The University of Western Australia

School of Mechanical, Materials and Mechatronics Engineering

Omni-Directional Wheelchair

Honours Thesis

Benjamin Woods

10218282

Bachelor of Engineering (Mechatronics)

Faculty of Engineering, Computing and Mathematics

Supervisors:

Associate Professor Thomas Bräunl

Mr. Chris Croft

Centre for Intelligent Information Processing

School of Electrical and Electronics Engineering

30 October, 2006

Benjamin Woods
12 Anaconda Place
Sorrento, WA 6020

30th October 2006

Professor Mark Bush
The Dean
Faculty of Engineering, Computing and Mathematics
The University of Western Australia
35 Stirling Highway
Crawley, WA 6009

Dear Professor Bush,

It is with pleasure that I present this thesis, entitled "Omni-Directional Wheelchair" in partial fulfilment of the requirements for the degree of Bachelor of Engineering. I hereby certify that both this document and the proceedings described within are my own work unless otherwise stated.

Yours sincerely,

Benjamin Woods
10218282

Abstract

Since the beginning of 2004, the University of Western Australia's Centre for Intelligent Information Processing Systems (CIIPS) has been developing an omni-directional wheelchair. Omni-directional vehicles can turn and drive in any direction, including directly sideways. Therefore, an omni-directional wheelchair allows the user to navigate through a confined environment with less difficulty than would otherwise be possible with a conventional wheelchair.

This project aims to improve the driving accuracy, human interface and comfort of the already existing omni-directional wheelchair found in the mobile robotics laboratory at the University of Western Australia. This will be accomplished by altering the wheels, batteries, motor driver cards, joystick, control software, chassis and suspension system.

Acknowledgements

I would like to thank the following people for helping me produce this work. Without them, it would surely not have been possible.

- Associate Professor Thomas Bräunl for your guidance throughout the life of this project (even whilst in Germany).
- Mr. Chris Croft for catching the handball from Thomas.
- Dr. Nathan Scott for your unforgiving eye and advice regarding the wheelchair suspension system.
- The friendly staff and students within CIIPS and the EE workshops. It has been a pleasure to work with you for a year.
- My family and friends for putting up with my scarce free time over the past year and providing sound advice.

In addition to this, two organisations have been a great help in this project:

- DNM for supplying the university with the 4 shock absorbers necessary for the project - at no cost! Along with shock absorbers for mountain bikes, DNM also provide a large range of other suspension systems for bicycles and dirt bikes. More details available at http://www.dnmsuspension.com/.
- TADWA for their supply of knowledge regarding practical wheelchair design. TADWA provide a range of help for disabled people in WA. More details available at http://www.technicalaidwa.org.au/.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Objectives 2

1.3 Thesis Structure 2

2 Literature Survey **3**

2.1 Mecanum Wheels and Alternatives 3

2.2 Control Methods 5

2.3 Human Interface 6

3 Hardware **7**

3.1 General Arrangement 7

3.2 Mecanum Wheels 9

3.2.1 Technical Details 9

3.2.2 Kinematics 9

3.2.3 Wheel Rims 12

3.3 Motors 13

3.4 EyeBot 14

3.5 Batteries 15

3.6 Position Sensitive Devices 16

3.7 Footrests 16

4 Motor Driver Cards **17**

4.1 Introduction 17

4.2 Selection 18
4.3 Installation 19
4.4 Programming 20

5 Joystick **23**
5.1 Introduction 23
5.2 Selection 23
5.3 Installation 26
5.4 Programming 28

6 Low Level Driving Routines **31**
6.1 Introduction 31
6.2 Design 32

7 Suspension System **35**
7.1 Introduction 35
7.2 Design 36

8 Results, Testing and Simulation **43**
8.1 Motor Control 43
8.2 Joystick 43
8.3 Suspension and Chassis 46
8.4 Modelling and Simulation 47

9 Conclusion **49**
9.1 Outcomes 49
9.2 Recommendations 49

Bibliography **51**

A Code **55**
A.1 ODW.h 55
A.2 ODW.c 57
A.3 ODW_MotorCtrl.h 60

A.4 ODW_MotorCtrl.c 65
A.5 ODW_Joystick.h 71
A.6 ODW_Joystick.c 75
A.7 ODW_IR.h 80
A.8 ODW_IR.c 82
A.9 Makefile 87
A.10 Makeincl 88
A.11 test.c 89
A.12 mctest.c 90
A.13 joytest.c 92
A.14 joytest2.c 97
A.15 psdtest.c 103

B Mechanical and Electrical Designs **105**

B.1 Joystick Circuit 105
B.2 Suspension Designs 107

C Information and Brochures **121**

C.1 Roboteq AX1500 122

List of Figures

2.1 NASA OmniBot mobile base (Lippitt & Jones 1998) 3
2.2 Airtrax Sidewinder lift truck (Airtrax 2006) 4
2.3 New Mecanum wheel design (McCandless 2001) 4

3.1 The Mecanum wheel design 9
3.2 Force components in the Mecanum wheel (seen from below) 10
3.3 Wheel rotations for different driving directions (seen from below) . . 10
3.4 The Mecanum wheel rims after machining 13
3.5 A 24V brushed DC motor 13
3.6 EyeBot controller MK4 (front and back views) 14
3.7 Original car batteries and new sealed, deep-cycle gel batteries 15
3.8 Position sensitive devices (PSDs) 16
3.9 Wheelchair footrests donated by TADWA 16

4.1 Pulse-width modulation (PWM) signal 18
4.2 The new Roboteq AX1500 motor controller cards 19
4.3 Roboteq input/output power connections 20
4.4 Daisy-chain serial cable for Roboteq AX1500 cards 20

5.1 A 2 axis joystick compared to a 3 axis joystick 24
5.2 The 3 joysticks purchased and tested 25
5.3 The USB connection originally on the joystick 27
5.4 The new joystick circuit 27

6.1 Flow chart for the main ODW code 33

6.2 Flow charts for the IR and Joystick modules 34
6.3 Flow chart for the MotorCtrl module 34

7.1 DNM DV-22 bicycle shock absorbers 36
7.2 Battery box design 37
7.3 Battery box lid and chair mount 38
7.4 Perpendicular requirement for the adopted Mecanum wheel design . . 39
7.5 A basic but inadequate suspension design 39
7.6 Two alternative trailing arm designs 39
7.7 Trailing arm connection to the chassis 40
7.8 The forces experienced by the trailing arm suspension 40
7.9 A steel gusset is used to add strength 41
7.10 The shock absorber angle determines the effective spring constant . . 42

8.1 Actual motor speeds achieved 44
8.2 A textual joystick testing program 45
8.3 A graphical joystick testing program 45
8.4 The wheelchair model in EyeSim 48

9.1 The wheelchair after all alterations and improvements 50

B.1 The joystick circuit schematic 105
B.2 The joystick PCB design 106

List of Tables

3.1 Omni-directional wheelchair components summary - February 2006 . 8

4.1 RS232 settings for Roboteq AX1500 cards 20
4.2 Character strings for communication with daisy-chained Roboteq cards 21
4.3 Communication language for Roboteq AX1500 cards 22

5.1 The 3 joysticks purchased and tested 25
5.2 EyeBot I/O pins used 28
5.3 RoBIOS commands for reading buttons 29
5.4 RoBIOS commands for reading potentiometers 29

8.1 Actual motor speeds achieved 44

Nomenclature

Acronyms

ASCII	American Standard Code for Information Interchange
DOF	Degrees Of Freedom
EEPROM	Electrically Erasable Programmable Read-Only Memory
FSJ	Force Sensing Joystick
I/O	Input/Output
IR	Infra-Red
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MotorCtrl	Motor control
ODV/ODW	Omni-Directional Vehicle/Wheelchair
PSD	Position Sensitive Device
PSJ	Position Sensing Joystick
PWM	Pulse-Width Modulation
RS232	RETMA Standard 232
SHS	Square Hollow Section
USB	Universal Serial Bus
WC	WheelChair

Mathematical Notation

r	The radius of the Mecanum wheels
d	The width of the wheelchair (from wheel to wheel)
s	The length of the wheelchair (from wheel to wheel)
V_X	The forwards/backwards component of the wheelchair velocity (positive forwards)
V_Y	The left/right component of the wheelchair velocity (positive left)
$\dot{\varphi}$	The rotation speed of the wheelchair (positive counter clockwise)
θ_i	The rotational speed of wheel i (positive forwards)
x	The left/right axis of the joystick (positive right)
y	The forwards/backwards axis of the joystick (positive forwards)
z	The rotational axis of the joystick (positive clockwise)
t	The throttle of the joystick (positive up)

Chapter 1

Introduction

1.1 Background

Navigating a wheelchair through a confined or congested space can be extremely difficult. Conventional wheelchairs require an accurate approach path and a large amount of free space to undertake simple maneuvers such as driving through a doorway. One solution to this problem would be the development of a wheelchair that was able to drive directly sideways; otherwise known as an omni-directional wheelchair. The Centre for Intelligent Information Processing Systems (CIIPS) at the University of Western Australia has been working on the development of such a wheelchair since the beginning of 2004.

Omni-directional vehicles are able to drive in any direction in the 2D plane as well as rotate at the same time. In other words, they have three degrees of freedom. These vehicles differ from conventional drive arrangements (such as the Ackermann arrangement found in automobiles or the differential drive arrangement found in many scooters) in their ability to drive sideways.

Whilst working for the Swedish company Mecanum AB in 1973, Bengt Ilon came up with a design which, when used in a rectangular arrangement of 4 wheels, would allow such omni-directional motion. The wheel design, which was patented (Ilon 1975), is now known as the Mecanum wheel. The omni-directional wheelchair developed in this project uses this wheel design.

1.2 Objectives

At the beginning of 2006, the omni-directional wheelchair being developed was far from complete and had many flaws. The goals of this project were to make improvements to both the hardware and the software of the already existing wheelchair. Specifically, improvements were made to the already existing wheels, batteries, motor controller cards and chassis. Additionally, a joystick and chair (with both armrests and footrests) were added, progressing the project from a proof-of-concept design to a fully functioning wheelchair.

An entirely separate project has also been conducted on the wheelchair by Mei Leong for her final-year engineering project. Her project focused on both the development of the chair and the use of the position sensitive devices (PSDs) for semi-autonomous control of the wheelchair's motion. This included advanced driving methods such as obstacle avoidance, door-driving and wall following.

1.3 Thesis Structure

Chapter 2 presents background information on other relevant projects being carried around the world. It is divided into three parts describing different wheel designs that can provide omni-directional motion, control methods for the omni-directional wheelchair and the available options for the human interface of a wheelchair. Chapter 3 provides an overview of all of the hardware components used in the omni-directional wheelchair, focusing particularly on the Mecanum wheel design and the motors, micro-controller, batteries, sensors and footrests used. Chapters 4, 5 and 6 describe the motor controller cards and joystick used, and the software written to communicate with them and produce the desired motion from the given input. Chapter 7 describes a new chassis and suspension design which will provide a more accurate drive system, and a more comfortable experience for the user. Chapters 8 and 9 present the results of the project, a brief summary and recommendations for additional work on the project.

Chapter 2

Literature Survey

2.1 Mecanum Wheels and Alternatives

Omni-directional vehicles (ODVs) are by no means a new concept. Ilon (1975) details the design of the Mecanum wheel, which allows the omni-directional movement of a vehicle. This wheel is commonly used in robotic applications requiring a high degree of manoeuvrability, such as those experienced by NASA for hazardous environment exploration (Lippitt & Jones 1998) and Airtrax for their range of forklift trucks, aerial work platforms and mobility platforms (Airtrax 2006).

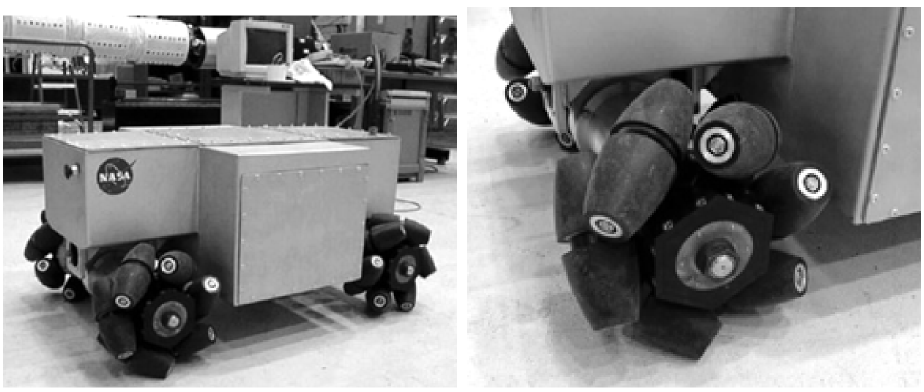


FIGURE 2.1: NASA OmniBot mobile base (Lippitt & Jones 1998)

Whilst normal wheels have a line contact with the ground, Mecanum wheels have a point contact with the floor in the ideal case. Due to the infinite pressure which would result from a point contact, either the floor or the wheel must deflect,

FIGURE 2.2: Airtrax Sidewinder lift truck (Airtrax 2006)

resulting in an area of contact. Regardless, the higher contact pressures that occur with Mecanum wheels can be minimised by using fewer rollers, with 6 rollers being found to be optimal (Dickerson & Lapin 1991). This fact was used by McCandless (2001) when developing his new Mecanum wheel design at the University of Western Australia.

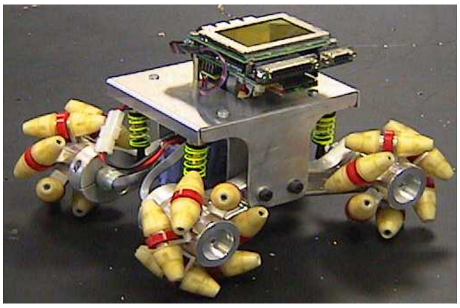


FIGURE 2.3: New Mecanum wheel design (McCandless 2001)

One disadvantage of the Mecanum design is the inefficient use of the kinetic energy supplied to the wheels by the motors. Due to the rotation of the exterior rollers, only a component of the force at the perimeter of the wheel is applied to the ground and the resulting force only partially contributes to the motion of the vehicle. Diegel, Badve, Bright, Potgieter & Tlale (2002) address this problem by introducing two new wheel designs, one with lockable rollers and the other with rotatable rollers. Although these designs are more efficient, their increased complexity makes them almost impractical in a university project with a limited budget.

Other designs use balls to facilitate the omni-directional movement, such as those used by West & Asada (1992) with the balls arranged along a crawler, and by Wada & Asada (1998) and Tahboub & Asada (2000) where four balls are used as the points of contact with the ground. These designs are relatively complicated and provide very few additional advantages over the Mecanum wheel design. As a result they are less likely to be adopted in a commercial wheelchair application, where manufacturing and maintenance costs are greatly reduced by simplicity.

2.2 Control Methods

The forward and inverse kinematics of the rectangular Mecanum wheel arrangement used in this project are derived by Viboonchaicheep, Shimada & Kosaka (2003). Unfortunately, using the wheel rotations and forward kinematic equations to determine the vehicle's current motion (also known as position rectification) is not possible with the Mecanum wheels being used in this project. This is due to the high level of slip experienced during normal operation providing inaccurate predictions of the ODV's velocity. As an alternative, a visual dead-reckoning system using a camera and optical flow analysis can be used to determine the change in the ODV's position (Nagatani, Tachibana, Sofne & Tanaka 2000), (Shimada, Yajima, Viboonchaicheep & Samura 2005) and (Cooney, Xu & Bright 2004). If accurate feedback of the wheelchair's position and/or velocity is required in a future project, this technique would be the most appropriate.

It is not uncommon for the user of an electric wheelchair to experience strong vibrations whilst driving. These vibrations have sometimes been known to excite the user's internal organs at their natural frequency, causing discomfort and sometimes nausea. By avoiding the natural frequency of the chair and human organs using frequency shape control, this effect can be minimised (Terashima, Miyoshi, Urbana & Kitagawa 2004) and (Urbano, Terashima, Miyoshi & Kitagawa 2005). Although this lies outside the scope of this project, if this problem is experienced at a future date it can be rectified using the methods described in these two papers.

2.3 Human Interface

An alternative to the common position sensing joystick (PSJ) found in most wheelchairs is the force sensing joystick (FSJ) or isometric joystick. This joystick remains stationary, but measures the degree of force placed on it in both the x and y axes. This requires virtually no range of hand motion, and allows easy optimisation for each individual user. For example, these joysticks can be used to reduce the effects of hand tremors on the vehicle's motion (Ding, Cooper & Spaeth 2004). Tests show that the PSJ and the FSJ provide similar accuracy and ease of use (Jones, Cooper, Albright & DiGiovine 1998) and (Cooper, Jones, Fitzgerald, Boninger & Albright 2000). The joystick used in this project has a third axis which is used to control the third degree of freedom of the wheelchair. As a result, the wheelchair continues to use the more conventional and intuitive PSJ.

For those individuals who do not have the ability to move their hands accurately, there are the options of chin operated FSJs (Guo, Cooper, Boninger, Kwarciak & Ammer 2002), or ultra-sonic non-contact head and voice activated control (Coyle 1995). Finally, a touch screen displaying video footage of the current surroundings can be used to control the vehicle (Kamiuchi & Maeyama 2004). Due to the additional complexity of controlling an ODV, this project assumes a user who has wrist movement which is acceptable for controlling a joystick.

As a user friendly alternative to obstacle avoidance, a variable impedance joystick that increases the impedance of tilting the joystick in the direction of an obstacle can be used (Kitagawa, Kobayashi, Beppu & Terashima 2001) and (Urbano, Terashima, Miyoshi & Kitagawa 2004). This ensures the wheelchair's motion always obeys the users instructions even when obstacles are present, rather than altering the trajectory of the vehicle. As an alternative to this method, the wheelchair requires the user to manually enable the obstacle avoidance system. This ensures any altered wheelchair trajectories are at least expected by the user, eliminating the need for this complex arrangement.

Chapter 3

Hardware

3.1 General Arrangement

At the beginning of 2006, the omni-directional wheelchair project at the University of Western Australia was a work in progress. The project was started by Iwasaki (2005), who used the Mecanum wheels and low-level driving routines developed by Voo (2000) to construct the basic frame and control software for this wheelchair. In addition to this work, a suspension system and alternative Mecanum wheel design had been developed for the miniature omni-directional driving robots used in the CIIPS department (McCandless 2001).

This project is an extension of these works, with the major aim of improving the ease of use and accuracy of the wheelchair by altering its hardware. Table 3.1 shows the state of the wheelchair components at the beginning of 2006. In summary, this project made improvements to the Mecanum wheels, batteries, footrests, motor controller cards, human interface, control software and chassis, whilst improvements were also made to the chair, armrests and high-level driver assistance system by Leong (2006).

TABLE 3.1: Omni-directional wheelchair components summary - February 2006

Qty	Component	Exists?	Suitable?	Requirements	Refer to...
4	Mecanum wheels	✓	✓	Machine down rims	Section 3.2
4	24V brushed DC motors	✓	✓	None	Section 3.3
1	EyeBot controller	✓	✓	Repair & install serial port 2	Section 3.4
2	12V batteries	✓	✗	Purchase replacements	Section 3.5
6	PSD sensors	✓	✓	None	Section 3.6
2	Footrests	✗		Design & install	Section 3.7
4	Motor controller cards	✓	✗	Purchase replacements & install	Chapter 4
1	Joystick	✗		Purchase & modify	Chapter 5
1	Control software	✓	✗	Rewrite for new cards & joystick	Chapter 6
1	Metal chassis	✓	✗	Redesign & add suspension	Chapter 7
1	Chair	✗		Design & install	(Leong 2006)
2	Armrests	✗		Design & install	(Leong 2006)

3.2 Mecanum Wheels

3.2.1 Technical Details

The Mecanum wheel works by using passive rollers around it's circumference at an angle offset from the axis of the wheel rotation. In the case where four of these wheels are used in combination, the rollers are at an angle of 45° (see Figure 3.1).

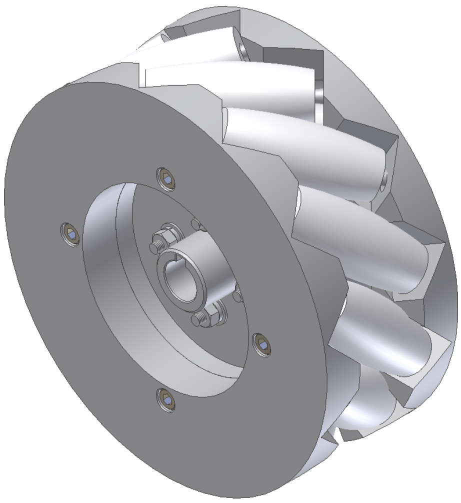


FIGURE 3.1: The Mecanum wheel design

As the wheel is made to rotate, the force exerted on the ground only consists of that component of the force along the axis of the rollers. The other component of the force does not affect the motion of the vehicle as it simply works to rotate the passive rollers. Hence, the resulting force on the vehicle from this particular wheel is in a direction 45° to the wheel axis (see Figure 3.2). By controlling the rotation of each individual wheel (and therefore every individual force), the vehicle can be made to move in any desired direction (see Figure 3.3).

3.2.2 Kinematics

Due to the increased complexity of the wheels used in ODVs, the algorithms required to control the vehicle's motion are very different to the algorithms used in a standard differential drive or Ackermann arrangement. The inverse kinematic equations are used when determining the required rotational speed of the motors to fulfill the

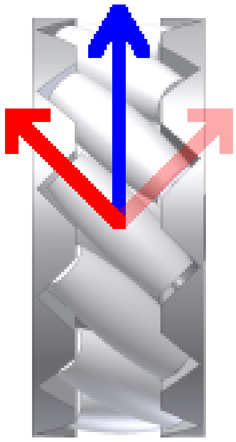


Figure 3.2: Force components in the Mecanum wheel (seen from below)

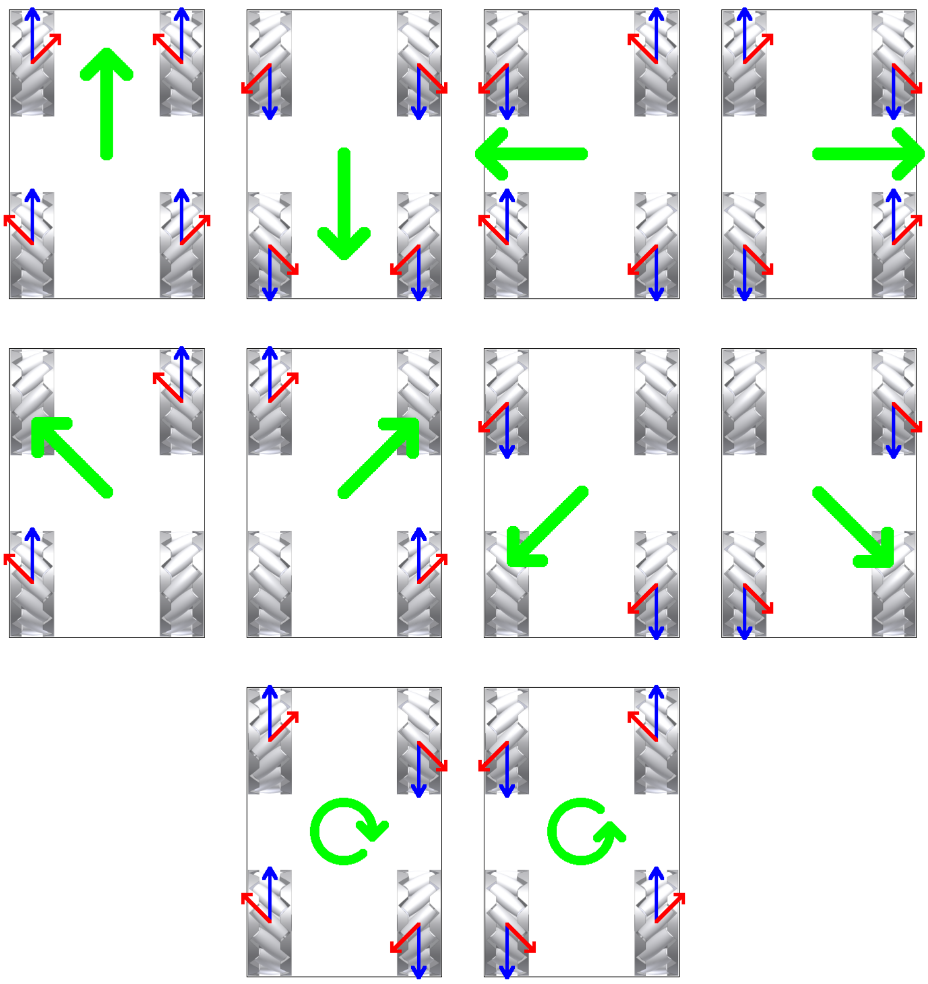


Figure 3.3: Wheel rotations for different driving directions (seen from below)

desired motion of the ODV. Similarly, the forward kinematic equations can be used to determine the current trajectory of the ODV from the current rotational speeds of the motors. The forward and inverse kinematics of the Mecanum wheel were derived by Viboonchaicheep, Shimada & Kosaka (2003) and are shown below in Equations 3.1 and 3.2 respectively.

$$\begin{Bmatrix} V_X \\ V_Y \\ \dot{\varphi} \end{Bmatrix} = 2\pi r \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ \frac{-1}{2(d+s)} & \frac{1}{2(d+s)} & \frac{-1}{2(d+s)} & \frac{1}{2(d+s)} \end{bmatrix} \begin{Bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{Bmatrix} \tag{3.1}$$

$$\begin{Bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{Bmatrix} = \frac{1}{2\pi r} \begin{bmatrix} 1 & -1 & \frac{-(d+s)}{2} \\ 1 & 1 & \frac{(d+s)}{2} \\ 1 & 1 & \frac{-(d+s)}{2} \\ 1 & -1 & \frac{(d+s)}{2} \end{bmatrix} \begin{Bmatrix} V_X \\ V_Y \\ \dot{\varphi} \end{Bmatrix} \tag{3.2}$$

These formulas convert a desired wheelchair motion (measured in meters per second and radians per second) into a required set of angular velocities for the wheels (measured in radians per second) and vice versa. However, the high level of wheel slippage experienced in the wheelchair would prevent the desired motion from being accurately achieved. This effectively makes any feedback that could be provided by shaft encoders useless, and as a result none are used on the wheelchair.

As an alternative, this project uses a method where the motion of the wheelchair is set relative to it's maximum speed. For example, if the user requests the wheelchair drive forward at 50% speed, the wheels will rotate forwards at half of their maximum speed. Similarly, if the user requests the wheelchair spin at 50% speed, the motors run at half their maximum speed in the correct directions.

Using this method, the scaling factors for the dimensions of the wheelchair and it's wheels (d, s & r) are not required. The ratio between directional driving and rotation is now lost, but the percentages between maximum speed and rotation are kept in tact. The resulting forward and inverse kinematic equations are shown in

Equations 3.3 and 3.4 respectively. This implementation also requires the wheel speeds are occasionally scaled down to ensure the maximum wheel speed is kept at 100% in all situations (see Appendix A.4).

$$\begin{Bmatrix} V_X \\ V_Y \\ \dot{\varphi} \end{Bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{Bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{Bmatrix} \tag{3.3}$$

$$\begin{Bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{Bmatrix} = \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix} \begin{Bmatrix} V_X \\ V_Y \\ \dot{\varphi} \end{Bmatrix} \tag{3.4}$$

3.2.3 Wheel Rims

One of the recommendations made by Iwasaki (2005) in the final section of his thesis was to "machine down the Mecanum wheel rims to increase clearance from the ground and avoid its contact with carpet or other soft surfaces". More simply, only the white plastic rollers of the Mecanum wheel design shown in Figure 3.1 are intended to make contact with the ground; the metal wheel rims are merely for structural support. If the rims were to come into contact with the ground, the wheel would begin to behave like it's more conventional counterpart, destroying the wheelchair's ability to move omni-directionally.

During initial tests, the metal rims were found to be making contact with the ground in some situations (such as when driving on an uneven, or carpeted surface). To fix this problem, the diameter of the wheel rims was reduced until the holes for the roller pins were almost exposed (about 5mm off the radius). This was acceptable since the force on the rims from the pins would always be towards the centre of the wheel. As an additional safety feature, a chamfer was also added to the outside edge of the rims to prevent serious injury if the wheels were to run over somebody. These alterations are shown in Figure 3.4.

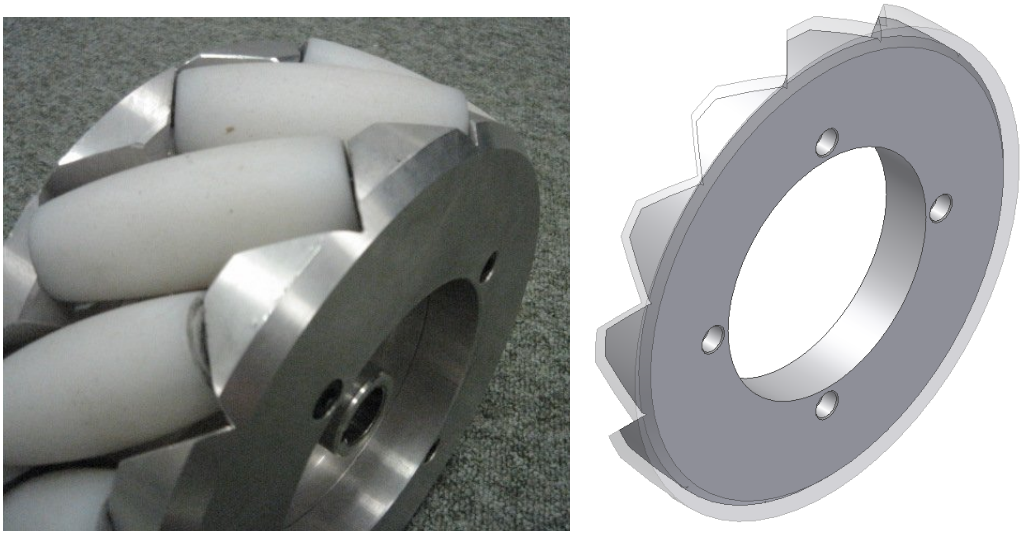


FIGURE 3.4: The Mecanum wheel rims after machining

3.3 Motors

The omni-directional wheelchair uses 4 motors to drive it's 4 independent Mecanum wheels. The 24V brushed DC motors used are made by Fortress and are commonly found on scooters used by the disabled (see Figure 3.5). They are rated to 15A, but generally only ever use up to 3A in normal operation.

FIGURE 3.5: A 24V brushed DC motor

The motors have built in brakes as well as a switch which provides two modes for the brakes:

Switch up: Always off, and

Switch down: On, unless a potential of 20V is applied across the brake inputs.

Currently, these brakes are not being used since they act very suddenly, causing a large jerk to the user if the wheelchair is still moving. It would be possible to implement a braking mechanism which activates the brakes 1 second after the motors have come to a halt.

3.4 EyeBot

The wheelchair makes use of the EyeBot controller developed at the University of Western Australia (Bräunl 2005a). It reads the inputs from the user and the sensors, and calculates a suitable wheelchair trajectory. The EyeBot uses a 25MHz and 32 bit Motorola 68332 chip and has 1Mb of RAM and 512KB of ROM. Communication with other devices is available via serial, parallel, digital input and analogue input and output ports. Figure 3.6 shows the front and back of the EyeBot MK4 used on the wheelchair.

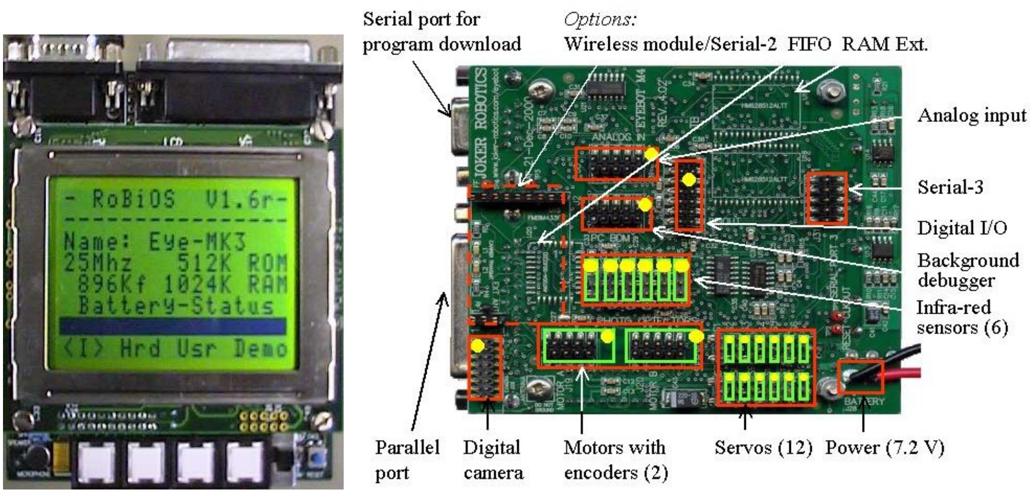


FIGURE 3.6: EyeBot controller MK4 (front and back views)

3.5 Batteries

Initially, the wheelchair used two 12V lead-acid car batteries connected in series to provide the required 24V. However, car batteries are designed for a 5 second burst of power to crank the engine, followed by 30 minutes of charging from the car's alternator. This makes them unsuitable for the omni-directional wheelchair which would require constant low-current power for many hours.

The batteries were replaced with two 12V deep-cycle lead-acid batteries with a rating of 40Ah. Since each motor uses approximately 3A at a maximum, this should provide for at least 3.5 hours of continuous usage. In reality, the motors are not continuously drawing 3A, and the batteries last for approximately 7 hours of semi-continuous usage.

The replacement batteries have two additional features to ensure they are suitable for this application:

- The batteries are sealed to ensure the acid inside does not spill on the user if they are tipped upside-down in a crash.
- The acid inside the batteries is a gel (rather than the typical liquid) to prevent the wheelchair's vibration from causing bubbles to form on the lead plates inside, affecting their performance and life.

The original Exide LM380C batteries, along with the purchased FirstPower LFP1240G batteries, are shown in Figure 3.7

FIGURE 3.7: Original car batteries and new sealed, deep-cycle gel batteries

3.6 Position Sensitive Devices

The wheelchair makes use of Position Sensitive Devices (PSDs) for it's driver-assistance software (Leong 2006). The six Sharp GP2D02 sensors use an infrared transmitter and receiver to detect the linear distance to the closest obstacle in the direction of the beam. They are shown in Figure 3.8.

FIGURE 3.8: Position sensitive devices (PSDs)

3.7 Footrests

Two standard wheelchair footrests were kindly donated by TADWA for use on the wheelchair. They were mounted in front of the two front motors once the new chassis and suspension system had been completed (see Figure 3.9). To prevent the footrests getting in the way of the user, they can be rotated up whilst the user gets into or out of the wheelchair.

FIGURE 3.9: Wheelchair footrests donated by TADWA

Chapter 4

Motor Driver Cards

4.1 Introduction

The speed of a DC motor is proportional to the voltage applied across its inputs, whilst the torque is proportional to the current it is drawing. Therefore, to control the speed of a DC motor, an analogue voltage ranging from $-V_{max}$ to V_{max} can be used. This requires the use of a digital to analogue converter, which is very expensive.

A common alternative is to use a constant supply voltage which is continuously being switched on and off by a controller. When this is done at a very high frequency, the input voltage is effectively reduced by the percentage of time that the signal is off (see Figure 4.1). This technique is known as pulse-width modulation (PWM) and allows the speed of a DC motor to be controlled by varying the pulse-width ratio of the input signal (also known as the duty cycle).

Initially, the EyeBot controller's digital outputs were used to generate a 5V and 8.191 kHz PWM signal for each motor. These signals were then fed into four motor controller cards designed by the electronic workshop, where they were amplified to 24V before being passed to the motors. Unfortunately the circuits in these cards were not designed to cope with the high currents generated when the motors were required to suddenly change direction, resulting in irreparable damage to the MOSFETs. The controller cards either needed to be redesigned to accommodate this sort of activity,

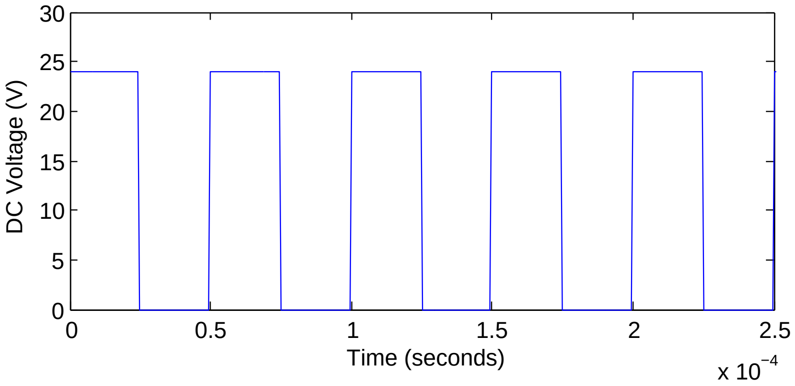


FIGURE 4.1: Pulse-width modulation (PWM) signal

or replaced with commercial DC motor controller cards. To save valuable workshop time and ensure the new cards came with a warranty, it was decided that some commercial cards would be purchased.

4.2 Selection

The following considerations were deemed important when selecting the new DC motor controller cards:

- To fit within the project budget, the cards should be relatively cheap whilst being of high enough quality to successfully perform the tasks required.
- The cards should be capable of handling a 24V signal with a current of up to 15A per motor (in the case of current spikes).
- The cards should include protections against over-heating, over-current and incorrect polarity.
- If possible, the cards should be available through a local supplier in Australia.
- The cards should either accept 5V PWM signals as inputs, or allow serial communication using the RS232 protocol.

The most suitable cards were found to be the Roboteq AX1500 controllers, which are unfortunately only manufactured in and distributed from the USA (see Figure 4.2). These cards can work with DC motor voltages between 12V and

40V, and are rated for continuous currents of up to 20A per motor or spikes of up to 150A per motor (see Appendix C.1). Since the cards control 2 motors each (either independently or using sum and difference commands), only 2 needed to be purchased for the 4 motors.

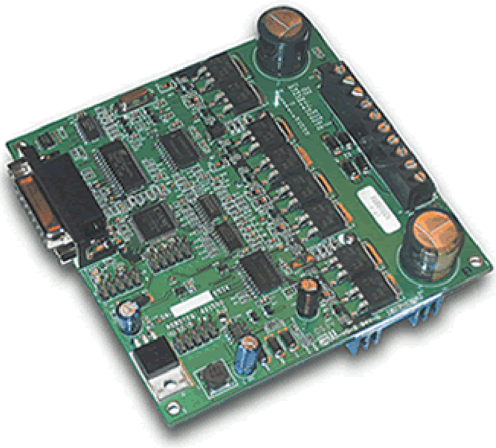


FIGURE 4.2: The new Roboteq AX1500 motor controller cards

4.3 Installation

The two Roboteq cards are used to drive the four wheelchair motors independently. One card controls both the front left and front right motors, whilst the other controls the back left and back right motors. Each card requires 4 power inputs (+24V supply and ground for each motor) and produces 4 power outputs (the PWM signal and ground for each motor). These connections are all made on one side of the card, as shown in Figure 4.3.

Additionally, the two cards are both connected to the second serial port on the EyeBot using the daisy-chain cable shown in Figure 4.4. This allows the EyeBot controller to communicate with the cards using only one of it's serial port, leaving the other free for communication with a PC. To allow the daisy-chaining of these cards, their EEPROM first needed to be flashed with the latest firmware available from Roboteq. The daisy-chain settings were then activated by sending the character strings "`^00 01`" and "`^00 11`" to the first and second cards respectively (Roboteq 2005).

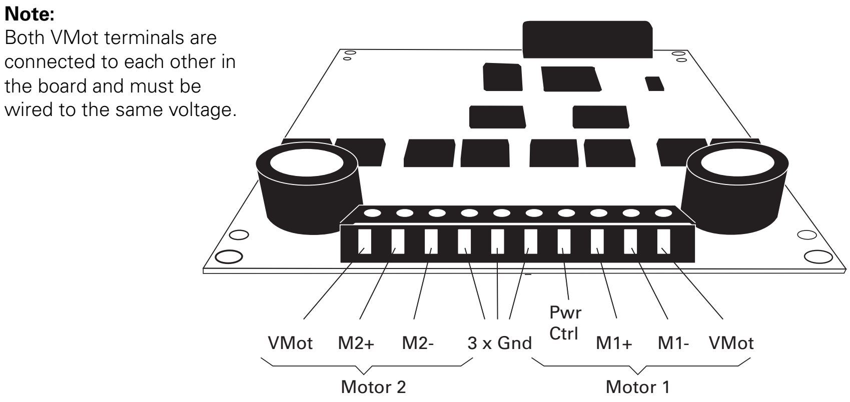


Figure 4.3: Roboteq input/output power connections

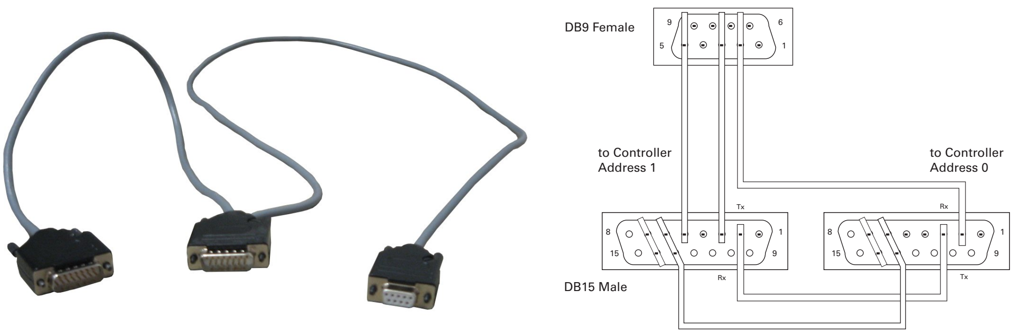


Figure 4.4: Daisy-chain serial cable for Roboteq AX1500 cards

4.4 Programming

Communication with the cards is done via the serial cable shown in Figure 4.4 using the RS232 protocol. As documented in the Roboteq AX1500 manual, the cards only accept the RS232 settings shown in Table 4.1. These settings are therefore used when initialising the second serial port on the EyeBot controller, prior to any communication.

Table 4.1: RS232 settings for Roboteq AX1500 cards

RS232 Parameter	Roboteq AX1500 Setting
Baud Rate	9600 bit/s
Word Size	10 bits
Start Bits	1
Data Bits	7
Parity	Even
Stop Bits	1
Flow Control	None

Once an appropriate serial connection has been established, the cards are controlled by sending and receiving the character strings shown in Table 4.3. This makes it extremely simple to interface with the cards, with the most common command being to set the motor speeds (eg. !A3F to instruct the first motor on card 1 to rotate forward at 50% speed).

Since the two controller cards are being daisy-chained to the one serial port, the commands to each card must be differentiated. This is done by moving the first character in the instruction 1 place along the ASCII table (by adding 1 to the it's hexadecimal notation). This creates the command set shown in Table 4.2.

TABLE 4.2: Character strings for communication with daisy-chained Roboteq cards

	Card 1			Card 2		
Event	**ASCII**	**HEX**	**Example**	**ASCII**	**HEX**	**Example**
Command	!	21	!a55	"	22	"a55
Query	?	3F	?a	@	40	@a
Config	^	5E	^00	_	5F	_00
Encoder	*	2A	*A8	+	2B	+A8
Reset	%	25	%rrrrrr	'	26	'rrrrrr

Table 4.3: Communication language for Roboteq AX1500 cards

Event	Send to Roboteq	Receive from Roboteq	Description
Power up prompt		`Roboteq v1.7b 02/01/05` `?`	Firmware version and date ? is random & can be ignored
Enter RS232 Mode	`\r\r\r\r\r\r\r\r\r\r`	`OK`	10 carriage returns Only required if not default mode
Bad Command	`???`	`-`	Unknown or incorrect command
Set motor speed	`!Mnn`	`+`	M selects the motor and direction: A ⇒ Motor 1, forward direction a ⇒ Motor 1, reverse direction B ⇒ Motor 2, forward direction b ⇒ Motor 2, reverse direction nn = Hexadecimal digits from 00 to 7F
Query motor speed	`?v` or `?V`	`nn` `mm`	nn = Motor 1 speed from 00 to 7F mm = Motor 2 speed from 00 to 7F No direction information given.
Query motor current	`?a` or `?A`	`nn` `mm`	nn = Motor 1 current in amps mm = Motor 2 current in amps
Query heatsink temperatures	`?m` or `?M`	`nn` `mm`	nn = Thermistor 1 from 00 to FF `mm` = Thermistor 2 from 00 to FF
Query battery voltages	`?e` or `?E`	`nn` `mm`	nn = Main battery from 00 to FF mm = Internal 12V from 00 to FF
Read controller setting	`^mm`	`DD`	mm = Parameter number DD = Current parameter value
Modify controller setting	`^mm nn`	`+`	mm = Parameter number nn = New parameter value
Reset controller	`%rrrrrr`		Will reset and display prompt

Chapter 5

Joystick

5.1 Introduction

Many people with disabilities rely on wheelchairs for use throughout their daily life. As a result, a wheelchair's control mechanism should be simple and accurate. Unfortunately, controlling a vehicle with 3 degrees of freedom is not immediately intuitive. Of the many human interface designs discussed in Section 2.3, a position-sensing joystick is the most familiar and simple to use, and was therefore selected for this wheelchair.

5.2 Selection

Joysticks come in all shapes and sizes, with many different features available. When selecting the appropriate joystick for the wheelchair, two important considerations were taken into account:

- Whether the joystick should be analogue or digital, and
- Whether the joystick should have 2 or 3 axes.

A digital joystick uses multiple on/off micro-switches to determine the direction in which the stick is being pushed. Using such a joystick would require an accompanying dial or pair of buttons to control the speed of the wheelchair. Conversely, an analogue joystick (often) uses potentiometers to sense both the magnitude and direction in which the stick is being pushed. In this way, the speed of the wheelchair is directly proportional to the degree by which the stick has been moved from it's origin. Of the two methods, the former allows simpler joystick control for disabled users with poor fine-motor skills, whilst the latter provides a more intuitive interface.

Most joysticks have 2 axes, allowing the stick to move from side to side (the x axis) and forwards/backwards (the y axis). A 3 axis joystick additionally allows the stick to be twisted about it's axis (the z axis), as shown in Figure 5.1. Using a 3 axis joystick allows direct control of each of the wheelchair's degrees of freedom, whilst a 2 axis joystick requires a toggle button to alternate between two driving modes. In the default mode, the x axis controls the rotation of the wheelchair, whilst in the alternative "strafe mode" the x axis controls the sideways motion of the wheelchair.

FIGURE 5.1: A 2 axis joystick compared to a 3 axis joystick

3 joysticks (described in Table 5.1 and shown in Figure 5.2) were purchased for the wheelchair and tested with a PC.

Table 5.1: The 3 joysticks purchased and tested

Joystick Model	**# axes**	**Control**
QuickShot QS-130F	2	Digital
Logitech Wingman Light	2	Analogue
Microsoft Sidewinder	3	Analogue

After using all three joysticks, it was clear that as the functionality of the joystick increased, the ease of use decreased. Therefore, a decision had to be made about the target market for this wheelchair. Since it would be difficult to accurately maneuver the wheelchair using either digital or 2 axis control, it was decided that the Microsoft Sidewinder joystick would be used. This would unfortunately make it difficult for users with certain disabilities (such as those which limit hand motion) to control the wheelchair. It would, however, be possible to customise the design for users with these disabilities.

Figure 5.2: The 3 joysticks purchased and tested

5.3 Installation

The Microsoft Sidewinder joystick purchased came with a USB cable for connection with a PC (see Figure 5.3). Unfortunately, the EyeBot controller to which it is connected does not have a USB host. This problem was overcome by removing the internal circuit board and replacing it with a custom circuit board that directly accesses the joystick's potentiometers and buttons. This board was made exactly the same size and uses exactly the same potentiometer connectors as the original board, allowing it to fit easily into place (see Appendix B.1).

Using a multimeter to measure the resistance of the joystick's potentiometers, it was found that the x, y, and z axes have potentiometers with a resistance of 10kΩ, whilst the throttle potentiometer has a resistance of 20kΩ. The circuit board connects these potentiometers to the analogue inputs of the EyeBot, which reads their current value (see Table 5.2). However, since the EyeBot supplies a voltage of 5V and the analogue inputs can read voltages between 0V and 4.1V, a resistor is required between the supply voltage and each potentiometer. Using simple voltage division, it is possible to find the necessary resistances for the x, y, z potentiometers (a) and the throttle potentiometer (b).

$$\frac{0.9}{5} = \frac{a}{10\text{k}+a} \qquad \frac{0.9}{5} = \frac{b}{20\text{k}+b}$$

$$a = 2.2\text{k}\Omega \qquad b = 4.4\text{k}\Omega$$

The joystick has 4 buttons on it's base and 7 buttons at the top of the stick. Only the buttons on the base are used by the wheelchair, since only 4 buttons are required and the user could accidentally press the buttons on the stick. The circuit board directly connects these buttons to the digital inputs of the EyeBot, which reads their current settings (see Table 5.2). When a button is not pushed, the open switch causes no current to flow through that branch resulting in the 5V source voltage being read by the EyeBot. When a button is pushed, the closed switch causes current to flow through that branch and a voltage drop across the resistor, resulting in 0V being read by the EyeBot.

FIGURE 5.3: The USB connection originally on the joystick

FIGURE 5.4: The new joystick circuit

Table 5.2: EyeBot I/O pins used

Signal	Wire Colour	EyeBot Pin
X Axis	Yellow	Analogue Input 5
Y Axis	Green	Analogue Input 6
Z Axis	Brown	Analogue Input 7
Throttle	Grey	Analogue Input 8
Button 5	Blue	Digital I/O 9
Button 6	Thin Black	Digital I/O 10
Button 7	Purple	Digital I/O 11
Button 8	Orange	Digital I/O 12
+5V	Red	Digital I/O 13
Ground	Thick Black	Digital I/O 16

5.4 Programming

Once the 4 buttons and 4 potentiometers are correctly connected to the digital and analogue input pins on the EyeBot, the RoBIOS operating system makes it simple to read their current values. The x, y and z potentiometers are used to control the motion of the wheelchair left/right, forwards/backwards and clockwise/anticlockwise respectively. The throttle potentiometer acts as an overall maximum speed control; at it's minimum position the maximum speed of the wheelchair is 20%, at it's maximum position the maximum speed of the wheelchair is 100%, and moving between these positions causes a gradual increase in the wheelchair's maximum speed. The 4 buttons are used to control the advance driving modes of the wheelchair (Leong 2006).

All of the digital inputs are read in one command: `OSReadInLatch(0)`. This command returns a 16 bit number, with each bit representing the current state of the corresponding digital I/O pin. "AND masking" the number with a mask with only the corresponding bit set to 1, and then dividing by the same mask causes the number to be 0 if the button is pushed and 1 otherwise. This bit is then inverted (a logical NOT) to give the desired result (see Table 5.3).

TABLE 5.3: RoBIOS commands for reading buttons

Button	Mask	Command
5	`BUTTON5 = 0x10`	`!((OSReadInLatch(0) & BUTTON5)/BUTTON5)`
6	`BUTTON6 = 0x20`	`!((OSReadInLatch(0) & BUTTON6)/BUTTON6)`
7	`BUTTON7 = 0x30`	`!((OSReadInLatch(0) & BUTTON7)/BUTTON7)`
8	`BUTTON8 = 0x40`	`!((OSReadInLatch(0) & BUTTON8)/BUTTON8)`

The analogue inputs are each read by one command: `OSGetAD(CHANNEL)` (where CHANNEL corresponds to the analogue input to read from). This command returns an integer between 0 and 1000 (representing 0V and 4.1V respectively). Since the joystick's minimum and maximum values for each axis will not be exactly 0 and 1000, the joystick must be calibrated. After calibration, moving the joystick to the minimum/maximum position on each axis should ideally result in a 0/1000 reading respectively. This number is then scaled to a number representing the percentage of the potentiometer's movement (see Table 5.4).

Calibration can be performed manually each time the EyeBot is started by introducing a routine that requires the user move the joystick to the bottom left position with the joystick twisted fully left and the throttle at a minimum and then pushing a button. The user then moves the joystick to the top right, twisted fully right with the throttle at a maximum and pushes a button. The EyeBot would read the minimum and maximum values for each axis and use them to convert the raw readings to the desired values. As it turns out, the raw minimum and maximum values stay relatively constant between uses and can therefore be hard coded into the software, removing the need for this arduous calibration routine (see Table 5.4).

TABLE 5.4: RoBIOS commands for reading potentiometers

Pot.	Channel	Raw Min	Raw Max	Desired Min	Desired Max
X	4	95	905	-100	100
Y	5	920	75	-100	100
Z	6	145	845	-100	100
Throttle	7	1000	58	0	100

After calibration, the joystick input values are finally adjusted using threshold values on each axis. This causes the input from an axis to be taken as zero unless it is above a specified (and customisable) value. This is required for three reasons:

- The joystick is extremely sensitive around it's origin, and any small accidental movements would cause the wheelchair to drive.
- People with disabilities may have difficulty holding the joystick perfectly still at the origin without shaking it.
- By thresholding each axis individually, driving directly forward is also made easier by preventing a small component of the wheelchair's velocity from being in the sideways direction.

After testing, it was found that a threshold value of 15% was optimal for most wheelchair users on each axis.

Chapter 6

Low Level Driving Routines

6.1 Introduction

For any driving robot, the low-level driving routines are the simple components of the onboard software that determine the desired driving positions, velocities and/or accelerations. This involves interfacing with all input and feedback devices on the vehicle and calculating the required motor speeds before sending them to the motors.

In our current wheelchair design, there are 3 pieces of hardware with which the low-level driving routines must communicate:

- the 3-axis joystick to read the human input,
- the infra-red remote control receiver as an alternative means of human input, and
- the motor controller cards to ensure the correct motor speeds are achieved.

Higher level driving routines are also being developed which make use of the wheelchair's onboard position-sensitive devices (PSDs) to perform automatic functions such as door-driving, wall following and obstacle avoidance (Leong 2006).

6.2 Design

The software has been written entirely in the C programming language and compiled using the gcc68 compiler for the Motorola HC68332 chip (Bräunl 2005a). Although C itself is not an object-oriented language, it can be made modular by using separate text files for the different software components. Header files are used to define the necessary variables and functions, as well as include documentation and comments on their use. These files have the .h prefix (eg. ODW.h). The bulk of the code is included in the C source files, with filenames using the .c prefix (eg. ODW.c).

The wheelchair's code has 4 main components or modules, each modelled around the hardware with which it is interfacing. A brief description of each component is provided below.

ODW: This component of the software includes the main function which initially starts the desired modules. Once these modules have been successfully started, the main function continually loops and updates the LCD display, whilst reading any EyeBot key inputs. When key 4 is pressed, the program stops the running modules and exits gracefully.

ODW_IR: This module interfaces with the infra-red remote control receiver, allowing the wheelchair to be controlled by a Nokia remote control. The module initialises the receiver and waits for a key input. When a key input is received, the appropriate action (defined in the corresponding header file) is taken. It is important to note that the functionality of this module is effectively destroyed if the Joystick module is also activated.

ODW_Joystick: This module interfaces with the modified 3-axis Microsoft Sidewinder joystick discussed in Chapter 5. When started, the module first calibrates the joystick and then continually reads the joystick's current position and button values. This data is converted into the appropriate motor speeds using the inverseKinematics function, which are then set using the setWCSpeed function (both of which are found in the ODW_MotorCtrl module). The button values can also be used to activate the advanced driving routines (Leong 2006).

ODW_MotorCtrl: This module serves two purposes; to supply functions which provide a simple means for setting and reading the current wheelchair speed, and to interface with the motor driver cards discussed in Chapter 4. The latter works by continually reading the desired motor speeds and sending them to the controller cards via the RS232 protocol.

This technical description is represented graphically in the following software flow charts. For a full listing of the developed software, see Appendix A.

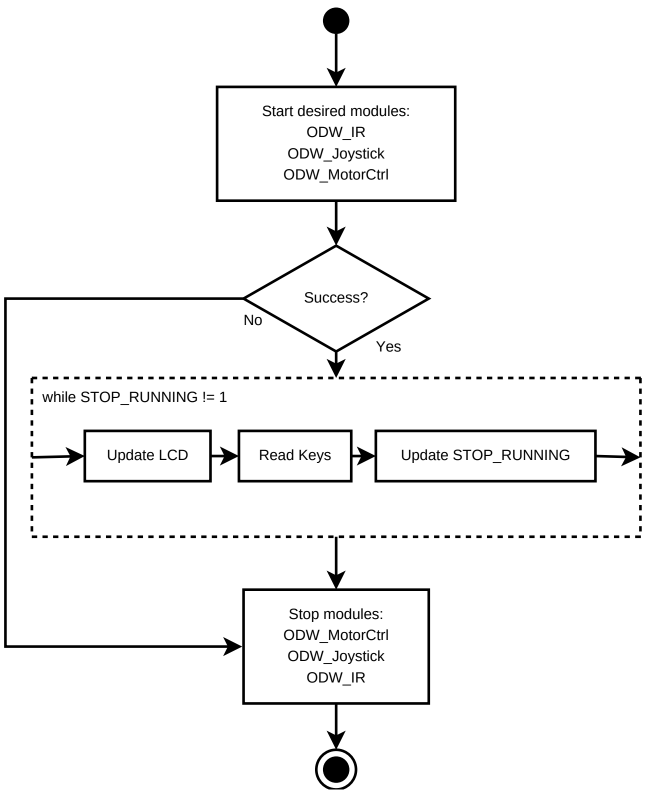


FIGURE 6.1: Flow chart for the main ODW code

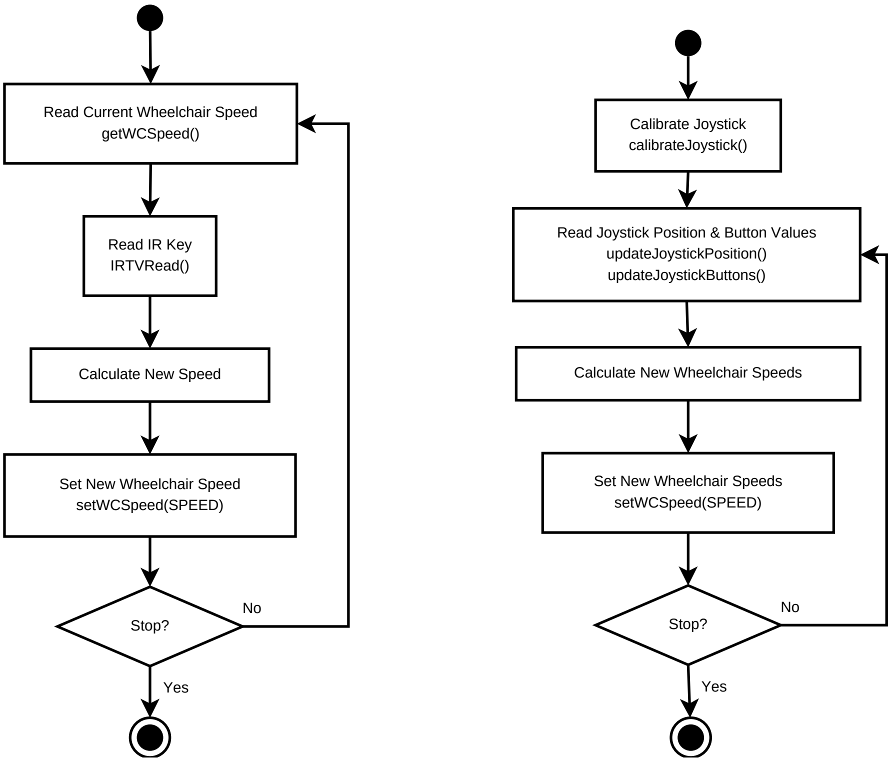


FIGURE 6.2: Flow charts for the IR and Joystick modules

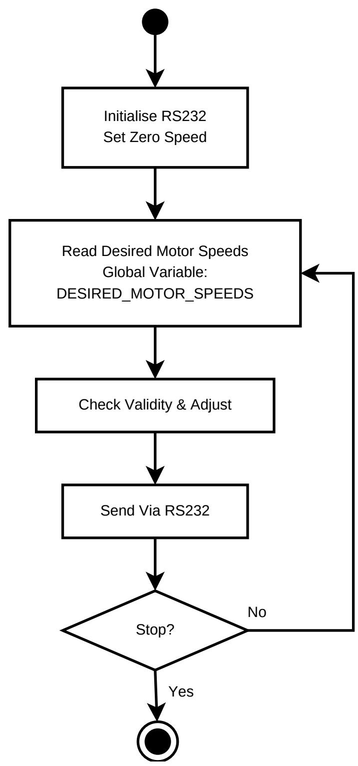


FIGURE 6.3: Flow chart for the MotorCtrl module

Chapter 7

Suspension System

7.1 Introduction

At the beginning of 2006, the omni-directional wheelchair had no suspension system; the wheels were directly coupled to the motors which were in turn directly mounted onto the chassis. This initial setup was only intended to be a temporary solution, until a superior design was developed.

As discussed in Section 3.2.2, the Mecanum wheels used on the omni-directional wheelchair cause a large degree of slip during it's normal operation. This makes the wheelchair difficult to drive accurately, and deprecates any feedback that could be obtained from shaft encoders. Part of this slip could be attributed to the rigid design, which would cause only 3 of the 4 wheels to be in contact with the ground when driving on uneven terrain. This would theoretically cause the wheelchair to drive at 45° to the design direction.

This problem was overcome by designing and building a suspension system for the wheelchair. Under the weight of the wheelchair and it's user, each wheel is now being constantly pushed down to the ground. In addition, the wheelchair is more capable of handling rough terrain such as grassed or paved areas without causing excessive user discomfort.

One final problem that was overcome by the inclusion of this system was the issue of vibrations felt by the user during medium to high speed driving. These vibrations could mainly be attributed to the minor machining inaccuracies of the Mecanum wheels which resulted in a sudden jerk each time the roller in contact the ground was alternated. Since the rollers were made of an inflexible plastic and there was no suspension built into the design, the user would be left feeling these effects. This was found to be particularly bad when driving on a rough terrain.

7.2 Design

Suspension systems often rely on a spring and a damper used in combination to absorb both the small vibrations and the large movements. The spring acts as the main absorber, whilst the damper acts to mitigate the oscillatory effects over time. Many bicycle designs now include a small spring/damper shock absorber directly below the seat in the main frame. These shock absorbers are available in many different sizes, with varying spring stiffness and damping coefficients to match.

DNM Suspension is a Thai company which designs, manufactures and sells these shock absorbers for both mountain bicycles and motorbikes. Four of their DV-22 shocks absorbers were selected for the wheelchair, and were kindly donated to the project (see Figure 7.1). They come with a 350lbs spring, use an oil damping system, weigh 180 grams, have an eye-to-eye uncompressed length of 125mm, bushing width of 24mm and hole diameter of 8mm.

FIGURE 7.1: DNM DV-22 bicycle shock absorbers

Since the wheels mount directly onto the shafts of the motors, these shock absorbers are placed between the main chassis and the motors. The chassis therefore serves three purposes:

- it holds/contains the batteries, electronics, sensors and wiring,
- it acts as a base for the actual chair the user sits on, and
- it acts as a strong frame to hold the motors.

For strength, the chassis has been made entirely of $30 \times 30 \times 3$mm angle iron. It has been made large enough to hold the two batteries and electronics between them, and also allows for enough width to mount two motors back to back with approximately 5mm between them. The angle iron faces in towards the centre of the battery box on the bottom to provide a flat surface for the batteries to rest on. Conversely, the angle iron faces outwards from the centre of the box at the top to allow space for the batteries to be inserted and removed. This lip also acts as a strong handle for lifting the wheelchair when required. This design can be seen in Figure 7.2.

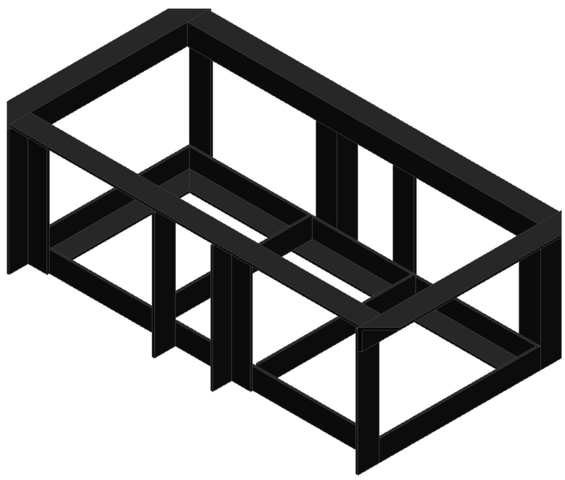


FIGURE 7.2: Battery box design

The chair, which was designed by Leong (2006), was mounted on top of this battery box by welding it's support arm onto a thick steel plate lid the same size as the top of the box. If this lid had simply been screwed to the top of the box, the entire chair would need to be lifted off each time maintenance was required on the batteries or the internal electronics. Instead, the lid was attached to the box with two hinges on the rear side, and locked down on the front side with a standard barrel bolt. This allows the internal components of the battery box to be exposed by simply unlocking the barrel bolt and tilting the chair back. This design can be seen in Figure 7.3.

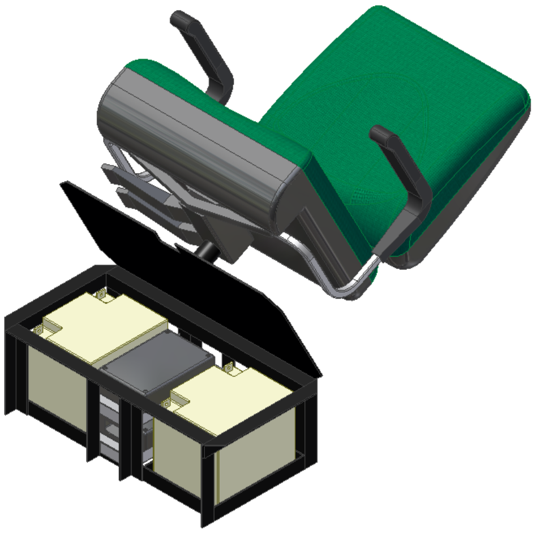


FIGURE 7.3: Battery box lid and chair mount

The final, and most important part of the design involved connecting the motors to the chassis and incorporating a suspension system. This system is the most likely section of the design to fail under the large forces and stress resulting from any sudden impacts. In addition, the Mecanum wheel design being used on the wheelchair requires that the wheels remain perpendicular to the floor at all times. If the wheels were on an angle, the rims would contact the ground making omni-directional motion impossible (see Figure 7.4).

Each wheel requires independent suspension to allow movement without affecting the wheelchair chassis or the other wheels. The most basic independent suspension design positions the shock absorber directly between the motors and the chassis (see Figure 7.5). This design does not provide strength against sideways or rotational motion of the motors, making it unsuitable for use in the wheelchair.

A slightly more complicated design uses a trailing arm which runs from the motors to the chassis, where it is connected in such a way that it is only allowed

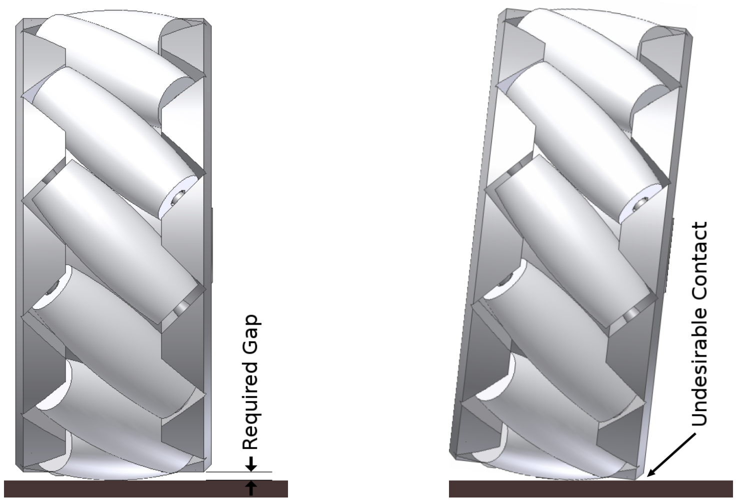


FIGURE 7.4: Perpendicular requirement for the adopted Mecanum wheel design

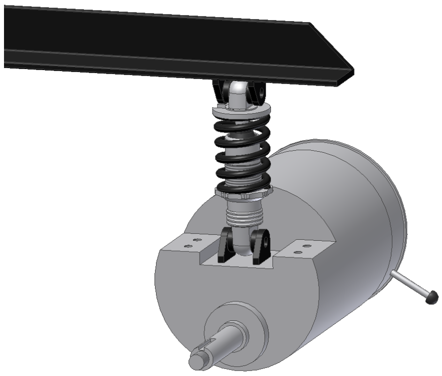


FIGURE 7.5: A basic but inadequate suspension design

to rotate in the vertical direction. The shock absorber then connects to both the chassis and the arm to limit this motion. The trailing arms can either extend to the sides of the chassis, or to the front and back of the chassis. Since the Mecanum wheels need to remain perpendicular to the ground the latter was chosen for this design (see Figure 7.6).

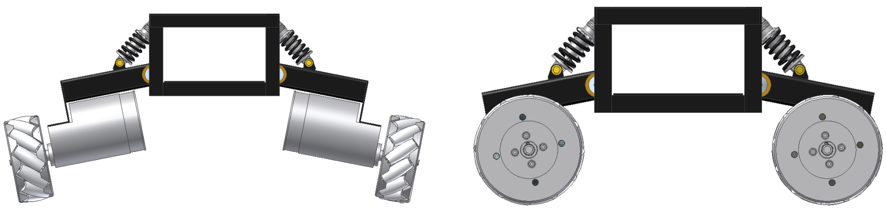


FIGURE 7.6: Two alternative trailing arm designs

Rather than using a simple plate for the trailing arm, a $50 \times 50 \times 3$mm square-hollow-section (SHS) tube was used. This allows the arm to remain rigid and resist both bending and twisting forces which would otherwise have resulted in failure. This arm is bolted to the motors at one end and welded to a 20mm diameter solid steel rod at the other end. This rod runs perpendicular to the trailing arm and is held by rotational bearings at each end. The bearings are housed in pressed-metal straps which are bolted onto the main chassis of the wheelchair. This design is shown in Figure 7.7.

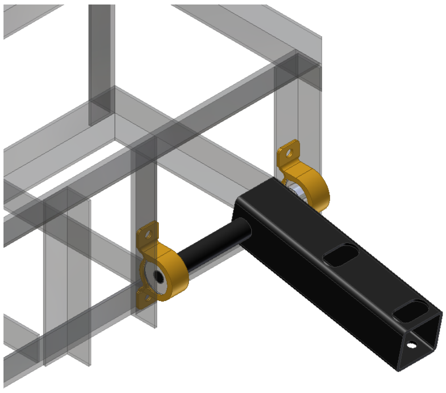


Figure 7.7: Trailing arm connection to the chassis

Figure 7.8 shows the forces that could be experienced by the wheels and motors (in red), and the corresponding forces and torques each independent suspension system will need to overcome (in green).

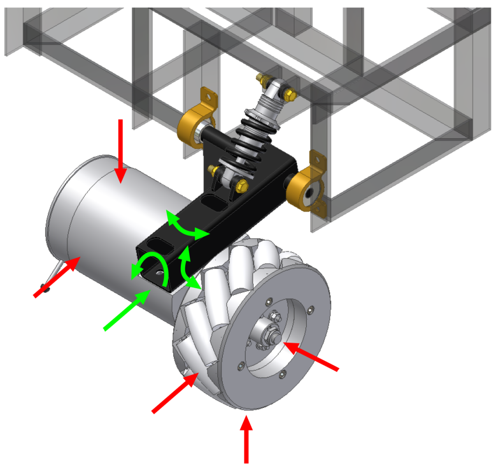


Figure 7.8: The forces experienced by the trailing arm suspension

The design in Figure 7.7 successfully resists all of the forces shown in Figure 7.8 except those which act to rotate the trailing arm sideways. The torque generated by this force is amplified by the long trailing arm, and the connection between the arm and the rod could easily fail. For additional strength, a triangular gusset was added between the trailing arm and the rod (shown in Figure 7.9). This also stops any motion of the trailing arm from side to side, preventing the motors from contacting.

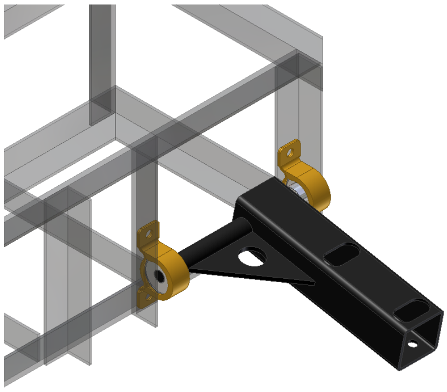


FIGURE 7.9: A steel gusset is used to add strength

Finally, the shock absorbers are connected between the trailing arms and the sides of the battery box. Four small mounting blocks were welded onto the arms and box to achieve this. The mounting angle of the shock absorbers between the arm and the chassis determines the extent to which the springs will be compressed or expanded for a given amount of movement of the wheels. This therefore dictates how easily the wheels will be able to move up and down. The compression of the springs was tested on a prototype, and the angle of the shock absorbers was chosen to cause the springs to compress approximately 50% under the weight of the chair and a standard user. This allows leeway for the wheels to move both up and down from the default position, as required. An extra set of holes were drilled for the bearings to allow an alternative setting for slightly heavier users (see Figure 7.10).

The final designs of the new wheelchair chassis and suspension system can be seen in Appendix B.2.

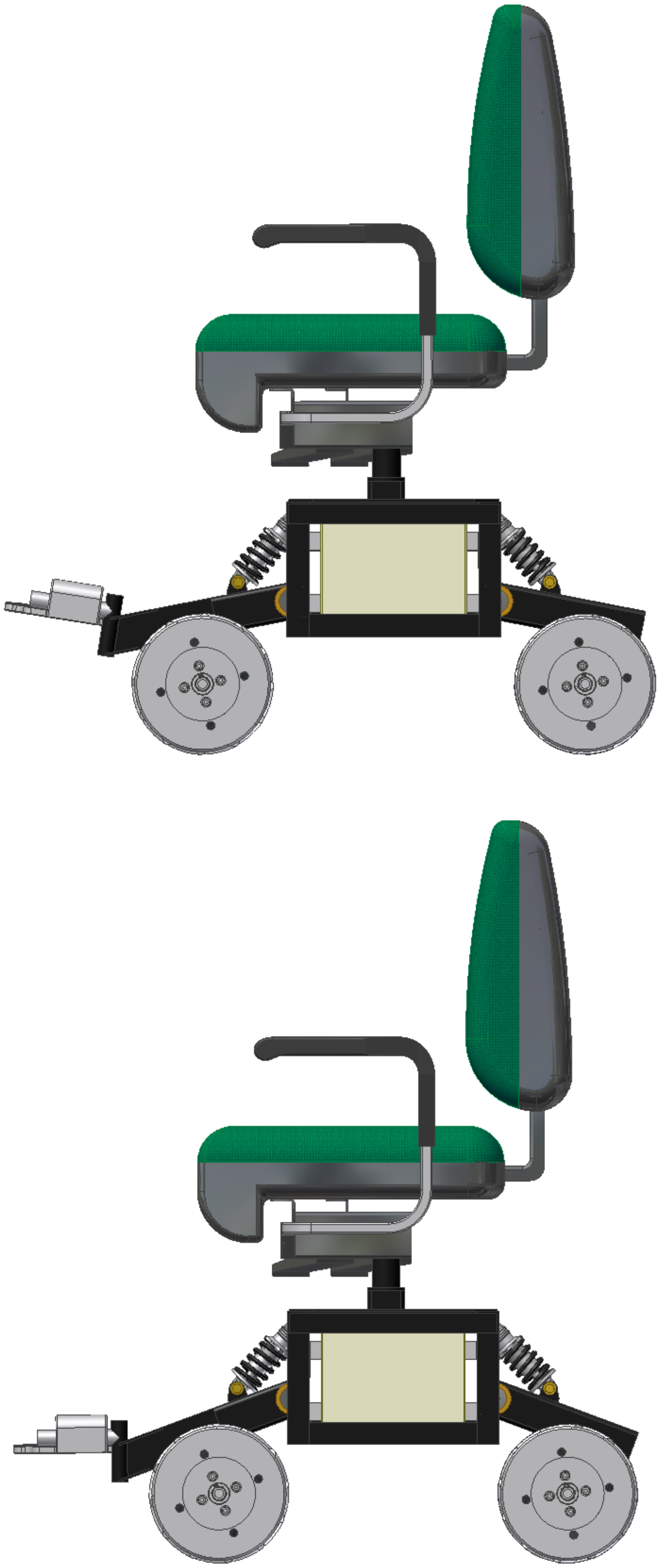


Figure 7.10: The shock absorber angle determines the effective spring constant

Chapter 8

Results, Testing and Simulation

8.1 Motor Control

The speeds of the four independent DC motors on the wheelchair are now controlled by the new Roboteq AX1500 controller cards. This is done by sending ASCII characters through the daisy-chain serial cable using the RS232 protocol (see Chapter 4). Tests were performed on the motors, by measuring the actual speeds achieved for certain requested values. Even though no feedback is used to control the motor speeds, they are in general very accurate. The test results can be seen in Table 8.1 and Figure 8.1. If the motors were to become inaccurate in the future, it would be possible to write simple software instructions to calibrate them.

8.2 Joystick

The 3-axis joystick installed in this project allows highly accurate control of each of the wheelchair's degrees of freedom. Although this style of joystick is new to most users, the design is intuitive and easy to learn. To prevent accidental wheelchair motion resulting from small movements of the stick near the zero position of an axis, independent threshold values were implemented on each axis. These values can be customised to also allow users with deteriorated fine-motor skills (such as those with Parkinson's disease) to use the joystick.

Table 8.1: Actual motor speeds achieved

Requested Speed			FL Fwd !Ann	FL Rev !ann	FR Fwd !bnn	FR Rev !Bnn	BL Fwd "Bnn	BL Rev "bnn	BR Fwd "ann	BR Rev "Ann
/127	*nn*	*%*	*RPM*	*RPM*	*RPM*	*RPM*	*RPM*	*RPM*	*RPM*	*RPM*
12	0C	9.45	-	-	-	-	-	-	-	-
25	19	19.69	-	-	-	-	-	-	43	42
38	26	29.92	59	58	57	57	56	56	65	63
50	32	39.37	77	76	75	75	75	75	86	83
63	3F	49.61	97	97	94	95	95	95	108	104
76	4C	59.84	117	117	114	114	114	114	130	125
88	58	69.29	135	135	131	131	133	133	150	144
101	65	79.53	155	155	151	150	153	153	172	165
114	72	89.76	173	175	170	169	173	173	193	186
127	7F	100	192	193	189	186	192	192	214	205

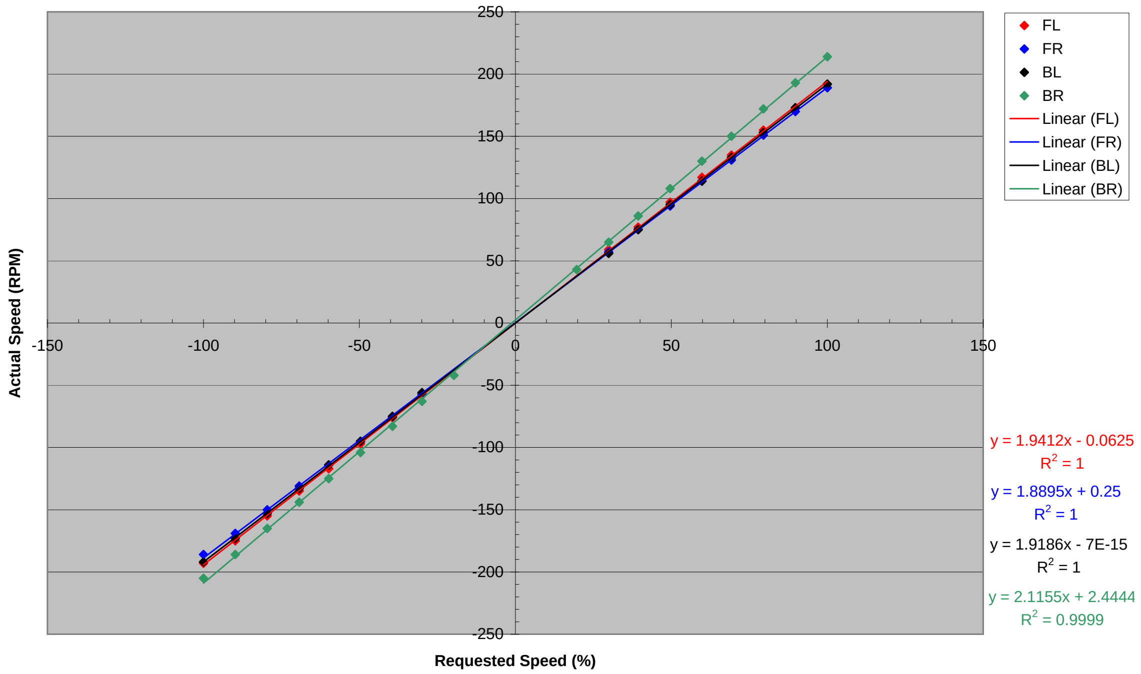


Figure 8.1: Actual motor speeds achieved

Two programs were developed for testing the joystick on the EyeBot. The first program, shown in Appendix A.13, prints both raw and calibrated joystick values to the LCD, as well as the current values of the buttons (see Figure 8.3). This program was used for the initial joystick testing and for finding the calibration constants required to keep the joystick readings within the desired range. The second program, shown in Appendix A.14, plots the current joystick values onto a map on the LCD (see Figure 8.2). This program is useful for determining the appropriate threshold values for each axis, and for visualising the actual joystick position.

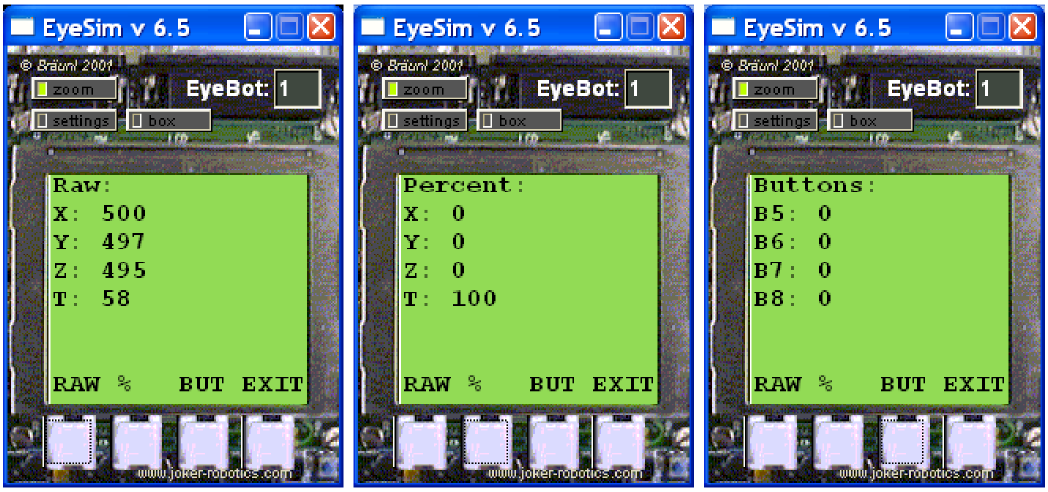


FIGURE 8.2: A textual joystick testing program

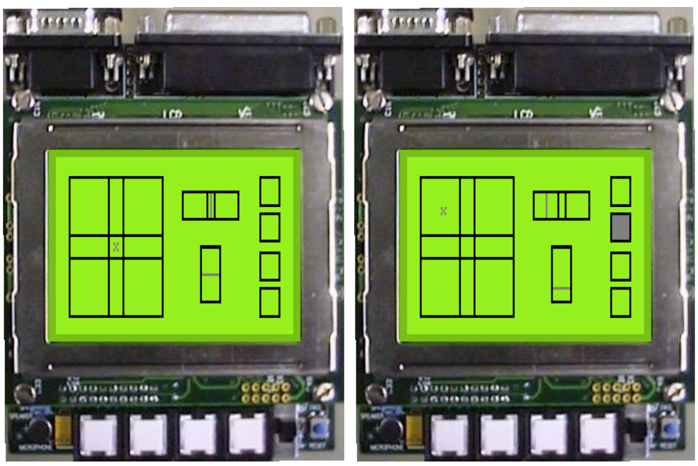


FIGURE 8.3: A graphical joystick testing program

8.3 Suspension and Chassis

After the installation of the trailing-arm suspension system, the wheelchair's motion was found to be much more accurate on uneven surfaces. Motions which used to cause the Mecanum wheels to slip (such as sideways and diagonal driving) can now be executed with a higher degree of accuracy. This is due to the wheels always being pushed down to the ground by the springs in the shock absorbers. In addition, the vibrations felt by the user when driving the wheelchair are now mostly absorbed by the shock absorbers. This greatly improves the level of comfort for the user.

Maintenance can be performed on the wheelchair by simply unlocking the barrel-bolt on top of the battery box, and tilting the entire chair backwards to expose the electronic circuits and batteries. This prevents any injuries that would otherwise result from the user lifting the entire chair off the battery box to access the internal components.

An unexpected side-effect of the suspension system can be seen when the wheelchair is driven directly to the side. These movements cause the springs on the side away from the direction of motion to compress, resulting in a slight tilt of the chair in this direction. This is due to the combination of the angular forces from the Mecanum wheels, and cannot easily be overcome. Although this may at first startle the user, there are no devastating effects and overtime the user learns to anticipate it.

8.4 Modelling and Simulation

The new chassis and suspension system were designed with the aid of AutoDesk Inventor version 10. Inventor is the 3D modelling version of AutoDesk's popular AutoCAD software, and is similar to other 3D CAD programs such as Solid Edge and Solid Works. The model was used to ensure that all the components would actually fit together as planned, and that the workshop would be able to get the required access to all the nuts and bolts that needed to be fitted and tightened.

The model developed in AutoDesk Inventor was then converted to a Milkshape 3D model, developed by chUmbaLum sOft. This allowed the model to be used in the EyeSim EyeBot simulation software developed at the University of Western Australia (Bräunl 2005b). A program called Deep Exploration, developed by Right Hemisphere, was used to convert the Inventor assembly files (with extensions .iam and .ipt) into a AutoDesk drawing interchange/exchange format file (with extension .dxf). This file was then imported into Milkshape 3D where it was scaled, coloured, and saved (with extension .ms3d) ready for use in EyeSim. A picture of the wheelchair model in EyeSim is shown in Figure 8.4.

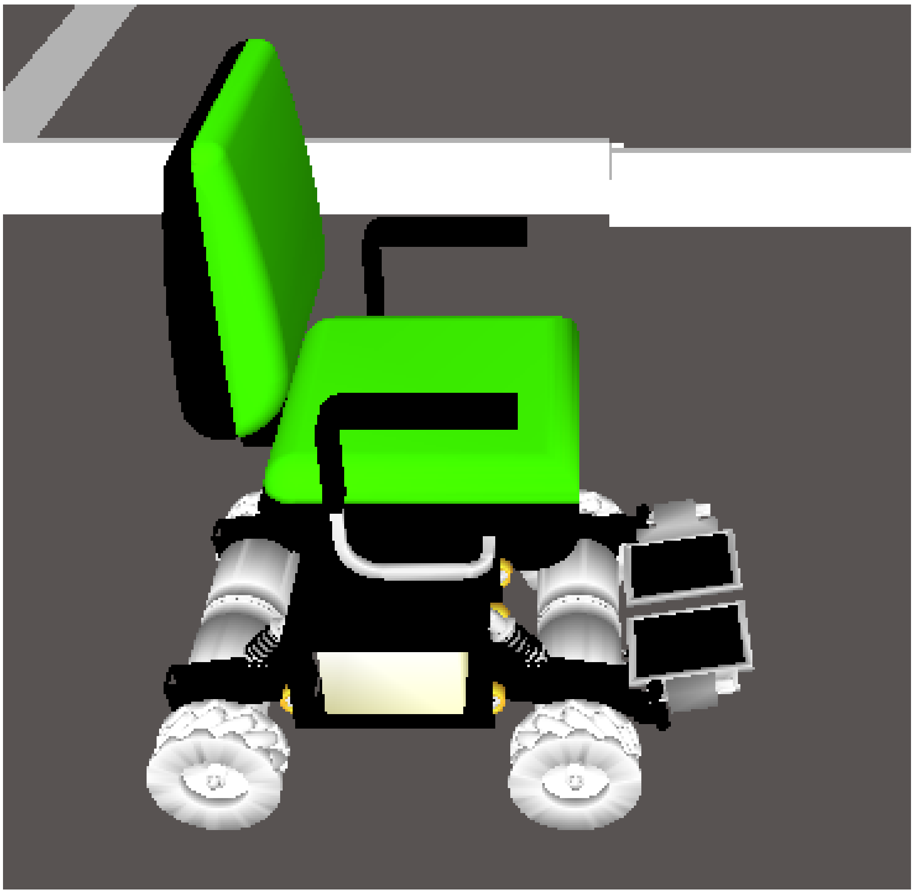


Figure 8.4: The wheelchair model in EyeSim

Chapter 9

Conclusion

9.1 Outcomes

The omni-directional wheelchair being developed at the University of Western Australia's Centre for Intelligent Information Processing Systems (CIIPS) allows the user to easily manoeuvre in what would otherwise be an extremely complicated environment. This project made improvements to the Mecanum wheels, batteries, motor driver cards, human interface, control software, chassis and suspension system.

These improvements transformed the partially working prototype into a fully usable wheelchair (see Figure 9.1). The result is much higher driving accuracy and a greatly improved overall experience for the user in both comfort and easy of use. On the whole, the project was extremely successful and will provide a very solid test bed for advanced driving and mapping projects in the future.

9.2 Recommendations

The following recommendations are made for future projects involving the wheelchair.

- Replace the current Mecanum wheels with an alternative material and design. The current wheels are inherently slippery, and a rubbery material would most likely prevent any slip from occurring. In addition to this, a soft rubber

FIGURE 9.1: The wheelchair after all alterations and improvements

would have a shock absorbing effect and would aid the suspension system in mitigating vehicle vibrations. To prevent issues with the rims contacting the ground, the new design should use a fork to hold each external roller, similar to that developed by McCandless (2001). The final result would be similar to that used by Lippitt & Jones (1998).

- With the new Mecanum wheels mentioned above, the issue of wheel slip should be overcome. This would allow feedback from the motors to actually provide useful information for the driving routines. Shaft encoders should be installed on each wheel, and the information used to provide both velocity and position feedback for the wheelchair. A simple PID control system should be developed for the overall wheelchair velocity and position, rather than for each individual wheel.

- Using the feedback from the motors and PID control system mentioned above, software similar to the $V\omega$ interface could be written for the wheelchair, allowing for movements in certain directions for a specified distance. This could be extended to provide fully autonomous driving through a known environment, using a map of the walls and obstacles. For example, the wheelchair could be made to drive from room 3.13 of the Electrical Engineering building at the University of Western Australia, to room 4.04 (making use of the elevator).

Bibliography

Airtrax. (2006). *Omni-Directional Technology: Changing the way vehicles move*, [Online]. Available from: <http://www.airtrax.com/> [September 2006].

Bräunl, T. 2003, *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*, Springer, New York.

Bräunl, T. (2005a). *EyeBot - Online Documentation*, [Online], The University of Western Australia. Available from: <http://robotics.ee.uwa.edu.au/eyebot/index.html> [April 2006].

Bräunl, T. (2005b). *EyeSim - Mobile Robot Simulator*, [Online], The University of Western Australia. Available from: <http://robotics.ee.uwa.edu.au/eyebot/doc/sim/sim.html> [September 2006].

Cooney, J.A., Xu, W.L., Bright, G. 2004, 'Visual Dead-Reckoning for Motion Control of a Mecanum-Wheeled Mobile Robot', *Mechatronics*, vol. 14, pp. 623–637.

Cooper, R.A., Jones, D.K., Fitzgerald, S., Boninger, M.L. & Albright, S.J. 2000, 'Analysis of position and isometric joysticks for powered wheelchair driving', *IEEE Transactions on Biomedical Engineering*, vol. 47, pp. 902–910. Available from: IEEE Xplore, [April 2006].

Coyle, E.D. 1995, 'Electronic wheelchair controller designed for operation by hand-operated joystick, ultrasonic noncontact head control and utterance from a small word-command vocabulary', *IEEE Colloquium on New Developments* *in Electric Vehicles for Disabled Persons*, pp. 3/1–3/4. Available from: IEEE Xplore, [April 2006].

Dickerson, S. & Lapin, B. 1991, 'Control of an omni-directional robotic vehicle with Mecanum wheels', *Proceedings of the National Telesystems Conference*, vol. 1, pp. 323–328. Available from: IEEE Xplore, [April 2006].

Diegel, O., Badve, A., Bright, G., Potgieter, J. & Tlale, S. 2002, 'Improved mecanum wheel design for omni-directional robots', *Proceedings of the Australasian Conference on Robotics and Automation*, pp. 117–121.

Ding, D., Cooper, R.A., Spaeth, D. 2004, 'Optimized joystick controller', *Proceedings of the 26th Annual International Conference of the Engineering in Medicine and Biology Society*, vol. 2, pp. 4881–4883. Available from: IEEE Xplore, [April 2006].

Guo, S., Cooper, R.A., Boninger, M.L., Kwarciak, A. & Ammer, B. 2002, 'Development of power wheelchair chin-operated force-sensing joystick', *Proceedings of the Second Joint Engineering in Medicine and Biology*, vol. 3, pp. 2373–2374. Available from: IEEE Xplore, [April 2006].

Ilon, B.E. 1975, *Wheels for a course stable selfpropelling vehicle movable in any desirable direction on the ground or some other base*, US Patent 3876255

Iwasaki, Y. 2005, *Omni-Directional Wheelchair*, Honours Thesis, The University of Western Australia.

Jones, D.K., Cooper, R.A., Albright, S. & DiGiovine, M. 1998, 'Powered wheelchair driving performance using force- and position-sensing joysticks', *Proceedings of the IEEE 24th Annual Northeast Bioengineering Conference*, pp. 130–132. Available from: IEEE Xplore, [April 2006].

Kamiuchi, S. & Maeyama, S. 2004, 'A novel human interface of an omni-directional wheelchair', *13th IEEE International Workshop on Robot and Human Interactive Communication*, pp. 101–106. Available from: IEEE Xplore, [April 2006].

Kitagawa, L., Kobayashi, T., Beppu, T. & Terashima, K. 2001, 'Semi-autonomous obstacle avoidance of omnidirectional wheelchair by joystick impedance control', *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 4, pp. 2148–2153. Available from: IEEE Xplore, [April 2006].

Leong, M. 2006, *Active User Omni-Directional Wheelchair*, Honours Thesis, The University of Western Australia.

Lippitt, T.C. & Jones, W.C. 1998, 'OmniBot Mobile Base', *KSC Research and Technology Report* [Online], NASA, USA. Available from: <http://rtreport.ksc.nasa.gov/techreports/98report/09-ar/ar06.html> [April 2006].

McCandless, A. 2001, *Design and construction of a robot vehicle chassis*, Honours Thesis, The University of Western Australia.

Nagatani, K., Tachibana, S., Sofne, M. & Tanaka, Y. 2000, 'Improvement of odometry for omnidirectional vehicle using optical flow information', *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 1, pp. 468–473. Available from: IEEE Xplore, [April 2006].

Roboteq. 2005, *AX1500 User's Manual* [Online]. 1.7b. Available from: <http://www.roboteq.com/> [June 2006].

Shimada, A., Yajima, S., Viboonchaicheep, P. & Samura, K. 2005, 'Mecanum-wheel vehicle systems based on position corrective control', *32nd Annual Conference of IEEE Industrial Electronics Society*, pp. 2077–2082. Available from: IEEE Xplore, [April 2006].

Tahboub, K.A. & Asada, H.H. 2000, 'Dynamics analysis and control of a holonomic vehicle with a continuously variable transmission', *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 3, pp. 2466–2472. Available from: IEEE Xplore, [April 2006].

Terashima, K., Miyoshi, T., Urbana, J. & Kitagawa, H. 2004, 'Frequency shape control of omni-directional wheelchair to increase user's comfort', *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 3, pp. 3119–3124. Available from: IEEE Xplore, [April 2006].

Urbano, J., Terashima, K., Miyoshi, T. & Kitagawa, H. 2004, 'Impedance control for safety and comfortable navigation of an omni-directional mobile wheelchair', *Proceedings of the International Conference on Intelligent Robots and Systems*, vol. 2, pp. 1902–1907. Available from: IEEE Xplore, [April 2006].

Urbano, J., Terashima, K., Miyoshi, T. & Kitagawa, H. 2005, 'Velocity control of an omni-directional wheelchair considering user's comfort by suppressing vibration', *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3169–3174. Available from: IEEE Xplore, [April 2006].

Viboonchaicheep, P., Shimada, A. & Kosaka, Y. 2003, 'Position rectification control for Mecanum wheeled omni-directional vehicles', *The 29th Annual Conference of the IEEE Industrial Electronics Society*, vol. 1, pp. 854–859. Available from: IEEE Xplore, [April 2006].

Voo, C.Y. 2000, *Low level driving routines for the omni-directional robot*, Honours Dissertation, The University of Western Australia.

Wada, M. & Asada, H. 1998, 'A holonomic omnidirectional vehicle with a reconfigurable footprint mechanism and it's application to wheelchairs', *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 1, pp. 774–780. Available from: IEEE Xplore, [April 2006].

West, M. & Asada, H. 1992, 'Design of a holonomic omnidirectional vehicle', *Proceeding of the IEEE International Conference on Robotics and Automation*, vol. 1, pp. 97–103. Available from: IEEE Xplore, [April 2006].

Appendix A

Code

A.1 ODW.h

```
/**
 * ODW.h
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   This is the header file for the general ODW code.
 *
 * Note:
 *   The types, variables and functions defined here should ↩
     be available
 *   regardless of the implementation.
 */


// GLOBAL VARIABLES

/**
 * If set to 1, the program and wheelchair control will stop.
 */
int STOP_RUNNING;


// FUNCTIONS

```

```
/**
 * Input:   NONE
 * Output:  0  = ok
 *        -1 = error
 * Semantics:   Initialise and start all the functions of ↩
    the ODW
 */
int ODWInit( void );

/**
 * Input:   NONE
 * Output:  0  = ok
 *        -1 = error
 * Semantics:   Stop and release all functions of the ODW
 */
int ODWStop( void );
```

A.2 ODW.c

```
/**
 * ODW.c
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   This file contains the main function and other general initialisation and
 *   stop functions for the ODW code.
 */


#include "eyebot.h"
#include <math.h>
#include <stdlib.h>
#include "ODW.h"
#include "ODW_MotorCtrl.h"
#include "ODW_IR.h"
#include "ODW_Joystick.h"


int ODWInit()
{
    int result;

    STOP_RUNNING = 0;

    // Motor Control
    result = startMotorCtrl();
    if( result != 0 ) return result;

    OSWait(10);

    // Joystick Control
    result = startJoystick();
    if( result != 0 ) return result;

    OSWait(10);

    // Infra-red Remote Control
    result = startIR();
    if( result != 0 ) return result;

```

```
    return result;
}


int ODWStop()
{
    int result;

    // Infra-red Remote Control
    result = stopIR();
    if( result != 0 ) return result;

    OSWait(10);

    // Joystick Control
    result = stopJoystick();
    if( result != 0 ) return result;

    OSWait(10);

    // Motor Control
    result = stopMotorCtrl();
    if( result != 0 ) return result;

    return result;
}


int main(int argc, char *argv[])
{
    int result;

    // Start all control modules
    LCDPrintf("Starting ODW...\n");
    result = ODWInit();
    if( result != 0 )
    {
    LCDPrintf("Start ODW Failed!\n");
    LCDPrintf("Stopping & Exiting\n");
    ODWStop();
    return result;
    }

    LCDPrintf("Success!\n");
    OSWait(100);

    // Update LCD and loop until KEY4 pressed
    while( STOP_RUNNING != 1 )
    {
```

```
    WCSpeed wcspeed = getWCSpeed();

    LCDClear();
    LCDMenu("", "", "", "Exit");
    LCDPrintf("x: %f\n", wcspeed.x);
    LCDPrintf("y: %f\n", wcspeed.y);
    LCDPrintf("w: %f\n\n", wcspeed.w);
    LCDPrintf("IRstep: %d\n", IRstep);

    if( KEYRead() == KEY4 ) STOP_RUNNING = 1;
    }

    // Stop all control modules
    LCDPrintf("Stopping ODW...\n");
    result = ODWStop();
    if( result != 0 )
    {
    LCDPrintf("Stop ODW Failed!\n");
    LCDPrintf("Exiting anyway...\n");
    return result;
    }

    LCDPrintf("Success!\n");

    // Exit program
    return 0;
}
```

A.3 ODW_MotorCtrl.h

```
/**
 * ODW_MotorCtrl.h
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   This is header file for the code that controls the motors and the speed of the ODW.
 *
 * Note:
 *   The types, variables and functions defined here should be available
 *   regardless of the implementation.
 */


// DEFINITIIONS

/**
 * Set to 1 if using daisy-chaining of Roboteq cards.
 * Set to 0 otherwise.
 */
#define DAISY 1

/**
 * Required settings for serial communication with Roboteq cards
 * Baud:          9600 kb/s
 * Handshaking:   None
 * Start bits:    1
 * Data bits:     7
 * Parity:        Even
 * Stop bits:     1
 */
#define ROBOTEQ_BAUD SER9600
#define ROBOTEQ_HANDSHAKE NONE

/**
 * 100/Hz
 * Frequency at which to transmit speed signals to motors
 */
#define MOTOR_CTRL_FREQ 100/2

```

```
/**
 * ODW physical specifications
 */
#define ODW_WIDTH 5.20
#define ODW_LENGTH 5.40
#define WHEEL_RADIUS 0.95

/**
 * Speed value for stopping the ODW
 */
#define WC_STOPPED (WCSpeed) {0,0,0}

/**
 * Address of values to change in MC68332 chip to set values↩
    for serial communication
 */
#define sccr1 0xfc0a


// TYPE DEFINITIONS

/**
 * Holds speed values for each individual motor in the ODW ↩
   in rads/sec
 */
typedef struct
{
    int FL;
    int FR;
    int BL;
    int BR;
} WCMotorSpeeds;

/**
 * Holds speed values for the overall motion of the ODW
 */
typedef struct
{
    double x;
    double y;
    double w;
} WCSpeed;

/**
 * Holds position values for the overall motion of the ODW
 */
typedef struct
{
    double x;
```

```
    double y;
    double phi;
} WCPosition;


// GLOBAL VARIABLES

/**
 * The currently desired individual motor speeds.
 * Max value per motor = 127 = 0x7F
 */
WCMotorSpeeds DESIRED_MOTOR_SPEEDS;

/**
 * The currently desired overall ODW speed.
 * Max value per axis = 100
 */
WCSpeed DESIRED_WC_SPEED;

/**
 * The handle for the timer which continues to
 * send the currently desired motor speeds to the
 * motor controller cards.
 */
TimerHandle MotorCtrlTimer;


// FUNCTIONS

/**
 * Input:   (wcmotorspeeds) The motor speeds to convert
 * Output:  The corresponding overall speed of the ODW
 * Semantics:   Convert individual motor speeds into the ↩
    overall WC speed
 */
WCSpeed forwardKinematics( WCMotorSpeeds wcmotorspeeds );

/**
 * Input:   (wcspeed) The overall speed of the ODW to convert
 * Output:  The corresponding speeds of the individual motors
 * Semantics:   Convert the overall WC speed into required ↩
    individual motor speeds
 */
WCMotorSpeeds inverseKinematics( WCSpeed wcspeed );

/**
 * Input:   NONE
 * Output:  0  = ok
 *      -1 = error
```

```
 * Semantics:   Set up and start the control of the motors
 */
int startMotorCtrl( void );

/**
 * Input:   NONE
 * Output:  0  = ok
 *         -1 = error
 * Semantics:   Stop the control of the motors and release the serial ports
 */
int stopMotorCtrl( void );

/**
 * Input:   NONE
 * Output:  NONE
 * Semantics:   Control the motors using the global parameters
 */
void MotorCtrl( void );

/**
 * Input:   (wcspeed) The speed values to set for the ODW
 * Output:  0  = ok
 *         -1 = error
 * Semantics:   Set the speed of the overall ODW (rather than each motor independently)
 */
int setWCSpeed( WCSpeed wcspeed );

/**
 * Input:   NONE
 * Output:  The currently desired speed for the ODW
 * Semantics:   Get the current desired speed of the overall ODW (rather than each individual motor)
 */
WCSpeed getWCSpeed( void );

/**
 * Input:   (wcmotorspeeds) The speed of each motor to set
 * Output:  0  = ok
 *         -1 = error
 * Semantics:   Set the speed of each motor independently (rather than the overall ODW)
 */
int setWCMotorSpeeds( WCMotorSpeeds wcmotorspeeds );

/**
 * Input:   NONE
```

```
 * Output:  The currently desired motor speeds
 * Semantics:   Get the current desired speed of each ↩
    individual motor (rather than the overall ODW)
 */
WCMotorSpeeds getWCMotorSpeeds( void );
```

A.4 ODW_MotorCtrl.c

```
/**
 * ODW_MotorCtrl.c
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   This is code to control the motors and the speed of the ODW.
 *   This implementation uses the Roboteq AX1500 cards installed.
 */


#include "eyebot.h"
#include "ODW_MotorCtrl.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>


WCSpeed forwardKinematics( WCMotorSpeeds wcmotorspeeds )
{
    WCSpeed wcspeed;

    // Find ODW speeds by converting motor speeds
    wcspeed.x = ( wcmotorspeeds.FL + wcmotorspeeds.FR + wcmotorspeeds.BL + wcmotorspeeds.BR ) * 100.0 / (127.0*4);
    wcspeed.y = ( -wcmotorspeeds.FL + wcmotorspeeds.FR + wcmotorspeeds.BL - wcmotorspeeds.BR ) * 100.0 / (127.0*4);
    wcspeed.w = ( -wcmotorspeeds.FL + wcmotorspeeds.FR - wcmotorspeeds.BL + wcmotorspeeds.BR ) * 100.0 / (127.0*4);

    return wcspeed;
}


WCMotorSpeeds inverseKinematics( WCSpeed wcspeed )
{
    WCMotorSpeeds wcmotorspeeds;

```

```
    // Find motor speeds by converting ODW speed components
    wcmotorspeeds.FL = round( ( wcspeed.x - wcspeed.y - wcspeed.w ) * 127.0/100.0 );
    wcmotorspeeds.FR = round( ( wcspeed.x + wcspeed.y + wcspeed.w ) * 127.0/100.0 );
    wcmotorspeeds.BL = round( ( wcspeed.x + wcspeed.y - wcspeed.w ) * 127.0/100.0 );
    wcmotorspeeds.BR = round( ( wcspeed.x - wcspeed.y + wcspeed.w ) * 127.0/100.0 );

    // If any of the motor speeds are outside [-127, 127] then scale all speeds down
    // the fastest motor is spinning at 100% speed (either -127 or 127).
    double scaling_factor = 1;
    if( (wcmotorspeeds.FL >  127) && ( 127.0/wcmotorspeeds.FL < scaling_factor) ) scaling_factor =  127.0 / wcmotorspeeds.FL;
    if( (wcmotorspeeds.FR >  127) && ( 127.0/wcmotorspeeds.FR < scaling_factor) ) scaling_factor =  127.0 / wcmotorspeeds.FR;
    if( (wcmotorspeeds.BL >  127) && ( 127.0/wcmotorspeeds.BL < scaling_factor) ) scaling_factor =  127.0 / wcmotorspeeds.BL;
    if( (wcmotorspeeds.BR >  127) && ( 127.0/wcmotorspeeds.BR < scaling_factor) ) scaling_factor =  127.0 / wcmotorspeeds.BR;
    if( (wcmotorspeeds.FL < -127) && (-127.0/wcmotorspeeds.FL < scaling_factor) ) scaling_factor = -127.0 / wcmotorspeeds.FL;
    if( (wcmotorspeeds.FR < -127) && (-127.0/wcmotorspeeds.FR < scaling_factor) ) scaling_factor = -127.0 / wcmotorspeeds.FR;
    if( (wcmotorspeeds.BL < -127) && (-127.0/wcmotorspeeds.BL < scaling_factor) ) scaling_factor = -127.0 / wcmotorspeeds.BL;
    if( (wcmotorspeeds.BR < -127) && (-127.0/wcmotorspeeds.BR < scaling_factor) ) scaling_factor = -127.0 / wcmotorspeeds.BR;

    wcmotorspeeds.FL = round( scaling_factor * wcmotorspeeds.FL );
    wcmotorspeeds.FR = round( scaling_factor * wcmotorspeeds.FR );
    wcmotorspeeds.BL = round( scaling_factor * wcmotorspeeds.BL );
    wcmotorspeeds.BR = round( scaling_factor * wcmotorspeeds.BR );

```

```
    return wcmotorspeeds;
}


int startMotorCtrl()
{
    // Initialise the second serial port for transmission to Roboteq cards
    LCDPrintf("Init SERIAL2...\n");
    int result = OSInitRS232( ROBOTEQ_BAUD, ROBOTEQ_HANDSHAKE, SERIAL2 );
    // The next line sets even parity for serial interface 2
    (*(((volatile BYTE*)Ser1Base)+3)) = 0x1a;

    // If not using daisy chaining, also initialise the first serial port
    if( !DAISY ) {
        LCDPrintf("Init SERIAL1...\n");
        result = result && OSInitRS232( ROBOTEQ_BAUD, ROBOTEQ_HANDSHAKE, SERIAL1 );
        // The next line sets even parity for serial interface 1
        (*((volatile unsigned short*)sccr1)) = 0x042c;
    }

    setWCSpeed( WC_STOPPED );

    MotorCtrlTimer = OSAttachTimer( MOTOR_CTRL_FREQ, MotorCtrl );

    return (result);
}


int stopMotorCtrl()
{
    // Stop continuous transmission of speed signals
    int result = !OSDetachTimer( MotorCtrlTimer );

    // Send one last transmission of zero values
    setWCSpeed( WC_STOPPED );
    MotorCtrl();

    return result;
}


void MotorCtrl()
{
```

```c
    // Grab a snapshot of the currently desired motor speeds
    int FL = DESIRED_MOTOR_SPEEDS.FL;
    int FR = DESIRED_MOTOR_SPEEDS.FR*-1; // FR motor spins backwards with +ve signal
    int BL = DESIRED_MOTOR_SPEEDS.BL;
    int BR = DESIRED_MOTOR_SPEEDS.BR*-1; // BR motor spins backwards with +ve signal

    // Limit speeds to allowed values between -127 and 127
    if( FL > 127 ) FL = 127;
    if( FR > 127 ) FR = 127;
    if( BL > 127 ) BL = 127;
    if( BR > 127 ) BR = 127;
    if( FL < -127 ) FL = -127;
    if( FR < -127 ) FR = -127;
    if( BL < -127 ) BL = -127;
    if( BR < -127 ) BR = -127;

    // Convert integer values into hexidecimal characters
    char sFL[3], sFR[3], sBL[3], sBR[3];
    snprintf( sFL, 3, "%X", abs(FL) );
    snprintf( sFR, 3, "%X", abs(FR) );
    snprintf( sBL, 3, "%X", abs(BL) );
    snprintf( sBR, 3, "%X", abs(BR) );

    // Ensure all character arrays contain exactly 2 characters
    // (Pad the first character with a 0 if only 1 character is used)
    if( FL < 16 && FL > -16 )
    {
    sFL[1] = sFL[0];
    sFL[0] = '0';
    }
    if( FR < 16 && FR > -16 )
    {
    sFR[1] = sFR[0];
    sFR[0] = '0';
    }
    if( BL < 16 && BL > -16 )
    {
    sBL[1] = sBL[0];
    sBL[0] = '0';
    }
    if( BR < 16 && BR > -16 )
    {
    sBR[1] = sBR[0];
    sBR[0] = '0';
    }
```

```

    // Transmit commands for the front left motor
    OSSendCharRS232( '!', SERIAL2 );
    if ( FL < 0 ) OSSendCharRS232( 'a', SERIAL2 );
    else OSSendCharRS232( 'A', SERIAL2 );
    OSSendCharRS232( sFL[0], SERIAL2 );
    OSSendCharRS232( sFL[1], SERIAL2 );
    OSSendCharRS232( '\r', SERIAL2 );

    // Transmit commands for the front right motor
    OSSendCharRS232( '!', SERIAL2 );
    if ( FR < 0 ) OSSendCharRS232( 'b', SERIAL2 );
    else OSSendCharRS232( 'B', SERIAL2 );
    OSSendCharRS232( sFR[0], SERIAL2 );
    OSSendCharRS232( sFR[1], SERIAL2 );
    OSSendCharRS232( '\r', SERIAL2 );

    // Transmit commands for the back right motor
    OSSendCharRS232( '!'+DAISY, SERIAL1+DAISY );
    if ( BR < 0 ) OSSendCharRS232( 'a', SERIAL1+DAISY );
    else OSSendCharRS232( 'A', SERIAL1+DAISY );
    OSSendCharRS232( sBR[0], SERIAL1+DAISY );
    OSSendCharRS232( sBR[1], SERIAL1+DAISY );
    OSSendCharRS232( '\r', SERIAL1+DAISY );

    // Transmit commands for the back left motor
    OSSendCharRS232( '!'+DAISY, SERIAL1+DAISY );
    if ( BL < 0 ) OSSendCharRS232( 'b', SERIAL1+DAISY );
    else OSSendCharRS232( 'B', SERIAL1+DAISY );
    OSSendCharRS232( sBL[0], SERIAL1+DAISY );
    OSSendCharRS232( sBL[1], SERIAL1+DAISY );
    OSSendCharRS232( '\r', SERIAL1+DAISY );
}


int setWCSpeed( WCSpeed wcspeed )
{
    // Convert desired ODW speed into required wheel speeds
    WCMotorSpeeds wcmotorspeeds = inverseKinematics( wcspeed
        );

    return setWCMotorSpeeds( wcmotorspeeds );
}


WCSpeed getWCSpeed()
{
    WCMotorSpeeds wcmotorspeeds = getWCMotorSpeeds();

```

```
    // Convert wheel speeds into resulting ODW speed
    return forwardKinematics( wcmotorspeeds );
}


int setWCMotorSpeeds( WCMotorSpeeds wcmotorspeeds )
{
    // Limit motor speeds to allowed values (between -127 ↩
       and 127)
    if( wcmotorspeeds.FL > 127 ) wcmotorspeeds.FL = 127;
    if( wcmotorspeeds.FR > 127 ) wcmotorspeeds.FR = 127;
    if( wcmotorspeeds.BL > 127 ) wcmotorspeeds.BL = 127;
    if( wcmotorspeeds.BR > 127 ) wcmotorspeeds.BR = 127;
    if( wcmotorspeeds.FL < -127 ) wcmotorspeeds.FL = -127;
    if( wcmotorspeeds.FR < -127 ) wcmotorspeeds.FR = -127;
    if( wcmotorspeeds.BL < -127 ) wcmotorspeeds.BL = -127;
    if( wcmotorspeeds.BR < -127 ) wcmotorspeeds.BR = -127;

    DESIRED_MOTOR_SPEEDS = wcmotorspeeds;

    return 0;
}


WCMotorSpeeds getWCMotorSpeeds()
{
    return DESIRED_MOTOR_SPEEDS;
}
```

A.5 ODW_Joystick.h

```
/**
 * ODW_Joystick.h
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   This is the header file for the code that initialises
    and reads from the ODW's joystick.
 *
 * Note:
 *   The types, variables and functions defined here should
    be available
 *   regardless of the implementation.
 */


// DEFINITIIONS

/**
 * 100/Hz
 * The frequency with which to read from the joystick
 */
#define JOY_FREQ 100/10

/**
 * The  bitmasks for extracting the bit values for
    associated buttons
 */
#define BUTTON5 0x10
#define BUTTON6 0x20
#define BUTTON7 0x40
#define BUTTON8 0x80

/**
 * The analogue channels to which each axis is connected
 */
#define X_CHANNEL 4
#define Y_CHANNEL 5
#define Z_CHANNEL 6
#define T_CHANNEL 7

/**
 * The threshold values for each axis
```

```c
 */
#define X_THRESHOLD 15
#define Y_THRESHOLD 15
#define Z_THRESHOLD 15

/**
 * With the throttle set to the minimum level, the maximum possible ODW speed
 * will be limited to 100/THROTTLE_DIVISOR %
 */
#define THROTTLE_DIVISOR 5.0


// TYPE DEFINITIIONS

/**
 * Holds position values for each axis in the joystick.
 */
typedef struct
{
    int x;
    int y;
    int z;
    int t;
} JoyPos;

/**
 * Holds bit values for each button on the joystick.
 */
typedef struct
{
    BYTE b5;
    BYTE b6;
    BYTE b7;
    BYTE b8;
} ButtonState;


// GLOBAL VARIABLES

/**
 * Hold the minimum, central and maximum values for each axis in the joystick
 */
JoyPos JOY_MIN, JOY_MAX, JOY_CURRENT;

/**
 * Holds integer values representing button states
 */
```

```
ButtonState BUT_CURRENT;

/**
 * TimerHandle for Joystick timer
 */
TimerHandle JoystickTimer;


// FUNCTIONS

/**
 * Input:    NONE
 * Output:   0   = ok
 *          -1 = error
 * Semantics:   Callibrates the minimum and maximum values for each axis
 *          in the joystick and sets the global variables.
 */
int callibrateJoystick( void );

/**
 * Input:    NONE
 * Output:   0   = ok
 *          -1 = error
 * Semantics:   Callibrates the joystcik, and starts the reading of joystick values
 */
int startJoystick( void );

/**
 * Input:    NONE
 * Output:   0   = ok
 *          -1 = error
 * Semantics:   Stops the reading of joystick values
 */
int stopJoystick( void );

/**
 * Input:    NONE
 * Output:   NONE
 * Semantics:   Updates the current settings of the joystick, and sets the
 *              ODW speed appropriately
 */
void Joystick( void );

/**
 * Input:    NONE
 * Output:   NONE
```

```
 * Semantics:   Get the current position of each axis in the joystick.
 *              The result will be stored in the global variable JOY_CURRENT
 */
void updateJoystickPosition( void );

/**
 * Input:   NONE
 * Output:  NONE
 * Semantics:   Get current state of each button on the joystick base
 *              The result will be stored in the global variable BUT_CURRENT
 */
void updateJoystickButtons( void );

/**
 * Input:       NONE
 * Output:      The current position of the joystick
 * Semantics:   Get the current position of the joystick
 */
JoyPos getJoystickPosition( void );

/**
 * Input:       NONE
 * Output:      The current state of all 4 joystick buttons
 * Semantics:   Get the current state of the 4 buttons
 */
ButtonState getJoystickButtons( void );

/**
 * Input:       Button code
 *              0 = Any buttons pushed
 *              5 = Button 5
 *              6 = Button 6
 *              7 = Button 7
 *              8 = Button 8
 * Output:      A bit value for whether the buttons are currently being pushed
 *              -1 = Illegal button code
 *               0 = Not pushed
 *               1 = Being pushed
 * Semantics:   Get the current state of a button on the joystick, or
 *              determine whether any button is currently being pressed.
 */
BYTE isButtonPushed( int button_code );
```

A.6 ODW_Joystick.c

```
/**
 * ODW_Joystick.c
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   This is the code for initialising and reading from the ↩
    ODW's joystick.
 */


#include "eyebot.h"
#include "ODW_Joystick.h"
#include "ODW_MotorCtrl.h"
#include <math.h>


int callibrateJoystick()
{
    // Auto callibration (magic numbers)
    JOY_MIN = (JoyPos) {95, 920, 145, 1000};
    JOY_MAX = (JoyPos) {905, 75, 845, 58};

    /*
    // Manual callibration
    LCDClear();
    LCDPrintf("Move joy to...\n");

    LCDPrintf("bottom left\n");
    LCDPrintf("& push button 5\n");
    while( !isButtonPushed(5) );
    AUBeep();
    JOY_MIN = joy();
    OSWait(50);

    LCDPrintf("top right\n");
    LCDPrintf("& push button 6\n");
    while( !isButtonPushed(6) );
    AUBeep();
    JOY_MAX = joy();
    LCDPrintf("Done!\n");
    OSWait(100);
    */
```

```c

    return 0;
}


int startJoystick()
{
    int result = callibrateJoystick();

    if ( result != 0 ) return result;

    JoystickTimer = OSAttachTimer( JOY_FREQ, Joystick );

    if ( JoystickTimer == 0 ) result = 1;
    else result = 0;

    return result;
}


int stopJoystick()
{
    int result = !OSDetachTimer( JoystickTimer );

    return result;
}


void Joystick()
{
    double divisor;

    // Read current joystick values and update global ↩
        variables accordingly
    updateJoystickPosition();
    updateJoystickButtons();

    // Adjust speed values to account for throttle position
    divisor = THROTTLE_DIVISOR - (THROTTLE_DIVISOR-1)/100 * ↩
        JOY_CURRENT.t;

    // Set the ODW speed accordingly
    setWCSpeed( (WCSpeed) {round(JOY_CURRENT.y/divisor), ↩
        round(JOY_CURRENT.x/divisor), ↩
        round(JOY_CURRENT.z/divisor)} );
}


void updateJoystickPosition()
```

```
{
    // Read raw values from joystick
    JOY_CURRENT.x = OSGetAD( X_CHANNEL );
    JOY_CURRENT.y = OSGetAD( Y_CHANNEL );
    JOY_CURRENT.z = OSGetAD( Z_CHANNEL );
    JOY_CURRENT.t = OSGetAD( T_CHANNEL );

    // Adjust for min/max and threshold region
    // Convert to absolute value between 0 and 100 (does not account for +ve and -ve)
    double x = 100*( abs(200*(JOY_CURRENT.x-JOY_MIN.x)/(JOY_MAX.x-JOY_MIN.x)-100) - X_THRESHOLD )/( 100-X_THRESHOLD );
    double y = 100*( abs(200*(JOY_CURRENT.y-JOY_MIN.y)/(JOY_MAX.y-JOY_MIN.y)-100) - Y_THRESHOLD )/( 100-Y_THRESHOLD );
    double z = 100*( abs(200*(JOY_CURRENT.z-JOY_MIN.z)/(JOY_MAX.z-JOY_MIN.z)-100) - Z_THRESHOLD )/( 100-Z_THRESHOLD );

    // Check limits of 0 and 100 are enforced
    if(x<0) x=0;
    if(y<0) y=0;
    if(z<0) z=0;
    if(x>100) x=100;
    if(y>100) y=100;
    if(z>100) z=100;

    // Convert back to positive and negative values between -100 and 100
    if(JOY_CURRENT.x > 500) x = -1*x;
    if(JOY_CURRENT.y > 500) y = -1*y;
    if(JOY_CURRENT.z > 500) z = -1*z;

    // Round to integers
    JOY_CURRENT.x = (int) round(x);
    JOY_CURRENT.y = (int) round(y);
    JOY_CURRENT.z = (int) round(z);

    // Correct throttle to account for the min/max
    JOY_CURRENT.t = round(100 * (JOY_CURRENT.t - JOY_MIN.t)/(JOY_MAX.t - JOY_MIN.t));
    if(JOY_CURRENT.t<0) JOY_CURRENT.t=0;
    if(JOY_CURRENT.t>100) JOY_CURRENT.t=100;
}


void updateJoystickButtons()
{
```

```
    // Read raw values from joystick, and extract associated
     bit
    BUT_CURRENT.b5 = !((OSReadInLatch( 0 ) &
       BUTTON5)/BUTTON5);
    BUT_CURRENT.b6 = !((OSReadInLatch( 0 ) &
       BUTTON6)/BUTTON6);
    BUT_CURRENT.b7 = !((OSReadInLatch( 0 ) &
       BUTTON7)/BUTTON7);
    BUT_CURRENT.b8 = !((OSReadInLatch( 0 ) &
       BUTTON8)/BUTTON8);
}

JoyPos getJoystickPosition( void )
{
    JoyPos joypos;

    joypos.x = JOY_CURRENT.x;
    joypos.y = JOY_CURRENT.y;
    joypos.z = JOY_CURRENT.z;
    joypos.t = JOY_CURRENT.t;

    return joypos;
}

ButtonState getJoystickButtons( void )
{
    ButtonState bs;

    bs.b5 = BUT_CURRENT.b5;
    bs.b6 = BUT_CURRENT.b6;
    bs.b7 = BUT_CURRENT.b7;
    bs.b8 = BUT_CURRENT.b8;

    return bs;
}

BYTE isButtonPushed( int button_code )
{
    switch ( button_code ) {
        case 0:
            return ( BUT_CURRENT.b5 || BUT_CURRENT.b6 ||
               BUT_CURRENT.b7 || BUT_CURRENT.b8 );
            break;
        case 5:
            return BUT_CURRENT.b5;
            break;
        case 6:
            return BUT_CURRENT.b6;
            break;
```

```
        case 7:
            return BUT_CURRENT.b7;
            break;
        case 8:
            return BUT_CURRENT.b8;
            break;
    }

    return -1;
}
```

A.7 ODW_IR.h

```
/**
 * ODW_IR.h
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   This is header file for the code that allows control of the ODW via an
 *   infra-red remote control.
 *
 * Note:
 *   The types, variables and functions defined here should be available
 *   regardless of the implementation.
 */


/**
 * Key Code      Meaning
 *
 * 0             No Key
 * RC_0          0 Key
 * RC_1          1 Key
 * RC_2          2 Key
 * RC_3          3 Key
 * RC_4          4 Key
 * RC_5          5 Key
 * RC_6          6 Key
 * RC_7          7 Key
 * RC_8          8 Key
 * RC_9          9 Key
 * RC_PLUS       + Key
 * RC_MINUS      - Key
 * RC_FF         >> Key
 * RC_RW         << Key
 * RC_STOP       Stop Key
 * RC_PLAY       Play Key
 * RC_STANDBY    On/Off Key
 * RC_OK         OK Key
 */
```

```
// DEFINITIIONS

/**
 * 100/Hz
 * The frequency with which to check for an IR key press
 */
#define IR_FREQ 100/10


// GLOBAL VARIABLES

/**
 * Handle for the Infra Red Timer which checks for an
 * infra red key press with frequency determined by IR_FREQ.
 */
TimerHandle IRTimer;

/**
 * The size of a step increase in speed.
 */
int IRstep;


// FUNCTIONS

/**
 * Input:    NONE
 * Output:   0   = ok
 *          -1 = error
 * Semantics:   Initialise and start the control by IR ↩
    remote control
 */
int startIR( void );

/**
 * Input:    NONE
 * Output:   0   = ok
 *          -1 = error
 * Semantics:   Check for and deal with a key press
 */
void IRKey( void );

/**
 * Input:    NONE
 * Output:   0   = ok
 *          -1 = error
 * Semantics:   Stop the control by IR remote control
 */
int stopIR( void );
```

A.8 ODW_IR.c

```c
/**
 * ODW_IR.c
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   This is code to allow control of the ODW via an infra-red remote control.
 */


#include "eyebot.h"
#include "ODW_IR.h"
#include "ODW_MotorCtrl.h"
#include "irtv.h"
#include "IRnokia.h"


int startIR( void )
{
    // Initialise IR using appropriate settings
    int result = IRTVInit(SPACE_CODE,15,0,0x03ff,SLOPPY_MODE,1,10);

    IRstep = 10;
    IRTimer = OSAttachTimer( IR_FREQ, IRKey );

    return result;
}


int stopIR( void )
{
    int result = !OSDetachTimer( IRTimer );
    // Terminate IR
    IRTVTerm();

    return result;
}


void IRKey( void )
{
```

```
    int IRKey = IRTVRead();
    int x, y, w;
    WCSpeed currentwcspeed = getWCSpeed();

    switch( IRKey )
    {
        case 0:
            // No key pressed
            break;

        case RC_0:
            // 0 key pressed: STOP!
            setWCSpeed( WC_STOPPED );
            break;

        case RC_1:
            // 1 key pressed: Forward-Left
            if ( currentwcspeed.x == currentwcspeed.y && currentwcspeed.w == 0 )
            {
                x = currentwcspeed.x;
                y = currentwcspeed.y;
                if ( x < 0 ) x = 0;
                if ( y < 0 ) y = 0;
                x = x + IRstep;
                y = y + IRstep;
            } else {
                x = IRstep;
                y = IRstep;
            }
            setWCSpeed( (WCSpeed) {x, y, 0} );
            break;

        case RC_2:
            // 2 key pressed: Forward
            if ( currentwcspeed.y == 0 && currentwcspeed.w == 0 )
            {
                x = currentwcspeed.x;
                if ( x < 0 ) x = 0;
                x = x + IRstep;
            } else {
                x = IRstep;
            }
            setWCSpeed( (WCSpeed) {x, 0, 0} );
            break;

        case RC_3:
            // 3 key pressed: Forward-Right
```

```
            if ( currentwcspeed.x == -1*currentwcspeed.y && currentwcspeed.w == 0 )
            {
                x = currentwcspeed.x;
                y = currentwcspeed.y;
                if ( x < 0 ) x = 0;
                if ( y > 0 ) y = 0;
                x = x + IRstep;
                y = y - IRstep;
            } else {
                x = IRstep;
                y = -1 * IRstep;
            }
            setWCSpeed( (WCSpeed) {x, y, 0} );
            break;

        case RC_4:
            // 4 key pressed: Left
            if ( currentwcspeed.x == 0 && currentwcspeed.w == 0 )
            {
                y = currentwcspeed.y;
                if ( y < 0 ) y = 0;
                y = y + IRstep;
            } else {
                y = IRstep;
            }
            setWCSpeed( (WCSpeed) {0, y, 0} );
            break;

        case RC_5:
            // 5 key pressed: STOP!
            setWCSpeed( WC_STOPPED );
            break;

        case RC_6:
            // 6 key pressed: Right
            if ( currentwcspeed.x == 0 && currentwcspeed.w == 0 )
            {
                y = currentwcspeed.y;
                if ( y > 0 ) y = 0;
                y = y - IRstep;
            } else {
                y = -1 * IRstep;
            }
            setWCSpeed( (WCSpeed) {0, y, 0} );
            break;

```

```
        case RC_7:
            // 7 key pressed: Backward-Left
            if ( -1*currentwcspeed.x == currentwcspeed.y && currentwcspeed.w == 0 )
            {
                x = currentwcspeed.x;
                y = currentwcspeed.y;
                if ( x > 0 ) x = 0;
                if ( y < 0 ) y = 0;
                x = x - IRstep;
                y = y + IRstep;
            } else {
                x = -1 * IRstep;
                y = IRstep;
            }
            setWCSpeed( (WCSpeed) {x, y, 0} );
            break;

        case RC_8:
            // 8 key pressed: Backward
            if ( currentwcspeed.y == 0 && currentwcspeed.w == 0 )
            {
                x = currentwcspeed.x;
                if ( x > 0 ) x = 0;
                x = x - IRstep;
            } else {
                x = -1 * IRstep;
            }
            setWCSpeed( (WCSpeed) {x, 0, 0} );
            break;

        case RC_9:
            // 9 key pressed: Backward-Right
            if ( -1*currentwcspeed.x == -1*currentwcspeed.y && currentwcspeed.w == 0 )
            {
                x = currentwcspeed.x;
                y = currentwcspeed.y;
                if ( x > 0 ) x = 0;
                if ( y > 0 ) y = 0;
                x = x - IRstep;
                y = y - IRstep;
            } else {
                x = -1 * IRstep;
                y = -1 * IRstep;
            }
            setWCSpeed( (WCSpeed) {x, y, 0} );
            break;
```

```

        case RC_PLUS:
            // + key pressed: Increase Step Size
            IRstep = IRstep + 10;
        if ( IRstep > 100 ) IRstep = 100;
        if ( IRstep < 5 ) IRstep = 5;
            break;

        case RC_MINUS:
            // - key pressed: Decrease Step Size
            IRstep = IRstep - 10;
        if ( IRstep > 100 ) IRstep = 100;
        if ( IRstep < 5 ) IRstep = 5;
            break;

        case RC_FF:
            // >> key pressed: Rotate Right
            if ( currentwcspeed.x == 0.0 && currentwcspeed.y↩
                == 0.0 )
            {
                w = currentwcspeed.w;
                if ( w > 0 ) w = 0;
                w = w - IRstep;
            } else {
                w = -1 * IRstep;
            }
            setWCSpeed( (WCSpeed) {0, 0, w} );
            break;

        case RC_RW:
            // << key pressed: Rotate Left
            if ( currentwcspeed.x == 0.0 && currentwcspeed.y↩
                == 0.0 )
            {
                w = currentwcspeed.w;
                if ( w < 0 ) w = 0;
                w = w + IRstep;
            } else {
                w = IRstep;
            }
            setWCSpeed( (WCSpeed) {0, 0, w} );
            break;

        case RC_STOP:
            // Stop key pressed: STOP!
            setWCSpeed( WC_STOPPED );
            break;
    }
}
```

A.9 Makefile

```
include Makeincl

LIBS       = -lm
CFLAGS     =
AFLAGS     =

ASMSOURCE = $(wildcard *.s)
CSOURCE   = $(wildcard *.c)


all:    ODW.hex

ODW.hex:        ODW.o ODW_MotorCtrl.o ODW_Joystick.o ODW_IR.o
        $(CC68) $(CFLAGS) -o ODW.hex ODW.o ODW_MotorCtrl.o ODW_Joystick.o ODW_IR.o $(LIBS)

clean:
        -$(RM)  $(addsuffix .hex,$(basename $(CSOURCE)) $(basename $(ASMSOURCE))) \
                $(addsuffix .elf,$(basename $(CSOURCE))) \
                $(addsuffix .o,$(basename $(CSOURCE))) \
                $(addsuffix .o,$(basename $(ASMSOURCE))) \
                *.hex core

%.o:    %.c
        $(CC68) $(CFLAGS) -c -o $@ $<

%.o:    %.s
        $(AS68) $(AFLAGS) -c -o $@ $<
```

A.10 Makeincl

```
# Shell script works for Linux and Windows
# Thomas Braunl, March 2004

# Set current version
mc = /home/bwoods/ROBIOS
CC68 = gcc68
AR68 = m68k-eyebot-elf-ar
RANLIB68  = m68k-eyebot-elf-ranlib
SREC2BIN = srec2bin

# Detect if running on Unix or DOS.  'ver' exists on DOS only
ifeq ($(sh ver),)
 PLATFORM = UNIX
 COPY = cp -f
 RM = rm -f
 TMPDIR = /tmp
 AS68 = gas68

else
 PLATFORM = DOS/Windows
 COPY    = copy
 RM      = del
 TMPDIR  = .
 AS68 = gas68

endif
```

A.11 test.c

```
/**
 * test.c
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   Test the basic use of the Eyebot LCD.
 */

#include "eyebot.h"

int main( int argc, char* argv[] )
{
    LCDPrintf("Hello, World!\n");

    OSWait(500);

    return 0;
}
```

A.12 mctest.c

```
/**
 * mctest.c
 *
 * Author:
 *    Benjamin Woods (10218282)
 *    The University of Western Australia
 *    Bachelor of Engineering & Bachelor of Commerce
 *    Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *    Test the communication with the Roboteq cards and the ↩
     resulting motor speeds
 */

#include "eyebot.h"

#define BAUD SER9600
#define HANDSHAKE NONE
#define INTERFACE SERIAL2

#define sccr1 0xfc0a

int main( int argc, char *argv[] )
{
    int key, error;
    char ch;

    LCDClear();
    LCDMenu("GO", "", "", "STOP");

    LCDPrintf("Init...\n");
    OSInitRS232( BAUD, HANDSHAKE, INTERFACE );

    // Set RS232 on SERIAL2 to 7Bit EvenParity and 1 Stopbit
    (*(((volatile BYTE*)Ser1Base)+3)) = 0x1a;

    // Set RS232 on SERIAL1 to 7Bit EvenParity and 1 Stopbit
    //(*((volatile unsigned short*)sccr1)) = 0x042c;

    do
    {
        key = KEYGet();
    } while ( key != KEY1 && key != KEY4 );

    OSWait(10);

    if( key == KEY4 ) return 0;
```

```

    LCDPrintf("Start...\n");

    while( KEYRead() != KEY4 )
    {
        error = OSRecvRS232(&ch, INTERFACE);
        if( error == 0 ) LCDPutChar(ch);
        else if( error != 1 ) LCDPrintf("Error: %d\n", error);
        ch = 0;

        OSSendCharRS232('!', INTERFACE);
        OSSendCharRS232('a', INTERFACE);
        OSSendCharRS232('5', INTERFACE);
        OSSendCharRS232('5', INTERFACE);
        OSSendCharRS232('\r', INTERFACE);
    }

    LCDPrintf("Stop...\n");
    OSSendCharRS232('!', INTERFACE);
    OSSendCharRS232('a', INTERFACE);
    OSSendCharRS232('0', INTERFACE);
    OSSendCharRS232('0', INTERFACE);
    OSSendCharRS232('\r', INTERFACE);

    return 0;
}
```

A.13 joytest.c

```c
/**
 * joytest.c
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   Test the joystick by displaying readings as numbers on the screen.
 *   The LCD can be set (using the Eyebot keys) to display either:
 *     - Raw values read from the 4 potentiometers in the joystick
 *     - Percentage and thresholded values of the joysticks position
 *     - 0's and 1's representing the states of the 4 joystick buttons
 */

#include "eyebot.h"
#include <limits.h>
#include <math.h>

#define BUTTON5 0x10
#define BUTTON6 0x20
#define BUTTON7 0x40
#define BUTTON8 0x80
#define X_CHANNEL 4
#define Y_CHANNEL 5
#define Z_CHANNEL 6
#define T_CHANNEL 7
#define X_THRESHOLD 15
#define Y_THRESHOLD 15
#define Z_THRESHOLD 15

typedef struct
{
    int x;
    int y;
    int z;
    int t;
} JoyPos;

int STOP_RUNNING;
```

```
int percent, buttons;
JoyPos MAX, MIN;

int button( int MASK )
{
    return (int) !((OSReadInLatch( 0 ) & MASK)/MASK);
}

JoyPos joy()
{
    JoyPos joypos;

    joypos.x = OSGetAD( X_CHANNEL );
    joypos.y = OSGetAD( Y_CHANNEL );
    joypos.z = OSGetAD( Z_CHANNEL );
    joypos.t = OSGetAD( T_CHANNEL );

    return joypos;
}

void callibrate()
{
    // Auto callibration (magic numbers)
    MIN = (JoyPos) {95, 920, 145, 1000};
    MAX = (JoyPos) {905, 75, 845, 58};

    /*
    // Manual callibration
    LCDClear();
    LCDPrintf("Move joy to...\n");

    LCDPrintf("bottom left\n");
    LCDPrintf("& push button 5\n");
    AUTone(5000, 50);
    OSWait(100);
    AUTone(5000, 50);
    OSWait(100);
    AUBeep();
    MIN = joy();
    OSWait(300);

    LCDPrintf("top right\n");
    LCDPrintf("& push button 6\n");
    AUTone(5000, 50);
    OSWait(100);
    AUTone(5000, 50);
    OSWait(100);
    AUBeep();
    MAX = joy();
```

```

    LCDPrintf("Done!\n");
    OSWait(100);
    */
}

int main( int argc, char *argv[] )
{
    STOP_RUNNING = 0;
    percent = 0;
    int b5, b6, b7, b8, key;
    JoyPos joypos;

    while( STOP_RUNNING == 0 )
    {
        LCDClear();
        LCDMenu("RAW", "%", "BUT", "EXIT");

        if( buttons == 0 )
        {
            joypos = joy();

            if( percent == 0 )
            {
                LCDPrintf("Raw:\n");
            } else {
                double x = 100*( ↩
                   abs(200*(joypos.x-MIN.x)/(MAX.x-MIN.x)-100) ↩
                   - X_THRESHOLD )/( 100-X_THRESHOLD );
                double y = 100*( ↩
                   abs(200*(joypos.y-MIN.y)/(MAX.y-MIN.y)-100) ↩
                   - Y_THRESHOLD )/( 100-Y_THRESHOLD );
                double z = 100*( ↩
                   abs(200*(joypos.z-MIN.z)/(MAX.z-MIN.z)-100) ↩
                   - Z_THRESHOLD )/( 100-Z_THRESHOLD );

                if(x<0) x=0;
                if(y<0) y=0;
                if(z<0) z=0;
                if(x>100) x=100;
                if(y>100) y=100;
                if(z>100) z=100;

                if(joypos.x < 500) x = -1*x;
                if(joypos.y > 500) y = -1*y;
                if(joypos.z < 500) z = -1*z;

                joypos.x = (int) round(x);
                joypos.y = (int) round(y);
```

```
                joypos.z = (int) round(z);

                joypos.t = round(100 * (joypos.t - MIN.t)/(MAX.t - MIN.t));
                if(joypos.t<0) joypos.t=0;
                if(joypos.t>100) joypos.t=100;

                LCDPrintf("Percent:\n");
            }
            LCDPrintf("X: %d\n", joypos.x);
            LCDPrintf("Y: %d\n", joypos.y);
            LCDPrintf("Z: %d\n", joypos.z);
            LCDPrintf("T: %d\n", joypos.t);
        } else {
            b5 = button( BUTTON5 );
            b6 = button( BUTTON6 );
            b7 = button( BUTTON7 );
            b8 = button( BUTTON8 );
            LCDPrintf("Buttons:\n");
            LCDPrintf("B5: %d\n", b5);
            LCDPrintf("B6: %d\n", b6);
            LCDPrintf("B7: %d\n", b7);
            LCDPrintf("B8: %d\n", b8);
            if( b5 || b6 || b7 || b8 ) AUTone(b5*1000+b6*1500+b7*3000+b8*3500, 50);
        }

        key = KEYRead();
        switch( key )
        {
            case KEY1:
                percent = 0;
                buttons = 0;
                break;
            case KEY2:
                percent = 1;
                buttons = 0;
                callibrate();
                break;
            case KEY3:
                buttons = 1;
                percent = 0;
                break;
            case KEY4:
                STOP_RUNNING = 1;
                break;
            default:
                break;
        }
```

```

            OSWait(10);
        }

        LCDPrintf("Stopping...\n");

        return 0;
}
```

A.14 joytest2.c

```
/**
 * joytest2.c
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   Test the joystick using a graphical display on the ↩
   Eyebot LCD.
 */

#include "eyebot.h"
#include <limits.h>
#include <math.h>

#define BUTTON5 0x10
#define BUTTON6 0x20
#define BUTTON7 0x40
#define BUTTON8 0x80

#define X_CHANNEL 4
#define Y_CHANNEL 5
#define Z_CHANNEL 6
#define T_CHANNEL 7

#define X_THRESHOLD 15
#define Y_THRESHOLD 15
#define Z_THRESHOLD 15
#define JOY_FREQ 100/5

typedef struct
{
    int x;
    int y;
    int z;
    int t;
} JoyPos;

typedef struct
{
    BYTE b5;
    BYTE b6;
    BYTE b7;
    BYTE b8;
```

```
} ButtonState;

int STOP_RUNNING;
JoyPos JOY_MAX, JOY_MIN, JOY_CURRENT, JOY_OLD;
ButtonState BUT_CURRENT, BUT_OLD;

int xy_realx( int x )
{
    return (round(x/4.0) + 32);
}

int xy_realy( int y )
{
    return (32 - round(y/4.0));
}

int z_realx( int z )
{
    return ( round(z*0.15)+83 );
}

int t_realy( int t )
{
    return ( 52-round(t*0.2) );
}

void xy_plot(int x, int y, int col)
{
    LCDSetPixel( xy_realx(x), xy_realy(y), col );
}

void xy_drawline( int x1, int y1, int x2, int y2, int col )
{
    LCDLine( xy_realx(x1), xy_realy(y1), xy_realx(x2), xy_realy(y2), col );
}

void xy_drawrectangle( int x1, int y1, int x2, int y2, int col )
{
    xy_drawline( x1, y1, x1, y2, col );
    xy_drawline( x1, y2, x2, y2, col );
    xy_drawline( x2, y2, x2, y1, col );
    xy_drawline( x2, y1, x1, y1, col );
}

void xy_drawcross( int x, int y, int col )
{
    xy_drawline( x-2, y-2, x+2, y+2, col );
```

```
    xy_drawline( x-2, y+2, x+2, y-2, col );
}

void drawrectangle( int x1, int y1, int x2, int y2, int col )
{
    LCDLine( x1, y1, x1, y2, col );
    LCDLine( x1, y2, x2, y2, col );
    LCDLine( x2, y2, x2, y1, col );
    LCDLine( x2, y1, x1, y1, col );
}

void drawmap()
{
    xy_drawrectangle( -100, -100, 100, 100, 1 );
    xy_drawline( X_THRESHOLD, -100, X_THRESHOLD, 100, 1 );
    xy_drawline( -1*X_THRESHOLD, -100, -1*X_THRESHOLD, 100, ↩
       1 );
    xy_drawline( -100, Y_THRESHOLD, 100, Y_THRESHOLD, 1 );
    xy_drawline( -100, -1*Y_THRESHOLD, 100, -1*Y_THRESHOLD, ↩
       1 );

    drawrectangle( 68, 12, 98, 22, 1 );
    LCDLine( z_realx(-1*Z_THRESHOLD), 12, ↩
       z_realx(-1*Z_THRESHOLD), 22, 1 );
    LCDLine( z_realx(Z_THRESHOLD), 12, z_realx(Z_THRESHOLD),↩
        22, 1 );

    drawrectangle( 78, 52, 88, 32, 1 );

    drawrectangle( 110, 7, 120, 17, 1 );
    drawrectangle( 110, 20, 120, 30, 1 );
    drawrectangle( 110, 34, 120, 44, 1 );
    drawrectangle( 110, 47, 120, 57, 1 );
}

void drawzt( int z, int t, int col )
{
    LCDLine( z_realx(z), 12, z_realx(z), 22, col );
    LCDLine( 78, t_realy(t), 88, t_realy(t), col );
}

void drawbuttons( BYTE b5, BYTE b6, BYTE b7, BYTE b8, int ↩
   col )
{
    if( b5 ) LCDArea( 111, 8, 119, 16, col );
    if( b6 ) LCDArea( 111, 21, 119, 29, col );
    if( b7 ) LCDArea( 111, 35, 119, 43, col );
    if( b8 ) LCDArea( 111, 48, 119, 56, col );
}
```

```c

void drawreadings()
{
    xy_drawcross( JOY_OLD.x, JOY_OLD.y, 0 );
    drawzt( JOY_OLD.z, JOY_OLD.t, 0 );
    drawbuttons( BUT_OLD.b5, BUT_OLD.b6, BUT_OLD.b7, BUT_OLD.b8, 0 );

    drawmap();

    JOY_OLD.x = JOY_CURRENT.x;
    JOY_OLD.y = JOY_CURRENT.y;
    JOY_OLD.z = JOY_CURRENT.z;
    JOY_OLD.t = JOY_CURRENT.t;

    BUT_OLD.b5 = BUT_CURRENT.b5;
    BUT_OLD.b6 = BUT_CURRENT.b6;
    BUT_OLD.b7 = BUT_CURRENT.b7;
    BUT_OLD.b8 = BUT_CURRENT.b8;

    xy_drawcross( JOY_OLD.x, JOY_OLD.y, 1 );
    drawzt( JOY_OLD.z, JOY_OLD.t, 1 );
    drawbuttons( BUT_OLD.b5, BUT_OLD.b6, BUT_OLD.b7, BUT_OLD.b8, 1 );
}

void button_beep()
{
    if( BUT_CURRENT.b5 || BUT_CURRENT.b6 || BUT_CURRENT.b7 || BUT_CURRENT.b8 )
    AUTone(BUT_CURRENT.b5*1000+BUT_CURRENT.b6*1500+BUT_CURRENT.b7*3000+BUT_CURR
     50);
}

void joy_update()
{
    BUT_CURRENT.b5 = !((OSReadInLatch( 0 ) & BUTTON5)/BUTTON5);
    BUT_CURRENT.b6 = !((OSReadInLatch( 0 ) & BUTTON6)/BUTTON6);
    BUT_CURRENT.b7 = !((OSReadInLatch( 0 ) & BUTTON7)/BUTTON7);
    BUT_CURRENT.b8 = !((OSReadInLatch( 0 ) & BUTTON8)/BUTTON8);

    JOY_CURRENT.x = 200*(OSGetAD( X_CHANNEL ) - JOY_MIN.x)/(JOY_MAX.x - JOY_MIN.x) - 100;
    JOY_CURRENT.y = 200*(OSGetAD( Y_CHANNEL ) - JOY_MIN.y)/(JOY_MAX.y - JOY_MIN.y) - 100;
```

```
    JOY_CURRENT.z = 200*(OSGetAD( Z_CHANNEL ) - JOY_MIN.z)/(JOY_MAX.z - JOY_MIN.z) - 100;
    JOY_CURRENT.t = 100*(OSGetAD( T_CHANNEL ) - JOY_MIN.t)/(JOY_MAX.t - JOY_MIN.t);
}

void callibrate()
{
    // Auto callibration (magic numbers)
    JOY_MIN = (JoyPos) {95, 920, 145, 1000};
    JOY_MAX = (JoyPos) {905, 75, 845, 58};

    /*
    // Manual callibration
    LCDClear();
    LCDPrintf("Move joy to...\n");

    LCDPrintf("bottom left\n");
    LCDPrintf("& push button 5\n");
    AUTone(5000, 50);
    OSWait(100);
    AUTone(5000, 50);
    OSWait(100);
    AUBeep();
    JOY_MIN.= joy();
    OSWait(300);

    LCDPrintf("top right\n");
    LCDPrintf("& push button 6\n");
    AUTone(5000, 50);
    OSWait(100);
    AUTone(5000, 50);
    OSWait(100);
    AUBeep();
    JOY_MAX.= joy();

    LCDPrintf("Done!\n");
    OSWait(100);
    */
}

int main( int argc, char *argv[] )
{
    STOP_RUNNING = 0;
    int key = 0;

    callibrate();

```

```
    TimerHandle joyHandle = OSAttachTimer( JOY_FREQ, joy_update );

    while( STOP_RUNNING == 0 )
    {
    button_beep();

    drawreadings();

        key = KEYRead();
        if( key == KEY4 ) STOP_RUNNING = 1;

        OSWait(15);
    }

    LCDPrintf("Stopping...\n");
    OSDetachTimer( joyHandle );

    return 0;
}
```

A.15 psdtest.c

```c
/**
 * psdtest.c
 *
 * Author:
 *   Benjamin Woods (10218282)
 *   The University of Western Australia
 *   Bachelor of Engineering & Bachelor of Commerce
 *   Final Year Mechatronics Engineering Project 2006
 *
 * Description:
 *   Test the position sensitive devices on the wheelchair
 */

#include <eyebot.h>

int main()
{
    PSDHandle psdright;
    int start, key, current, stop, release;
    psdright = PSDInit(PSD_RIGHT);
    start = PSDStart(psdright, TRUE);
    LCDPrintf("Start: %d\n", start);
    LCDMenu("GO", "", "", "END");
    do {
        key = KEYGet();
    } while( key!=KEY1 && key!=KEY4 );

    while( key!=KEY4 && start==0 )
    {
        if( PSDCheck() ) current = PSDGet(psdright);
        if( current == PSD_OUT_OF_RANGE ) LCDPrintf("Out of range\n");
        else LCDPrintf("Right: %d\n", current);
        key = KEYRead();
        OSWait(10);
    }

    stop = PSDStop();
    LCDPrintf("Stop: %d\n", stop);
    OSWait(10);

    release = PSDRelease();
    LCDPrintf("Release: %d\n", release);
    OSWait(10);

    return 0;
}
```

Appendix B

Mechanical and Electrical Designs

B.1 Joystick Circuit

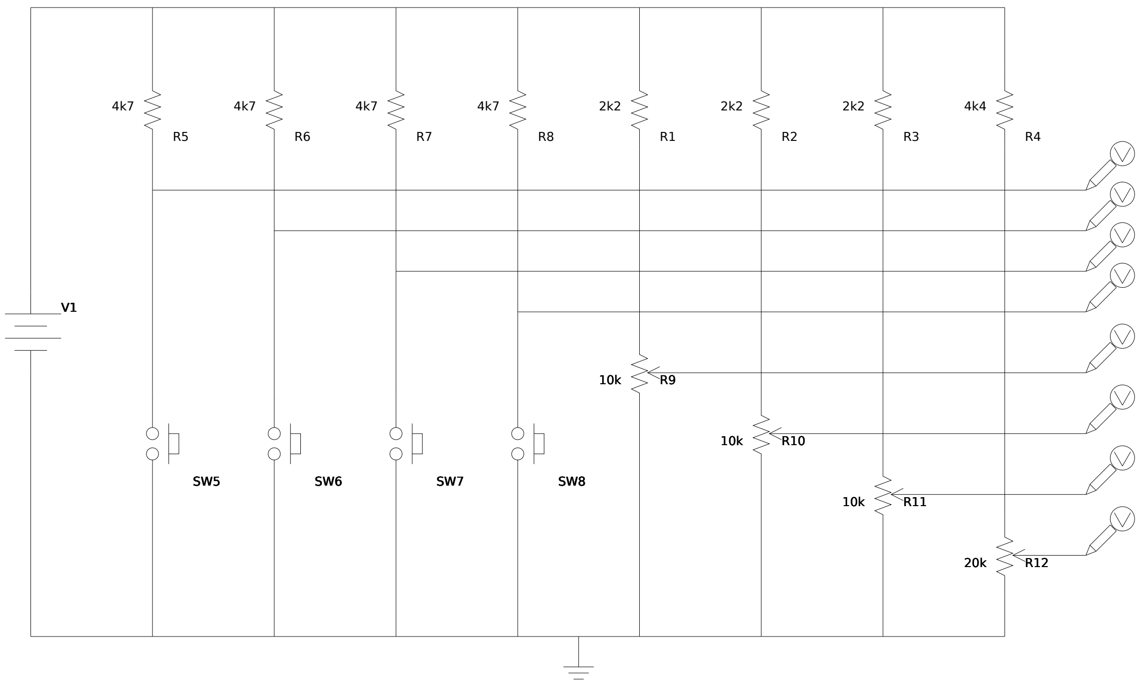


FIGURE B.1: The joystick circuit schematic

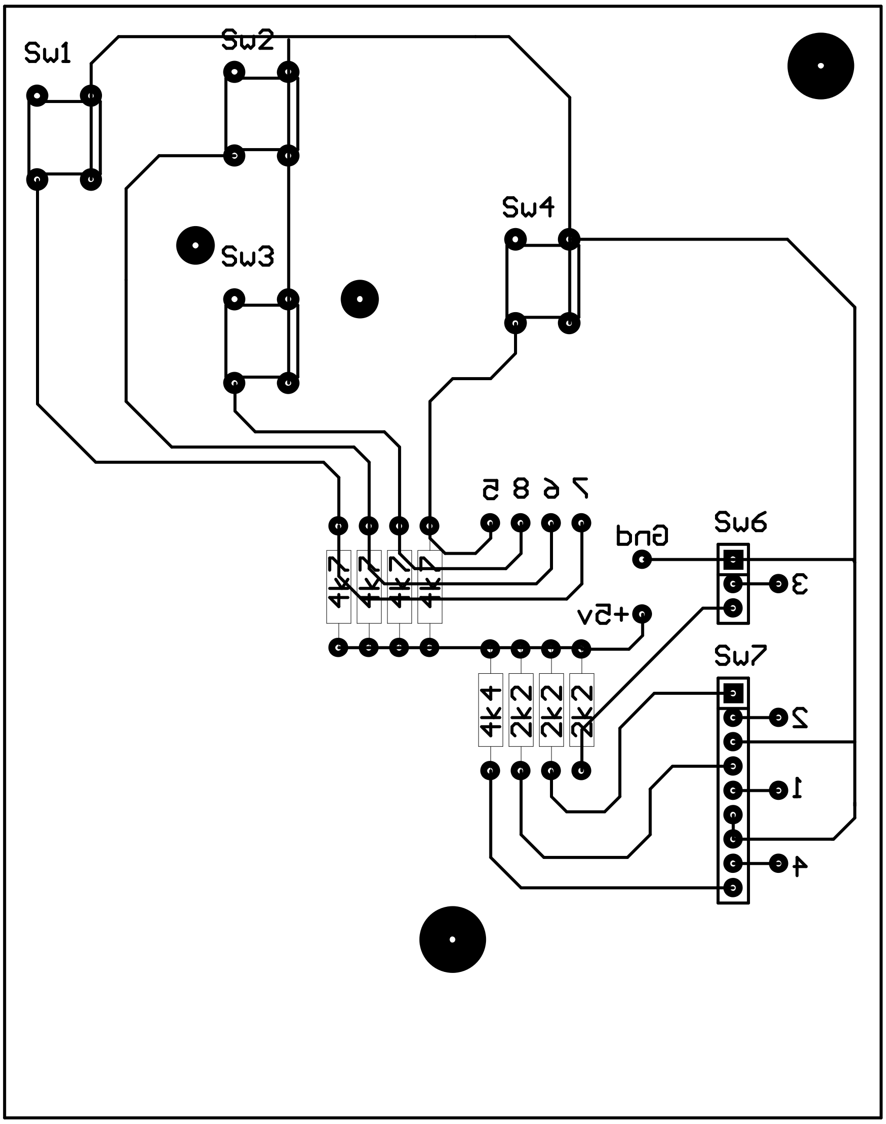
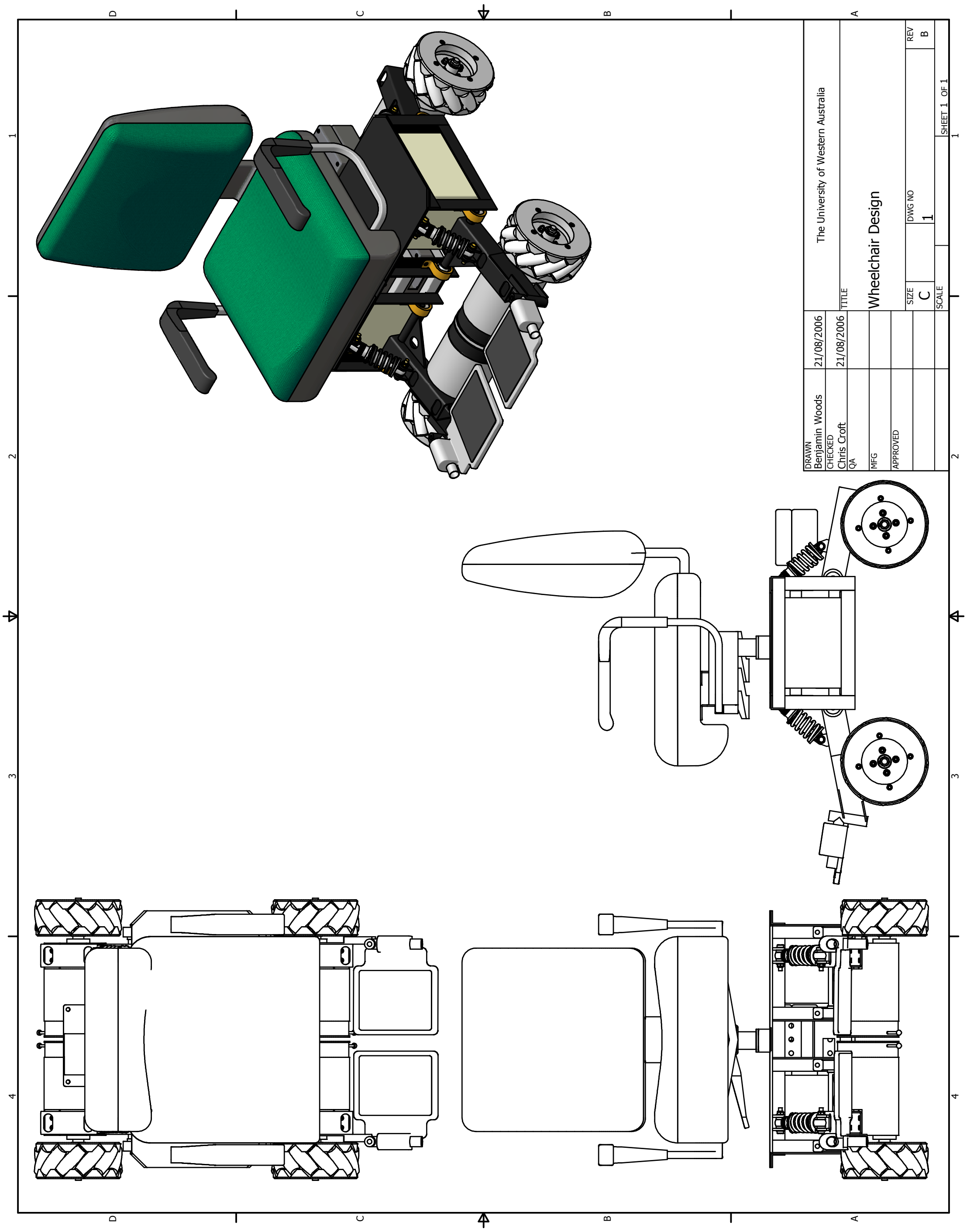
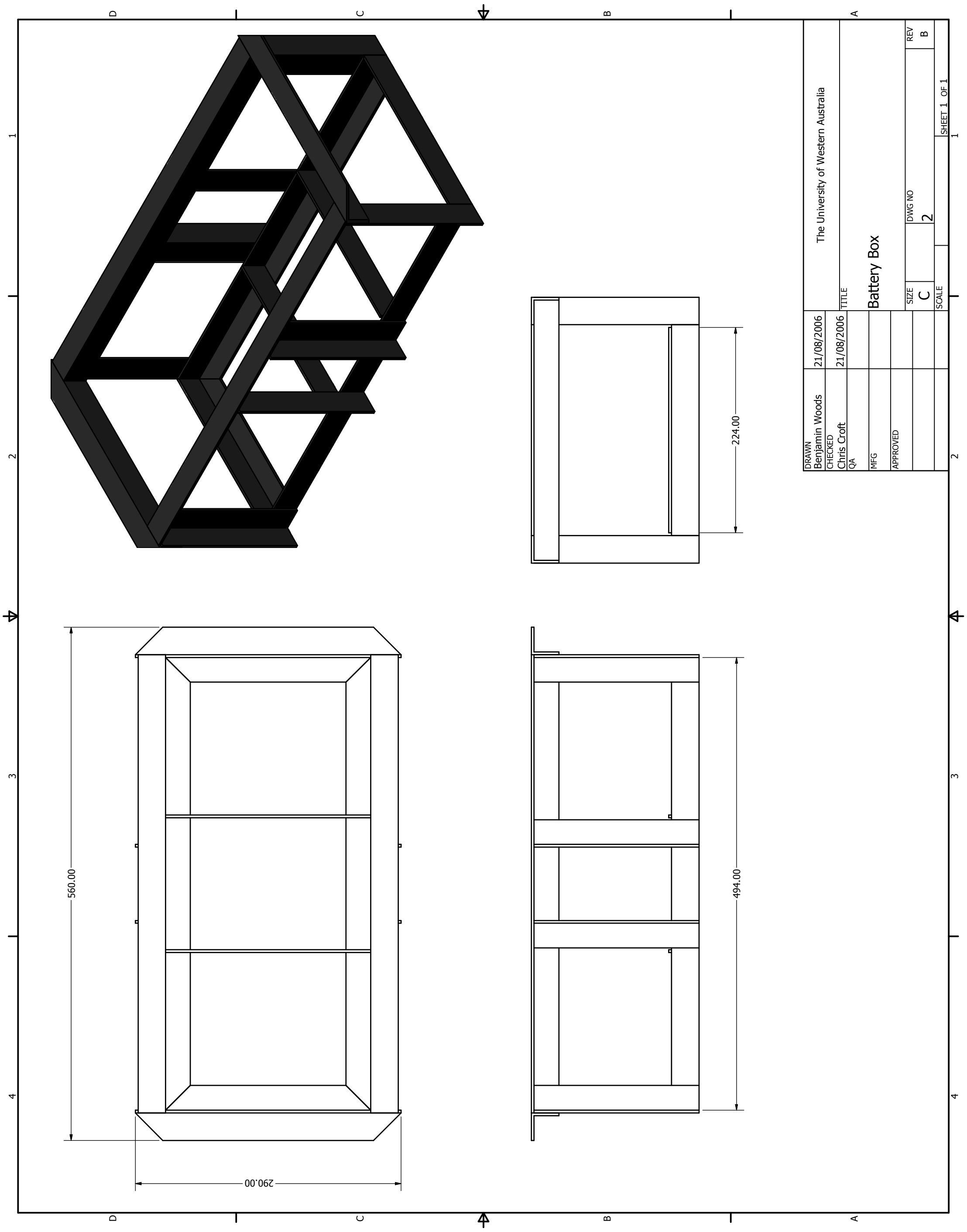
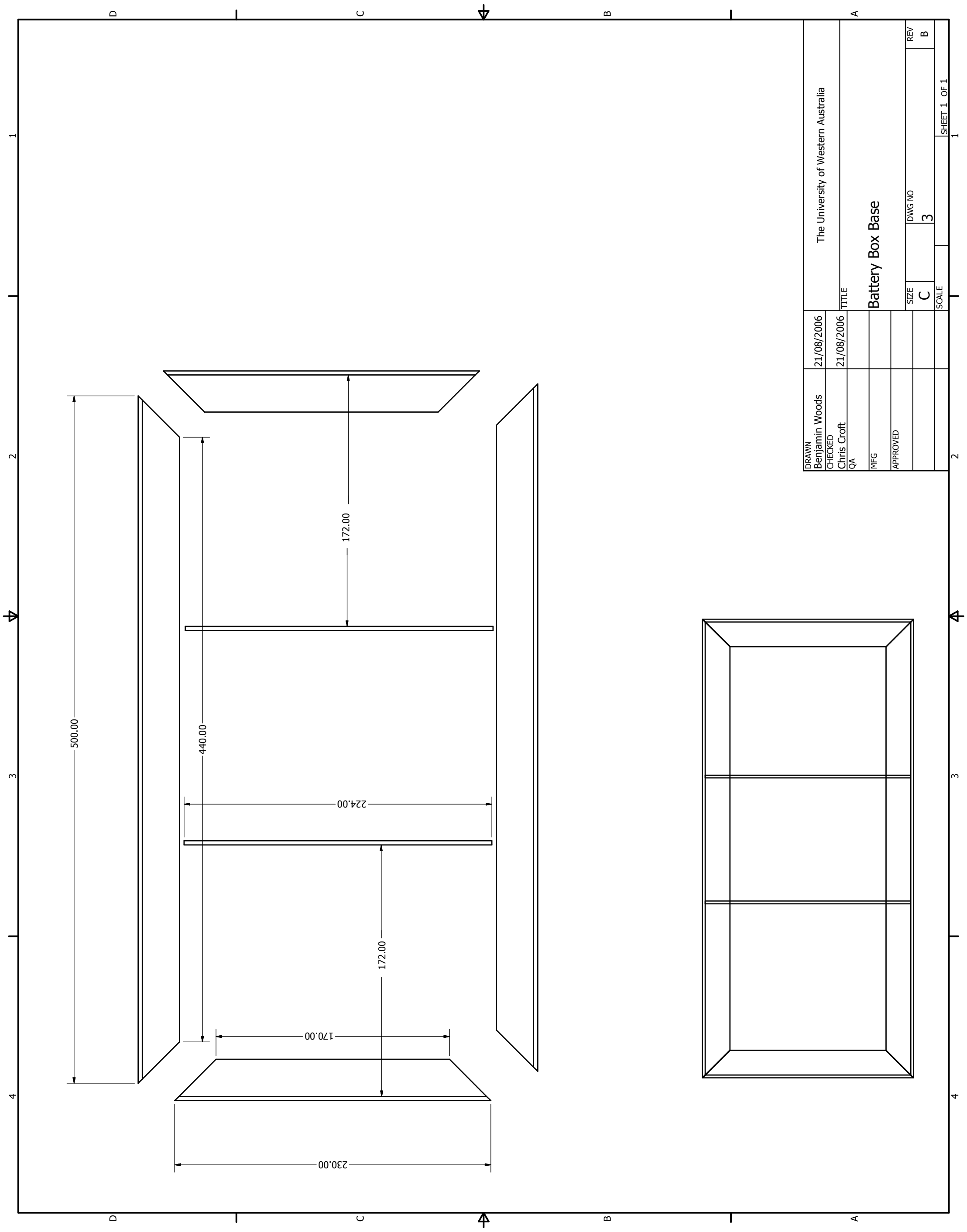


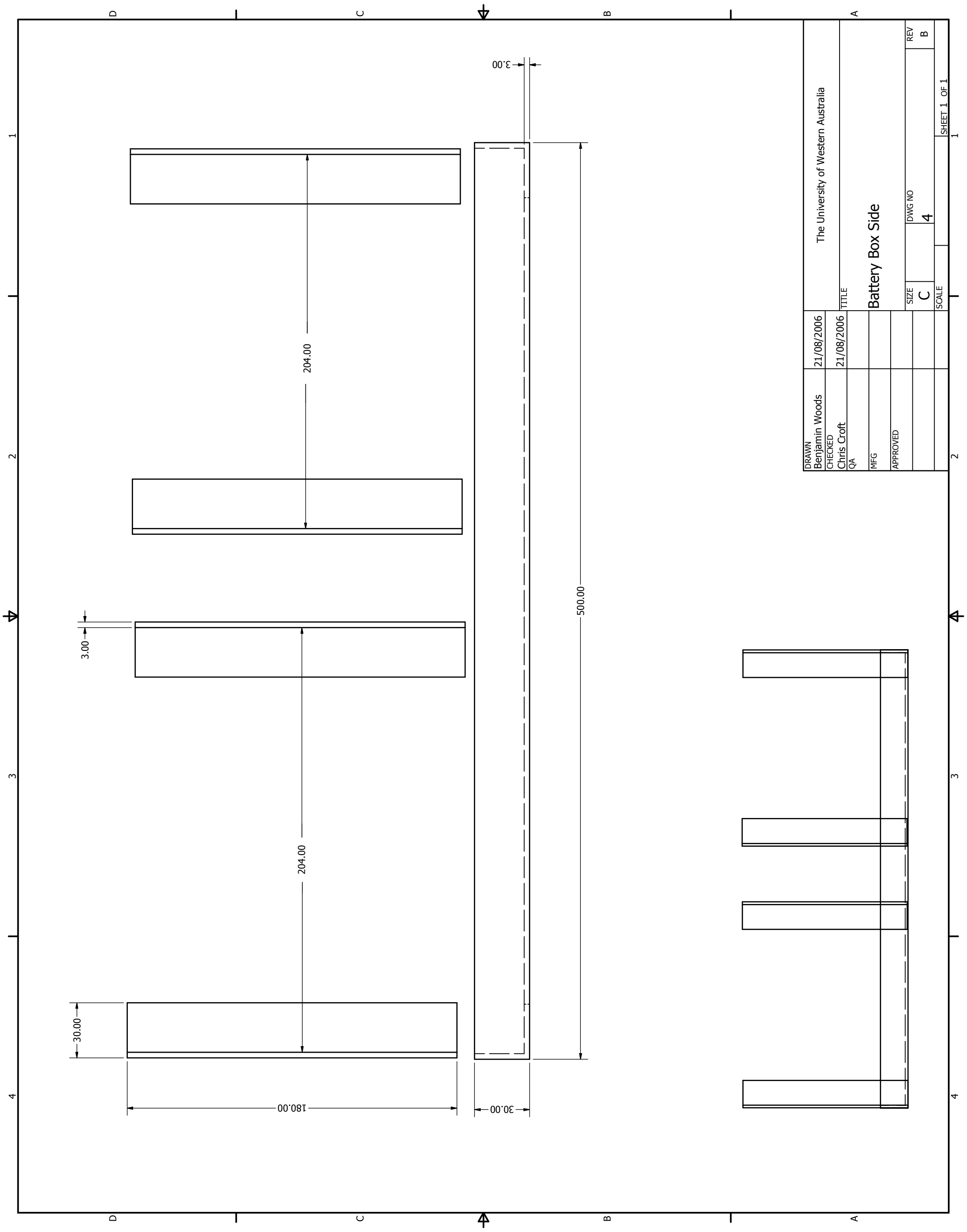
FIGURE B.2: The joystick PCB design

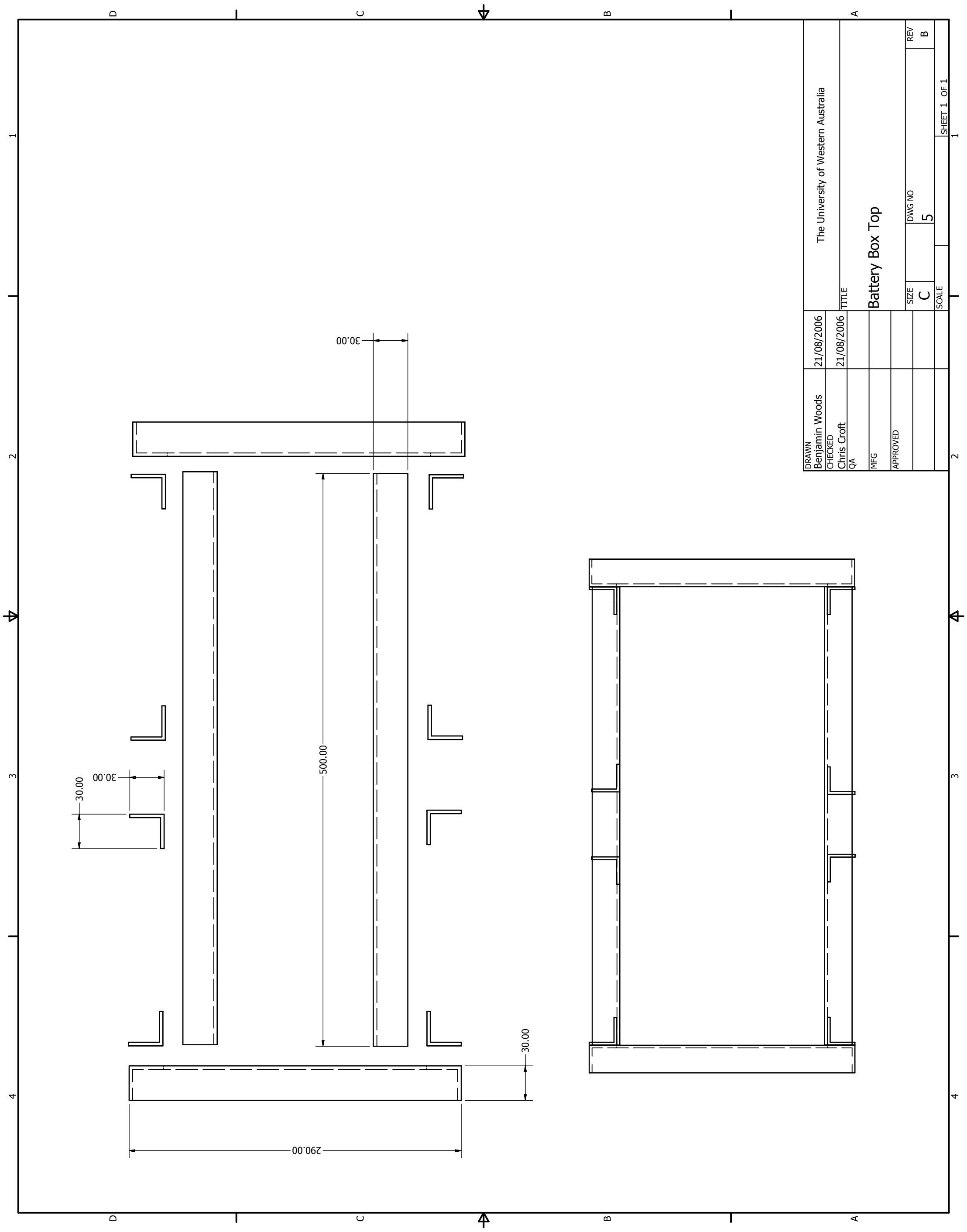
B.2 Suspension Designs

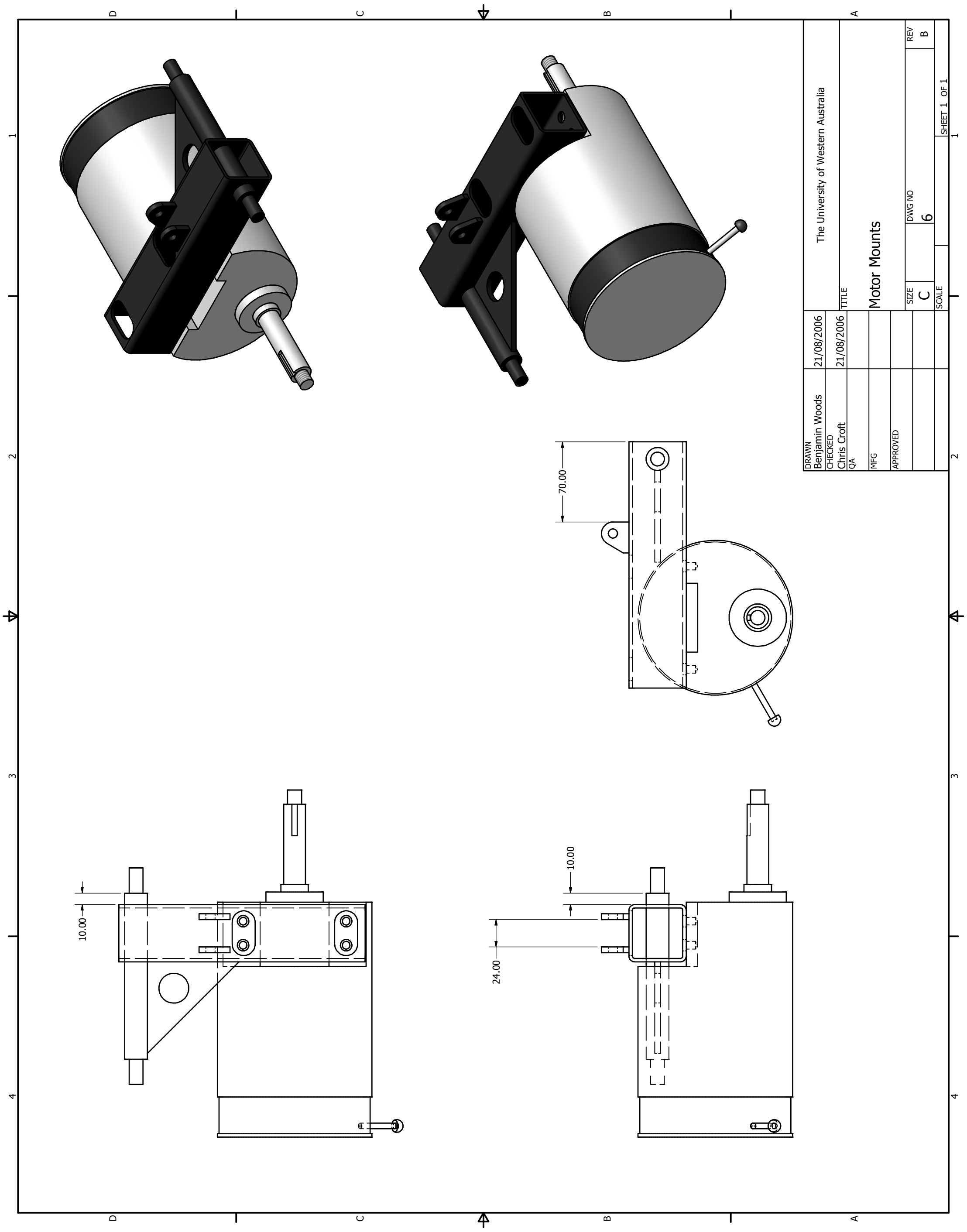


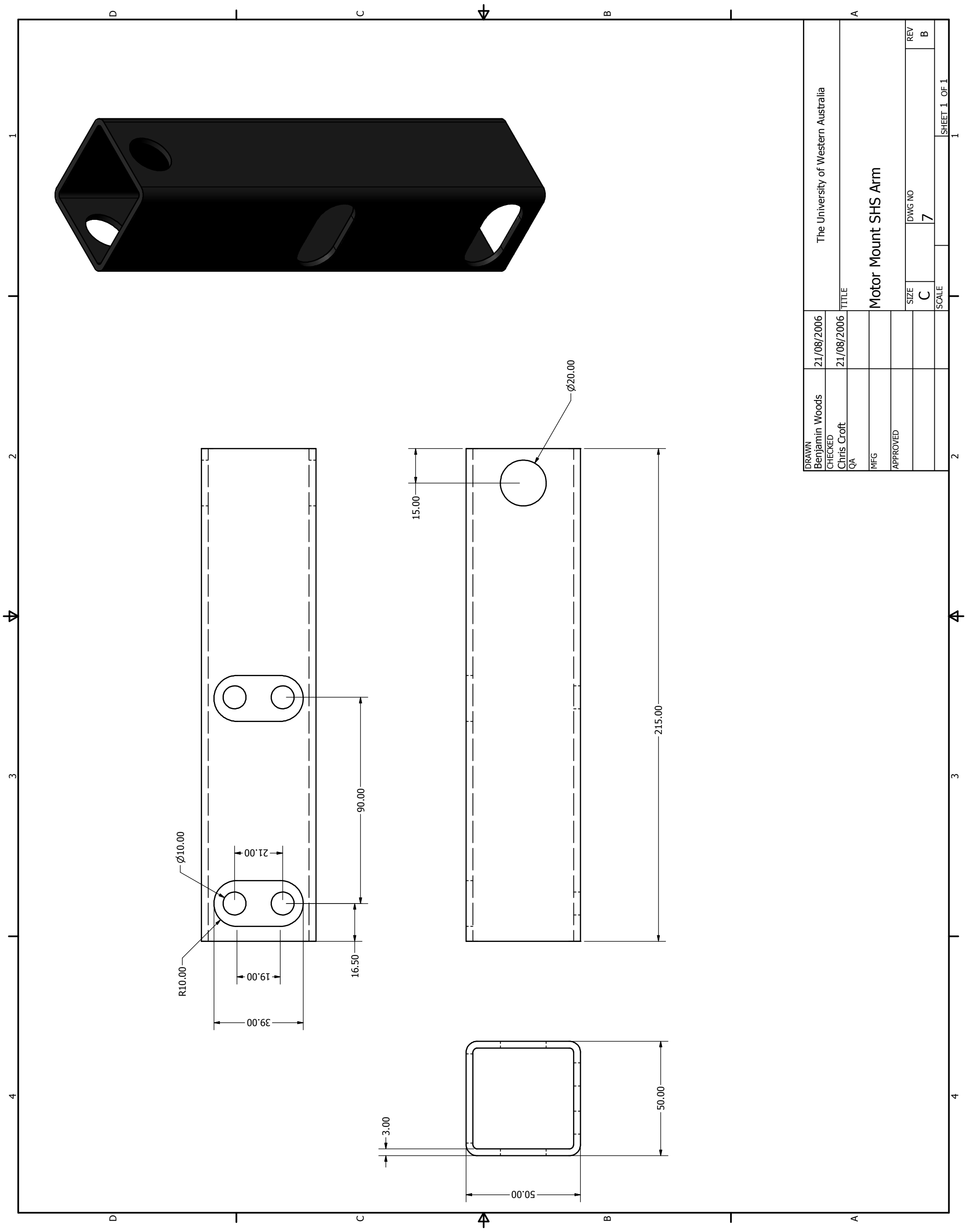

The University of Western Australia
TITLE
Battery Box
DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
SIZE
C
DWG NO
2
REV
B
SCALE
SHEET 1 OF 1
224.00
560.00
494.00
290.00

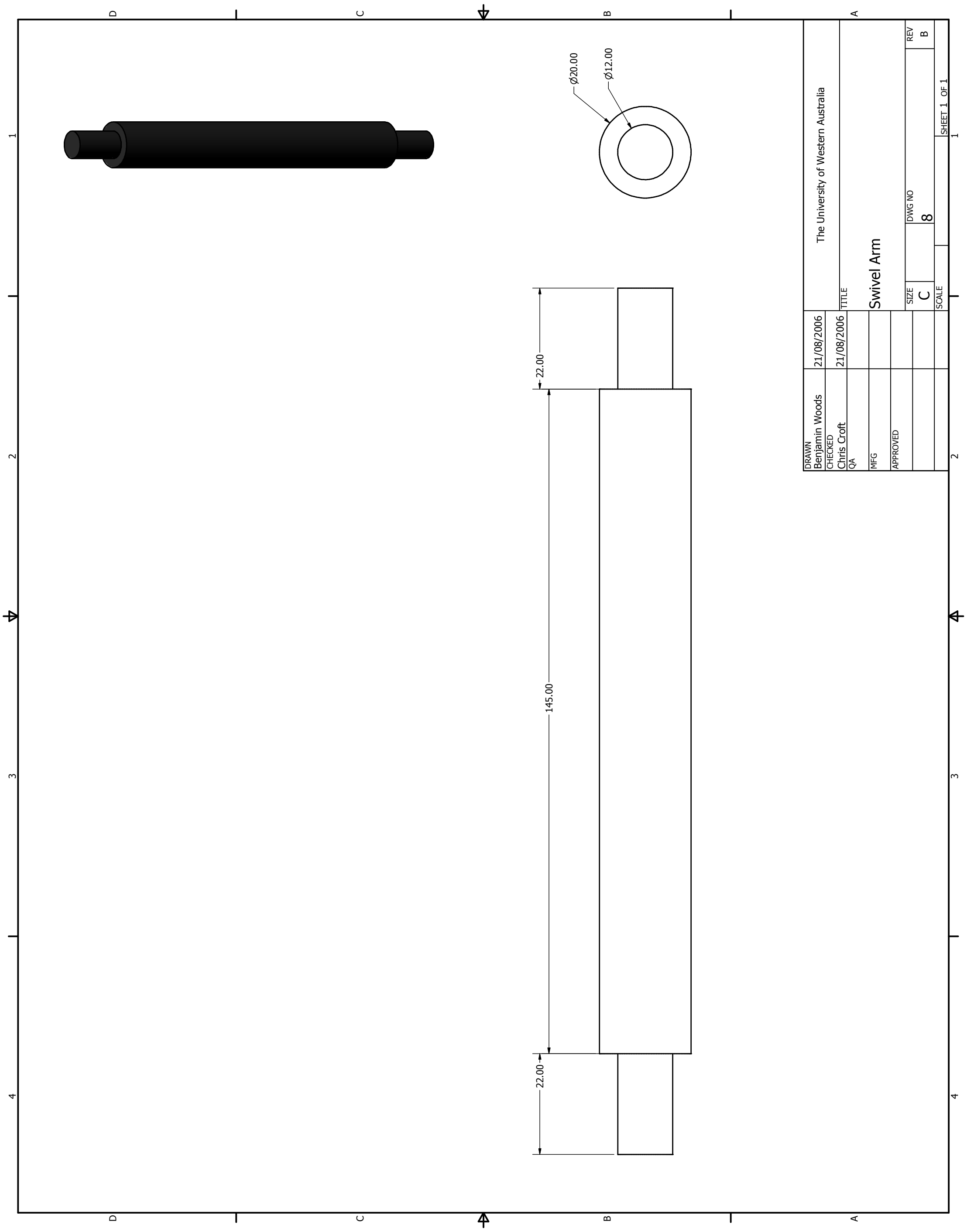

The University of Western Australia
TITLE
Battery Box Base
DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
SIZE
C
DWG NO
3
REV
B
SCALE
SHEET 1 OF 1
500.00
440.00
172.00
172.00
224.00
170.00
230.00

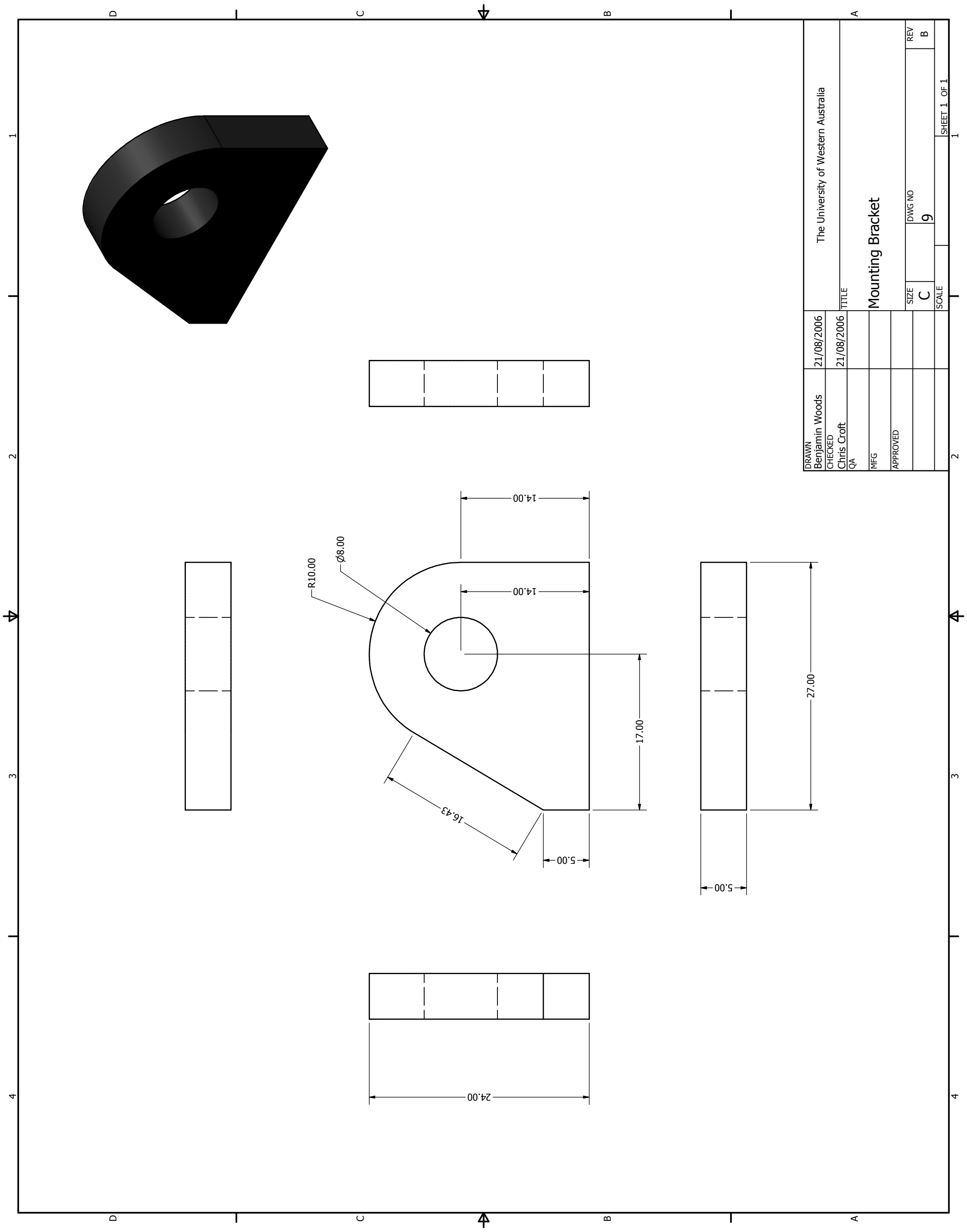

The University of Western Australia
DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
TITLE
Battery Box Side
SIZE
C
DWG NO
4
REV
B
SCALE
SHEET 1 OF 1
3.00
204.00
500.00
3.00
204.00
30.00
180.00
30.00

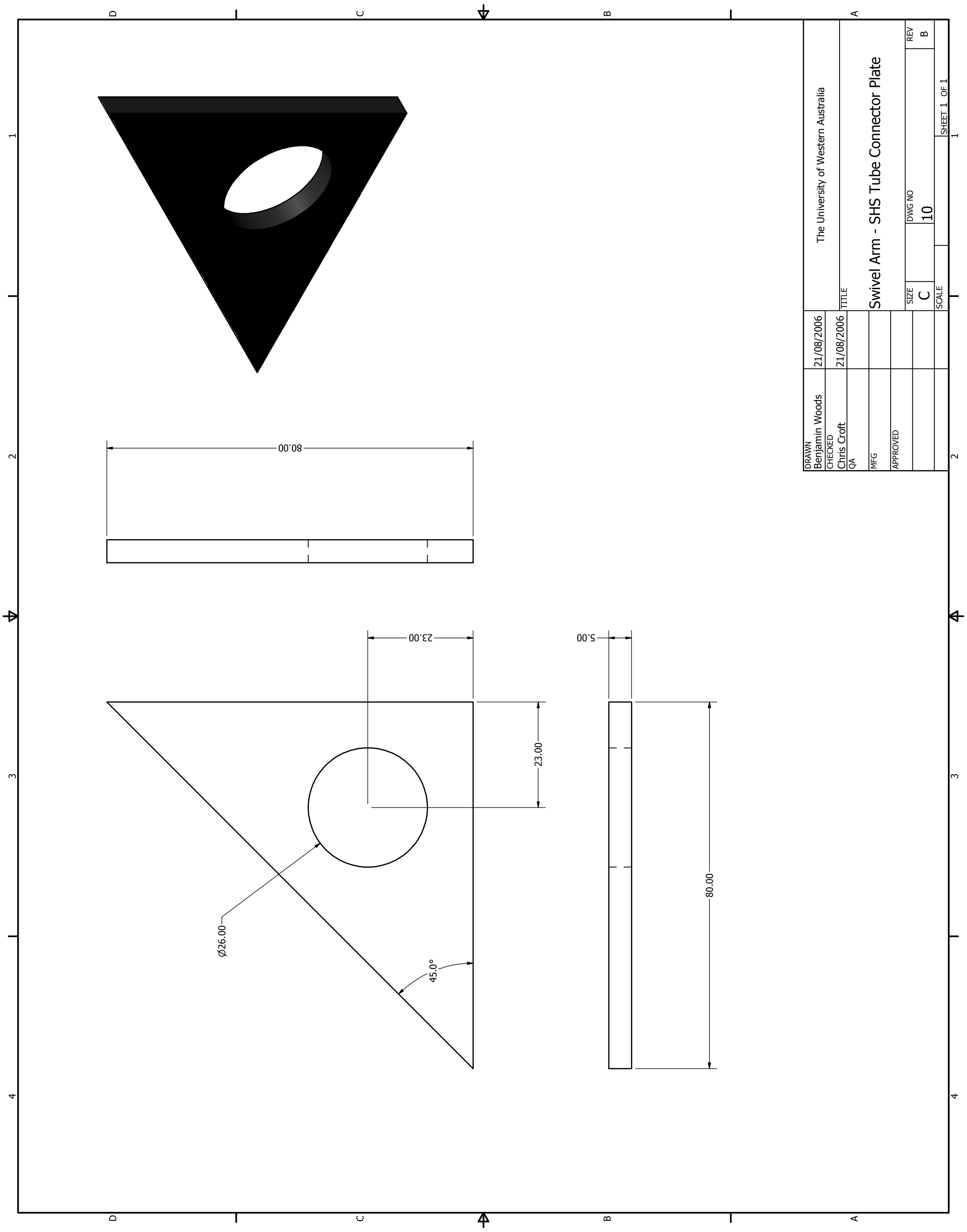

The University of Western Australia
DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
TITLE
Battery Box Top
SIZE
C
DWG NO
5
REV
B
SCALE
SHEET 1 OF 1
30.00
500.00
290.00

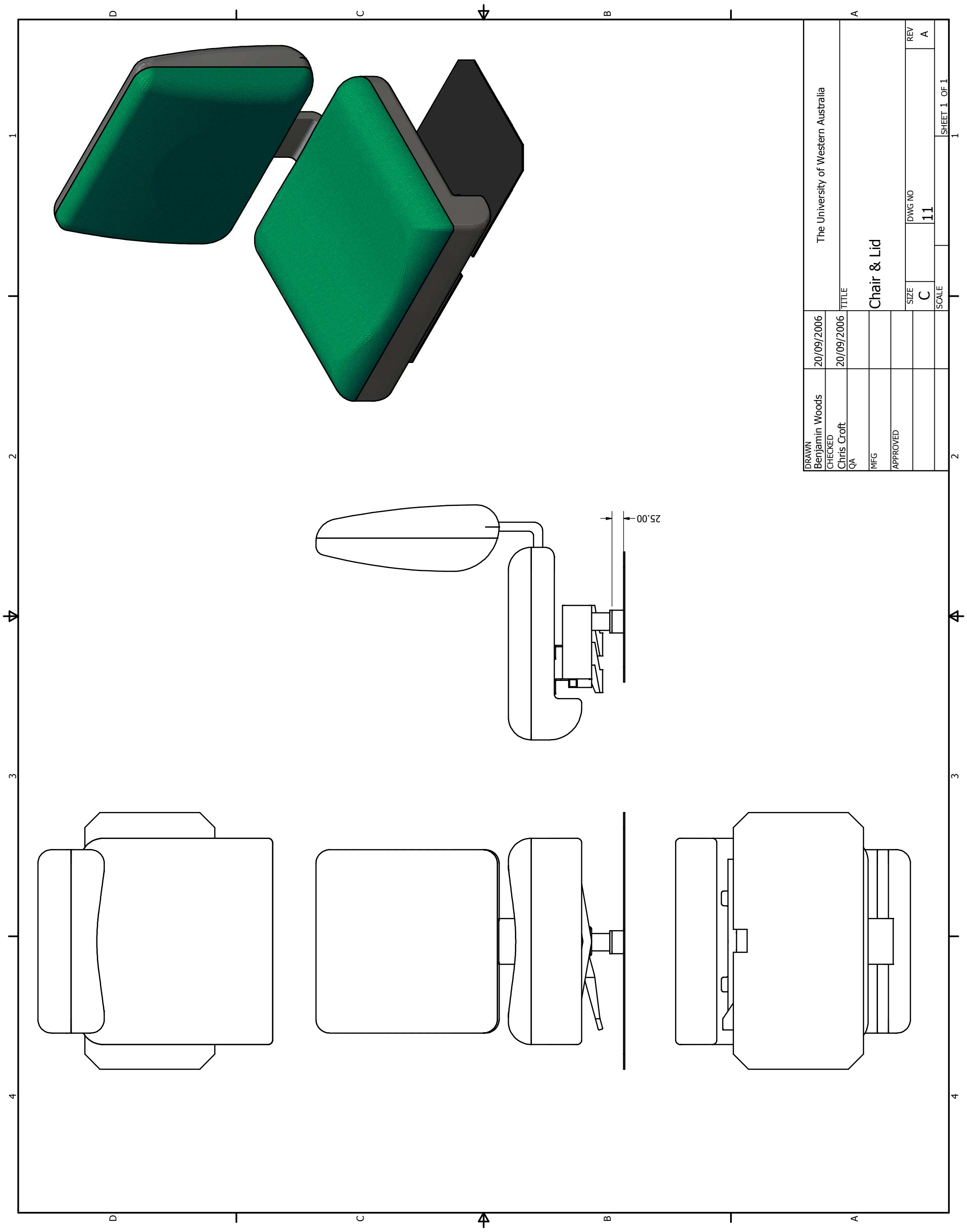

The University of Western Australia
Motor Mounts
DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
TITLE
SIZE
C
DWG NO
6
REV
B
SCALE
SHEET 1 OF 1
70.00
10.00
10.00
24.00

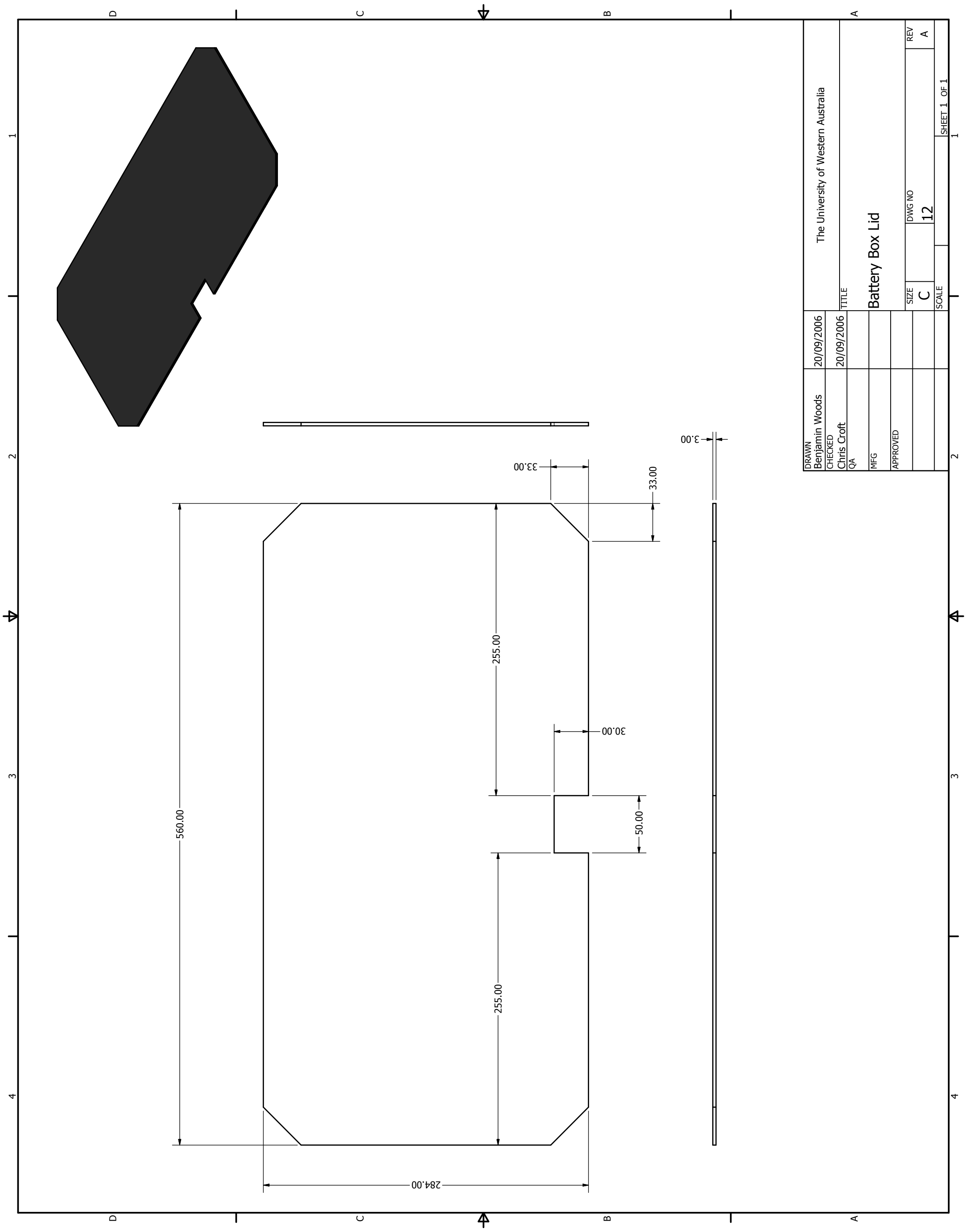

DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
The University of Western Australia
TITLE
Motor Mount SHS Arm
SIZE
C
DWG NO
7
REV
B
SCALE
SHEET 1 OF 1
Ø20.00
15.00
215.00
Ø10.00
21.00
90.00
R10.00
19.00
16.50
39.00
3.00
50.00
50.00

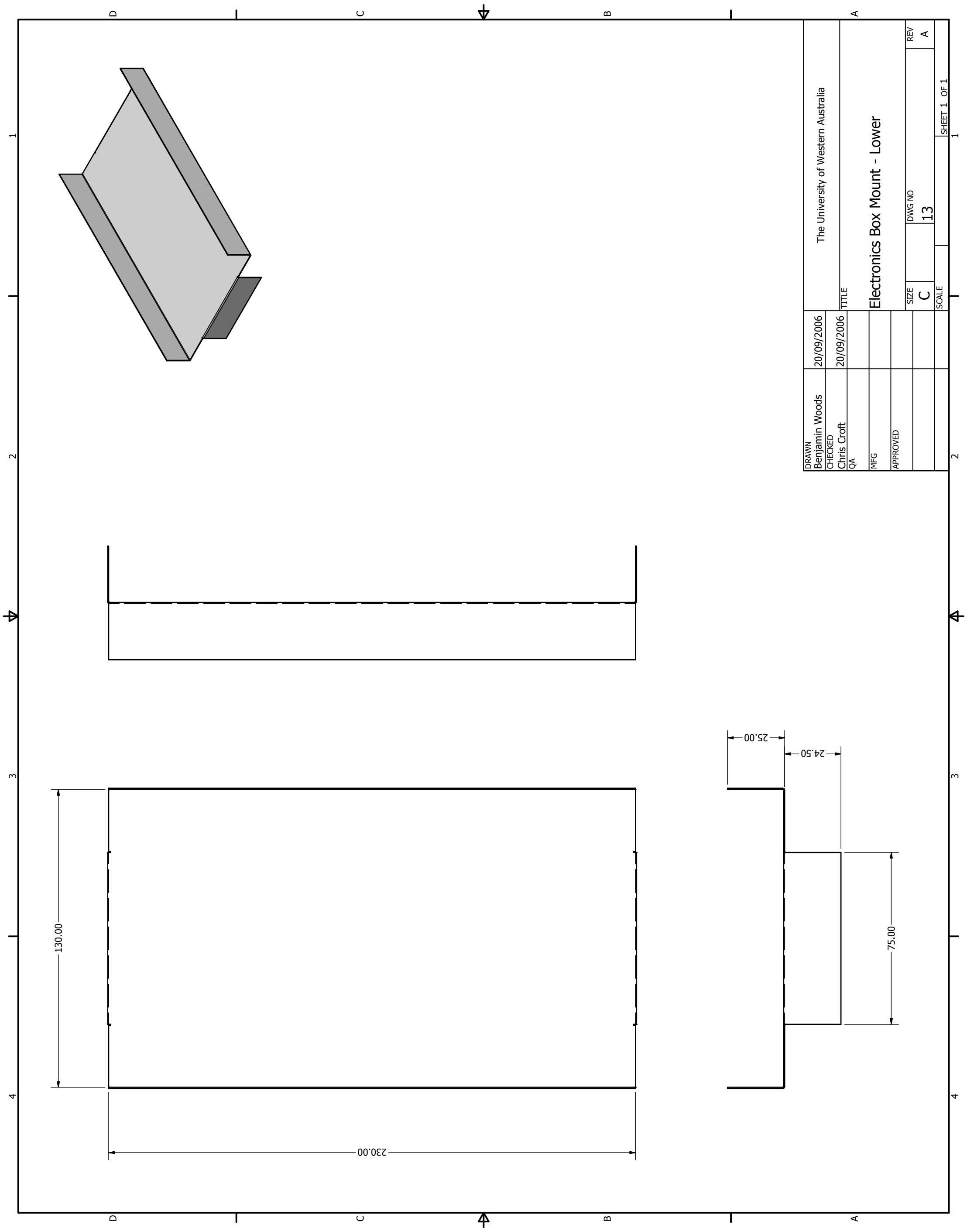

Ø20.00
Ø12.00
22.00
145.00
22.00
DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
The University of Western Australia
TITLE
Swivel Arm
SIZE
C
DWG NO
8
REV
B
SCALE
SHEET 1 OF 1

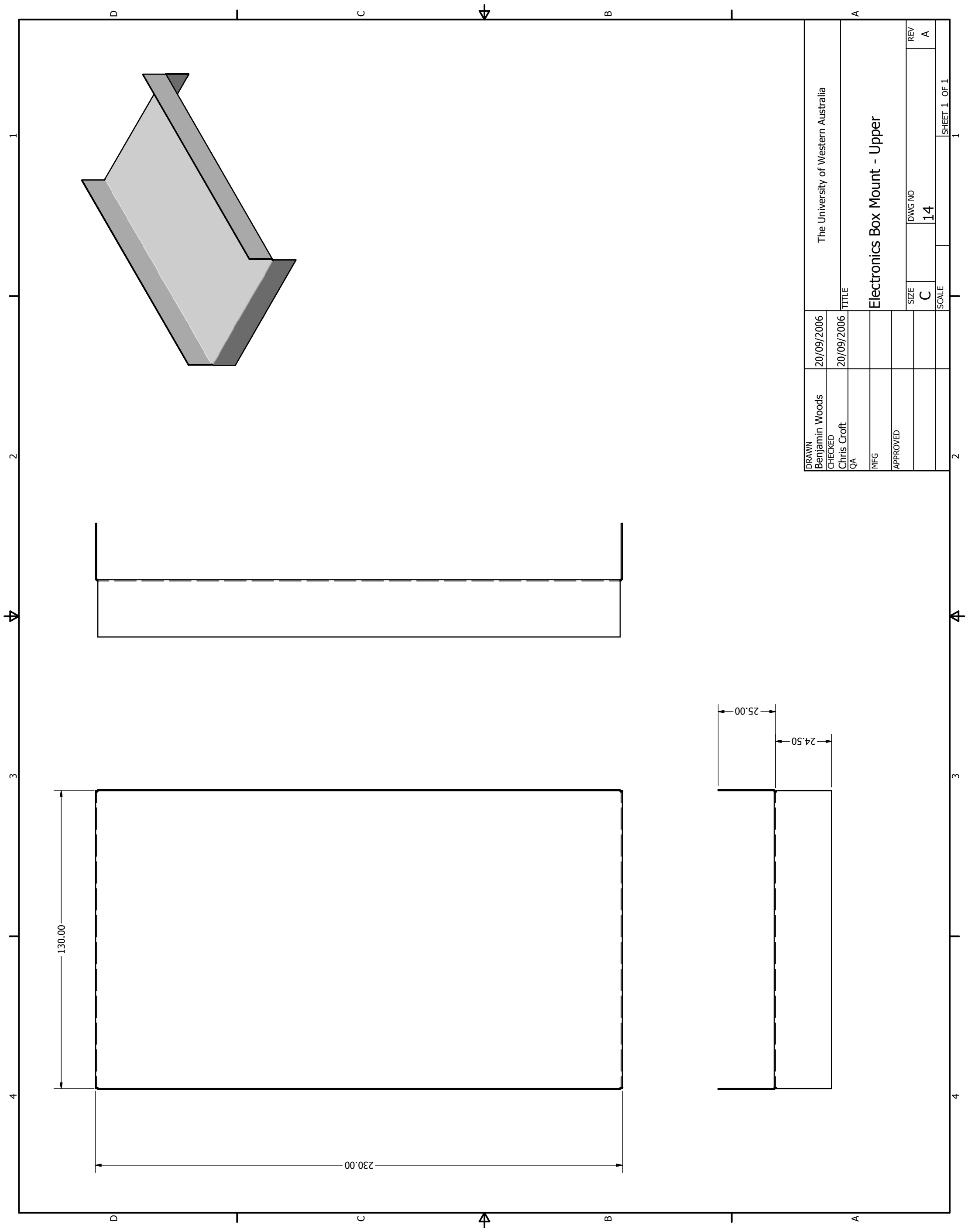

The University of Western Australia
TITLE
Mounting Bracket
DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
SIZE
C
DWG NO
9
REV
B
SCALE
SHEET 1 OF 1
14.00
14.00
R10.00
Ø8.00
17.00
27.00
16.43
5.00
5.00
24.00


The University of Western Australia
Swivel Arm - SHS Tube Connector Plate
DRAWN
Benjamin Woods
21/08/2006
CHECKED
Chris Croft
21/08/2006
QA
MFG
APPROVED
TITLE
SIZE
C
DWG NO
10
REV
B
SCALE
SHEET 1 OF 1
80.00
23.00
5.00
23.00
80.00
Ø26.00
45.0°


25.00
DRAWN
Benjamin Woods
20/09/2006
CHECKED
Chris Croft
20/09/2006
QA
MFG
APPROVED
The University of Western Australia
TITLE
Chair & Lid
SIZE
C
DWG NO
11
REV
A
SCALE
SHEET 1 OF 1


The University of Western Australia
TITLE
Battery Box Lid
DRAWN
Benjamin Woods
20/09/2006
CHECKED
Chris Croft
20/09/2006
QA
MFG
APPROVED
SIZE
C
DWG NO
12
REV
A
SCALE
SHEET 1 OF 1
560.00
284.00
255.00
255.00
50.00
30.00
33.00
33.00
3.00


The University of Western Australia
TITLE
Electronics Box Mount - Lower
DRAWN
Benjamin Woods
20/09/2006
CHECKED
Chris Croft
20/09/2006
QA
MFG
APPROVED
SIZE
C
DWG NO
13
REV
A
SCALE
SHEET 1 OF 1
130.00
230.00
25.00
24.50
75.00


The University of Western Australia
TITLE
Electronics Box Mount - Upper
DRAWN
Benjamin Woods
20/09/2006
CHECKED
Chris Croft
20/09/2006
QA
MFG
APPROVED
SIZE
C
DWG NO
14
REV
A
SCALE
SHEET 1 OF 1
130.00
230.00
25.00
24.50

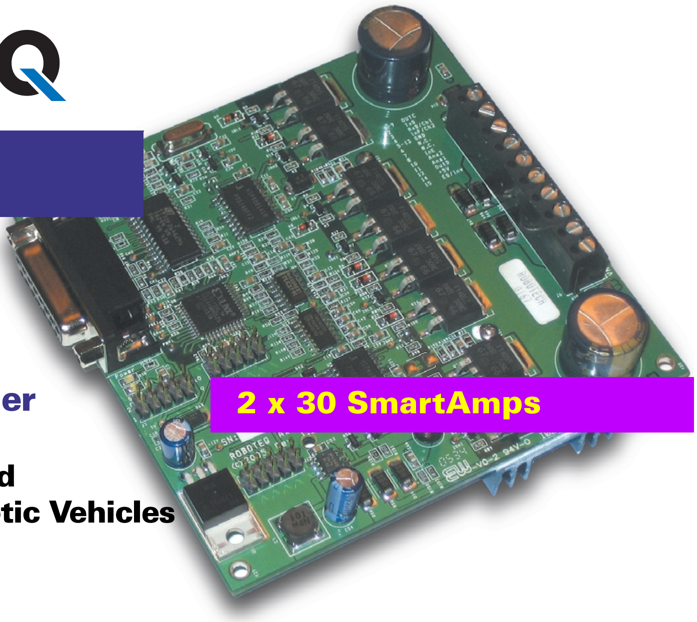
Appendix C

Information and Brochures

C.1 Roboteq AX1500

RoboteQ

AX1500



Dual Channel Forward/Reverse Digital Robot Controller

for Computer Guided and Remote Controlled Robotic Vehicles

Roboteq's AX1500 controller is designed to convert commands received from a R/C radio, Analog Joystick, wireless modem, or microcomputer into high voltage and high current output for driving one or two DC motors. Designed for maximal ease-of-useby professionals and hobbyist alike, it is delivered with all necessary cables and hardware and is ready to use in minutes.

The controller's two channels can either be operated independently or mixed to set the direction and rotation of a vehicle by coordinating the motion on each side of the vehicle. The motors may be operated in open or closed loop speed mode. Using low-cost position sensors, they may also be set to operate as heavy-duty position servos.

The AX2850 version is equipped with quadrature optical encoders inputs for precision speed or position operation.

Numerous safety features are incorporated into the controller to ensure reliable and safe operation.

The controller can be reprogrammed in the field with the latest features by downloading new operating software from Roboteq.

Applications

- Light duty robots
- Terrestrial and Underwater Robotic Vehicles
- Automatic Guided Vehicles
- Electric vehicles
- Police and Military Robots
- Hazardous Material Handling Robots
- Telepresence Systems

Key Features	Benefits
Microprocessor digital design	Accurate, reliable, and fully programmable operation. Advanced algorithms
R/C mode support	Connects directly to simple, low cost R/C radios
RS232 Serial mode support	Connects directly to computers for autonomous operation or to wireless modem for two-way remote control
Analog mode support	Connects directly to analog joystick
Header for Optional Optical encoder	Stable speed regardless of load. Accurate measurement of travelled distance
Built-in power drivers for two motors	Supports all common robot drive methods
Up to 30A output per channel	Suitable for a wide range of motors
Programmable current limitation	Protects controller, motors, wiring and battery.
Open loop or closed loop speed control	Low cost or higher accuracy speed control
Closed loop position control	Create low cost, ultra-high torque jumbo servos
Data Logging Output	Capture operating parameters in PC for analysis
Built-in DC/DC converter	Operates from a single 12V-40V battery
Compact Board Level Design	Lightweight and easy to incorporate in most applications
Field upgradable software	Never obsolete. Add features via the internet

Technical Features

Microcomputer-based Digital Design

- Multiple operating modes
- Fully configurable using a connection to a PC
- Non-volatile storage of user configurable settings. No jumpers needed
- Simple operation
- Software upgradable with new features

Multiple Command Modes

- Serial port (RS-232) input
- Radio-Control Pulse-Width input
- 0-5V Analog Voltage input

Multiple Motor Control modes

- Independent channel operation
- Mixed control (sum and difference) for tank-like steering
- Open Loop or Closed Loop Speed mode
- Position control mode for building high power position servos
- Modes can be set independently for each channel

Optical Encoder Inputs (option)

- Two Quadrature Optical Encoders inputs
- 250kHz max. frequency per channel
- 32-bit up-down counters
- Inputs may be shared with four optional limit switches

Automatic Command Corrections

- Joystick min, max and center values
- Selectable deadband width
- Selectable exponentiation factors for each command inputs
- 3rd R/C channel input for accessory output activation

Special Function Inputs/Outputs

- 2 Analog inputs. Used as
  - Tachometer inputs for closed loop speed control
  - Potentiometer input for position (servo mode)
  - External temperature sensor inputs
  - User defined purpose (RS232 mode only)
- One Switch input configurable as
  - Emergency stop command
  - Reversing commands when running vehicle inverted
- Up to 2 general purpose outputs for accessories or weapon
  - One 24V, 2A output
  - One low-level digital output
- Up to 2 digital input signals

Built-in Sensors

- Voltage sensor for monitoring the main 12 to 40V battery
- Voltage monitoring of internal 12V
- Temperature sensors near each Power Transistor bridge

Advanced Data Logging Capabilities

- 12 internal parameters, including battery voltage, captured R/C command, temperature and Amps accessible via RS232 port
- Data may be logged in a PC or microcomputer
- Data Logging Software supplied for PC

Low Power Consumption

- On board DC/DC converter for single 12 to 40V battery system operation
- Optional 12V backup power input for powering safely the controller if the main motor batteries are discharged
- 100mA at 12V or 50mA at 24V idle current consumption
- Power Control input for turning On or Off the controller from external microcomputer or switch
- No consumption by output stage when motors stopped
- Regulated 5V output for powering R/C radio. Eliminates the need for separate R/C battery.

High Efficiency Motor Power Outputs

- Two independent power output stages
- Dual H bridge for full forward/reverse operation
- Ultra-efficient 5 mOhm ON resistance MOSFETs
- Four quadrant operation. Supports regeneration
- 12 to 40 V operation
- User programmable current limit up to 30A
- Standard Fast-on connectors for power supply and motors
- 16 kHz Pulse Width Modulation (PWM) output

Advanced Safety Features

- Safe power on mode
- Optical isolation on R/C control inputs
- Automatic Power stage off in case of electrically or software induced program failure
- Overvoltage and Undervoltage protection
- Watchdog for automatic motor shutdown in case of command loss (R/C and RS232 modes)
- Run/failure diagnostics on visible LEDs
- Programmable motors acceleration
- Built-in controller overheat sensors
- "Dead-man" switch input
- Emergency Stop input signal and button

Compact Design

- All-in-one board-level design.
- Efficient heat sinking. Operates without a fan in most applications.
- 4.25" (108mm) L, 4.25" W (108mm), 1" (25mm) H
- -20o to +70o C operating environment
- 3oz (85g)

Ordering Information

Model	Description
AX1500	Dual Channel DC Motor controller up to 30 SmartAmps per channel

RoboteQ
www.roboteq.com

8180 E.Del Plomo Dr.
Scottsdale, AZ 85258 - USA
602-617-3931