6(void)
{
    return valueADC6;    // Up/Down
}

#ifdef INSTRUCTOR_REMOTE
uint8_t getEStop(void)
{
    return !gDebouncedInputs[DEBOUNCE_INDEX_ESTOP].debounced_value;
}
#endif // INSTRUCTOR_REMOTE

#ifdef STUDENT_JOYSTICK
uint8_t isJoystickEnabled()
{
    return !(PINA & _BV(PA1));
}
}

uint8_t getBuddyButtons()
{
    uint8_t buttonMask = 0;
    if (!(PINA & _BV(PA4))) {
        buttonMask |= 0b00001000; // Forward
    }
    if (!(PINA & _BV(PA2))) {
        buttonMask |= 0b00010000; // Reverse
    }
    if (!(PINA & _BV(PA5))) {
        buttonMask |= 0b00100000; // Left
    }
    if (!(PINA & _BV(PA3))) {
        buttonMask |= 0b01000000; // Right
    }
    return buttonMask;
}
```


remote_hardware.c

```
}
#endif // STUDENT_JOYSTICK

ISR(ADC_vect)
{
    if (ADMUX == ADMUX_ADC5_bm)
    {
        valueADC5 = ADCH;

        ADMUX = ADMUX_ADC6_bm;
    }
    else
    {
        valueADC6 = ADCH;

        ADMUX = ADMUX_ADC5_bm;
    }
    ADCSRA |= _BV(ADSC); //Start conversion
}

// Debounce timer ISR
ISR(TIMER0_COMPA_vect)
{
    static uint8_t ledOnCount = 0;
#ifdef INSTRUCTOR_REMOTE
    int i;
    for (i = 0; i < DEBOUNCED_INPUT_COUNT; i++)
    {
        gDebouncedInputs[i].previous_values = (gDebouncedInputs[i].previous_values
<< 1) | ((*gDebouncedInputs[i].pin) & gDebouncedInputs[i].pin_bm) ? 1 : 0);

        if (gDebouncedInputs[i].previous_values == UINT8_MAX)
        {
            gDebouncedInputs[i].debounced_value = 1;
        }
        else if (gDebouncedInputs[i].previous_values == 0)
        {
            gDebouncedInputs[i].debounced_value = 0;
        }
    }
#endif // INSTRUCTOR_REMOTE

    if (gLEDDelayOn) {
        if (ledOnCount < 50) {
            ledOnCount++;
        } else {
            setLED();
        }
    } else {
        clrLED();
        ledOnCount = 0;
    }
}
}
```

Appendix F: Bill of Materials

Table 20: Connector Components

Category	Description	Manufacturer	Mfg part number	Supplier	Supplier part number	Price × 1	Quantity	Price × Qty
Battery cable (6 AWG)	Ring terminals (7 or 8 mm)	Molex	19071-0196	Mouser	538-19071-0196	\$0.66	4	\$2.64
	Disconnect housing	Anderson Power Products	6810G1-BK	Mouser	879-6810G1-BK	\$4.71	4	\$18.84
	Disconnect contact 6 AWG	Anderson Power Products	1319G6-BK	Mouser	879-1319G6-BK	\$3.26	8	\$26.08
	Panel mount housing	Anderson Power Products	1321-BK	Mouser	879-1321-BK	\$4.95	6	\$29.70
	Panel mount contact 6 AWG	Anderson Power Products	1319G6-BK	Mouser	879-1319G6-BK	\$3.26	6	\$19.56
	Panel mount plates (pair - complain to Mouser if single)	Anderson Power Products	1464G2	Mouser	879-1464G2	\$7.04	1	\$7.04
	Additional contact 4 AWG	Anderson Power Products	1319G4-BK	Mouser	879-1319G4-BK	\$3.14	1	\$3.14
	Additional contact 2 AWG	Anderson Power Products	1319-BK	Mouser	879-1319-BK	\$3.14	0	\$0.00
Motor (8 AWG)	PP75 red housing	Anderson Power Products	5916G7-BK	Mouser	879-5916G7-BK	\$1.47	4	\$5.88
	PP75 black housing	Anderson Power Products	5916G4-BK	Mouser	879-5916G4-BK	\$1.47	4	\$5.88
	contact	Anderson Power Products	5900-BK	Mouser	879-5900-BK	\$0.62	8	\$4.96
	Panel mount (pair)	Anderson Power Products	1463G1	Mouser	879-1463G1	\$4.19	2	\$8.38
Photointerrupter	Panel mount	Neutrik	RT3MP	Mouser	568-RT3MP-B	\$1.86	2	\$3.72
	Plug	Neutrik	RT3FC-B	Mouser	568-RT3FC-B	\$2.54	2	\$5.08
Connector to front	Panel mount (housing only)	Molex	15-06-0245	Mouser	538-15-06-0245	\$2.07	1	\$2.07
	Plug	Molex	39-01-2240	Mouser	538-39-01-2240	\$1.25	1	\$1.25
	Female Socket crimp contacts 16 AWG	Molex	45750-3111	Mouser	538-45750-3111	\$0.09	24	\$2.04
	Male Pin crimp contacts 16 AWG	Molex	46012-3142	Mouser	538-46012-3142	\$0.22	24	\$5.33
Charging connector	XLR receptacle (panel mount)	Neutrik	NC3FD-LX-0	Mouser	568-NC3FD-LX-0	\$3.32	1	\$3.32
	XLR receptacle (cable)	Switchcraft	AAA3FPZ	Mouser	502-AAA3FPZ	\$3.19	1	\$3.19
Fuses	8A or 10A charging							\$0.00
Wire	6 AWG Red (foot)			McMaster-Carr	6948K912	\$1.68	9	\$15.12
	6 AWG Black (foot)			McMaster-Carr	6948K911	\$1.68	9	\$15.12
	8 AWG Red (foot)			McMaster-Carr	6948K892	\$1.18	3	\$3.54
	8 AWG Black (foot)			McMaster-Carr	6948K891	\$1.18	3	\$3.54
	24AWG Black (foot)			McMaster-Carr	7587K921	\$0.09	150	\$13.79
	24AWG White (foot)			McMaster-Carr	7587K924	\$0.09	150	\$13.79
	16AWG Black (foot)			McMaster-Carr	7587K961	\$0.23	50	\$11.44
	16AWG Red (foot)			McMaster-Carr	7587K962	\$0.23	50	\$11.44
	16AWG Yellow (foot)			McMaster-Carr	7587K963	\$0.23	50	\$11.44
	3-conductor wire (foot)			McMaster-Carr	9936K25	\$1.47	20	\$29.40
Wire sleeves	Mesh sleeving			McMaster-Carr	9284K415	\$7.95	1	\$7.95
	D-sub standoffs			McMaster-Carr	93620A701	\$1.40	4	\$5.60

Table 21: Main Receiver Components

ID #	Manufacturer	MFG Part #	Description	Supplier	Supplier Part #	Schematic Ref	Price	Qty	Sub Total
Resistors									
1	Stackpole Electronics	RMCF0805JT10K0	10kΩ 0805	Digi-Key	RMCF0805JT10K0CT-ND	R4, R7, R8, R10, R11, R12, R13, R14, R15, R17, R18, R19, R20, R22, R23, R24, R27, R28, R31, R32, R33, R34, R36, R37, R41, R42, R48, R49, R51, R52, R53, R54, R55, R57, R65, R66, R67, R68, R72, R75, R77, R78, R79, R80, R81, R82, R83, R84	\$0.03	48	\$1.44
2	Stackpole Electronics	RMCF0402FT1M00	1M 0402	Digi-Key	RMCF0402FT1M00CT-ND	R1	\$0.04	1	\$0.04
3	Panasonic	ERJ-2RKF2202X	22k 0402	Digi-Key	P22.0KLCCT-ND	R2	\$0.10	1	\$0.10
4	Panasonic	ERJ-6ENF40R2V	40 0805	Digi-Key	P40.2CCT-ND	R3, R69, R70	\$0.10	3	\$0.30
5	Stackpole Electronics	RMCF0805FT147R	145 0805	Digi-Key	RMCF0805FT147RCT-ND	R16	\$0.04	1	\$0.04
6	Stackpole Electronics	RMCF0805FT100K	100k 0805	Digi-Key	RMCF0805FT100KCT-ND	R21, R43, R44, R45, R46, R47, R64	\$0.04	7	\$0.28
7	Panasonic	ERJ-6ENF1100V	110 0805	Digi-Key	P110CCT-ND	R35	\$0.10	1	\$0.10
8	Stackpole Electronics	RMCF0805FT12K4	12.5k 0805	Digi-Key	RMCF0805FT12K4CT-ND	R38	\$0.04	1	\$0.04
9	Stackpole Electronics	RMCF0805FT1K00	1k 0805	Digi-Key	RMCF0805FT1K00CT-ND	R39, R71, R73	\$0.04	3	\$0.12
10	Panasonic	ERJ-6ENF4020V	400 0805	Digi-Key	P402CCT-ND	R40	\$0.10	1	\$0.10
11	Stackpole Electronics	RMCF0805JT15R0	15 0805	Digi-Key	RMCF0805JT15R0CT-ND	R5, R6, R25, R26, R29, R30, R50, R56, R58, R59, R60, R61, R62, R63	\$0.03	8	\$0.24
12	Stackpole Electronics	RMCF0805FT220R	220 0805	Digi-Key	RMCF0805FT220RCT-ND	R62, R63	\$0.04	6	\$0.24
13	Stackpole Electronics	RMCF0805ZTOR00	0 0805	Digi-Key	RMCF0805ZTOR00CT-ND	R9, R74	\$0.03	2	\$0.06
14	Bourns Inc	3362P-1-203LF	20k Potentiometer	Digi-Key	3362P-203LF-ND	R76	\$1.05	1	\$1.05
Capacitors									
1	Kemet	T491C107K0162T	100uF 2312	Digi-Key	399-5214-1-ND	C11, C12	\$1.34	2	\$2.68
2	Kemet	C0805C103K5RACU	0.01uF 0805	Digi-Key	399-1158-1-ND	C10, C16, C21, C26, C31	\$0.05	5	\$0.25
3	Taiyo Yuden	TMK105B7103KV-F	10nF 0402	Digi-Key	587-1223-1-ND	C8	\$0.05	1	\$0.05
4	Taiyo Yuden	UMK105B7102KV-F	1nF 0402	Digi-Key	587-1220-1-ND	C9	\$0.05	1	\$0.05
5	Taiyo Yuden	UMK105B7222KV-F	2.2nF 0402	Digi-Key	587-1221-1-ND	C4	\$0.05	1	\$0.05
6	Taiyo Yuden	UMK105CG220J-F	22pF 0402	Digi-Key	587-1203-1-ND	C1, C2	\$0.06	2	\$0.12
7	Johanson Technology Inc	500R0751R08V4T	1pF 0402	Digi-Key	712-1266-1-ND	C6	\$0.19	1	\$0.19
8	Johanson Technology Inc	500R0754R78V4T	4.7pF 0402	Digi-Key	712-1166-1-ND	C5	\$0.26	1	\$0.26
9	AVX Corp	04025U1R58AT2A	1.5pF 0402	Digi-Key	478-5992-1-ND	C7	\$0.16	1	\$0.16
10	TDK Corporation	C1005X7R1E333K	33nF 0402	Digi-Key	445-4939-1-ND	C3	\$0.04	1	\$0.04
11	AVX Corp	0805SC104IAT2A	0.1uF 0805	Digi-Key	478-3352-1-ND	C17, C18, C19, C22, C23, C24, C27, C28, C29, C32, C33, C34, C35, C36, C37, C38, C41, C42, C45, C48, C49, C50	\$0.05	22	\$1.10
12	AVX Corp	0805PC102KAT1A	1nF 0805	Digi-Key	478-5541-1-ND	C15, C20, C25, C30, C51, C52	\$0.30	6	\$1.80
13	Murata Electronics	GRM21BR61E106KA73L	10uF 0805	Digi-Key	490-5523-1-ND	C13, C39, C43, C46	\$0.26	4	\$1.04
14	Taiyo Yuden	UMK212B1105KG-T	1uF 0805	Digi-Key	587-2229-1-ND	C14, C40, C44, C47	\$0.39	4	\$1.56
IC									
1	National Semiconductor	LM2674M-3.3/NOPB	3.3 voltage regulator	Digi-Key	LM2674M-3.3-ND	IC3	\$3.47	1	\$3.47
2	Nordic Semiconductor	nRF24L01P-T	Nordic 2.4GHz Wireless IC	Mouseer	949-NRF24L01P-T	U1	\$3.60	1	\$3.60
3	Allegro Microsystems	ACS716KLATR-12CB-T	±12A Current Sensor	Digi-Key	620-1443-1-ND	IC4, IC7, IC10, IC13	\$4.35	4	\$17.40
4	Atmel	ATXMEGA64A1-AU	Xmega64A1	Mouseer	556-ATXMEGA64A1-AU	IC5	\$7.50	1	\$7.50
5	Linear Technologies	LT1160CS#PBF	Half Bridge Driver	Digi-Key	LT1160CS#PBF-ND	IC1, IC2, IC6, IC8	\$5.65	4	\$22.60
6	NXP Semiconductors	74LVCT245DC,125	2-bit Level Translator	Digi-Key	568-5479-1-ND	IC9	\$0.62	1	\$0.62
7	Texas Instruments	SN74VLC8T245DGVPR	8-bit Level Translator	Digi-Key	296-19287-1-ND	IC11, IC12	\$1.26	2	\$2.52
Other									
1	Fairchild Semiconductor	FDD6630A	half bridge Nfet	Digi-Key	FDD6630ACT-ND	Q2, Q3, Q10, Q11, Q12, Q13, Q16, Q17	\$0.69	8	\$5.52
2	Pulse Electronics Corp	PE-53811SNL	118uH inductor for voltage regulator	Digi-Key	553-1395-ND	L4	\$1.89	1	\$1.89
3	Micro Commercial Co	SS24-TP	Schottky Diode	Digi-Key	SS24-TPMSCT-ND	D1, D3, D4, D5, D6	\$0.51	5	\$2.55
4	Lite-On Inc	LTST-C170TBKT	Debug LED 0805	Digi-Key	160-1579-1-ND	LED1	\$0.46	1	\$0.46
5	NXP Semiconductors	TL431M5DT,215	Shunt Voltage Reference	Digi-Key	568-4880-1-ND	D2	\$0.47	1	\$0.47
6	E-Switch	TL1015AF160QG	Reset Switch	Digi-Key	EG4344CT-ND	SW1	\$0.95	1	\$0.95
7	Johanson Technology Inc	L-07C2N75V6T	2.7nH 0402	Digi-Key	712-1415-1-ND	L2	\$0.10	1	\$0.10
8	Johanson Technology Inc	L-07C3N95V6T	3.9nH 0402	Digi-Key	712-1459-1-ND	L1	\$0.10	1	\$0.10
9	Johanson Technology Inc	L-07C8N2JV6T	8.2nH 0402	Digi-Key	712-1420-1-ND	L3	\$0.10	1	\$0.10
10	Abracon Corp	ABM38-16.000MHZ-10-1-U-T	16MHz Crystal	Digi-Key	300-8206-1-ND	Q1	\$1.50	1	\$1.50
11	Micro Commercial Co	SK106-TP	Flyback Schottky Diode	Digi-Key	SK106-TPCT-ND	D20, D21, D22, D23, D24, D25, D26, D27	\$1.00	8	\$8.00
12	Micro Commercial Co	BZV55C3V3-TP	3.3V Zener	Digi-Key	BZV55C3V3-TPMSCT-ND	D17, D18, D19, D48, D49	\$0.38	5	\$1.90
13	Semtech	UCLAMP3304A.TCT	3.3V TVS	Digi-Key	UCLAMP3304ACT-ND	D10, D11, D12, D13, D45, D46, D50, D51, D52	\$0.57	9	\$5.13
14	Semtech	UCLAMP0504A.TCT	5V TVS	Digi-Key	UCLAMP0504ACT-ND	D14, D15, D16, D47	\$0.49	4	\$1.96
15	Bourns Inc	SMBJ16CA	16V Bidirectional TVS	Digi-Key	SMBJ16CABCT-ND	D28, D29, D30, D31, D32, D33, D34, D35	\$0.30	8	\$2.40
16	Bourns Inc	SMAJ26A	26V Unidirectional TVS	Digi-Key	SMAJ26ABCT-ND	D36, D37, D38, D39, D40, D41, D42, D43	\$0.34	8	\$2.72
17	Newhaven Display Intl	NHD-0216K1Z-NSW-BBW-L	LCD	Digi-Key	NHD-0216K1Z-NSW-BBW-L-ND		\$11.70	1	\$11.70
18	Sharp Microelectronics	GP1A51HRJ00F	Photointerrupter	Digi-Key	425-1949-5-ND		\$1.44	2	\$2.88

Table 22: Main Receiver Components (Continued)

ID #	Manufacturer	MFG Part #	Description	Supplier	Supplier Part #	Schematic Ref.	Price	Qty	Sub Total
Connectors & Switches									
									\$0.00
1	Kobiconn	161-1640-EX	1/8" Mono Female	Mouser	161-1640-EX		\$1.04	6	\$6.24
2	CW Industries	GRS-4013C-0001	Mom-Off-Mom Rocker, Linear Actuators	Digi-Key	5W340-ND		\$1.62	1	\$1.62
3	TE Connectivity	PRASCI-16F-BBR00	Bumper Override Rocker Switch	Digi-Key	450-1041-ND		\$2.05	2	\$4.10
4	E-Switch	RS8BLKGRNFF2	DPST Power Switch	Digi-Key	EG1536-ND		\$2.75	1	\$2.75
5	E-Switch	PS1057ABLK	Pushbuttons for UI	Digi-Key	EG2041-ND		\$1.63	6	\$9.78
6	Apem	A01ESSP3	E-stop button	Mouser	642-A01ES-SP3		\$19.38	2	\$38.76
7	Apem	A0150B	E-stop switch block	Mouser	642-A0150B		\$6.77	2	\$13.54
8	Apem	A01YL1	E-stop sticker (optional)	Mouser	642-A01YL1		\$2.21	2	\$4.42
9	Lumex Opto	SSI-LXHR080GO	Actuator Indicator LED	Digi-Key	67-1168-ND		\$1.17	1	\$1.17
10	Amphenol Commercial	G1750910110EU	D-sub 9 plug panel mount	Mouser	523-G1750910110EU		\$0.58	2	\$1.16
11	Kobiconn	172-2136	E-Stop Mono Cable	Mouser	172-2136		\$2.35	1	\$2.35
12	FCI	68602-406HLF	Programming header	Mouser	649-68602-406HLF	PROG1	\$0.12	1	\$0.12
13	Molex	22-28-4360	Straight Male header	Digi-Key	WM6436-ND		\$1.18	4	\$4.72
14	TE Connectivity	881545-2	Shunt Jumpers	Digi-Key	A26242-ND		\$0.21	10	\$2.13
15	Phoenix Contact	1935187	Motor driver connector	Digi-Key	277-1579-ND	JP15	\$0.64	1	\$0.64
16	Keystone Electronics	7200	Hex standoff 3/8"	Digi-Key	7200K-ND		\$0.68	4	\$2.72
17	Sullins Connector Solutions	PPTC161LFBN-RC	Female pin header 16 pos single row	Digi-Key	S7014-ND		\$1.22	1	\$1.22
18	Sullins Connector Solutions	PPTC061LFBN-RC	Female pin header 6 pos single row	Digi-Key	S7004-ND		\$0.67	1	\$0.67
19	Sullins Connector Solutions	PPTC031LFBN-RC	Female pin header 3 pos single row	Digi-Key	S7001-ND		\$0.63	2	\$1.26
20	Sullins Connector Solutions	PPTC021LFBN-RC	Female pin header 2 pos single row	Digi-Key	S7000-ND		\$0.49	2	\$0.98
21	Sullins Connector Solutions	PPTC122LFBN-RC	Female pin header 24 pos double row	Digi-Key	S7080-ND		\$1.73	1	\$1.73
22	Sullins Connector Solutions	PPTC042LFBN-RC	Female pin header 8 pos double row	Digi-Key	S7072-ND		\$0.83	1	\$0.83

Table 23: Remote Components

ID #	Manufacturer	MFG Part #	Description	Supplier	Supplier Part #	Schematic Ref.	Price	Qty	Sub Total
Resistors									
1	Stackpole Electronics	RMCF0402FT1M00	1M 0402	Digi-Key	RMCF0402FT1M00CT-ND	R1	\$0.04	1	\$0.04
2	Susumu	RR0510P-223-D	22k ±1% 0402	Digi-Key	RR05P22.0KDCCT-ND	R2	\$0.08	1	\$0.08
3	Stackpole Electronics	RMCF0402JT100K	100k 0402	Digi-Key	RMCF0402JT100KCT-ND	R3, R7, R8, R9, R10	\$0.02	5	\$0.10
4	Stackpole Electronics	RMCF0402JT110R	110 0402	Digi-Key	RMCF0402JT110RCT-ND	R4	\$0.02	1	\$0.02
5	Stackpole Electronics	RMCF0402FT75K0	75k 0402	Digi-Key	RMCF0402FT75K0CT-ND	R6	\$0.04	1	\$0.04
6	Stackpole Electronics	RMCF0402FT330K	330k 0402	Digi-Key	RMCF0402FT330KCT-ND	R5	\$0.04	1	\$0.04
Capacitors									
7	Taiyo Yuden	TMK105B1104KV-F	0.1µF 0402	Digi-Key	587-1456-1-ND	C10	\$0.10	1	\$0.10
8	Taiyo Yuden	TMK105B7103KV-F	10nF 0402	Digi-Key	587-1223-1-ND	C8	\$0.10	1	\$0.10
9	Taiyo Yuden	UMK105B7102KV-F	1nF 0402	Digi-Key	587-1220-1-ND	C9	\$0.10	1	\$0.10
10	Taiyo Yuden	UMK105B7222KV-F	2.2nF 0402	Digi-Key	587-1221-1-ND	C4	\$0.10	1	\$0.10
11	Taiyo Yuden	UMK105CG220JV-F	22pF 0402	Digi-Key	587-1203-1-ND	C1, C2	\$0.10	2	\$0.20
12	TDK Corporation	C1005C0G1H010B	1pF 0402	Digi-Key	445-4854-1-ND	C6	\$0.10	1	\$0.10
13	TDK Corporation	C1005C0G1H4R7B	4.7pF 0402	Digi-Key	445-4878-1-ND	C5	\$0.10	1	\$0.10
14	TDK Corporation	C1005C0G1H1R5B	1.5pF 0402	Digi-Key	445-4858-1-ND	C7	\$0.10	1	\$0.10
15	TDK Corporation	C1005X7R1E333K	33nF 0402	Digi-Key	445-4939-1-ND	C3	\$0.10	1	\$0.10
16	TDK Corporation	C1608X5R0J226M	22µF 0603	Digi-Key	445-8028-1-ND	C11, C14, C15	\$0.38	3	\$1.14
17	Kemet	C0402C105K9PACTU	1µF 0402	Digi-Key	399-4873-1-ND	C12, C13	\$0.07	2	\$0.14
Inductors									
18	Johanson Technology Inc.	L-07C2N7SV6T	2.7nH 0402	Digi-Key	712-1415-1-ND	L2	\$0.10	1	\$0.10
19	Johanson Technology Inc.	L-07C3N9SV6T	3.9nH 0402	Digi-Key	712-1459-1-ND	L1	\$0.10	1	\$0.10
20	Johanson Technology Inc.	L-07C8N2JV6T	8.2nH 0402	Digi-Key	712-1420-1-ND	L3	\$0.10	1	\$0.10
21	TDK Corporation	CPL2512T2R2M	2.2µH 0805	Digi-Key	445-4172-1-ND	L4, L5	\$0.45	2	\$0.90
IC's									
22	Atmel	ATTINY461V-10MUR	Attiny461 microcontroller QFN32	Digi-Key	ATTINY461V-10MURCT-ND	IC1	\$3.05	1	\$3.05
23	Nordic Semiconductor	nRF24L01P-T	Nordic Wireless	Mouser	949-NRF24L01P-T	IC2	\$3.60	1	\$3.60
24	Texas Instruments	TPS61200DRCT	Adjustable Boost regulator	Digi-Key	296-21663-1-ND	IC3	\$3.74	1	\$3.74
25	Texas Instruments	TPS61202DSCR	5V Boost regulator	Digi-Key	296-24865-1-ND	IC4	\$3.17	1	\$3.17
Other									
26	Abracon Corp	ABM38-16.000MHZ-10-1-U-T	16MHz Crystal	Digi-Key	300-8206-1-ND	XTAL1	\$1.50	1	\$1.50
27	Semtech	UCLAMP3304A.TCT	3.3V TVS	Digi-Key	UCLAMP3304ACT-ND	D1	\$0.57	1	\$0.57
28	CHIPLED	LS Q976-NR-1-0-20-R18	LED RED 0603	Digi-Key	475-2512-1-ND	LED1	\$0.13	1	\$0.13
29	Fairchild Semiconductor	MBR0520L	Reverse battery protection diode	Digi-Key	MBR0520LCT-ND	D2	\$0.36	1	\$0.36
30	FCI	67996-206HLF	Header AVR programming	Digi-Key	609-3210-ND	AVR	\$0.29	1	\$0.29
31	Sparkfun		Thumb Joystick PS2 style	Sparkfun	COM-09032	JSTICK1	\$3.95	1	\$3.95
32	Kobiconn	161-1640-EX	3.5mm mono jack female	Mouser	161-1640-EX		\$1.04	4	\$4.16
33	Molex	22-28-4360	Male pin header	Digi-Key	WM6436-ND		\$1.18	1	\$1.18
34	Sullins Connector Solutions	PPTC042LFBN-RC	2x4 Female pin header	Digi-Key	S7072-ND		\$0.83	1	\$0.83
35	Sullins Connector Solutions	PPTC021LFBN-RC	2x1 Female pin header	Digi-Key	S7000-ND		\$0.49	1	\$0.49
36			1x1 Female pin header	Digi-Key				1	\$0.00
37	MPD (Memory Protection Devices)	BC2AAW	Battery Holder 2xAA	Digi-Key	BC2AAW-ND	BAT1	\$0.85	1	\$0.85
38	Cherry	PRK22J5DBBNN	On/off switch	Digi-Key	CH865-ND		\$1.36	2	\$2.72
39	CW Industries	GP8507A05BR	Red e-stop Button	Digi-Key	SW637-ND		\$2.11	1	\$2.11

Appendix G: Quick Start Guide

Power Wheelchair Trainer Quick Start Guide











<h3>Loading and Unloading a Learner</h3>	<p>5. Once the learner has been loaded, attach the front straps to the wheelchair's front tie down brackets and adjust the strap lengths to position the learner near the front of the platform (Figure 6).</p> <p>6. Engage the wheelchair brakes, then push the red lever on the rear Sure-Lok® tie downs to attach the straps to the wheelchair's rear tie down brackets. Then turn the knob to retract the rear tie downs until the wheelchair is firmly secured (Figure 7).</p>	<h3>Driving Profiles</h3>
<ol style="list-style-type: none"> 1. Ensure that all cables are properly plugged in and turn the power ON with the green power switch on the control panel (Figure 2). 2. Press and hold the platform switch down (Figure 2) until the lift platform touches the floor.   <p>Figure 1</p> <p>Figure 2</p> <ol style="list-style-type: none"> 3. Once the platform has been lowered, pull the two pins out (Figure 3) to detach and remove the front gate (Figure 4).   <p>Figure 3</p> <p>Figure 4</p> <ol style="list-style-type: none"> 4. Wheel the learner onto the platform with his or her back facing the power unit and ensure that the wheelchair's front casters are facing forward, parallel to the gray lines on the platform (Figure 5).  <p>Figure 5</p>	<h3>Drive Control</h3> <ol style="list-style-type: none"> 8. Mount the tray on the wheelchair by securing the armrest straps around the wheelchair's armrests. Then secure the long strap around the back of the wheelchair. 9. Mount the wireless joystick in any position of the tray by removing 3 tiles in the desired position, and make sure the arrow points forward. Up to four external switches may optionally be plugged in to the wireless joystick and secured to either the tray or wheelchair. The switch labeled "joystick" enables or disables the joystick only – any switches that are plugged in will work anytime the joystick power is on.   <p>Figure 6</p> <p>Figure 7</p>  <p>Figure 8</p>	<ol style="list-style-type: none"> 10. Before making changes to the profiles, make sure the joystick is turned OFF. Use the buttons below the LCD screen to navigate the menu interface. Right/left chooses the setting to change, and up/down changes the value. For example, scroll all the way to the left to get to the profile setting, then scroll up or down to change the current profile. A driving profile has a customized speed, acceleration, sensitivity, etc. for each learner. 11. Once an appropriate profile has been selected turn the joystick power ON. Also turn ON the therapist remote and make sure it is in close proximity to the power wheelchair trainer. The learner is now ready to drive.  <p>Figure 9</p>
		<h3>Therapist Remote</h3> <p>The power wheelchair trainer will not drive unless the therapist remote is powered ON and in close proximity to the trainer. The joystick on the therapist remote overrides all other control inputs. The LED will light up when the remote is out of range – in this case the Power Wheelchair Trainer will stop moving.</p>  <p>Figure 10</p>

Figure 49: Quick Start Guide Front

Charging

The charging plug is located at the rear of the control unit. To charge the device connect the charger to the charging plug (Figure 11) and use a 3-prong wall outlet, preferably with a surge protector. It is recommended to keep the charger plugged in anytime the Power Wheelchair Trainer is not in use to prolong the life of the batteries.



Figure 11

Disassembly for Transport

The power unit can be detached from the platform for easier transport in a vehicle. Before detaching the power unit it is recommended to drive the PWCT near the transport vehicle, lower the platform to the loading position, and connect the tie downs together (Figure 12).



Figure 12

1. Unplug the emergency stop cable (Figure 13).



Figure 13

2. Unplug the blue battery connector (Figure 14).



Figure 14

3. Unplug the gray battery connectors (Figure 15) and the motor connectors (Figure 16).



Figure 15



Figure 16

4. Remove the batteries and unplug the linear actuator connector (Figures 17 and 18).



Figure 17



Figure 18

5. Unlock the two white latches (Figure 19).



Figure 19

6. Lift the platform up and away from the power unit to disconnect the two. Let the power unit drop forward and pull the power unit away from the platform (Figure 20).



Figure 20

The power unit and platform can now be loaded into a vehicle for transport. To reassemble the platform and power unit, follow the previous steps in reverse order.

Profile settings

The following is a list of the available profile settings and their range of values for joystick and switch use unless indicated otherwise:

- Throw – forward, reverse, & turn (joystick only)
 - How far the joystick moves to reach top speed.
 - Setting: 0.05 to 2.50 in steps of 0.05
 - Higher number = joystick deflects less to reach top speed
- Top speeds – forward, reverse, & turn
 - Setting: 5 to 125 in steps of 5
 - 125 is really fast, 5 is really slow
- Sensitivity AKA tremor dampening
 - How quickly the wheelchair responds to joystick movement or switch activation.
 - Setting: 1 to 10
 - Higher number = quicker response
- Acceleration / deceleration
 - Setting: 1 to 25
 - Higher number means quicker acceleration / deceleration.
- Center Dead Band (joystick only)
 - Setting: 1 to 10
- Outer Dead Band (joystick only)
 - A circular band along the perimeter of the joystick where the motors optionally shut off.
 - Setting: Off, Immediate, seconds until shut-off
- Invert (joystick only)
 - Setting: On, Off
- Proportional as switch (joystick only)
 - Setting: On, Off
 - Converts the proportional joystick to a switch joystick.

Figure 50: Quick Start Guide Back

Power Wheelchair Trainer Operator Manual

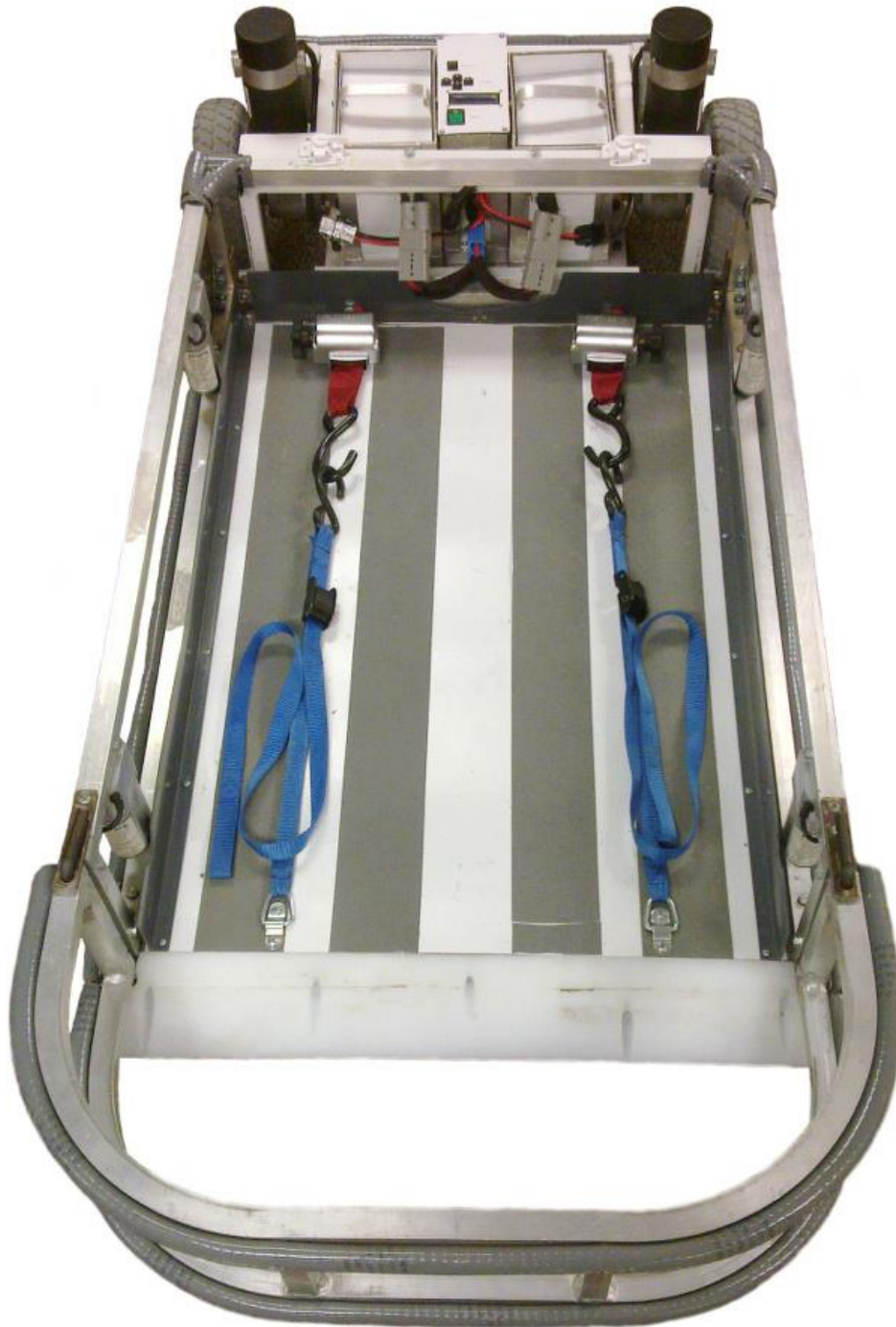


Figure 51: The Power Wheelchair Trainer

IX. Intended Use and Safety

IX.1 Intended Use

- The Power Wheelchair Trainer is intended to be used in conjunction with a power mobility training program developed by a physical therapist or occupational therapist.
- The Power Wheelchair Trainer is not to be used outdoors. The Power Wheelchair Trainer is intended only to be used indoors on smooth surfaces.

IX.2 Safety

- The Power Wheelchair Trainer is a prototype device and has not yet been approved by the Food and Drug Administration for clinical use.
- Two emergency stop switches are located on the back of the Power Wheelchair Trainer, and one emergency stop switch on the therapist remote. If the emergency stop has been engaged it will be indicated on the liquid crystal display (LCD), and the power must be turned off and on to make the Power Wheelchair Trainer operational again.
- Ensure that the remote, joystick, and trainer are powered OFF before storing or charging.
- Turn OFF the remote and joystick power before loading or unloading a learner.
- Ensure that an appropriate driving profile has been selected for the specific learner.
- Do not change driving profiles or settings while the Power Wheelchair Trainer is in use.
- The Power Wheelchair Trainer will not drive unless the therapist remote is powered ON and in close proximity to the trainer.
- The Power Wheelchair Trainer is to be used only while supervised by a qualified therapist.
- The supervising therapist must hold the remote at all times.

X. Parts of the system

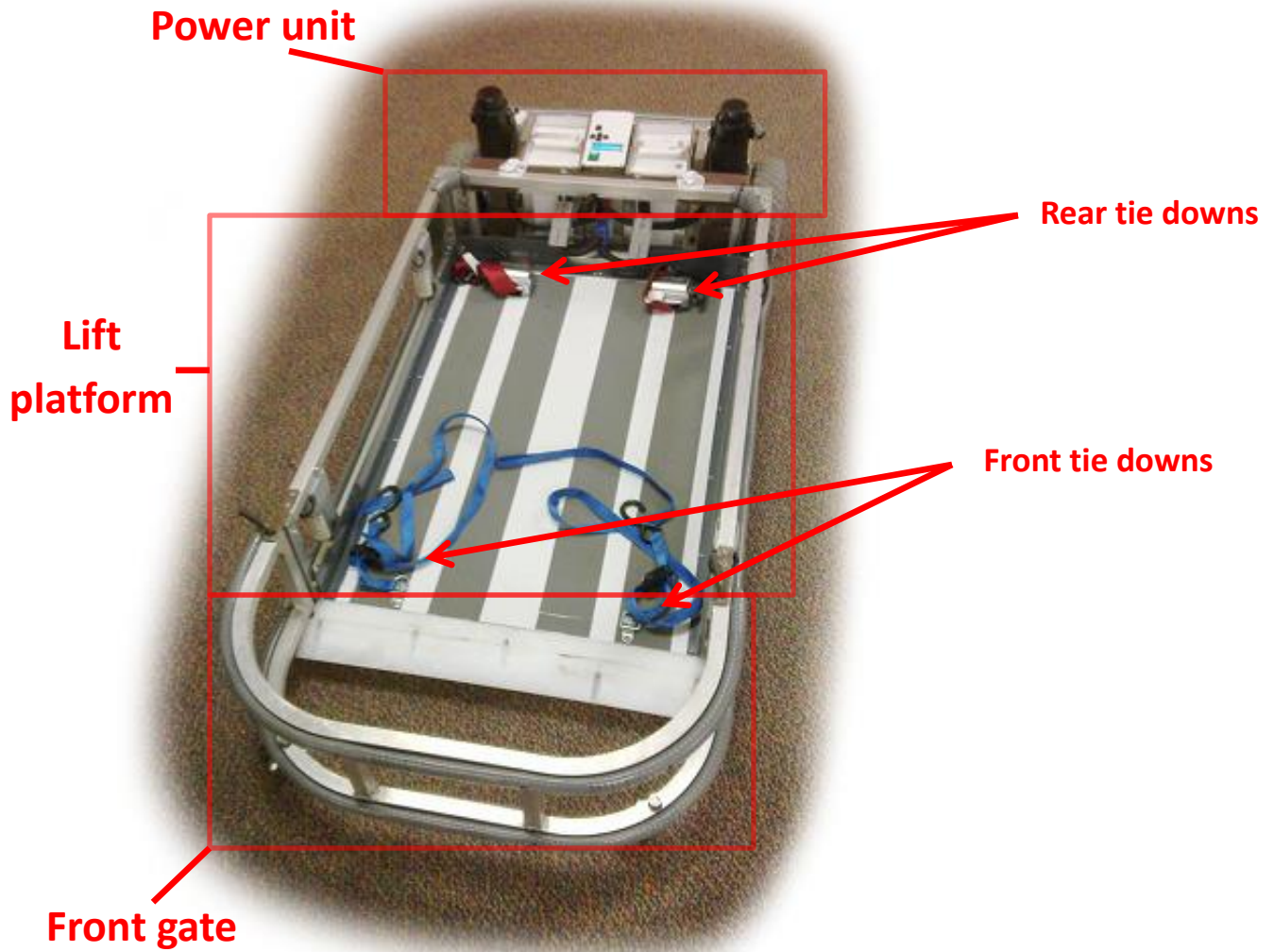


Figure 52: Major Parts of the Power Wheelchair Trainer

X.1 Power Unit

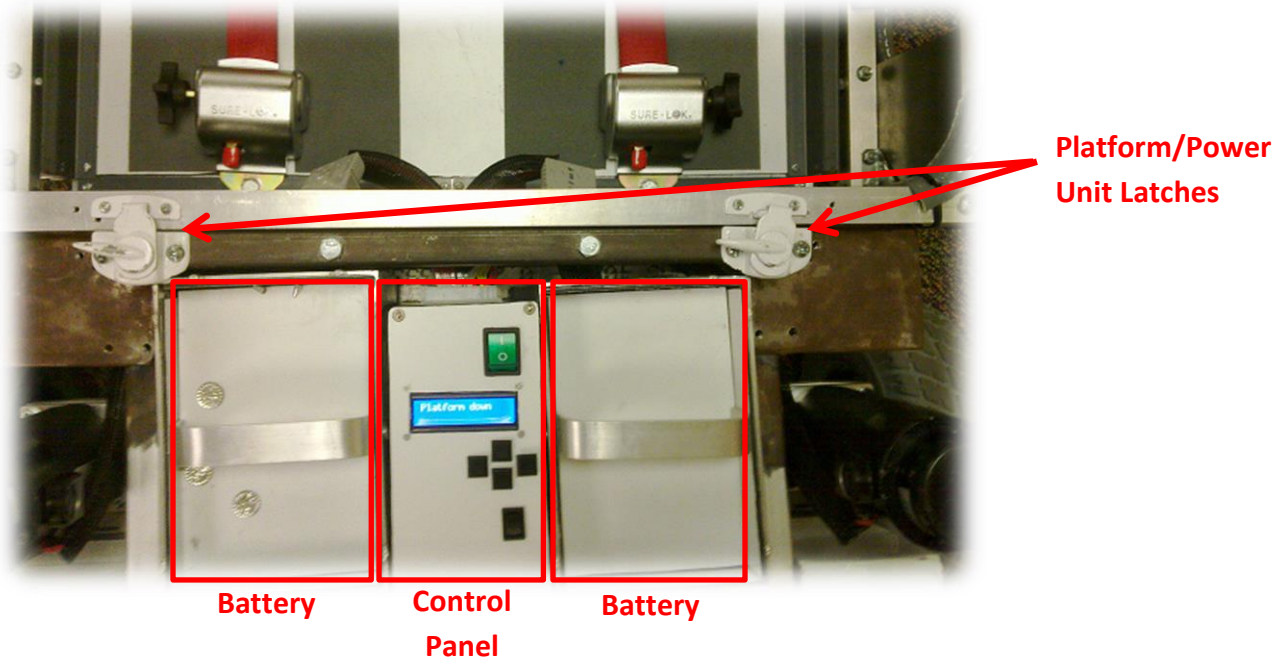


Figure 53: Power Unit Components

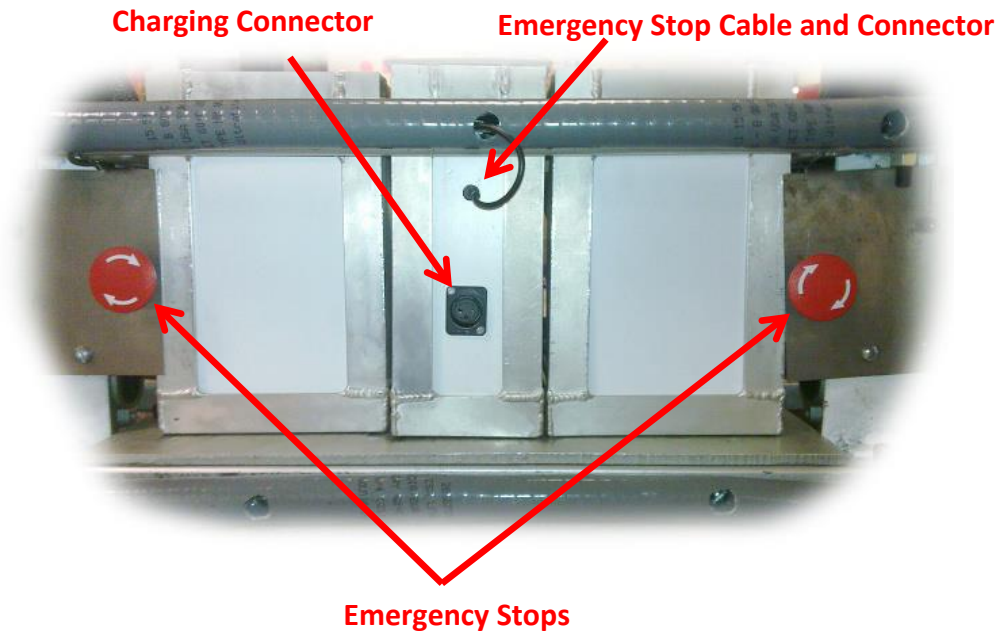


Figure 54: Rear Components

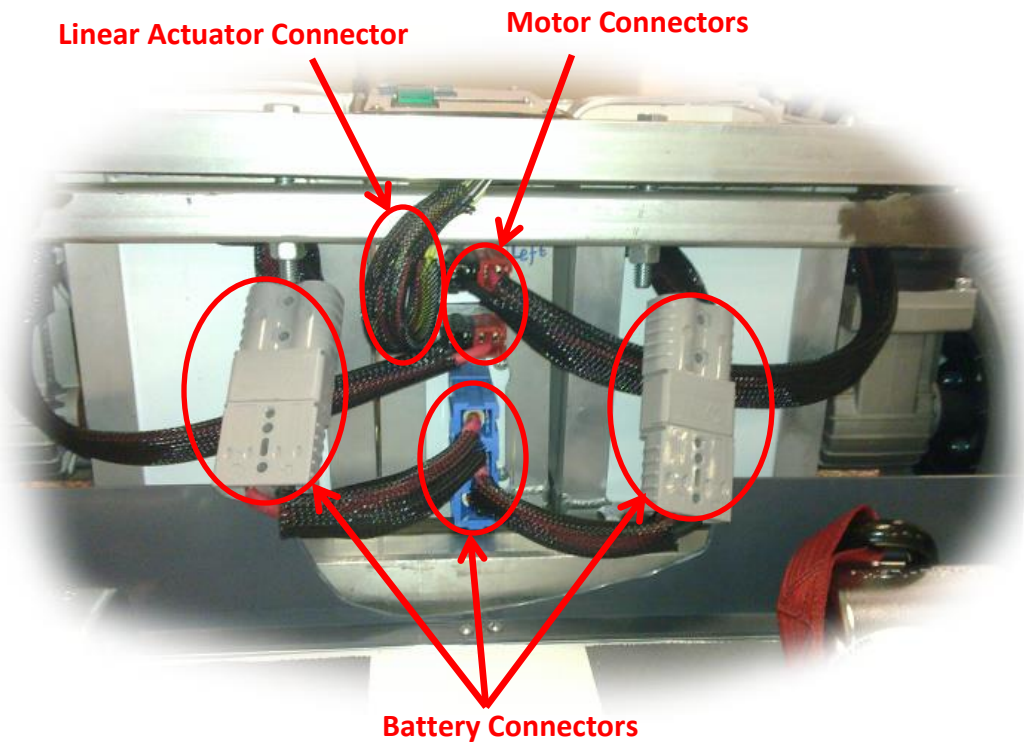


Figure 55: Cables and Connectors

X.2 Control Panel

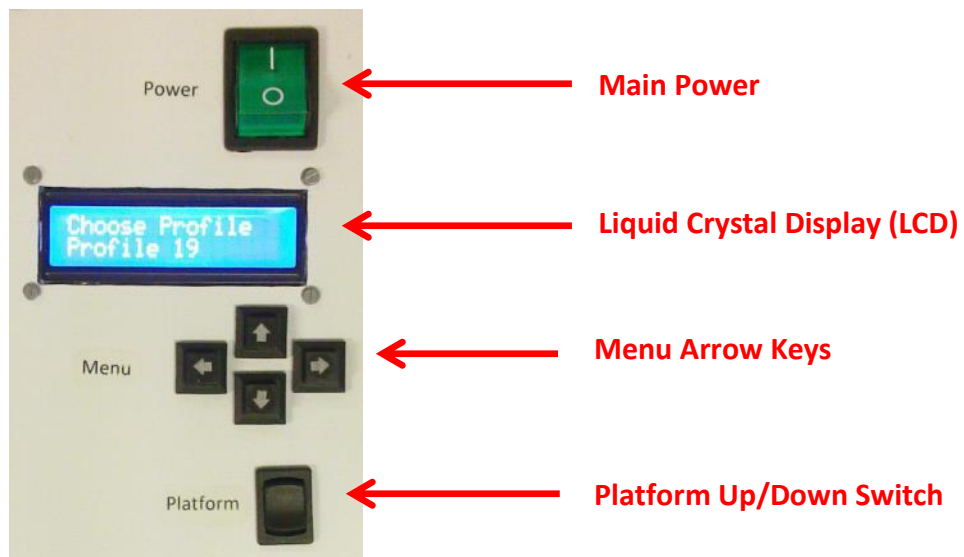


Figure 56: Control Panel Components

X.3 Learner Joystick

See Section XII.2.2 – Learner Joystick for a complete overview of the learner joystick function.

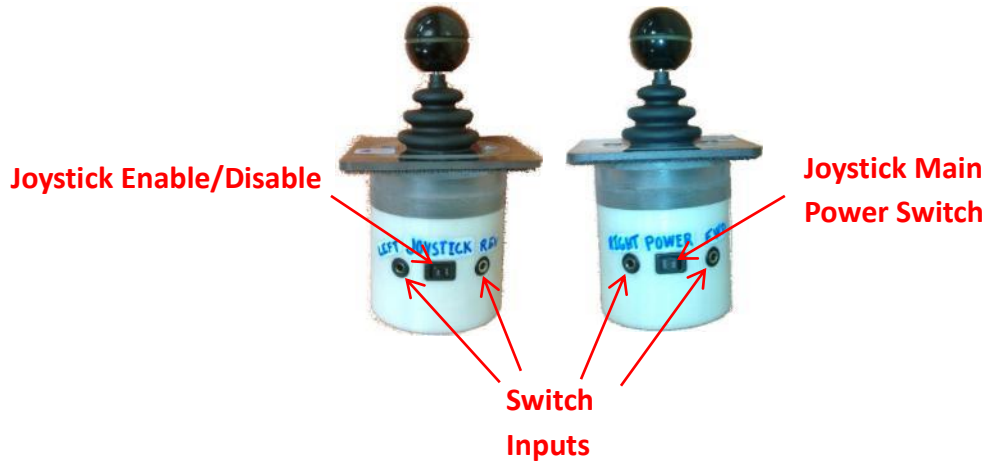


Figure 57: Wireless Joystick Components

X.4 Therapist Remote

See Section XII.2.3 – Therapist Remote for a complete overview of therapist remote function.



Figure 58: Therapist Remote Components

X.5 Tray

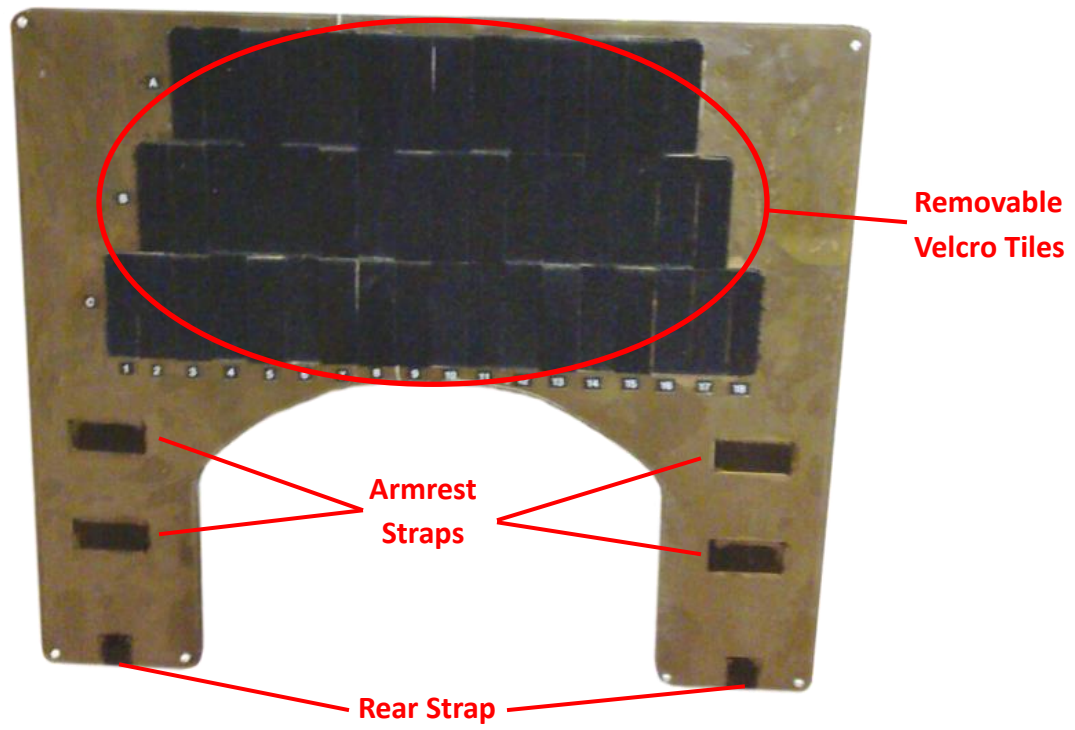


Figure 59: Tray

XI. Prior to Use

Ensure that the following cables are properly plugged in prior to use:

- Linear actuator plug (see Figure 55)
- Motor cables (see Figure 55)
- Battery cable (see Figure 55)
- Emergency stop cable (see Figure 54)

Ensure that the rear emergency stop switches are disengaged / pulled out (see Figure 54). If the emergency stop has been engaged it will be indicated on the liquid crystal display (LCD), and the power must be turned off and on to make the Power Wheelchair Trainer operational.

Ensure that the front gate pins are fully inserted (see Figure 62) and in line with the frame as shown in Figure 51.

Ensure that the main batteries are charged before daily use. Additionally, check the batteries in the joystick and remote.

XI.1 Charging

Before charging, ensure that the Power Wheelchair Trainer is turned OFF and the battery cable is properly connected. The charging plug is located at the rear of the power unit (see Figure 60). To charge the device, connect the charger to the charging plug and use a standard 115V, 60Hz, grounded 3-prong electrical outlet, preferably with a surge protector. The charger will indicate the current battery charge level once plugged into an electrical outlet. When the light on the charger is green, the batteries are fully charged and the charger will automatically stop charging.

The charging time will vary depending on the initial charge level and condition of the batteries. The time that the Power Wheelchair Trainer will run on a full charge varies and depends on usage, battery condition, and other environmental factors such as temperature, incline, type of flooring.

Due to the chemistry of the gel lead acid batteries, they will last longer when kept fully charged. Therefore it is recommended to keep the charger plugged in anytime the Power Wheelchair Trainer is not in use. If charging is not a possibility during storage, disconnect the battery cables to reduce the self-discharge rate and fully charge the batteries at least once every 3 months. Never store the Power Wheelchair Trainer with discharged batteries.

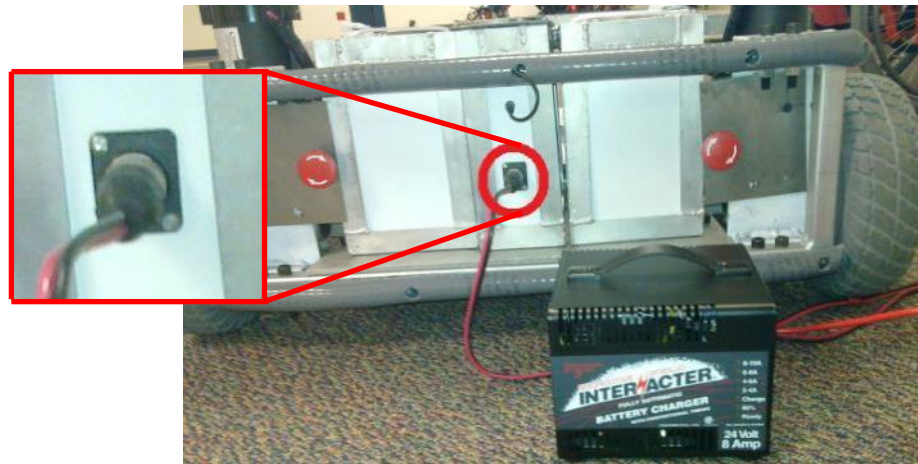


Figure 60: Location of Charging Plug

XII. Operation

XII.1 Loading and Unloading a Learner

1. Ensure that all cables are properly plugged in (see Section XI – Prior to Use) and turn the power ON with the green power switch on the control panel
2. Press and hold the platform down switch on the control panel until the platform touches the floor
3. Once the platform has been lowered, pull the two pins out to detach and remove the front gate

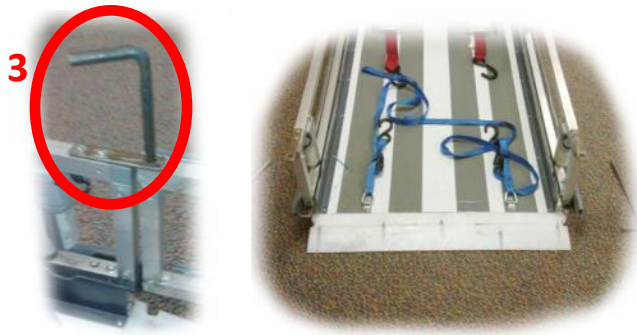


Figure 62: Front Gate Pins

4. Wheel the learner onto the platform with his or her back facing the power unit and ensure that the wheelchair's front casters are facing forward, parallel to the gray lines on the platform
5. Once the learner has been loaded, attach the front straps to the wheelchair's front tie down brackets and adjust the strap



Figure 61: Location of Power Switch and Platform Switch



Figure 63: Wheelchair on the Platform

lengths to position the learner near the front of the platform. Positioning the learner near the front helps with visual perception.

6. Engage the wheelchair brakes, then push the red lever on the rear Sure-lok tie downs to attach the straps to the wheelchair's rear tie down brackets. Turn the knob to retract the rear tie downs, and then re-check the front tie downs to make sure the wheelchair is firmly secured.

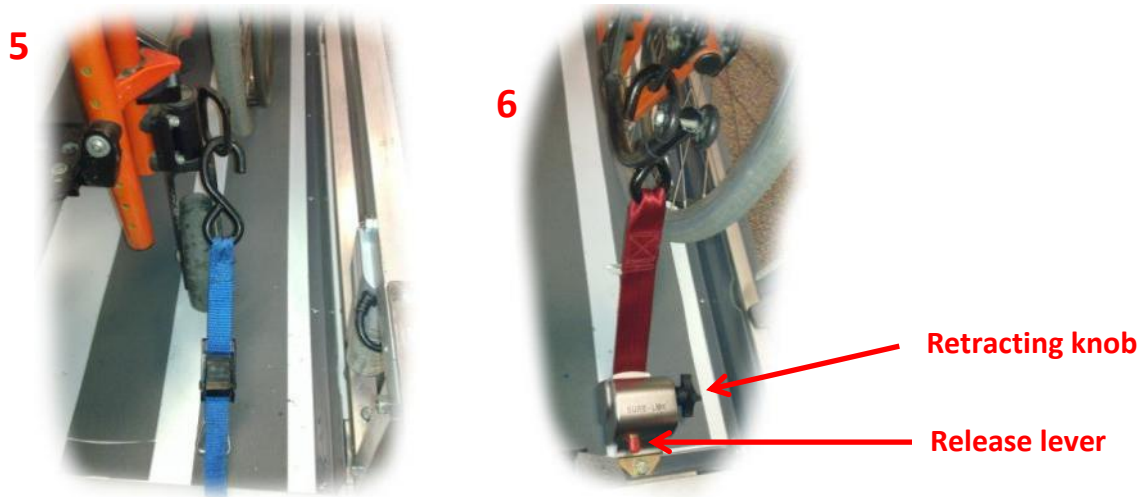


Figure 64: Front and Rear Tiedowns

7. Once the wheelchair has been securely locked in place, reattach the front gate and insert the locking pins all the way, making sure they line up with the frame. Raise the platform all the way up.

XII.2 Drive Control

XII.2.1 Tray Setup

1. With the learner sitting in the wheelchair, place the tray on the armrests.



Figure 65: Tray on Wheelchair

2. Secure the armrest straps around the wheelchair's armrests. There are two straps on each side.



Figure 66: Armrest Straps

3. Secure the tray's long rear strap around the back of the wheelchair. Be careful not to tighten the strap too much for the comfort of the learner.

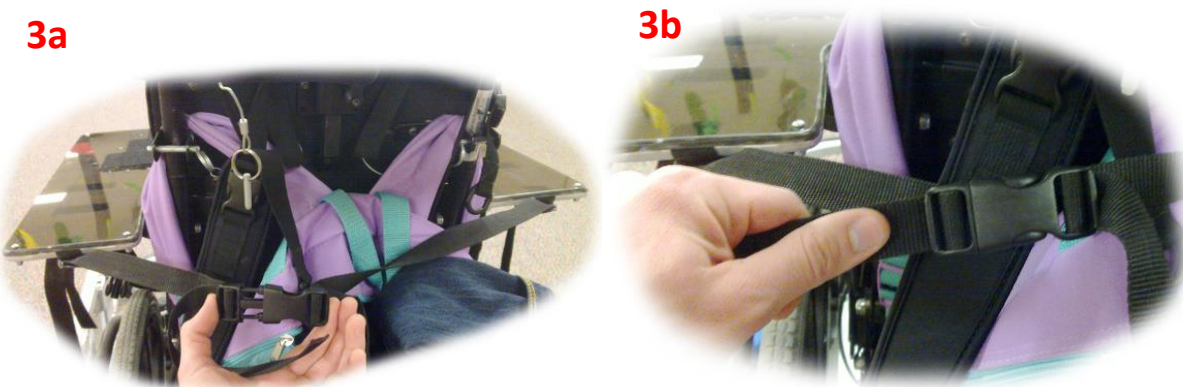


Figure 67: Rear Strap

XII.2.2 Learner Joystick

The wireless learner joystick may be mounted in any position in the tray by removing 3 tiles on the tray in the desired position. To help ensure consistent joystick positioning the tray tiles are labeled in a grid of letters and numbers. Ensure that the arrow on the joystick base is pointing forward (away from the learner), and secure the joystick with Velcro. The base of the joystick should be flush with the tray.

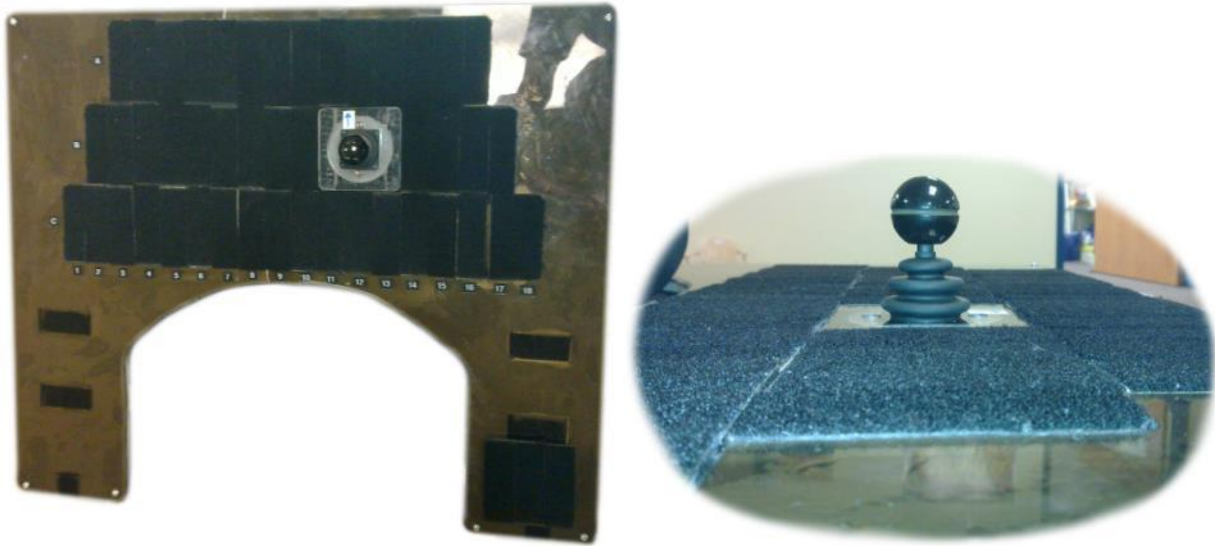


Figure 68: Joystick Mounted in Tray

Up to four external switches may be plugged in to the learner joystick and secured to the tray or wheelchair. Make sure all wires are tucked in the wheelchair or on the platform away from the wheels of the trainer.

The joystick and switch inputs may be used at the same time. The joystick, when

enabled, will override the switch inputs. The joystick may optionally be disabled while switches are

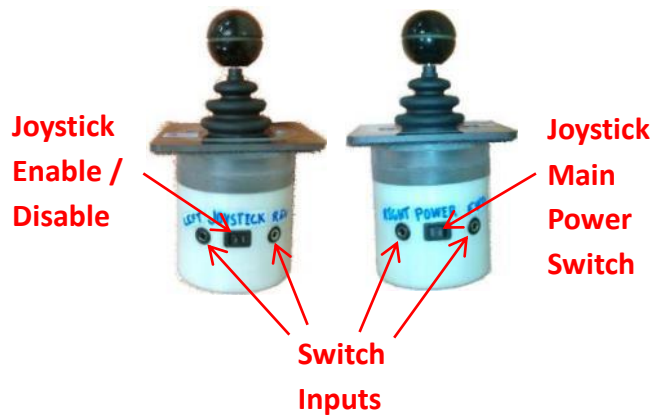


Figure 69: Wireless Joystick Components

being used for drive control. The joystick enable/disable switch affects the joystick only – any switches that are plugged in will work anytime the joystick main power is on.

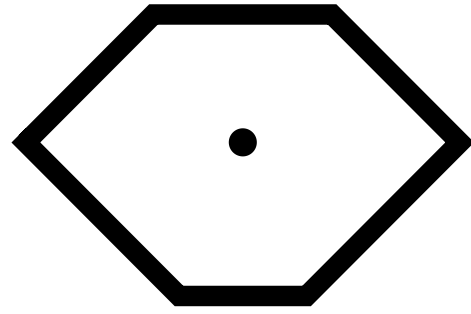


Figure 70 shows the “shape” of the learner joystick.

Figure 70: Learner Joystick Shape

XII.2.3 Therapist Remote

The joystick on the therapist remote overrides all other control inputs.

The Power Wheelchair Trainer will not drive unless the therapist remote is powered ON and in close proximity to the trainer. The light emitting diode (LED) on the therapist remote will light up when the wireless connection is lost between the remote and control unit – in this case the Power Wheelchair Trainer will stop moving.



Figure 71: Therapist Remote Components

The therapist remote has a red emergency stop switch that immediately shuts down operation of the Power Wheelchair Trainer. The intent of the emergency stop is to stop the Power Wheelchair Trainer in case of emergencies or potentially dangerous situations. If the emergency stop has been engaged (on the remote or power unit) it will be indicated on the liquid crystal display (LCD), and the power must be turned off and on to make the Power Wheelchair Trainer operational again.

There is a driving profile specifically for the therapist remote to ensure safe driving speeds while driving the Power Wheelchair Trainer with the therapist remote. All profile parameters are available in this special profile (see Section 4.3.1 – Profile Settings).

XII.3 Driving Profiles

Before changing any settings, please ensure that the wireless joystick and therapist remote are powered OFF to avoid accidentally moving the Power Wheelchair Trainer.

Use the four arrow keys below the liquid crystal display (LCD) screen to navigate the menu interface. Right/left chooses the setting to change, and up/down changes the value. For example, scroll all the way to the left to get to the profile setting, and then scroll up or down to change the current profile. A driving profile has a customized speed, acceleration, sensitivity, etc. for each learner (see Section XII.3.1 – Profile Settings). Once an appropriate profile is selected, turn the joystick power ON and turn the therapist remote power ON. The learner is now ready to drive.

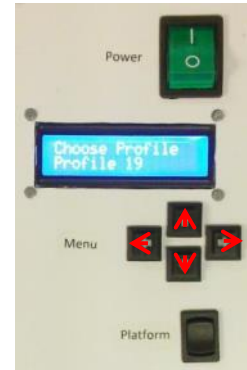


Figure 72: Navigating the Menu

XII.3.1 Profile Settings

The following is a list of the available profile settings and their range of values, along with reasonable values to be used as starting points. Each setting applies to both joystick and switch use unless indicated otherwise. Anytime a setting is changed it is immediately and automatically saved, and the settings are remembered even when the power is turned off.

It is recommended that a transport profile is set up so the therapist can easily drive the Power Wheelchair Trainer without changing any learner profiles.

All values are relative (i.e. no standard units) unless otherwise specified.

1. Profile

- How it appears on the LCD: “Choose Profile”
- Used for easily switching between driving profiles customized for different learners

- There is a fixed number of 20 driving profiles, plus one special profile that applies to the therapist remote joystick
- To edit the profile name, hold the left button for 2 seconds to enter the name edit mode.

Name edit mode is shown in Figure 73.

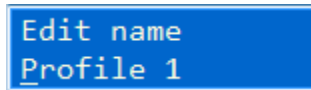


Figure 73: Name Edit Mode

Choose the character to modify with the right and left arrow keys as shown in Figure 74.

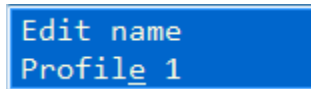


Figure 74: Choosing the Character to Modify

Modify the character with the up and down arrow keys as shown in Figure 75. The available profile name characters are: blank, uppercase A-Z, lowercase a-z, 0-9.

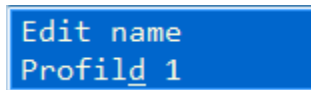


Figure 75: Modifying a Character

Hold the left button for 2 seconds to exit name edit mode.

2. Throw (forward, reverse, and turn)

- How it appears on the LCD: “Fwd Throw”, “Rev Throw”, “Turn Throw”
- This setting is not applicable when switches are used as input or when proportional as switch mode (setting 9) is active
- Definition: How far the joystick moves to reach top speed. A higher setting means the joystick moves less distance to obtain top speed. A lower number means the joystick moves farther to obtain top speed.

- Setting: 0.05 to 2.50 in steps of 0.05
- Reasonable starting values are 1.2, 0.8, and 1.0 for forward, reverse, and turn throw settings, respectively
- For a learner with minimal hand movement who will only move the joystick a small distance, values between 1.5 and 2.5 are recommended for all three throw settings

3. Top Speed (Forward, Reverse, Turn)

- How it appears on the LCD: “Fwd Speed”, “Rev Speed”, “Turn Speed”
- Definition: Maximum speed. A higher number means a higher top speed.
- Setting: 5 to 125 in steps of 5
 - Note: this is only an arbitrary numerical value. It is not miles per hour.
- Reasonable starting values are 35, 25, and 30 for forward, reverse, and turn speed, respectively
- If the Power Wheelchair Trainer is having trouble driving over thresholds in the floor while turning, or it is having trouble driving on carpet, try increasing the turn speed
- A heavier learner may require a higher top speed setting
- For a learner with minimal hand movement, it is recommended to set a relatively low top speed (around 30) for all three speed settings so they have the same response in all directions

4. Sensitivity

- How it appears on the LCD: “Sensitivity”

- Definition: How quickly the Power Wheelchair Trainer responds to joystick movement or activation of a switch. Also known as tremor dampening. A higher number means quicker response.
- Setting: 1 to 10
- A reasonable starting value is 8
- Typically sensitivity and acceleration/deceleration are adjusted together in order to achieve the desired behavior
- For learners who may have jerky or uncontrolled hand movement it is recommended to decrease the sensitivity

5. Acceleration/Deceleration

- How it appears on the LCD: “Acceleration”, “Deceleration”
- Definition: The maximum change in speed over time. A higher number means quicker acceleration or deceleration.
- Setting: 1 to 25
- A reasonable starting value is 22. A setting of 25 may result in jerkiness

6. Center dead band

- How it appears on the LCD: “Center DB”
- This setting is not applicable when switches are used as input or when proportional as switch mode (setting 9) is active
- Definition: A circular dead zone in the center of the joystick of configurable size where the joystick is considered to be centered. This setting determines how far the joystick handle

has to travel from the center for the motors to start moving. Figure 76 shows the “shape” of the joystick with the gray area in the middle corresponding to the center dead zone, where the size of the gray circle (the dead zone) is configurable. A larger setting means a larger dead zone.

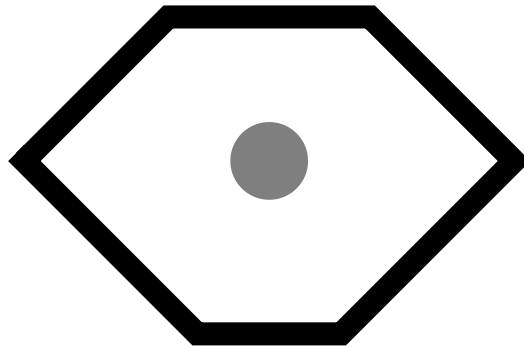


Figure 76: Center Dead Zone

- Setting: 1 to 10
- A reasonable starting value is 1
- This setting may be increased for learners who have jerky or uncontrolled hand (or upper extremity) movement if a lower sensitivity setting does not produce desirable results
- With an increased setting, the joystick handle must be moved further to be activated

7. Outer dead band

- How it appears on the LCD: “Outer DB”
- This setting is not applicable when switches are used as input or when proportional as switch mode (setting 9) is active
- Definition: A circular band along the perimeter of the joystick where the motors will optionally shut off immediately, or after a time-out. Figure 77 shows the “shape” of the joystick with the gray area corresponding to the outer dead zone. If a time setting is chosen, the motors will shut off if the joystick has been in the outer dead zone for the chosen

number of seconds. The joystick must return to center before the Power Wheelchair Trainer will start driving again (it is not necessary to cycle power).

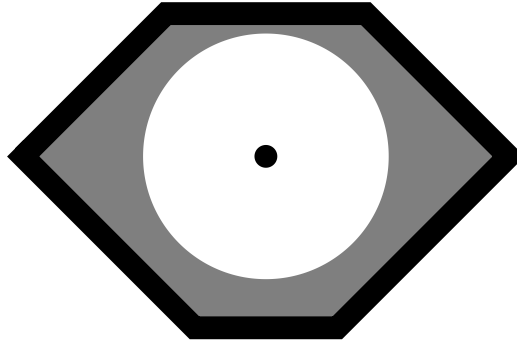


Figure 77: Outer Dead Zone

- Setting: Off, Immediate, seconds until motors shut off
- It is recommended to set a timeout for learners with high muscle tone who may be forceful with hand movement and lock their elbow into extension and be unable to voluntarily release

8. Invert

- How it appears on the LCD: "Invert"
- This setting is not applicable when switches are used as input
- Definition: swap forward / reverse
- Setting: Off, On
- It is recommended to turn this setting on for learners who have better control pulling the joystick toward their body than they do pushing it away

9. Proportional as switch

- How it appears on the LCD: "PropAsSwitch"

- This setting is not applicable when switches are used as input
- If proportional as switch mode is active, the following settings do not apply: throw (setting 2), center dead band (setting 6), outer dead band (setting 7)
- Definition: converts the proportional joystick into a switch joystick with four directions
- Setting: Off, On

XIII. Disassembly for Transport

The power unit may be detached from the platform to allow for easier vehicle transport. Before detaching the power unit it is recommended to drive the Power Wheelchair Trainer near the transport vehicle.

1. Lower the platform to the loading position, turn off the power, and connect the tie downs together.

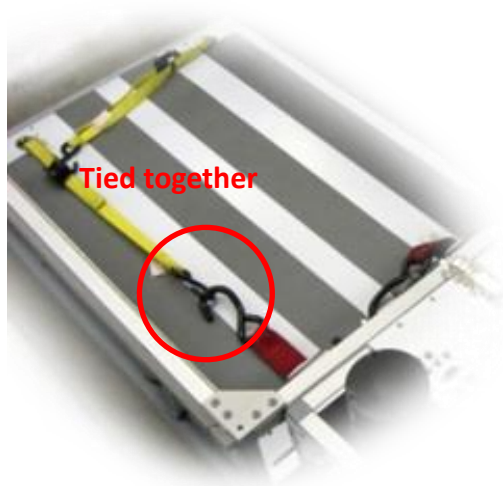


Figure 78: Tiedowns Tied Together

2. Unplug the emergency stop cable.



Figure 79: Emergency Stop Cable

3. Unplug the blue center battery connector.



Figure 80: Battery Connector

4. Unplug the two gray battery connectors and the motor connectors.

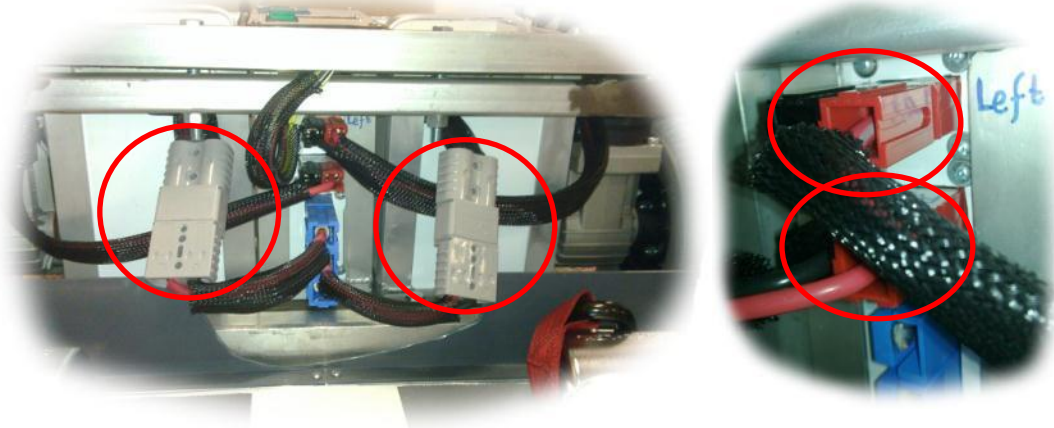


Figure 81: Battery and Motor Connectors

5. Remove the batteries and unplug the linear actuator connector to remove the control panel.



Figure 82: Linear Actuator Connector

6. Unlock the two white latches.

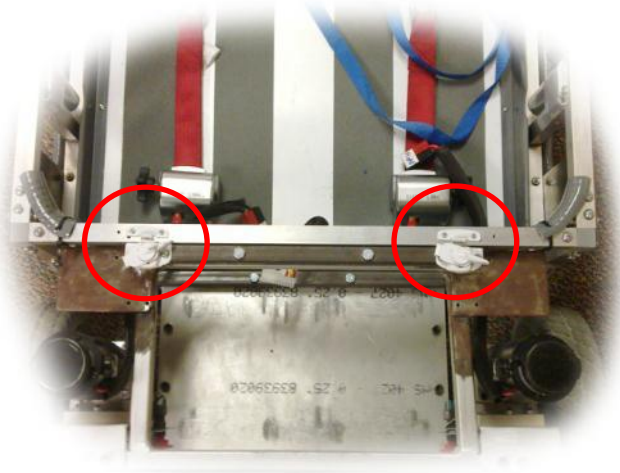


Figure 83: White Latches

7. Using proper body mechanics lift the platform up and away from the power unit to disconnect the two. Let the power unit drop forward and pull the power unit away from the platform.

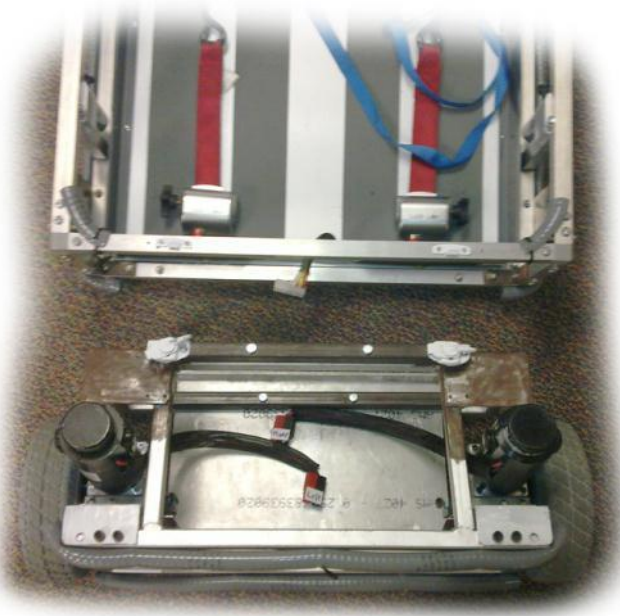


Figure 84: Power Unit Detached

8. If desired, remove the front gate.

The power unit and platform may now be loaded into a vehicle for transport. It is recommended that 2 people lift the power unit and the platform into and out of the vehicle. To reassemble the Power Wheelchair Trainer, follow the above steps in reverse order.

XIV. Troubleshooting

Symptom	Troubleshooting steps
The Power Wheelchair Trainer does not turn on, or the LCD is dark when the power switch is on	<ul style="list-style-type: none">• Ensure that all cables are properly connected (see Section XI – Prior to Use)• Ensure that the batteries are charged (see Section XI.1 – Charging)• Turn the trainer ON using the main power switch on the control panel (see Figure 61)
LCD displays “E-stop”	<ul style="list-style-type: none">• Ensure that the rear emergency stop cable is plugged in (see Section XI – Prior to Use)• Ensure that the rear emergency stop switches are disengaged/pulled out (see Figure 54: Rear Components)• Turn the main power off and on again to clear the emergency stop condition (see Figure 61)

Symptom	Troubleshooting steps
The LCD is on, but the trainer is not moving	<ul style="list-style-type: none"> • Ensure that the motor cables are properly connected (see Section XI – Prior to Use) • Ensure that the therapist remote is ON and in close proximity to the trainer, and the LED on the remote is off (see Section X.4 – Therapist Remote) • Ensure that the learner joystick is powered ON, and the joystick switch is ON (see Section X.3 – Learner Joystick) • Check the batteries in the remote and joystick and replace as necessary • It is possible that the active profile is configured inappropriately – try switching to another profile or use the recommended starting values described in Section XII.3.1 – Profile Settings
The LED on the therapist remote is on	<ul style="list-style-type: none"> • Ensure that the trainer is ON and that the therapist remote is in close proximity to the trainer (see Section X.4 – Therapist Remote)
The trainer continues to move after the joystick is released	<ul style="list-style-type: none"> • Increase the deceleration setting (see setting #5 in Section XII.3.1 – Profile Settings)
The wheels are spinning	<ul style="list-style-type: none"> • Decrease all three top speed settings (see setting #3 in Section XII.3.1 – Profile Settings)

Symptom	Troubleshooting steps
The trainer is unable to turn	<ul style="list-style-type: none"> • Increase the turn speed setting (see setting #3 in Section XII.3.1 – Profile Settings) • If increasing the turn speed does not solve the problem, increase the acceleration and sensitivity settings (see settings #5 and #4 in Section XII.3.1 – Profile Settings)
LCD displays “EEPROM corrupt” or any other error code	<ul style="list-style-type: none"> • Turn the main power off and on again (see Figure 61) • If the problem persists, take the Power Wheelchair Trainer back to the manufacturer

XV. Maintenance

Due to the chemistry of the gel lead acid batteries, they will last longer when kept fully charged. Therefore it is recommended to keep the charger plugged in anytime the Power Wheelchair Trainer is not in use. If charging is not a possibility during storage, disconnect the battery cables to reduce the self-discharge rate and fully charge the batteries at least once every 3 months. Never store the Power Wheelchair Trainer with discharged batteries.

If the trainer has not been used in a long time, perform all the checks described in Section XI – Prior to Use. Additionally, ensure that all bolts and screws are tightened properly.

The batteries and electronics are sensitive to extreme high or low temperatures. Do not expose the Power Wheelchair Trainer's batteries, electronics, and motors to temperatures below 0°C or above 55°C (32°F to 131°F). It's best to store the Power Wheelchair Trainer at or slightly below normal room temperature. Do not store the power unit electronics in a vehicle overnight.

XVI. References

AbleNet. (2012). *Switch selection guide*. Retrieved from

http://www.ablenetinc.com/Portals/0/KnowledgeBase/Selection_Grids/Switch_Selection_Grid.pdf

atan2. (n.d.). Retrieved from <http://www.cplusplus.com/reference/cmath/atan2/>

Atwell, A. K., & Pontin, T. K. (2000). *U.S. Patent No. 6,043,806 A*. Washington, DC: U.S. Patent and Trademark Office.

Bresler, M. I. (1990). Turtle trainer: A way to evaluate power mobility readiness. *Proceedings of the 13th Annual RESNA Conference*, 399-400.

Dimension Engineering. (2011). *Sabertooth 2x60 User's Guide*. Retrieved from

<https://www.dimensionengineering.com/datasheets/Sabertooth2x60.pdf>

Durkin, J. (2006). *Developing powered mobility with children who have multiple and complex disabilities: moving forward* (Doctoral dissertation), University of Brighton.

Durkin, J. (2009). Discovering powered mobility skills with children: 'Responsive partners' in learning. *International Journal of Therapy and Rehabilitation*, 16(6), 331.

Food & Drug Administration. (1997). *Design control guidance for medical device manufacturers*.

Retrieved from

<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm070642.pdf>

Hays, R. M. (1987). Childhood motor impairments: Clinical overview and scope of the problem. Paper presented at the *Childhood Powered Mobility: Developmental, Technical, and Clinical*

Perspectives. Proceedings of the RESNA First Northwest Regional Conference. Washington, DC: Rehabilitation Engineering and Assistive Technology Society of North America, 1-10.

Invacare. (2011, March). *Invacare® MK6i™ electronics field reference guide*. (Rev 1-03/11) [Field Reference Guide]. Elyria, Ohio: Invacare Corporation.

Jones, M. A., McEwen, I. R., & Neas, B. R. (2012). Effects of power wheelchairs on the development and function of young children with severe motor impairments. *Pediatric Physical Therapy : The Official Publication of the Section on Pediatrics of the American Physical Therapy Association*, 24(2), 131-40; discussion 140. doi:10.1097/PEP.0b013e31824c5fdc

Lange, M. L. (2010a). Proportional control for power wheelchairs: the right options can put your client in the driver's seat. *Advance for Occupational Therapy Practitioners* 26(10), 13. Retrieved from <http://occupational-therapy.advanceweb.com/Archives/Article-Archives/Proportional-Control-For-Power-Wheelchairs.aspx>

Lange, M. L. (2010b). Alternative Access: Options beyond joysticks for driving power wheelchairs. *Advance for Occupational Therapy Practitioners* 26(12), 13. Retrieved from <http://occupational-therapy.advanceweb.com/Archives/Article-Archives/Alternative-Access.aspx>

Livingstone, R. (2010). A critical review of powered mobility assessment and training for children. *Disability & Rehabilitation: Assistive Technology*, 5(6), 392-400.

Nilsson L. (2007). *Driving to Learn: the process of growing consciousness of tool use – a grounded theory of de-plateauing* (Doctoral dissertation). Lund University, Sweden. Retrieved from <https://lup.lub.lu.se/search/publication/548098>

Nordic Semiconductor. (2008). *nRF24L01+ Single Chip 2.4GHz Transceiver Product Specification v1.0*. Retrieved from

http://www.nordicsemi.com/kor/content/download/2726/34069/file/nRF24L01P_Product_Specification_1_0.pdf

PG Drives Technology. (2011). Programming. In *R-Net technical manual*. (SK77981/7) [Technical Manual]. Christchurch, UK: PG Drives Technology

Sitronix. (2006). *ST7066U Dot Matrix LCD Controller/Driver*. Retrieved from http://www.newhavendisplay.com/app_notes/ST7066U.pdf

Sparkfun. (2009). *Nordic FOB*. Retrieved from <https://www.sparkfun.com/products/retired/8602>

Sunplus Technology. (2003). *SPLC780D 16COM/40SEG Controller/Driver*. Retrieved from http://www.newhavendisplay.com/app_notes/SPLC780D.pdf

Tefft, D., Guerette, P., & Furumasu, J. (1999). Cognitive predictors of young children's readiness for powered mobility. *Developmental Medicine and Child Neurology*, 41(10), 665-670.

Teixeira, M. B., & Bradley, R. (2002). Overview. In *Design controls for the medical device industry*. London, England: CRC Press.

Åström, K. J., & Hägglund, T. (1995). *PID Controllers: Theory, Design, and Tuning* (2nd ed.). Research Triangle Park, NC: The International Society for Measurement and Control.