THE UNIVERSITY OF QUEENSLAND

Joint Controller Development For A Humanoid Robot

Simon Hall

October 2004

Simon Christopher Hall
188A Stanley Rd
CARINA QLD 4152

Professor Paul Bailes
Head of School of Information Technology and Electrical Engineering(Acting)
The University of Queensland
ST LUCIA QLD 4072

October 27, 2004

Dear Professor Bailes,

In accordance with the requirements of the Degree of Bachelor of Engineering (Mechatronic), I present the following thesis entitled, "Joint Controller Development for a Humanoid Robot". This work was performed under the supervision of Dr Gordon Wyeth.

I declare that the work submitted in this thesis is my own, except were acknowledged otherwise, and has not been previously submitted for a degree at any tertiary institution.

Yours Sincerely,

Simon Christopher Hall

Acknowledgements

The following people deserve a word of thanks for their assistance provided during the course of this project:

GuRoo, for his company during the many lonely late nights on level five in Axon building.

Damien Kee, for his invaluable assistance provided throughout the year, and for his ability to be a super absorbent whinge sponge.

Gordon Wyeth, for his supervision and assistance with technical issues.

Geoff Walker, for his valuable assistance with power circuitry diagnostics.

Keith, Dennis and Barry from the Electronics Workshop, for their assistance provided with soldering techniques and equipment use.

My fellow Undergrads, for making final year engineering seem a little bit more as though you actually have a life!

Lucas and Kelly, for all the much appreciated cooked dinners left in the fridge, and for doing all the grocery shopping.

My family and friends, for their much appreciated support throughout the year.

Abstract

The focus of this thesis is the development of DC motor controller boards for UQ's humanoid robot, "GuRoo". With the project now in its fourth year of development, the original lower limb DC motor controller boards were in need of an upgrade. A new hardware design was finalised in late 2003. Over the course of 2004, the progressive construction and testing of these new boards has brought them from blank PCB's to near fully operational status.

The new design incorporates a Motorola 68376 DSP superior to the Texas Instruments TMS320F243 processor controlling the old boards. This thesis details the progressive programming and testing of the new boards performed to achieve near full functionality. Software was developed to utilise pulse width modulation(PWM) and quadrature decoding capabilities of the 68376's TPU(Timer Processor Unit). Code was also developed to make use of the new processor's TouCAN(Controller Area Network) and QADC(Queued Analog to Digital Converter) modules. After verifying correct operation of these functionalities, the existing software of the original 2001 boards was updated to suit the new design.

An initialisation routine was developed for GuRoo's joint positioning on power-up. This was achieved through position control of GuRoo's motors about encoder index pulses. CAN communication enables the robots joints to be incremented between successive index positions.

Software was developed to demonstrate the new boards functionality, including the index positioning initialisation. This demonstration software proved that the control loop speed can now be drastically increased.

Through harnessing the new board's superior capabilities, further work following this thesis will enable GuRoo's joint control performance to be enhanced.

Table of Contents

List of Figures and Tables

1. Introduction

Research and development of humanoid robot technology is more than just a novel idea. Robots might not be widespread in today's society, but where they do exist their presence can be very beneficial, e.g., rescue robots performing tasks in environments that may be hazardous to human life. Reliable manoeuvrability and interaction of a robot within its environment is a common problem for any robotics project. Humans have contributed to this problem by adapting the world around us to suit our own geometries and capabilities. It therefore makes sense that the more human like a robot is, the easier it will be able to interact with us and our environment. Hence the reason for humanoid development is well justified.

The University of Queensland's humanoid robot, GuRoo, has been an ongoing project since 2001. Weighing in at 38kg, this 1.2m tall humanoid is fully autonomous, with the ability to crouch, stand on one leg and walk unaided at a speed of 0.1m/s. Now in its fourth year of development, this project involves the efforts of a large team of undergraduate and postgraduate students and staff of UQ. Joint control in particular, has proved to be a major focus of GuRoo's development.

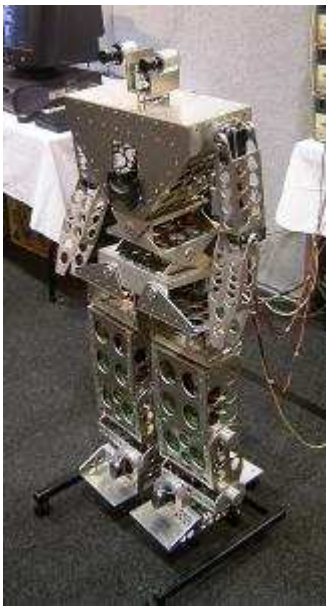


Figure 1.1: GuRoo The Humanoid Robot

GuRoo's original lower limb joint design was constructed in 2001. Since then, a new superior design has been developed. The topic of this thesis is the development and testing of this new design.

1.1 RoboCup

The immediate purpose of GuRoo is to participate in the International Robocup Competition, an ongoing quest to develop soccer playing humanoid robots by the year 2050.

Figure 1.2: Robocup. Humanoid Soccer - A Goal for 2050. [8]

RoboCup officially began in 1997, with the humanoid league of the competition commencing in 2002. The humanoid competition has included challenges such as walking, balancing, standing on one leg and freestyle. GuRoo competed in 2002 with respectable success. He obtained 7th place in the walking competition and the freestyle competition, and was the best competitor in the "stand on one leg" event [6]. Active joint control played a major part in GuRoo's success.

1.2 Thesis Goals and Overview

The primary goal of this thesis was to develop GuRoo's new 2004 lower limb DC motor controller boards to an operational status, sufficient for installation into GuRoo's existing system. Achieving this goal required building the new boards from blank PCB's whilst simultaneously developing software for the new board's Motorola 68376 DSP. Unfortunately, due to time constraints this goal has not been completely achieved. At present the boards are at a stage where they are almost ready for installation. A few more hardware and software issues require finalisation before installation may take place. Details of these minor issues will be discussed throughout this document.

A secondary goal for this thesis was to develop software for these new boards that would initialise each joint on power-up of the robot. Prior to this thesis, an initialisation process was non-existent. It was requested that initialisation purely involve the positioning of joints to encoder index pulses. A successful initialisation

routine has been developed that uses CAN to command each joint to increment to encoder index pulse positions.

This thesis has also intended to serve as a document which collates details of the new design and its features, and the design decisions that were made in order to justify its existence.

Chapter Two briefly outlines the history of GuRoo's joint control and provides a short description of the operation of the lower limb joint control. This is followed by a brief description of the new 2004 lower limb controller board design.

Chapter Three discusses the board development process required to achieve the required functionality.

Chapter Four Details the new software developed. This includes the updates that have been required for the existing software to suit the 68376, and the implementation of the new initialisation routine.

Chapter Five concludes this document, and discusses the outcomes of this thesis.

Chapter Six gives a brief outline of the further work that is required before the boards can be installed into GuRoo. It also discusses ideas for further development of the new joint controllers.

2. GuRoo's Joint Control

2.1 A History of GuRoo's Joint Control Development

The past three years have seen a major contribution of work towards developing GuRoo's joint control. The past theses that have directly influenced GuRoo's 2004 joint controller design are listed in Table 2.1.

Thesis Title	Author	Year Completed
Design and Implementation of a Distributed Digital Control System in an Industrial Robot.	James Kennedy	1999
Design of DC Motor Controllers for a Humanoid Robot	Jarad Stirzaker	2001
Design & Implementation of Small Scale Joint Controllers for a Humanoid Robot	Tim Cartwright	2001
Distributed Motion Controllers for a Humanoid Robot	Andrew Hood	2002
Gait Generation & Control Algorithms for a Humanoid Robot	Adam Drury	2002
Mobile Robot Electrical Design	Doug Turk	2003

Table 2.1: Past Theses Relating to GuRoo's Joint Control.

In 1999, James Kennedy[12] designed hardware and software for a distributed control system for actuation of a Kawasaki PUMA 560 industrial robot arm as shown in Figure 2.1. Kennedy's distributed control system consisted of a network of DC motor control boards distributed throughout the robot, with each board situated as close as possible to the motors it controlled. The joint control design that existed in the PUMA arm prior to his design utilised complex control electronics, with a bulky unreliable cable driving each joint from a central controller. Kennedy's design resulted in a simplified, more reliable joint control system, with a major reduction in the complexity and cost of the wiring harness. In many ways, Kennedy's design became

the foundation for the control system that exists in GuRoo and the subsequent new 2004 design.

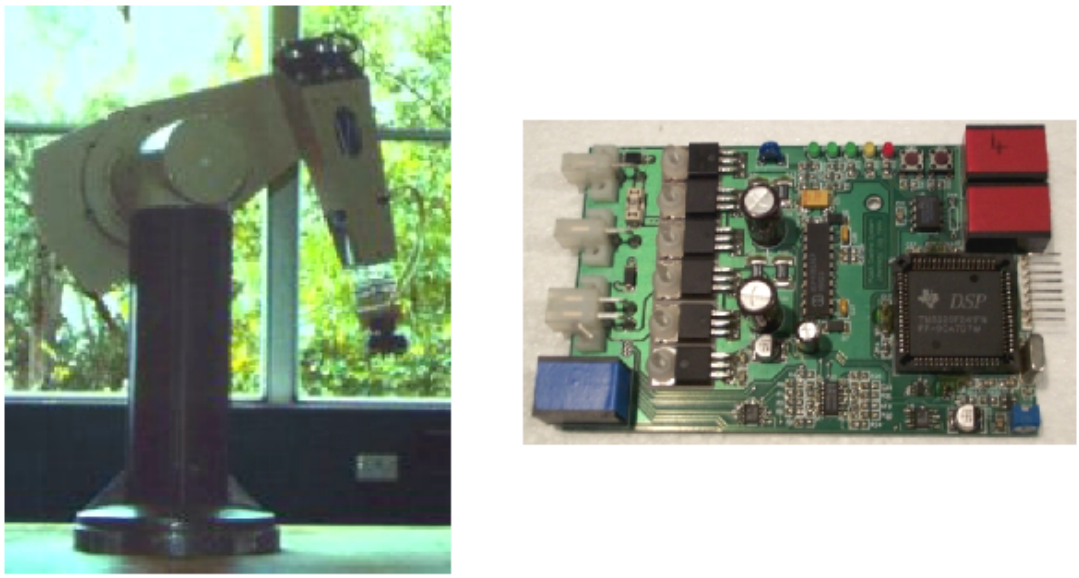


Figure 2.1: The PUMA Arm(left) & Kennedy's Distributed Controller Design(right). [4]

GuRoo's original and existing joint controller hardware was designed and developed by Stirzaker[4] and Cartwright[15] in 2001. Stirzaker's thesis focused on the design of the lower limb controller boards, as shown in Figure 2.2, for the control and actuation of GuRoo's fifteen lower limb Maxon DC motors. Cartwright's thesis focused on the development of an upper limb control board which actuates the eight light-weight servo motors in GuRoo's upper body. There were many consistencies between Stirzaker and Cartwright's designs. Stirzaker's lower limb design was then made fully operational by Hood[3] and Drury[6] in 2002. Hood's work focused on the development of low level software for control of the lower limb boards while Drury developed software for the PI velocity control algorithm and velocity profiles for gait patterns.

Hood also performed a complete review of the existing lower limb controller boards, and made specifications for a new board design. Finally, in Nov 2003, Kee[9] designed and manufactured the joint controller PCB used in this thesis. This finalised design incorporated the same microcontroller and memory configuration that is to be

incorporated into Turk's[13] new electrical design for UQ's mobile soccer playing robot project, the RoboRoo's.

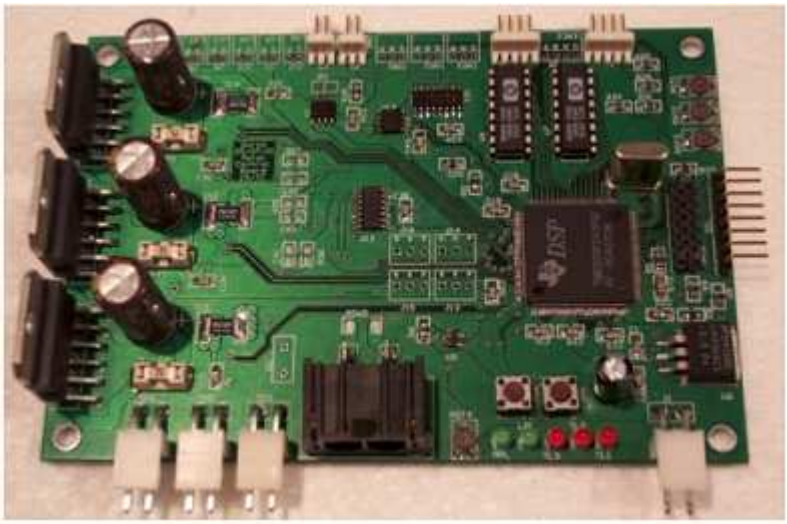


Figure 2.2: 2001 Lower Limb Motor Controller Board. [4]

2.2 GuRoo's Existing Joint Control

GuRoo's gait's are implemented by a distributed control system. Movement patterns are generated by an external computer. The external computer communicates with the various motor controller boards throughout its body. Each motor controller then processes the instructions it receives, to regulate each motor's speed such that GuRoo's body moves in a desired fashion.

2.2.1 Degrees of Freedom & Board Locations

Overall, there are currently 23 joints or "degrees of freedom" in GuRoo's body, as outlined in Figure 2.3. A DC motor actuates each joint. The neck, shoulder and elbow joints are actuated by eight, low power, low weight Hi-Tech HS705-MG RC servo motors. The lower limbs are actuated by more powerful, Maxon 70W RE32 brushed DC motors, fifteen in total. [17]

All 23 of these motors are controlled by six boards spread throughout GuRoo's chassis. Their locations are indicated in Figure 2.4. One motor controller board

(Board 6) is located on the back of the torso controlling all eight upper joints, while the lower limb controllers (Boards 1 to 5) are located as follows:

- One board in the stomach, controlling the three waist joints.
- One board in each thigh controlling the three respective hip joints.
- One board in each ankle controlling the respective knee and two ankle joints.

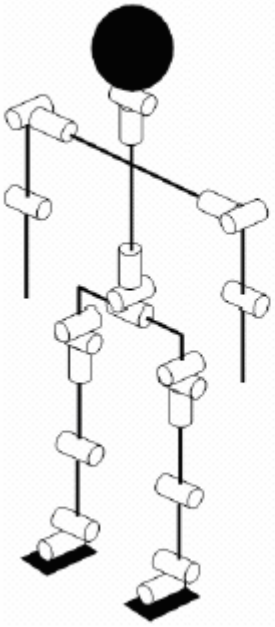


Figure 2.3: The 23 Degrees of Freedom in GuRoo's body. [1]

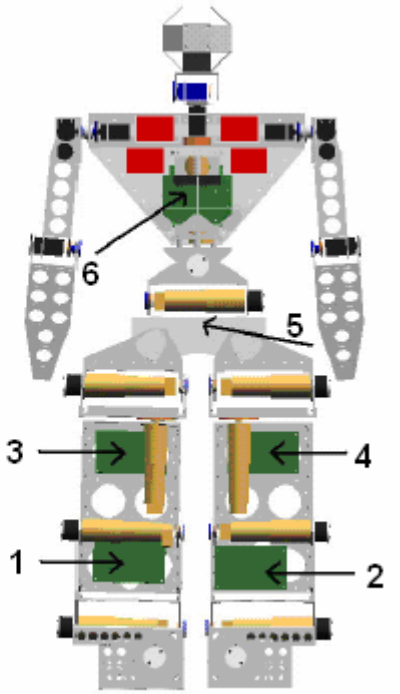


Figure 2.4: Board Placement in GuRoo's Body. [3]

Each lower limb motor controller board controls three motors. Desired joint velocities are transmitted through serial communication from an external computer to Board 6. The signals are then transmitted to each of the lower limb controllers through a Controller Area Network(CAN) serial communication protocol.

2.2.2 CAN Communication Network

CAN is a multi-master system with software identifiable nodes. Utilising a simple two-wire bus, the standard includes sophisticated error checking and a high bandwidth of up to 1Mbps [3]. Figure 2.5 outlines the flow of communication in GuRoo. When a message is sent through CAN, it is broadcast to all nodes, and software defineable message buffers at each node either accept or reject messages. Nodes are software programmable for message reception.

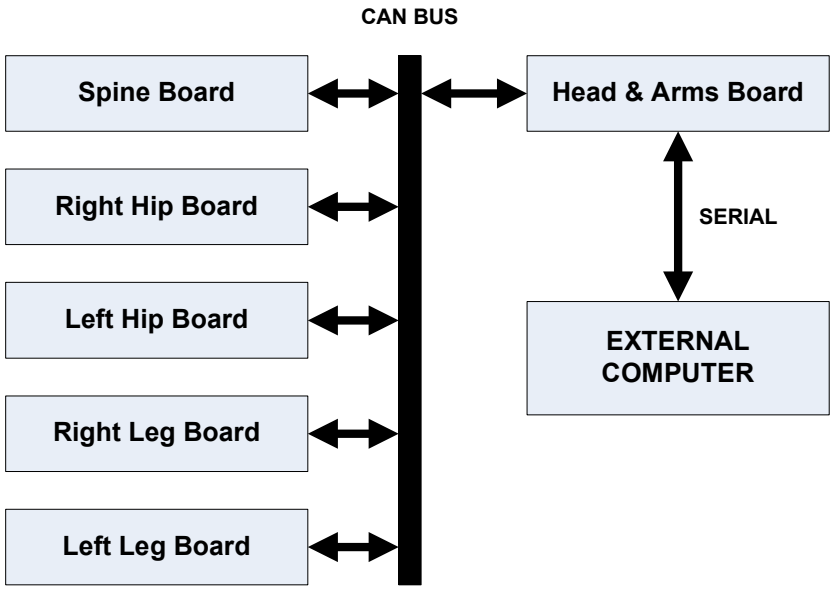


Figure 2.5: Flow of Control/Communication in GuRoo. [9]

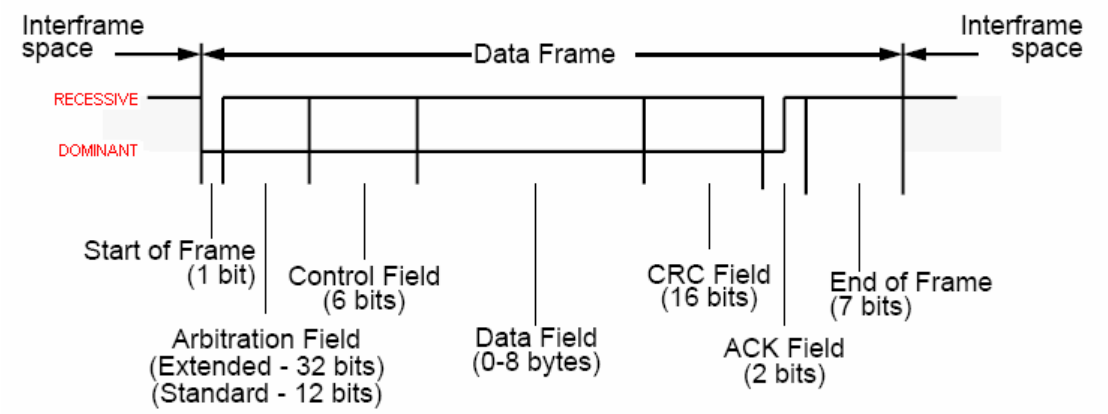


Figure 2.6: CAN Data Frame. [11]

Figure 2.6 outlines a CAN data frame. Each CAN frame can contain up to 8 bytes of data. The Arbitration(ID) field forms the first part of every message sent across a CAN network. Message prioritisation in the event of multiple nodes sending messages synchronously is given to the message with the lowest ID field. In GuRoo's communication network, each board has software defined CAN message buffers that will either accept or reject messages sent to them. This allows each board not only to receive the applicable incoming desired velocity settings but also to transmit performance information back to the external computer.

CAN makes use of a two wire bus, CANH (high) and CANL (low). The CAN bus can be placed in two states, dominant and recessive. In the dominant state both lines are driven to 2.5V, and when recessive, CANH is driven to 5V and CANL is driven to 0V. When a message is converted into CAN format, logical one bits are referred to as recessive bits and logical zero bits are referred to as dominant bits.

2.2.3 Motor Control Process

On reception of incoming desired motor velocity settings, the motor controllers perform all low level control. Less powerful processors are required and communication complexity is reduced through the local computation of motor control [12].

Figure 2.7 shows a block diagram of the lower limb motor controller board signal flow. Desired joint velocities arrive at each board at 50Hz from the CAN bus and are fed into the DSP message buffers. The DSP then uses these desired speed settings to run a PI velocity control loop at 250Hz. Motor drive is governed by pulse width modulation (PWM) of motor voltage. Feedback of motor current and motor position sensing allows for closed loop control of motor speeds.

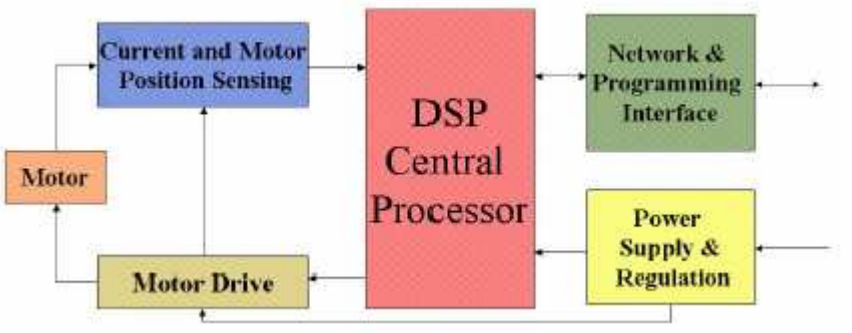


Figure 2.7: Block Diagram of Controller Board Components and Signal Flow. [3]

2.2.4 Pulse Width Modulation(PWM) – Bipolar and Unipolar Topologies

Velocity control is achieved through varying the voltage across the terminals of a motor. Pulse Width Modulation is the continuous fast switching of motor voltage. By varying the duty cycle from 0% to 100%, the effective voltage across a motor can be established from a set input voltage (V_{motor}).

Two basic topologies for the implementation of PWM are unipolar PWM and bipolar PWM. As described in Figure 2.8, bipolar PWM involves the switching of voltage between a positive and a negative set voltage(V_{motor}). Under this configuration a net positive voltage across the motor can be achieved for a positive duty cycle greater than 50%. This will drive the motor forward, provided that generated torque is greater than load torque. In the opposite case, if the net voltage across the motor is negative, the motor will be driven in reverse, provided that generated torque is greater than load torque. If the duty cycle is maintained at 50% the motor will remain stationary, provided there is no load torque applied.

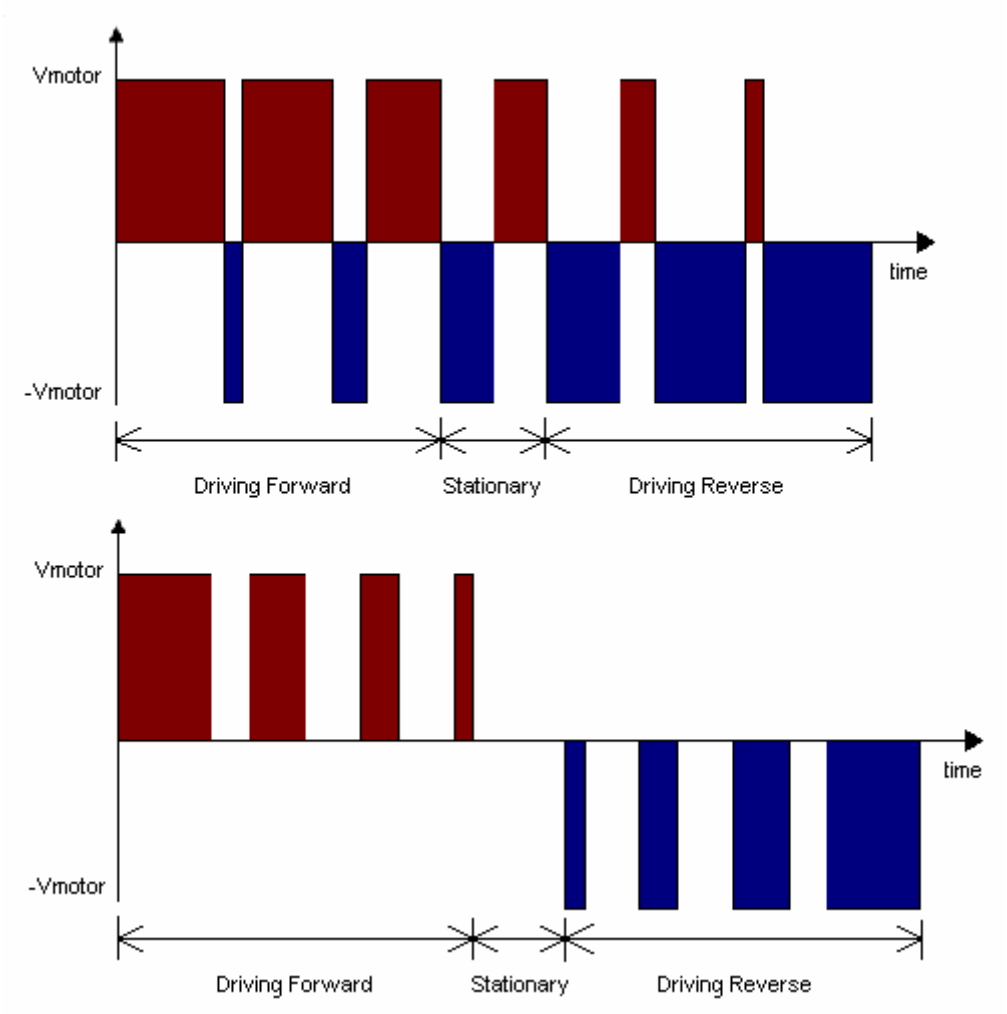


Figure 2.8: Bipolar PWM(above) and Unipolar PWM(below). [3]

Unipolar PWM involves only switching the voltage across the motor between 0 and V_{motor} or $-V_{motor}$ to achieve forward and reverse motion respectively. If a duty cycle of

0 is applied, the motor will remain stationary, again provided that the motor is unloaded.

The advantage of unipolar PWM is that ripple current can be significantly lower than that for bipolar switching. Ripple current is related to the inductance of the armature and the voltage ripple component of voltage across the motor's armature[16]:-

$$\mathbf{v_r \approx L_a(di_r/dt)}$$

where v_r = the voltage ripple component of armature voltage (V).
L_a = armature inductance (H).
i_r = the current ripple component of armature current (A).

Bipolar PWM effectively doubles the voltage swing across the motor terminals and drastically increases the voltage ripple component of armature voltage and consequently the ripple current. Power losses associated with this ripple current are[16]:-

$$\mathbf{P_{\Delta I} = R_a(I_r)^2}$$

where R_a = armature resistance (Ω).
I_r = RMS value of the current ripple component (A).

Observation of this I_r^2 relationship entails that power losses associated with ripple current can be significantly reduced by lowering the ripple current .

2.2.5 Motor Drive – The H-bridge

PWM is implemented through the use of an H-bridge. A simplified H-bridge configuration is shown in Figure 2.9. By varying switch states, the motor can be placed in the following states:

1) Driving forward (T1 on, T4 on)
2) Driving in reverse (T2 on, T3 on)
3) Braked to ground (T3 on, T4 on)
4) Braked to V_{motor} (T1 on, T2 on)
5) Neutral/ Floating (all switches off)

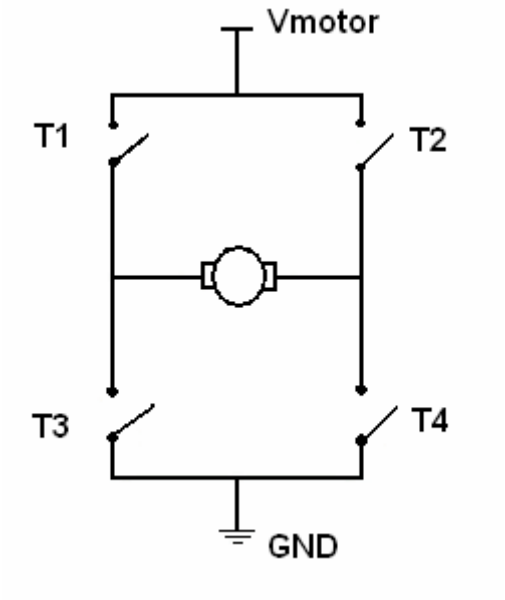


Figure 2.9: A Simplified H-bridge Configuration.

Bipolar switching only makes use of states 1 and 2. Unipolar switching involves states 1 and 2 and one of either 3 or 4. GuRoo's original control boards implemented bipolar PWM.

2.2.6 PI Velocity Control Implementation

PI, Proportional plus Integral velocity control is achieved through digital feedback from each motor, gearbox and encoder unit. As shown in Figure 2.10, input PWM duty cycles are fed to motor terminals proportional and integral to the error between the desired input velocity profile and the actual motor velocity. Motor velocities are expressed as functions of the difference in motor position per control loop. The integral component removes the steady state error in velocity control.

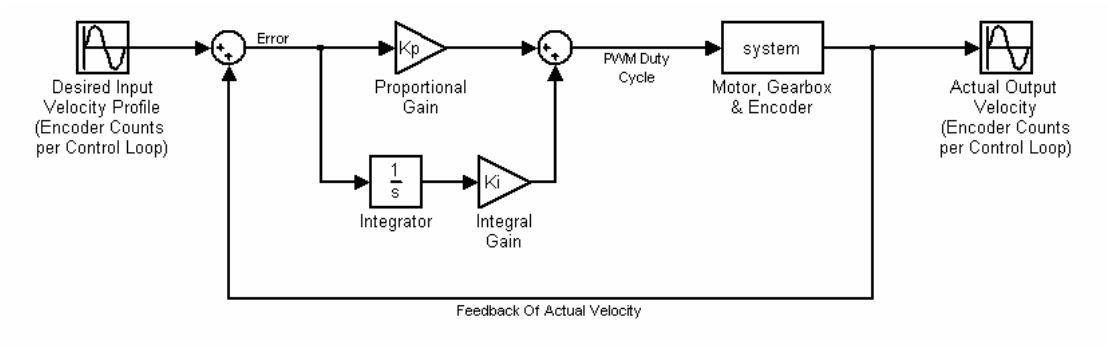


Figure 2.10: Implementation of PI Control in the Lower Limb Controllers.

2.2.7 Quadrature Decoding

Feedback of motor position is achieved through Quadrature Decoding of encoder position. Encoders are attached to each motor's rotating shaft. Quadrature decoding enables relative motor positions to be deciphered through the incrementing or decrementing of forward and reverse encoder counts respectively. This is achieved through the processing of two pulsating, 90 degree, out of phase waveforms(channels) output from each encoder. As described in Figure 2.11, if Channel A is leading, the motor is rotating clockwise and if Channel B is leading, the motor is rotating anti-clockwise.

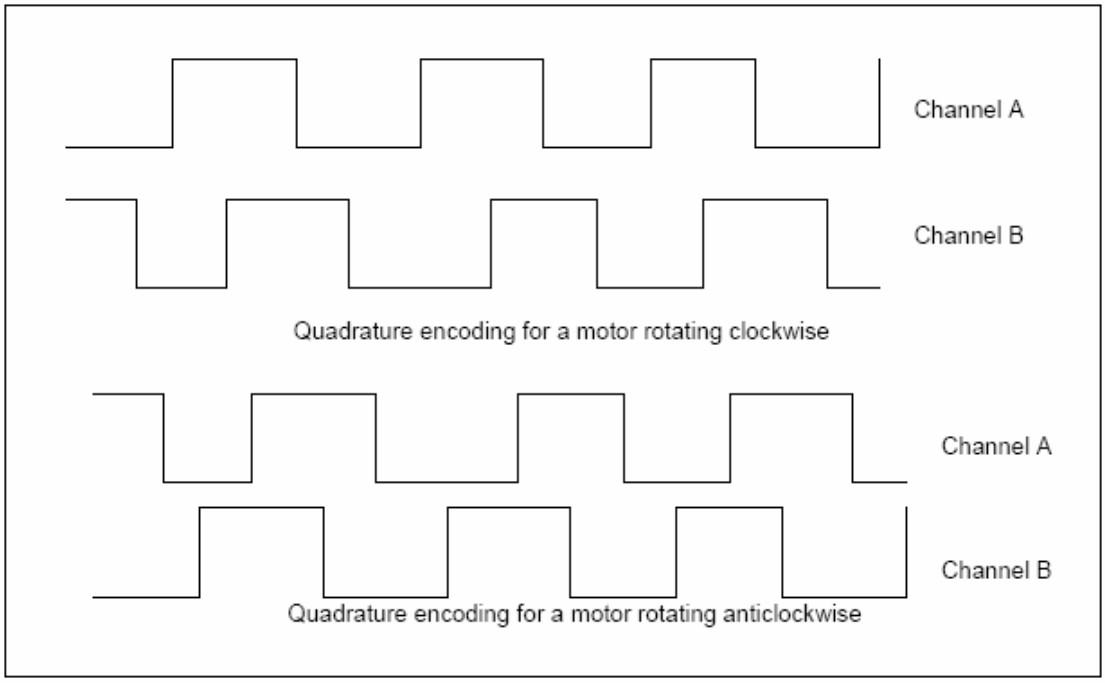


Figure 2.11: Encoder Waveforms for Forward and Reverse Position Feedback. [13]

The encoders on GuRoo's lower limbs are each accompanied by 500 pulse per revolution encoders. The existing system decodes for each channel on both rising and falling edges, to give a resolution of 2000 counts per revolution. This is then transferred through a 156:1 gearhead which gives a total theoretical resolution of 0.00115 degrees per revolution.

2.3 Existing Control Software for GuRoo's Lower Limb Boards

Operation of the existing joint controller code required closed loop control of joint velocities as outlined in Figure 2.12. All code is written in C++ which is compiled down to machine language and ported to the flash memory of the microcontrollers. The 2001 boards are each controlled by a Texas Instruments TMS320F243 16 bit DSP.

When GuRoo is first switched on, control of each motor is initiated by the `main()` function in startup.c. This source file initialises the DSP's appropriate register values and sets up a periodic interrupt that calls `b1_control()` in board1.c. `b1_control()` is the function that implements the PI velocity control algorithm.

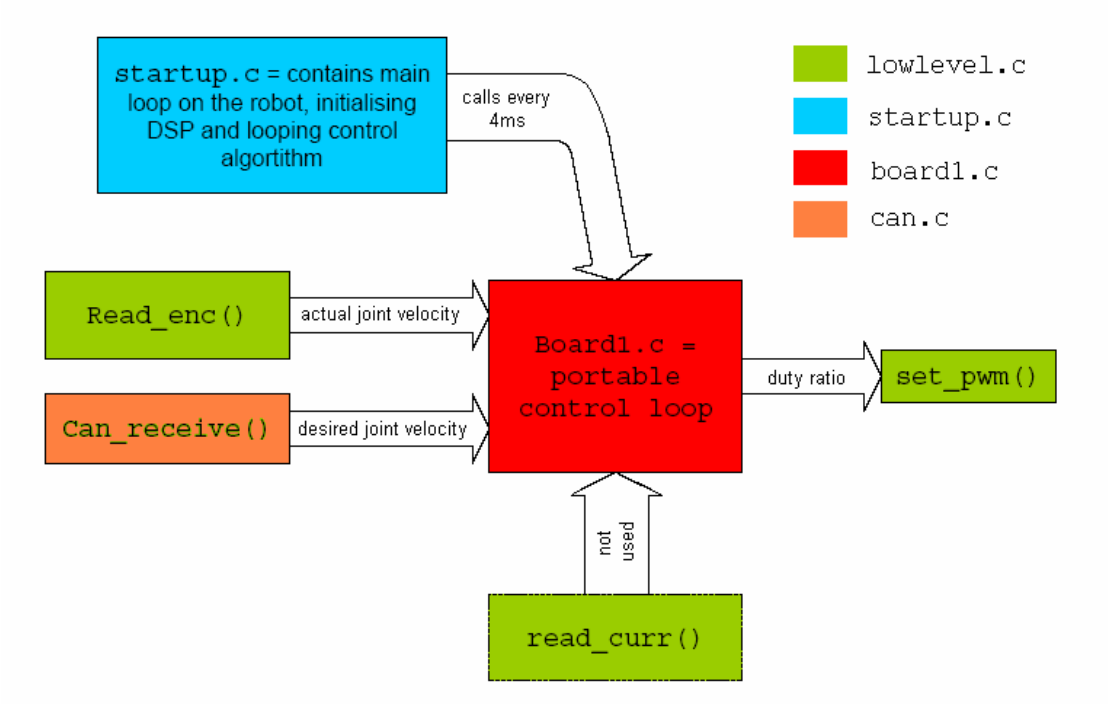


Figure 2.12: Block Diagram of 2001 Controller Code. [3]

The code has been kept modular so that all firmware code(lowlevel.c, can.c and startup.c) can be updated without modifiying the control algorithm code. What this means is that board1.c(`b1_control()`) requires no alteration. `b1_control()` is also used by the simulator and therefore maintaining this code as a separate module enables it to be used by both systems.

`b1_control()` achieves PI velocity control by calling upon methods in lowlevel.c and can.c. For each particular motor, the `can_receive()` function supplies the desired velocity from the CAN bus and the `read_enc()` function supplies the current encoder reading. From the desired velocity and current encoder reading, `b1_control()` dictates an estimate of the required PWM setting to achieve the desired velocity. This required PWM setting is set via the `set_pwm()` function.

The purpose of the `read_curr()` function is to continually monitor the armature current of each motor and if required, alter the required PWM setting to a maximum safe setting that will not damage the motor. This feature is currently not utilised on the 2001 boards.

2.3.1 PWM Duty Cycle Feathering

In 2001 Hood discovered that when low duty cycles were required there was not enough resolution for duty cycle settings. Originally, PWM duty cycle had a resolution of -100 to 100 for full reverse and full forward motion respectively (actually a number from 0 to 200 – Bipolar PWM). Under the direction of Dr Gordon Wyeth, a procedure called "feathering" was implemented to give finer resolution.

An integer between -1600 and 1600 is passed as a parameter for desired duty cycle. The lower four bits of this number are masked off to give a "*feather_count*" between 0 and 15. What remains is a duty value between -100 and 100. A "*pwm_low*" is then assigned as that duty value between 0 and 100 and a "*pwm_high*" value is assigned as that duty value plus one (or -1, if the duty value is between 0 and -100). The 100kHz timer which in fact generates the PWM, enables a *counter* to be incremented every 10us. If *feather_count* is less than *counter*, the duty cycle is set to *pwm_high*. If the

feather_count is less than *counter*, the duty cycle is set to *pwm_low*. When *counter* reaches 16 it is reset to 0.

The overall effect is that the PWM duty cycle can be passed a decimal amount between -100 and 100. The decimal value consists of *feather_count* / 16.

2.4 2001 Controller Design Flaws

The design specifics of the old controller boards will not be mentioned in this document. For a detailed description of the full 2001 design refer to Stirzaker[4]. Only the components of their design that impeded board performance will be described here.

Based upon Hood's analysis of the 2001 design, the following discussion outlines the flaws of the 2001 lower limb controllers. It is these issues that necessitated the design of the new 2004 controller boards.

2.4.1 Control Loop Speed

After the development of control code by Drury and Hood in 2002, it was found that the control loop was not able to operate at Stirzaker's originally anticipated speed of 2kHz. This was mainly due to limitations of the 2001 controller's CPU. The Texas Instruments TMS320F243 processors are only equipped with a single quadrature decoder channel. Since each board was required to control three motors, a single channel was not enough.

To compensate for this Stirzaker designed his boards to have two external quadrature decoders. These IC's process the encoder pulses and feed 16 bit encoder count values, one byte at a time, into the TMS chip via its data bus. This proved to be a major bottleneck for the control loop algorithm.

The end result is that each control loop actually requires 1.28ms to process, and thus 4ms(250Hz) has been allowed for each loop in order to cater for additional serial feedback of sensor data to an external computer for analysis. [3]

2.4.2 Power Consumption

The 2001 controllers feature motor drive circuitry that consumes power inefficiently.

Stirzaker chose an integrated motor driver circuitry package for the H-bridge circuitry, the L6203 from ST. Unfortunately these packages were found to very inefficient, requiring the dissipation of a large amount of heat. This required the mounting of large heat sinks adding weight to GuRoo's lower limbs. Hood's work uncovered that losses in the L6203 were mainly conduction losses of the switching devices within due to their large on resistance(R_{ON}). [3]

The L6203 package also required the use of bipolar switching, which as stated in section 2.2.4 is a relatively inefficient switching method. When GuRoo was first powered up, it was found that there were so much conduction losses that within minutes the MAXON motors became hot to touch. This was reduced by introducing separate inductors in series with the motors, proving that most of the losses were due to a large amount of ripple current dissipation. This halved the ripple current and the motors ran cold, but the addition of extra inductors also added unnecessary weight to GuRoo's lower limbs. [3]

T rectify these problems, Hood made design specifications for a more efficient semi-discrete motor driving circuitry.

2.4.3 Implementation of Design

The PCB layout of the 2001 controllers needed review. Positioning of the components on the board was found to be poor. There was a large amount of wasted space and few useful test points. Placing clip test leads to ground on the boards was inconvenient. The DSP itself was difficult to access. Motor power headers were not placed in order, which posed a possible threat to wiring up motors incorrectly. There was also a lack of debugging LED's on the board. The wiring harness proved to be complicated and impractical. [3]

2.4.4 Lack of Initialisation Software

It was found by Hood that initialisation of the robot prior to starting a gait routine was impractical. Power would be applied to GuRoo while on its stand followed by placing it on the ground and the positioning of each joint by eye[3]. A very time consuming and potentially difficult process.

Propositions for an initialisation routine have been made prior to this thesis. The original plan was to use mechanical stops or switches and driving GuRoo's limbs to their mechanical limits followed by retraction to a set point. Hood proposed an optical sensor alignment scheme.It was decided that this would add unnecessary complexity to the system.

Instead it was decided that utilising the index pulses of the encoders would provide a means for initialisation. Index pulse initialisation was achieved as part of this thesis and the successful results of a routine developed to do so are discussed in section 4.2.

2.5 2004 Lower Limb Controller Design

This section outlines the fundamental components of the new design. The full schematic is contained in Appendix A for board's 1-5 and Appendix B for board 6. Note that this schematic has been updated to include the adaptions made during the course of this thesis. The details of the adaptations made will be explained further in section 3.

2.5.1 Microcontroller

The microcontroller chosen for the new boards is the Motorola MC68376. It was in fact the original desired processor by Stirzaker & Cartwright in 2001, but due to cost and availability was not chosen [4]. Hood also outlined its superior aspects when considering processor selection for a new controller design. This processor has the following features:

- 32 bit architecture.
- On board TPU(Timer Processing Unit) with which 16 pins can be individually programmed to use its capture register (to decode quadrature pulses) or compare register (for PWM generation).
- Analog to Digital Converter for current sensing.
- TouCAN communication module.
- Operating speeds of up to 21MHz.

The most advantageous of these features is the TPU module's capabilities. It features 16 independent timer channels. This allows for three PWM outputs and six quadrature decoding lines(two are required for each encoder) leaving seven TPU lines to spare. This addresses the former quadrature decoding bottleneck as raised by Hood, of the 2001 controllers.

Note that Hood also expressed desirability for this processor but did not actually specify for it in his design, again due to issues of cost and availability. Subsequently the 2004 controller boards have all been designed to incorporate the Motorola 68376.

2.5.2 Memory Setup

The Motorola 68376 itself has limited on board EEPROM and SRAM, 8kB and 4kB respectively. Therefore the supply of larger external programmable flash and SRAM was a necessity.

Turk and Kee originally specified 64K x 16 of AS7C1026 SRAM from Alliance Semiconductor for its 16 bit bus width and fast access time[13]. The 16 bit bus width enables direct interfacing with the MC68376's 16 bit external memory bus, and hence single word accesses per bus cycle. At a later date, an AS7C4098 package was specified for the 2004 GuRoo boards. This package has the same pin configuration but has a larger capacity of 256K x 16, enabling a total supply of 512kB[14] to the microcontroller.

Two ST M29F010B 128K x 8 flash memory chips were specified by Turk and Kee for program memory. The two chips are interfaced in parallel with the MC68376

allowing 16 bit word accesses through the memory bus. This supplies a total of 256kB of EEPROM program memory to the microcontroller.

2.5.3 Motor Driver Circuitry

2.5.3.1 *H-bridge Design*

Hood mad specifications for a semi-discrete H-bridge design for the 2004 controller's motor driver circuitry. Although this requires more board space and greater circuit complexity, the trade off is improved power efficiency and device control [3].

Hood specified IRFZ44NS MOSFETS for the switching devices. At a later date, IRF530NS MOSFET's manufactured by International Rectifier were selected. The IRF530NS were chosen over the IRFZ44NS because they feature faster switching times[18]. Faster switching times equates to a reduction in MOSFET switching losses. The IRF530NS also feature a higher voltage rating of 100V as opposed to 55V for the IRFZ44NS, which is very close to the motor voltage supply of 42V. The circuit schematic for the 2004 controller H-bridge design is shown in Figure 2.13.

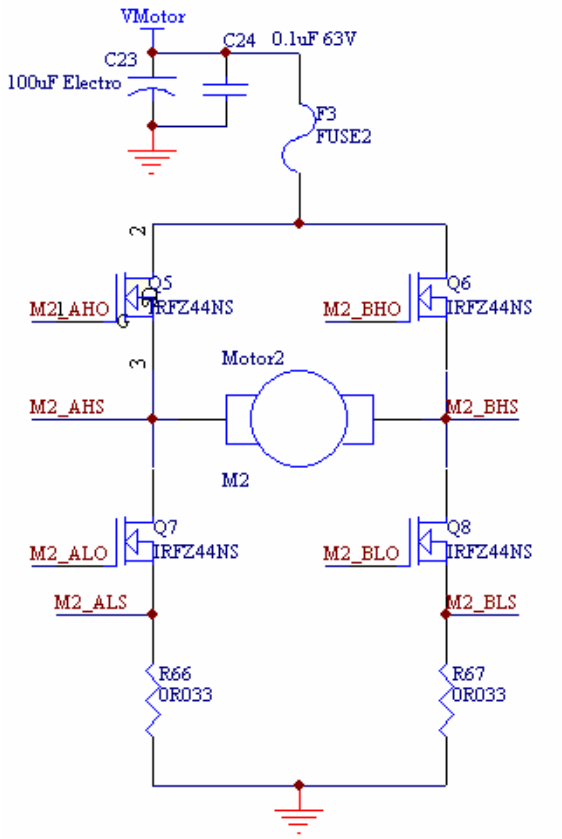


Figure 2.13: 2004 Controller H-bridge Design.

2.5.3.2 *MOSFET Driver Design*

In order to supply power to the high side of the H-bridge a bootstrap voltage must be applied to the gate of the desired MOSFET switch. As part of the semi-discrete design, Hood chose HIP4081 IC MOSFET drivers by Intersil. These drivers are capable of providing the sufficient bootstrap voltage with a switching frequency well above the required 100kHz. [3]

The circuit schematic for the MOSFET driver circuitry is shown in Figure 2.14. The purpose of the NAND gate configuration preceding input for the lower gate drivers, is to enable unipolar PWM. The PWM line is an output line from the TPU of the 68376. Varying combinations of PWM and PWMDIR enables the H-bridge to be configured for three modes of operation, driving forward, driving in reverse and braked to ground. The input truth logic for PWM and PWMDIR is outlined in Table 2.2.

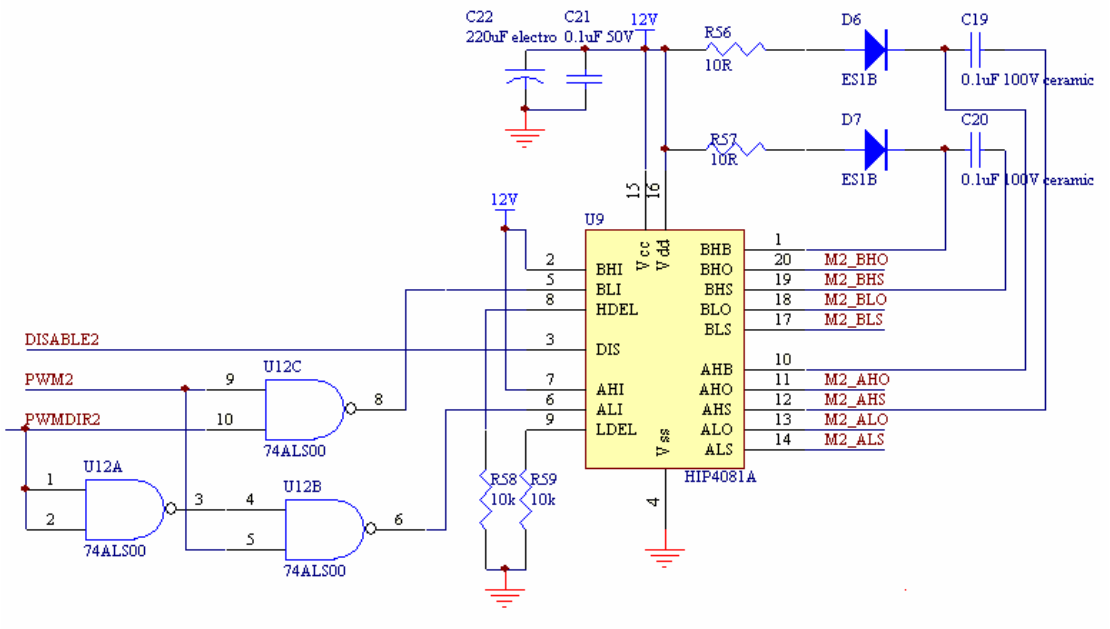


Figure 2.14: 2004 Controller H-bridge Driver Design.

PWM	PWMDIR	BLI	ALI	AHO	BHO	ALO	BLO	Motor
0	0	1	1	0	0	1	1	BRAKED TO GROUND
0	1	1	1	0	0	1	1	
1	0	1	0	1	0	0	1	FWD
1	1	0	1	0	1	1	0	REV

Table 2.2: Truth Table for Input Logic to MOSFET Drivers.

2.5.4 Current Sensing and Motor Protection

As shown in Figure 2.15, each H-bridge features a hardwired 5A fuse to prevent motor current from damaging the motors. Alternative to this, current is intended to be limited in software through the use of current sensing circuitry.

Figure 2.13 shows two low ohmic current sensing resistors that lie in either lower leg of the H-bridge. The voltage drop across these resistors is used to measure the current flowing through the H-bridge.

The voltage drop across each resistor is first fed through a low pass filter to dampen the voltage ripple present due to the PWM of motor voltage. Because the voltage measurements are so small, they are then amplified by LMC6082 op-amps from National Semiconductors, and fed to the ADC input channels of the DSP.

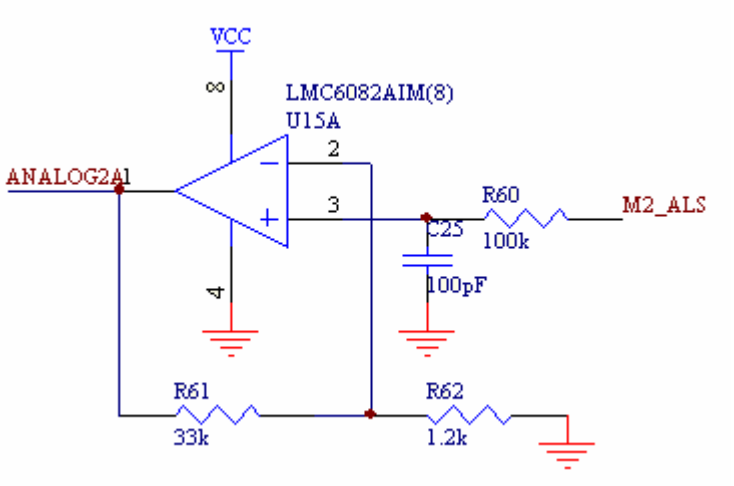


Figure 2.15: 2004 Controller Current Sensing Circuitry.

The amplifier has been designed to yield the following gain:-

$$\textbf{GAIN} = V_{OUT}/V_{IN} = 1 + (R_2/R_1)$$

$$\therefore \text{GAIN} = 1 + (33000/1200) = 28.5$$

where R_1 (R62) = 1.2 kΩ.

R_2 (R61) = 33 kΩ.

The Analog to Digital Converter of the 68376 has been setup through hardware to convert input voltages in the range of 0 to 5V. The resistances of the current sensing resistors in the lower legs of the H-bridge are 0.033Ω. The amplifier gain was designed so that for 5A of current flowing through the H-bridge the corresponding voltage input to the ADC channels are:-

$$V_{5A} = GAIN(I \times R_{SENSE})$$

$$\therefore V_{5A} = 28.5(5 \times 0.033) = 4.703 \text{ V}$$

This almost gives a 1:1 relationship between armature current and input voltage. The amplifier has been designed to convert up to 5A, because each motor driving circuitry has 5A fuse protection.

2.5.5 Power Supply

Power supply for the boards requires three different input voltages all sharing a common ground rail. All board logic requires +5V with the exception of the MOSFET drivers, which require +12V. Lower limb motor power is supplied directly from two 42V NiMH battery packs.

Figure 2.16 details the power supply circuitry for boards 1 to 5. Logic power is supplied from two 7.2V RC(radio controlled) car batteries which is regulated down to 5V. This is achieved through a 5V LM2940C 1A voltage regulator from National Semiconductor [19]. A 5A fuse preceding the regulator provides logic circuitry power protection.

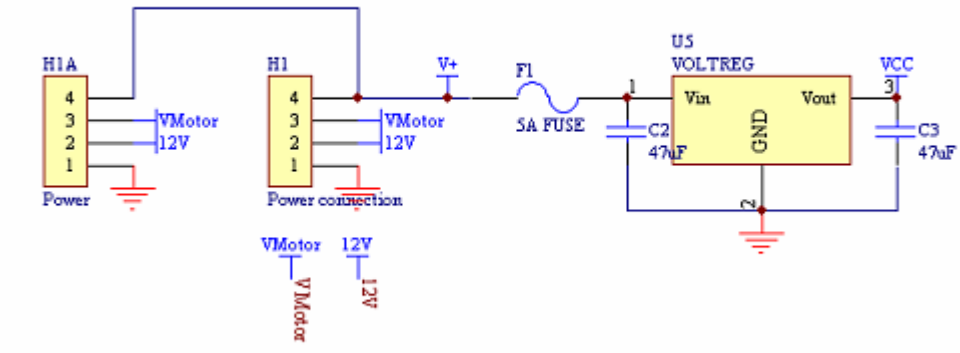


Figure 2.16: Lower Limb Board Power Circuitry.

Power supply for the MOSFET drivers is not yet specified. Hood made specifications for buck converters to derive 12V and 5V from the 42V NiMH battery packs. A final design decision is yet to be made for the supply of these voltage rails. It is anticipated that 12V will be converted from the 42V supply and transferred to both the MOSFET drivers and the 5V voltage regulators.

Power distribution throughout GuRoo is achieved through daisy chaining the power wiring harness from board to board. Twin power headers on each board allow for this.

2.5.6 Communication

Communication throughout GuRoo is achieved through a CAN protocol. CAN requires a two wire bus that is daisy chained from board to board. The Motorola 68376 is equipped with a TouCAN module that handles formatting of the CAN frames. These CAN frames are then further manipulated into the required CAN-H, CAN-L format for transmission across the network by a PCA82C251 CAN transceiver, manufactured by Phillips.

Successful communication requires that termination of the CAN network be imposed at terminal nodes of the network. Each board is capable of becoming a terminal node through the insertion a jumper that shorts the CANH and CANL pins of the transceiver through a 120Ω resistor. The presence of the resistor eliminates reflected signals[11].

The Motorola 68376 comes equipped with a Queued Serial Module(QSM). This module enables two wire RS232(SCI - Serial Communication Interface) transmission and reception and also features a Queued Serial Peripheral Interface(QSPI). Figure 2.17 outlines the communication circuitry present on the 2004 controllers. As yet, the SCI is the only part of the QSM that has been configured for use.

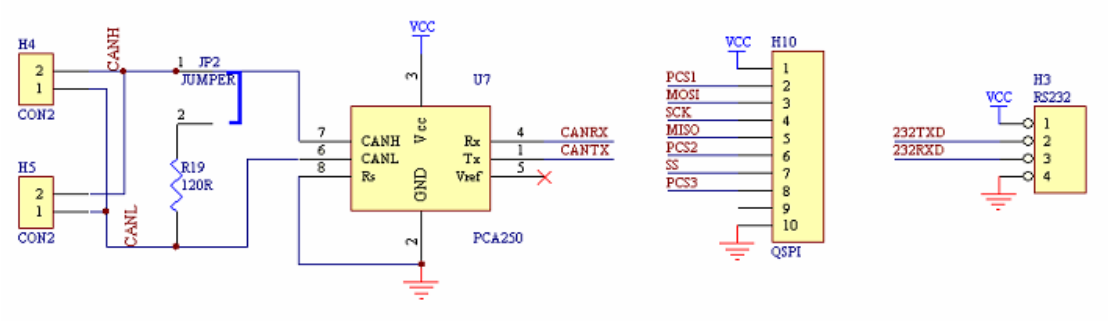


Figure 2.17: CAN Communication Circuitry(left). QSPI and RS232 Headers (right).

2.5.7 Additional Features

The new 2004 controllers feature eight debugging LED's enabling data to be viewed a whole byte at a time. Two push buttons can be programmed as external interrupts. Extra headers have been laid out for future use of the unused TPU channels and CTM lines. Each board is also capable of supporting a small speaker which may prove to be useful for debugging purposes in the future.

The new boards have also been setup to take advantage of index pulsing of the encoders. Index lines have been routed to the 68376's external interrupt lines. The use of these lines for initialisation of the robot is detailed in section 4.2.

Figure 2.18: ICD Programmer - In System Programming.

The Motorola 68376 also features a Background Debug Mode(BDM) which enables in system programming through an ICD(In Circuit Debugging) Programmer. This method of programming is very rapid.

The 2004 boards have also been equipped with 4-bit DIP switches that can be utilised to define each board's location(board ID). These switches enable software to define CAN message buffers on reset. It is also intended for all six boards to be programmed with the same code, with the relevant sections of code segmented out based upon the board ID defined by the switches.

2.5.8 Board Layout and Placement

Board layout has been drastically improved in this design. Motor connectors are clearly numbered. Motor, power, CAN, BDM and encoder headers are all easily accessible, and a ground header has been provided for oscilloscope measurements. The new 2004 controller board layout is detailed in Appendix C.

3. Development of the 2004 Controller Boards

In late April 2004, the 2004 Controller PCB's were received. One of these PCB's was populated with the basic components – microcontroller, power supply and LED's.

3.1 Board Development Plan

Board development involved a step by step approach. The plan was to initially construct a single prototype board, achieving the following milestones:-

- Setup code development environment using Microsoft Visual C++.
- Successfully program a board.
- Write an LED pattern program that utilises the push buttons.
- Develop serial(RS232) communication between board and PC for debugging purposes.
- Generate PWM using the 68376's TPU.
- Populate and test the motor driving circuitry for a single motor to be driven.
- Generate and test the quadrature decoding capabilities of the 69376's TPU.
- Develop a feedback loop utilising PWM and quadrature decoding.
- Populate and test the current sensing circuitry through configuration of the 68376's QADC module.
- Populate and test the two remaining motor driver circuitries.
- Configure the TouCAN module and test communication.
- Develop software to run a PI velocity control loop driving a GuRoo motor.
- Futher develop this software to simultaneously control the velocity of three GuRoo motors deriving their desired velocity from a separate board through the CAN network.

Once a single board was developed the plan was then to populate a further four boards. Software could then be updated to suit the new boards, followed by their installation into GuRoo.

All of the development stages listed were successfully completed over the course of this thesis, with the exception of final installation into GuRoo. It was also requested

that current sensing AD be conversions be calibrated and correlated with motor torque. This was attempted but unfortunately due to technical issues and time constraints could not be finalised.

3.2 Configuring the Motorola 68376

In order to program the 68376 a development environment was created using Microsoft Visual C++. It involved porting the existing RoboRoo's programming environment and modifying it for use on the GuRoo boards. The RoboRoo's are centrally controlled by a Motorola 68332, hence most of their software was compatible with the 68376. The development environment, "board.dsw", incorporating all new software has been submitted on the accompanying CD-ROM.

3.2.1 Initial Hardware Issues

Upon successful programming of the boards a hardware design fault was found that was inhibiting the processor from running. As shown in Figure 3.1, the MODCLK pin was being pulled low on reset when it was actually required to be pulled high. The 68376 is configured on power up or reset by holding certain pins high or low(reset conditions)[10]. By pulling this pin low on reset the processor is configured to use an external clock source driven onto the EXTAL pin. This was not the intended mode of operation. Instead the MODCLK pin was actually required to be held high on reset. This would allow the 68376's clock synthesiser to generate a clock source using the external 4.194304MHz crystal oscillator. Diode D1 was removed and a 10K pull up resistor was inserted between VDD and MODCLK.

Following this it was observed that the processor would stall after approximately five seconds of run time. Phantom interrupt requests were being generated by the unconnected, IRQ6 & IRQ7 pins. It was discovered that a reset condition was configuring the processor to allow external interrupts on power up. The processor was crashing because there were no interrupt routines configured in software. The problem was rectified by the insertion of a missing jumper link, JP1, which ensures DATA9 (pin 100) remains low during reset, as shown in Figure 3.1. By holding this pin low on reset, PORT F is configured for normal I/O, disabling the interrupt request lines.

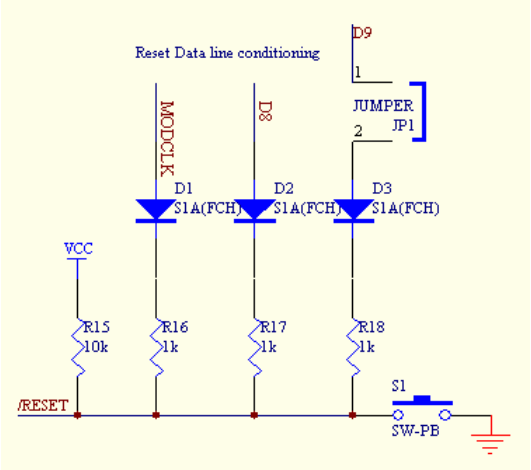


Figure 3.1: Original Reset Conditioning Circuitry.

It is possible to run a program without the jumper link present by setting PORT F to I/O mode early in software. However there is no foreseen benefit in doing this and it is not recommended, as index pulses from GuRoo's encoders which are also connected to PORTF, may trigger the same problematic behaviour.

3.2.2 System Clock Speed

The system clock speed, f_{sys}, is synthesised from an external 4.194304MHz crystal on reset. This is configured by setting the SYNCR register of the System Integration Module(SIM) during power up in socpwr.as. At present f_{sys} is synthesised to a maximum recommended speed of 20.972MHz.

3.3 Memory Map

The 68376 has been configured to utilise 24 bit addressing. Base registers for memory addressing the flash, SRAM and the internal registers and SRAM are all configured by an assembly file called "socpwr.as". This file contains the boot code for the processor when it is reset.

Figure 3.2 outlines the memory map for the 68376 memory configuration. The TPU SRAM is mapped by an initialisation routine for the TPU in software, `initTPU()`.

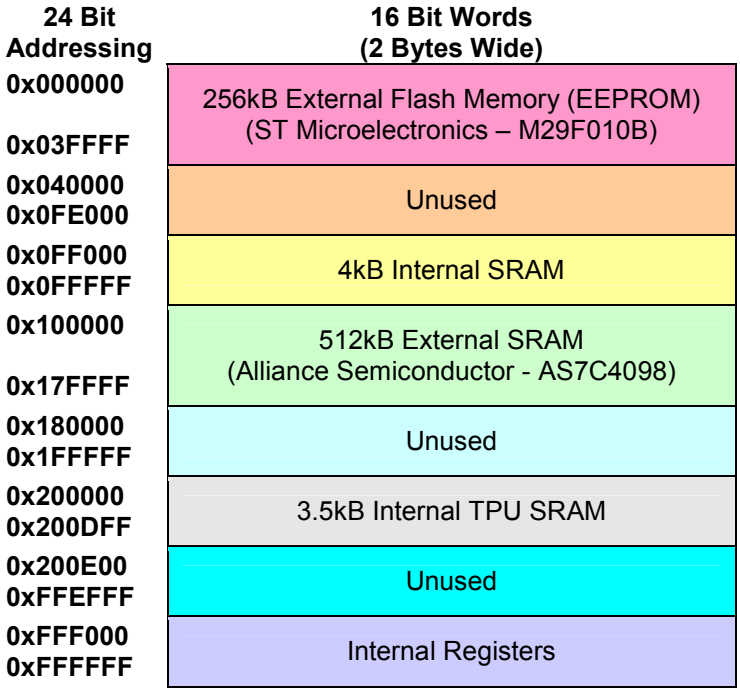


Figure 3.2: Internal and External Memory Addressing on the Motorola 68376.

3.4 Serial Communication Development

Using the Queued Serial Module(QSM) of the 68376, code was developed to transmit data from board to PC. Communication requires an RS232 connector and a DB9 cable, as shown in Figure 3.3, to connect to a PC COM port. The SCI enables the processor to send ASCII bytes, one character at a time. Output characters can be viewed and/or captured to a text file using a software package such as Microsoft Hyperterminal.

Setting up the SCI involved configuring the SCCR registers of the QSM module to enable the transmitter and select an appropriate baud rate. A baud rate of 38400 (bits/sec) has been implemented, as testing uncovered that this was the fastest setting that could successfully transmit to the PC. Testing involved adjusting the SCCR0 register to give a matching configuration for the predefined baud rates in Microsoft Hyperterminal. SCCR0 specifies the prescaling of an appropriate baud rate from the system clock(f_{sys}). Because f_{sys} is configured to an ambiguous speed of 20.972 MHZ, it is difficult to match baud rates between board and PC, and 38400 bits/s was the fastest speed setting that would function correctly.

Two functions, `transmitChar()` and `transmitShort()` were written to transmit *char* and *short* variables respectively. Transmitted numbers can be viewed in hexadecimal format using Hyperterminal or any other suitable communications program. These functions are contained in the updated lowlevel.c.

Care should be taken when using these functions as the baud rate is currently very slow and transmission times may hinder other processing. For debugging purposes, it is best to use CAN to transmit data to a central board for storage followed by SCI data transmission on completion of testing. Each board is equipped with 512kB of RAM which allows for a large amount of data storage.

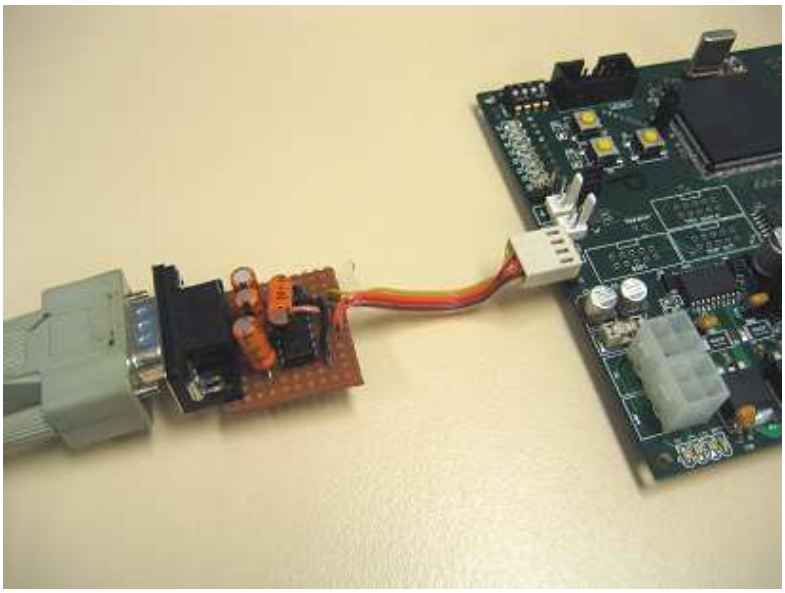


Figure 3.3: RS232 Connector for Board to PC Serial Communication.

3.5 Timer Processor Unit Development

Because the RoboRoo's use a Motorola 68332 processor, PWM and quadrature decoding software was already written for them. This was directly ported for use on the GuRoo boards. This software is all contained in tpu.c.

The Motorola 68376 processors that were supplied (package type (order number) – MC68376BGCFT25) contained the wrong pre-programmed TPU function set, which was incapable of PWM generation and QD[5]. Instead of writing custom made TPU

functions, a programming routine was written by Turk[13] to upload a RoboRoo TPU library function set to the flash each time the new boards are programmed. During reset, initialisation code transfers this function set from flash memory into the TPU RAM for use.

3.5.1 Pulse Width Modulation Generation

The TPU clock is derived from the system clock and has been configured to prescale for a maximum frequency of 5.243MHz. Therefore the minimum time quanta the TPU can utilise is 191ns. The `tpuPWM_init()` function in tpu.c, is used for configuring TPU channels for PWM. It is passed parameters for period and duty cycle as quantities of this time quanta. Once a TPU channel has been initialised for PWM, the duty cycle can be altered using the tpuPWM_set() function. Figures 3.4 and 3.5 show successful use these functions to generate PWM at approximately 100kHz and 50kHz respectively.

Because the time quantum is fixed at 191ns, increasing PWM frequency lowers the resolution of duty cycle settings. For example, for a theoretical PWM frequency of 104.9kHz, the PWM period should be set to 50 time quanta, giving a duty cycle resolution of 0 to 50. During velocity and position control testing this proved to cause problems when high frequencies of PWM were used and very low duty cycles were desired. An attempt was made to rectify this problem by implementing a simplified version of feathering, as used on the 2001 boards. This could still not provide a fine enough resolution. Further explanation of this attempt is detailed in section 4.1.2.1.

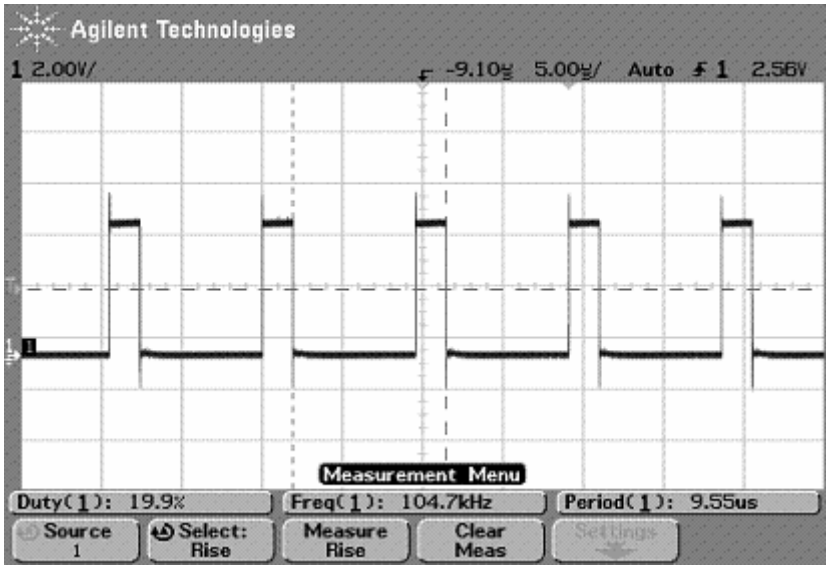


Figure 3.4: TPU Generating PWM (~100kHz, 20% duty cycle).

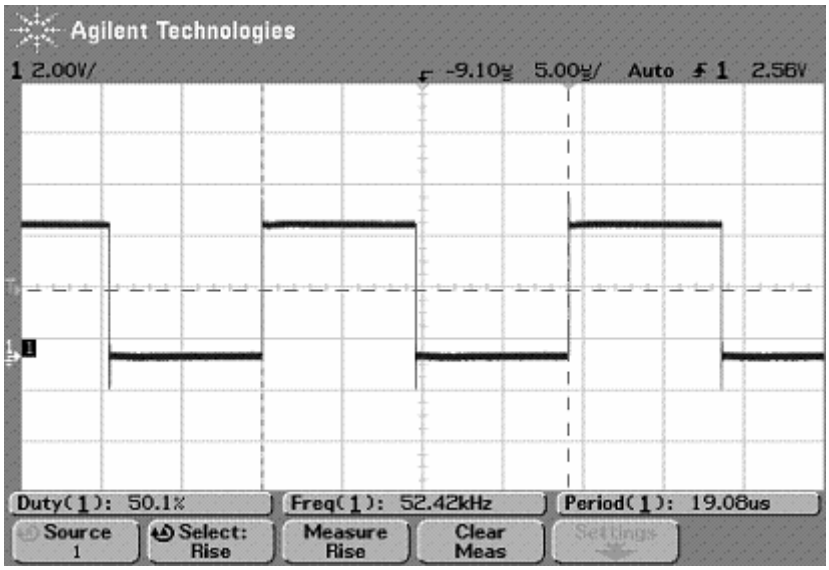


Figure 3.5: TPU Generating PWM (~50kHz, 50% duty cycle).

3.5.2 Quadrature Decoding

TPU channels can be configured as input capture channels for QD using the `QDEC_init()` function in tpu.c. This function is passed as parameters the two out of phase channels for decoding. Reading the decoded value is performed by using the `QDEC_read()` function in tpu.c to read the desired TPU channel.

Testing the QD capabilities involved running a motor at slow speed and outputting the encoder counts to the SCI. This testing showed the QD was able to increment and decrement as expected for forward and reverse rotation respectively.

Other testing included running and stopping a motor after a number of complete revolutions corresponding to a particular number of encoder clicks, to verify there was no accumulation of error. By observation, the motor was stopping precisely after complete revolutions and hence appeared to be functioning properly.

Further verification that the quadrature decoding functions correctly is supported by the serial output obtained from index positioning testing as shown in Table 4.1.

3.6 Current Sensing Development

The 68376 has a Queued Analog To Digital Converter Module(QADCM) for ADC processing. Operation of this module requires setting up a list("queue") of channels from which AD conversions are required. When ADC's takes place, the processor sequentially samples and converts each channel in the queue. The code developed triggers for these conversions to take place when required in software, running a single pass("scan") through the queue. Once the conversions are completed they can be read from memory.

3.6.1 Configuring the Analog to Digital Converter

Operation of this module is very sensitive to timing because sufficient time must be allowed for accurate conversions to take place. The QADC module runs off a separate clock, the Q-CLK, which is prescaled from the system clock. The Q-CLK frequency($f_{q\text{-}clk}$) and its duty must be set within a tolerable range to ensure correct operation of the module[5].

Setting up the Q-CLK requires programming the Q-CLK high time and low time. As recommended by the 68376 datasheet, a $f_{q\text{-}clk}$ of 1.0MHz was aimed for in order to ensure accurate conversions. Optimimum ADC performance is achieved when the duty cycle of the Q-CLK is as close as possible to 75% [5]. Based upon formulas

provided in the datasheet, the high time(t_{PSH}) was programmed to 739ns and the low time(t_{PSL}) to 262ns, giving a $f_{q\text{-}clk}$ of 0.999MHz, as shown in Figure 3.6.

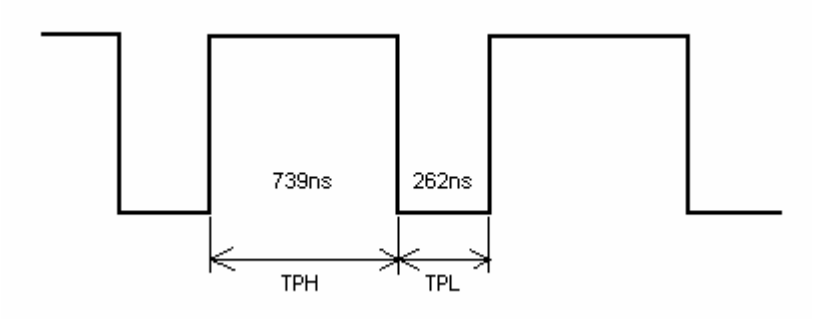


Figure 3.6: QADC Q-CLK Duty Cycle.

Overall conversion time can be programmed between 18 and 32 Q-CLK periods by programming the final sample time. Figure 3.7 outlines the sequence of events during an ADC conversion. An arbitrarily selected final sample time of 8 Q-CLK cycles was chosen. This results in an overall conversion time of 24 Q-CLK cycles, hence the overall theoretical conversion time is $24/f_{q\text{-}clk}$ = 24/0.999MHz = 24.02us.

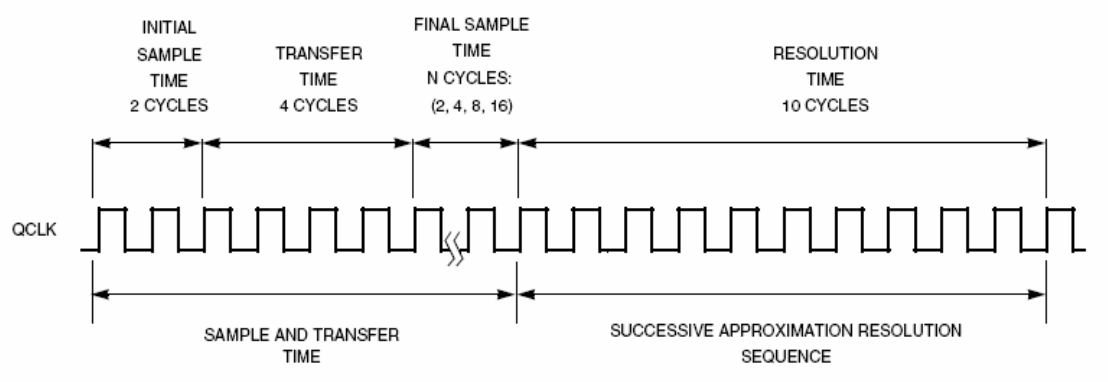


Figure 3.7: AD Conversion Timing. [5]

3.6.2 Conversion Times

If the control loop has to perform six conversions per control loop for the six corresponding current sensing resistors, then the overall length of time required to take ADC readings alone is approximately 6 x 24.02us = 144.1us. Considering that the control loop is intended to run at a speed of 2kHz(500us) the ADC conversions will form a major portion of the overall control loop processing time.

An ADC test program was developed to test conversion times. Actual conversion time for a single current sensing resistor was tested by switching on and off an LED before and after a conversion took place. Figure 3.8 shows the output signal generated across the LED on an oscilloscope. As can be seen, the actual time taken is 27.2us which is close to the theoretically calculated time of 24.02us. The additional 3.2us is due to the small amount of additional code surrounding the conversion and the natural inefficiency in the compiled code.

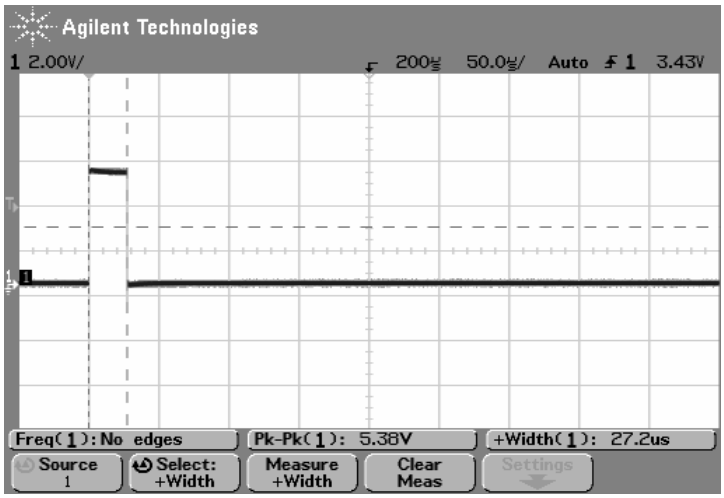


Figure 3.8: Time Taken for a Single AD Conversion.

The time taken for a queue of all six current sensing resistor readings was also tested. As shown in Figure 3.9, the time taken for six consecutive reading is 141.8us which is also close to the theoretical amount calculated.

This conversion time may be slightly decreased by reducing the final sample time. This will have little or no impact on accuracy of AD conversions. Testing would need to be performed to verify consistency in conversion accuracy.

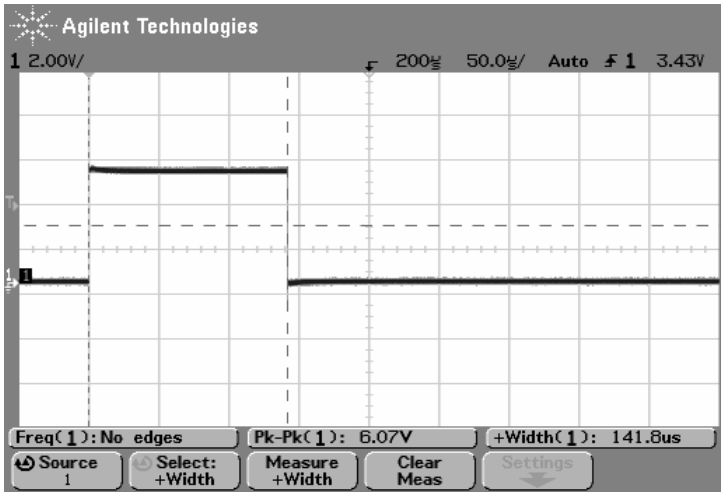


Figure 3.9: Time Taken For Six Consecutive AD Conversions.

If the QCLK is prescaled to its maximum operating frequency of 2.0972MHz, it is theoretically possible to have a minimum conversion time of 6.7uS. This would require enabling amplifier bypass mode, for which the minimum input sample time is 4 QCLK cycles. At an operating frequency of 20.972MHz, the QCLK period is 477ns. Amplifier bypass mode only requires an input sample(4 QCLK's) and resolution time(10 QCLK's)[5] resulting in a minimum conversion time of, 14 x 477ns = 6.7us. This is not recommended as conversion accuracy is reduced as maximum QCLK frequency is approached[5].

3.6.3 Resolution of Current Sensing

The QADC module performs conversions of ten bit resolution. The module utilises the input 5V and GND rails as reference voltages through VRH(voltage reference high) and VRL(voltage reference low) respectively. This gives a theoretical conversion resolution of 0 to 2^{10} (0 to 1024) corresponding to voltages between 0 and 5V. Therefore the AD conversions theoretically have a resolution of approximately 5V/1024 ≈ 4.9 mV per ADC bit increment.

With the existing op-amp configuration preceding input to the ADC module the theoretical current sensing range can be read from 0 to 5A corresponding to 0 to 4.7V. This therefore gives a theoretical resolution of 4.7/1024 = 4.6 mA per ADC bit increment.

It should be noted that this is only theoretical calibration factor. The tolerance of logical 5V is likely to be within 5.00V +/- 200mV with respect to the ground rail. Resistor tolerances and accuracy of the QADC module is also questionable. These factors will all contribute to inconsistencies with the theoretical calculations above.

3.6.4 Current Sensing Testing

Some quick tests were performed to check accuracy of the calculated calibration factor in the previous section. The LED's were used to output the 10 bit AD conversions for two samples of current flowing through the H-bridge. Testing was performed using a load consisting of a series connected 5W 47Ω resistor and a 470uH inductor. 15V was supplied to the motor voltage rail and PWM frequency was set to 50kHz. Figure 3.10 shows the DC current flowing for a 50% duty cycle and the corresponding AD reading displayed to the LEDs. Figure 3.11 shows the reading for a 100% duty cycle under the same conditions. A push button was utilised to display the upper 2 bits of the conversion, but for both test cases the upper two bits were zero. Readings under these conditions were observed to vary +/- one bit.

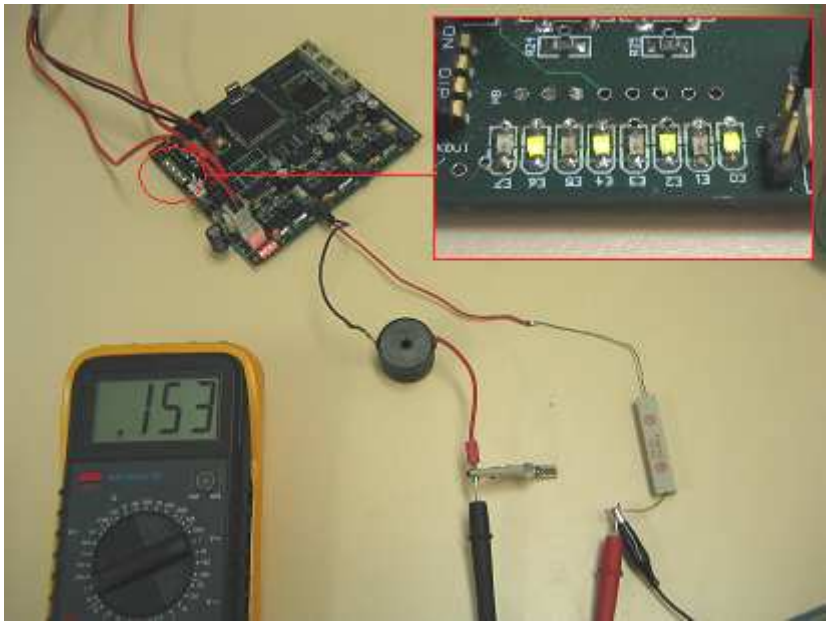


Figure 3.10: ADC Conversions for 50khz PWM, 50% duty cycle.

As shown in Figure 3.10, the conversion obtained a decimal value of 85. Multiplying this value by the theoretical calibration factor of 4.6 gives 85 x 4.6 = 391 mA, which is grossly different to the current displayed on the multimeter (153mA).

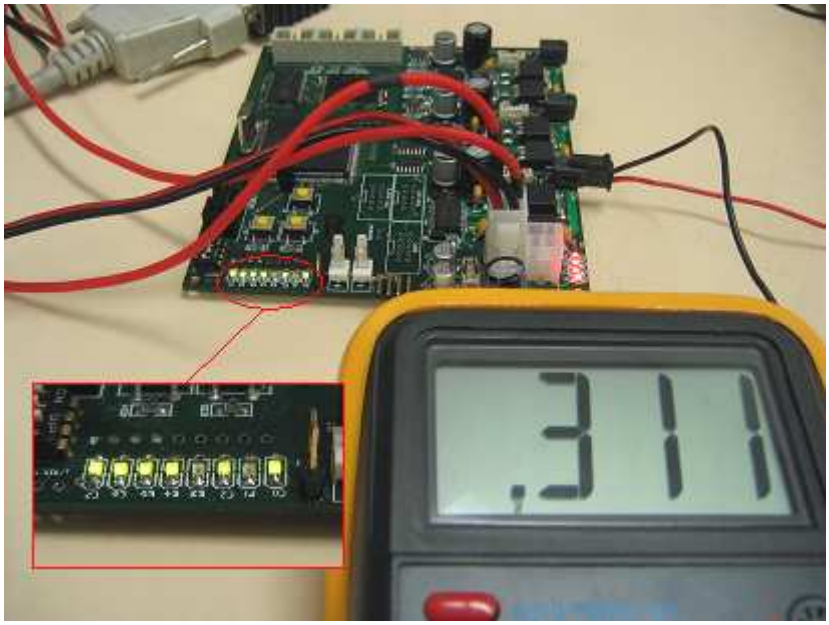


Figure 3.11: AD Conversion for 50khz PWM, 100% duty cycle.

Figure 3.11 shows that for a 100% duty cycle, a decimal conversion of 245 is obtained. Using the calibration factor yields an expected current flow of 1127mA, which is again grossly different to the current flow displayed by the multimeter.

The conclusion to these brief tests suggests that using the theoretical calibration factor to convert the 10 bit ADC reading to a current in mA is grossly inaccurate. It is possible that this calibration factor may hold more truth for higher current flow. Regardless, these results strongly suggest that ADC conversions need to be calibrated by correlation of tested current readings with AD conversions. This is proposed for future work following this thesis.

3.6.5 Calibration of Current Sensing & Motor Torque Correlation

An attempt was made to calibrate current sensing AD conversions and measure correlation of output torque. The apparatus shown in Figure 3.12 was intended for this testing.

Figure 3.12: Attempted Current Sensing and Torque Correlation Test Apparatus.

One of GuRoo's legs was clamped down to the workbench and a half metre length beam was mounted to the plate attached to the thigh motor. This enabled for torque loads of approximately up to 10Nm to be applied to the motor using a 2L water jug attached at 0.5m along the beam length. This remains under the 13.81Nm, maximum permissible continuous torque for the motors transmitted through the gearbox [17]. A set of hanging scales was to be used to measure the mass weight of the water container. The intended torque loads were to consist of the combined weight of the water jug, hanging scales and beam.

A controller was connected to the thigh motor and programmed to run a position control loop. The beam was able to be manoeuvred to desired positions using a push button on the controller board. Provided that the beam and motor shaft were level, and the position control was smooth, the torque counter-acted by the thigh motor could be measured. The torque component of the thigh plate could be considered void through

its symmetry about the rotor shaft provided that the beam remained level. A spirit level was used to ensure that the beam remained level.

The SCI was then used to transmit current sensing AD conversions to a PC. It was found that AD readings were erratic, with differences for a fixed load in the order of approximately +/- 300, and hence consistent readings were not possible. It was later discovered that the main reason for this was because the SCI header was connected whilst AD conversions were taking place. It is theorised that because the negative terminal of the laboratory power supply feeding the rails of the board was not grounded to mains, the QADC's 0V reference was being disrupted. This is because the SCI header's ground is connected through the DB9 cable to the PC's reference ground.

Completion of this task is proposed for future work. The current sensing circuitry in general is currently under further investigation. Once this is finalised, the same methodology can be applied to calibrate the current sensing and motor torque correlation.

3.7 CAN Communication Development

The 68376 is equipped with a TouCAN module for CAN communication. Developing functionality of this module required programming the appropriate configuration registers and testing for correct transmission of messages.

Each TouCAN module can utilise up to 16 message buffers that can be configured for transmission or reception of messages. Each reception buffer can be defined with an individual arbitration ID, and a global mask register pre-filter's messages for all buffers. This allows for messages to be sent specifically to individual buffers or multiple buffers within a single node or across multiple nodes.

3.7.1 TouCAN Module Configuration

Configuration of the touCAN module required specifying the CAN bit timing parameters outlined in Figure 3.13. The TouCAN module has been configured to clock directly from the system clock in order to achieve the maximum CAN bit rate of 1Mbit/s. Each nominal CAN bit time consists of the following segments:

- **Synchronisation Segment** – Each node synchronises to the transmitting node by ensuring that the first edge of a message lies within this segment.
- **Propogation Segment** – This compensates for delays on the CAN network.
- **Phase Segment 1 & 2 –** The CAN message bit value is sampled between these segments. Segment 2 allows time for the module to process the message bit. The sampling point is automatically altered by the TouCAN module by the Resynchronisation Jump Width(RJW) so that a receiving node can resynchronise with a transmitting node.

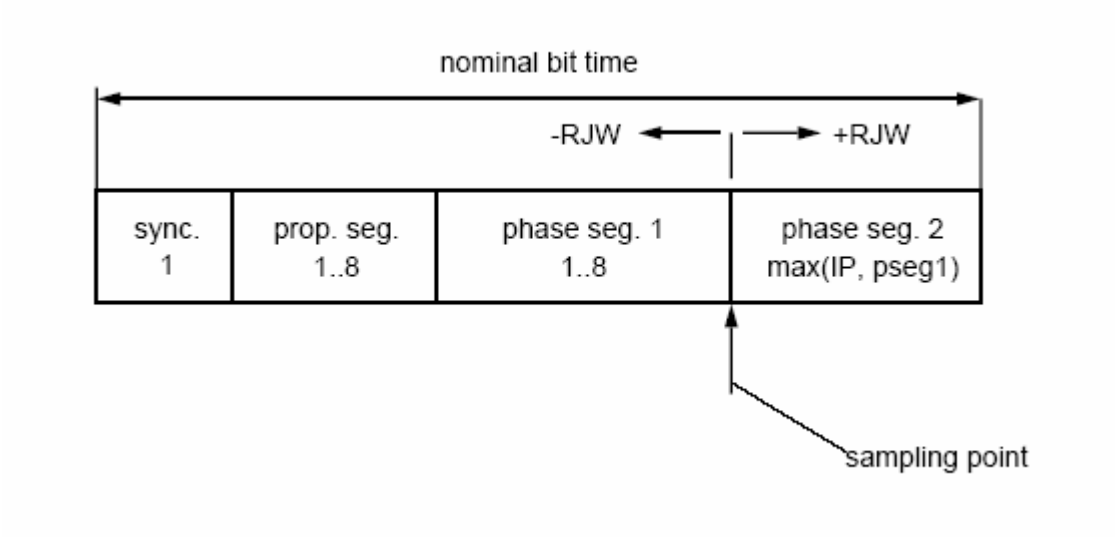


Figure 3.13: CAN Bit Timing Parameters. [11]

The bit time segments and the RJW are each comprised of an amount of system clock time quanta, specified by TouCAN control registers on initialisation, as follows:

- Synchronisation segment = 1 time quanta (fixed)
- Propagation segment = 6 time quanta
- Phase segment One = 7 time quanta
- Phase segment Two = 8 time quanta
- RJW = 2 time quanta

This allows for a total of 21 time quanta per CAN bit. RJW was arbitrarily assigned to 2 time quanta. Since system clock time quanta are each 47.7ns (f_{sys} = 20.972MHz), the total CAN bit time is therefore 21 x 47.7 = 1.00us. According to the 68376 datasheet, the only requirements for these parameters are that propagation segment two be greater than two time quanta, and total CAN bit time be greater than or equal to 9 time quanta. Therefore, the time quanta for each segment have been arbitrarily distributed to meet these requirements, and to make full use of the time between CAN bits of 1.0 us (1 MBit/s).

3.7.2 Message Transmission & Reception

The TouCAN module has only been configured for very basic communication. The software that has been written does not handle TouCAN error counters or utilise interrupts triggered by the TouCAN module. Brief testing did not uncover that these were required, however as future work it is recommended that routines be implemented to make use of these functions.

The TouCAN module supports both standard(11 bit) and extended(29 bit) identifier message formats. 11 bit message identification is ample for GuRoo's current distributed control system which only consists of 7 nodes, and therefore only the standard message format has been catered for in the new software. The standard CAN message buffer structure is outlined in Figure 3.14.

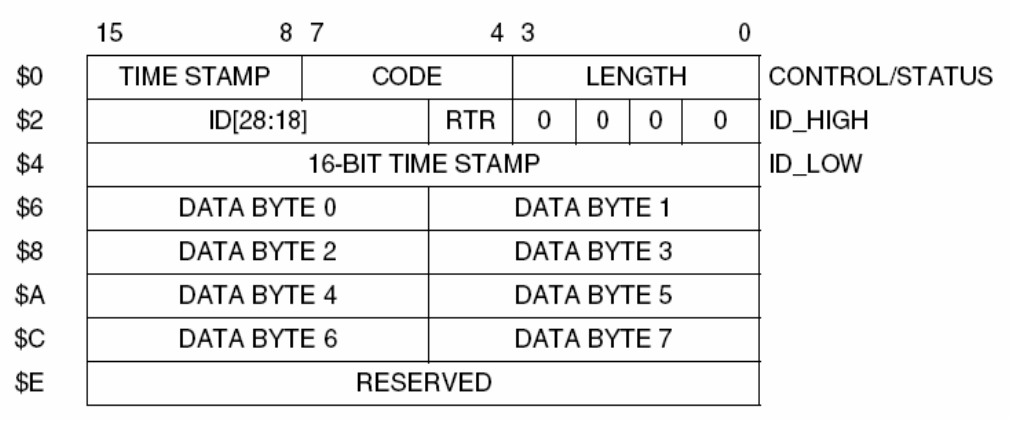


Figure 3.14: Standard CAN Message Buffer Structure. [5]

Message buffers are configured for transmission or reception by writing an appropriate word to the CODE field. When configured for message reception the ID[28:18] field defines the 11 bit reception ID for the buffer. Message transmission requires writing the ID of the destination buffer to this same field. The length of the message for transmission also requires definition in the LENGTH field. The actual data of the message either transmitted or received, is contained in the DATA BYTE [0 : 7] fields.

Each TouCAN module also utilises an 11 bit global receive mask register for message ID acceptance. There is a direct correspondence between each bit in this register and the ID bits in each buffer. When mask bits are set to 0, the corresponding bits in the ID fields of reception buffers are neglected for message arbitration. The software that is currently implemented sets this mask register to 0x7FF so that every ID bit must be checked for message buffer acceptance.

3.7.3 CAN Software Implementation

At present only simple CAN software has been implemented in can.c. Finalisation of can.c is proposed for future work following this thesis.

On reset, each board defines two buffers for active message reception. The first buffer utilises the board ID DIP switches to define its arbitration ID. The second buffer's ID is arbitrarily defined with 0x80 plus the defined board ID for handling incoming board commands. A separate buffer for command messages is required so that the robot's joints can be initialised on command after power up. Further details of the initialisation routine are contained in section 4.2.

The TouCAN module features a 16 bit interrupt flag register with each bit representing an individual message buffer. In the event that a message buffer receives a new message, its corresponding flag bit is set. Incoming message reception is checked regularly as part of the main loop through polling of these flag bits. Further details of software implementation of CAN are detailed in section 4.1.1 & 4.1.3.

A program was written to verify correct operation of the new CAN software. Two boards, with specific DIP switch settings defining board functionality, were connected through a CAN line. By pressing a push button on one board, a counter would be incremented and transmitted to the other board. When that message was received by the other board it would display the correct counter amount on the LEDs. If the board's ID's were changed, they would no longer function as required. This code verified truth in message reception and that the DIP switches could be used to define buffer arbitration ID's.

Further verification of correct CAN functionality was achieved through the implementation of the demonstration software discussed in section 4.3. The demonstration software transmits a sin wave velocity profile at 50Hz from an external board to a board installed in GuRoo's left leg. The board ID switches are not only used to define the message buffer ID's but also to define the sections of code relevant to each board. Figure 3.15 displays an example of a new velocity setting being transmitted to the board in GuRoo's leg.

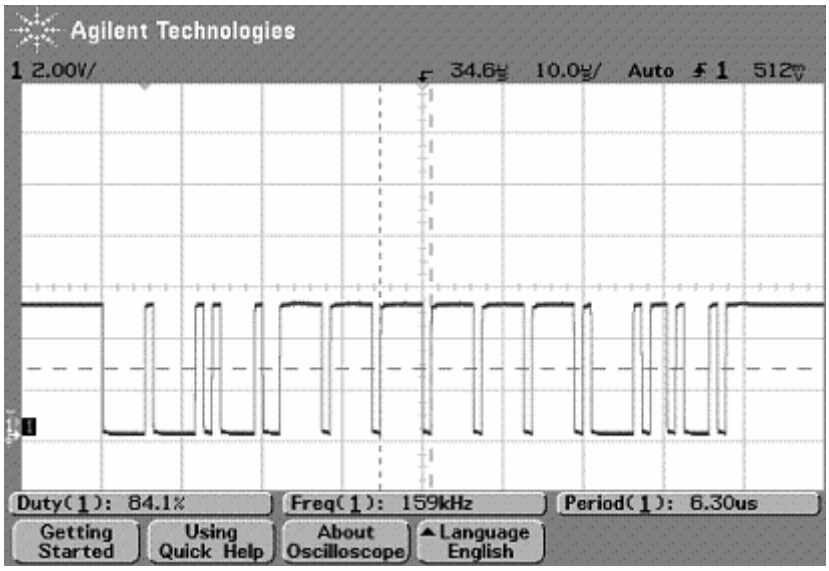


Figure 3.15: Transmission of a Velocity Profile CAN Frame.

3.8 MOSFET Driver Issues

The MOSFET driver's feature a disable pin. When this pin is held high the MOSFET driver maintains all MOSFET's in non-conducting state. It was found during board development that if the MOSFET driver's 12V voltage supply was switched on after motor voltage, the H-bridge would transform into a shoot-through condition. This behaviour was also observed if the reset button were to be pressed whilst the board was receiving 12V and motor voltage supplies.

In an attempt to solve this problem, pull up resistors were applied to each disable line to disable MOSFET's on reset. The disable lines were also re-routed to PORTE which exists in an undefined state on reset, thereby enabling the disable lines to remain high on reset by the pull up resistors.

It was later found, that this behaviour is specifically triggered if MOSFET drivers are enabled whilst the motor voltage supply is on. This problem still exists and finding a solution is proposed for future work.

Because the disable lines were re-routed to PORTE, the three LED lines that were disconnected are now re-routed to PORTC, as shown in Figure 3.16. `outputToLEDs()` was written so that the LEDs could continue to output data a whole character at a time.

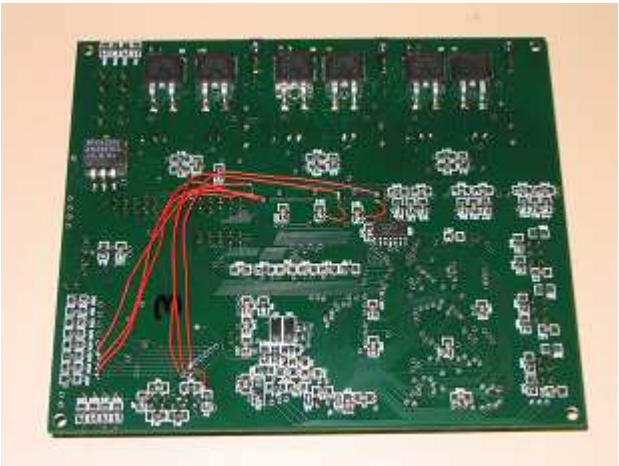


Figure 3.16: Re-routing of the MOSFET Driver Disable Lines.

3.9 2004 Lower Limb Controller Power Consumption

The existing power supply for GuRoo is currently being reconsidered. Now that a semi-discrete motor driving circuitry is used, an additional power rail of 12V is required for the MOSFET drivers. Hood proposed the use of a buck converter to achieve 12V from the 42V NiMH batteries followed by another buck converter for 12V down to 5V.

With the voltage regulators already in place, an input voltage to them of at least 6.25V is required for them to properly supply 5.0V [19]. It makes little sense to further convert 12V down to 6.25V or more. It is therefore envisaged that logic will be powered from 12V also. Although this involves wasted power dissipation through the voltage regulator, it is fairly insignificant with respect to the motor conduction losses.

The advantage of this plan is that the 7.2V RC batteries will no longer be required, reducing the weight in GuRoo's upper body. It will also reduce complexity of the power supply. The drawback is that more power will be required of the 42V supply.

In anticipation of this, the current required for the new boards was measured for the original and the envisaged arrangements, using a laboratory DC power supply and a multimeter. Table 3.2 lists the measurements taken.

Voltage Supplied	Logic Current Drawn	MOSFET Driver Current Drawn	Total Power Consumption
7.2V & 12.0V for Logic & MOSFET Drivers	140 mA	55 mA	1.7 W
Shared 12.0V	215 mA		2.6W

Table 3.1: 2004 Controller Power Consumption.

Therefore the total additional power loss under this new arrangement is approximately 0.9W. It should be noted that these are fairly crude measurements, but it gives an approximate figure of the power consumption of the 2004 boards, and a comparison between both configurations.

4. 2004 Controller Software

The existing software structure was generally maintained for the new boards but required modification to suit the new processor and the implementation of unipolar PWM. The Microsoft Visual C++ development environment was ported from the RoboRoo's and a considerable amount of their compatible code was salvaged for use on the 68376. The new software structure is shown in Figure 4.1. It is a little more complex than the original TMS320F243 software interaction outlined in Figure 2.12, but the original structure has been maintained.

Updating the software generally involved rewriting startup.c and lowlevel.c. An incomplete version of can.c has been written as part of this thesis to demonstrate board functionality but requires further development. A simplified version of board1.c was also written purely for demonstration purposes. The newly written startup.c, lowlevel.c and can.c are contained in Appendix D.

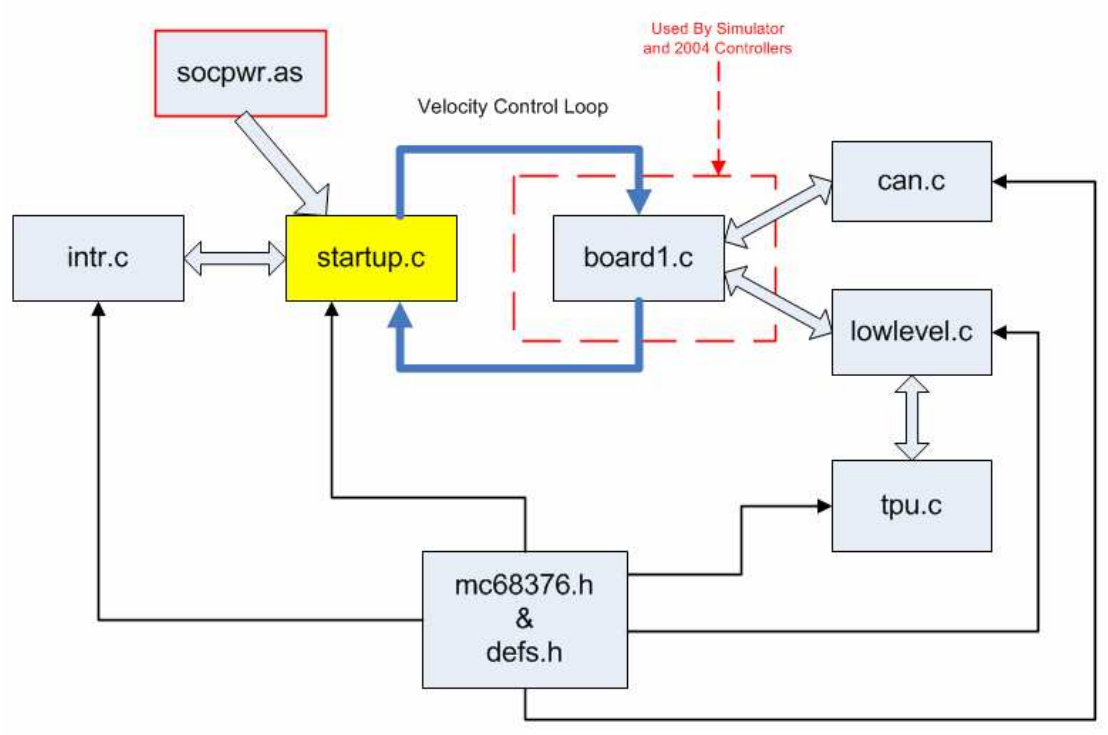


Figure 4.1: Interaction of Software on the 2004 Controllers.

Modularity has once again been maintained for ease of readability and future alterations:

- **socpwr.as** configures the processor on reset.
- **startup.c** initialises the processor for the necessary functionality and contains the `main()` function which calls upon the velocity control loop.
- **board1.c** contains `b1_control()`, the PI velocity control loop.
- **can.c** handles the software for CAN communication
- **lowlevel.c** contains the firmware functions for setting PWM, reading the encoders and reading current sensing through the ADC.
- **tpu.c** has been ported from RoboRoos software and handles the interaction of the TPU with lowlevel.c commands.
- **intr.c** has also been ported from RoboRoos software and handles interrupt processing code.
- **defs.h** defines global constant definitions and new data types for use by all other source files.
- **mc68376.h** maps register addressing for the 68376 modules.

4.1 Updated Software

4.1.1 startup.c and socpwr.as

On reset, the processor itself is initialised by socpwr.as, which configures the clock speed, the memory bus and base registers for memory addressing. Following this, flow of control is passed to the `main()` function in startup.c.

startup.c contains the source code that initialises I/O registers and configures the processor for PWM, quadrature decoding, ADC and SCI functionality. CAN communication is initialised by calling upon `setupCAN()` in can.c. Interrupt processing is set up through `intr_init()`, which is called from intr.c to intitialise the interrupt vector table. As discussed in section 3.5, `initTPU()` transfers TPU microcode from the flash memory to the TPU SRAM on reset. It also configures the TPU clock by calling upon `TPU_init()` in tpu.c.

Board ID is defined during initialisation by reading the DIP switches. At present this defines CAN buffer ID's for message reception. It is also intended for these DIP switches to define the gains that are used by each joint under control. The control software is currently functioning but not finalised. At present `b1_control()` runs the control loop at 1kHz. At this frequency the proportional and integral gains were required to be significantly lower then those defined for the old control loop speed of 250Hz. The present gains were selected through trial and error and as future work following this thesis, gains for the finalised control loop speeds will require calculation.

tpu.c and intr.c are source code files that have been directly ported from Roboroo's software. They only required minor modification for use on the GuRoo boards. In order to maintain their originality and modularity they have remained as separate files for interaction with the existing software structure.

Once initialisation is complete, startup.c then follows into a continuous loop that handles index positioning initialisation and velocity control. Initially, this loop periodically calls upon `posnCtrlIndex()` to perform index position control to initialise the robots limbs. Once an incoming velocity profile has been detected PI velocity control commences and the continuous loop instead periodically calls upon `b1_control()`. These routines are discussed further in sections 4.2 and 4.3 respectively.

Detection of the incoming velocity profile is currently polled in the main loop by calling `chkForMsgs()` from can.c. It is proposed that incoming velocity profiles will be updated by an interrupt routine for CAN message reception. Due to time constraints, interrupt processing has not yet been finalised.

4.1.2 lowlevel.c

lowlevel.c contains the functionality code for control loop processing. These functions have been updated so that b1_control() can call upon firmware without itself requiring modification .

4.1.2.1 `set_PWM()`

This procedure sets the duty cycle and direction of a particular motor based upon an `int` parameter passed to it between -1600 and 1600. The motor number is also passed as a parameter. Motors are numbered from 0 to 2 for motors labelled 1 to 3 respectively on the 2004 boards. The sign of the duty cycle parameter is then used to configure the MOSFET driver to control the H-bridge to drive forward or reverse, as detailed by the input logic shown in Table 2.2. At present a PWM frequency of approximately 15.4kHz is used, giving a duty cycle resolution of 0 to 800. Brief testing showed that this was the required resolution for fine duty cycles. The duty cycle is then divided by two, and negated if appropriate, to give an `int` amount between 0 and 800. This number is then used to set the duty cycle through `tpuPWM_set()`.

As discussed, as PWM frequency is increased, the resolution of duty cycle settings is decreased. Because of this it was found that at high PWM frequencies, when very low duty cycle settings were required for position control and velocity control, the lowest duty cycle was still too high. Instead of neatly maintaining position or zero velocity, the motors would continuously oscillate back and forth due to constantly overshooting a desired position. This behaviour was particularly noticeable in joints under a small amount of load. This problem was also experienced by Hood in 2002, for which the solution of "feathering" PWM was developed.

An attempt was made to implement a simplified version of feathering on the 2004 boards, but brief testing showed that the resolution was still inadequate. To remain consistent with the original software, `set_PWM()` was programmed to accept an input desired duty cycle parameter between -1600 and 1600. The bottom five bits were masked off to give a number from 0 to 31 and the desired duty cycle was then logically right shifted 5 bits to give a *duty* from -50 to 50. This masked off number between 0 and 31 then became the *feather_count* and a *pwm_high and pwm_low* value of *duty*+1 and *duty* respectively were assigned. Using a PWM frequency of 100kHz gave a resolution of 50 duty cycle increments.

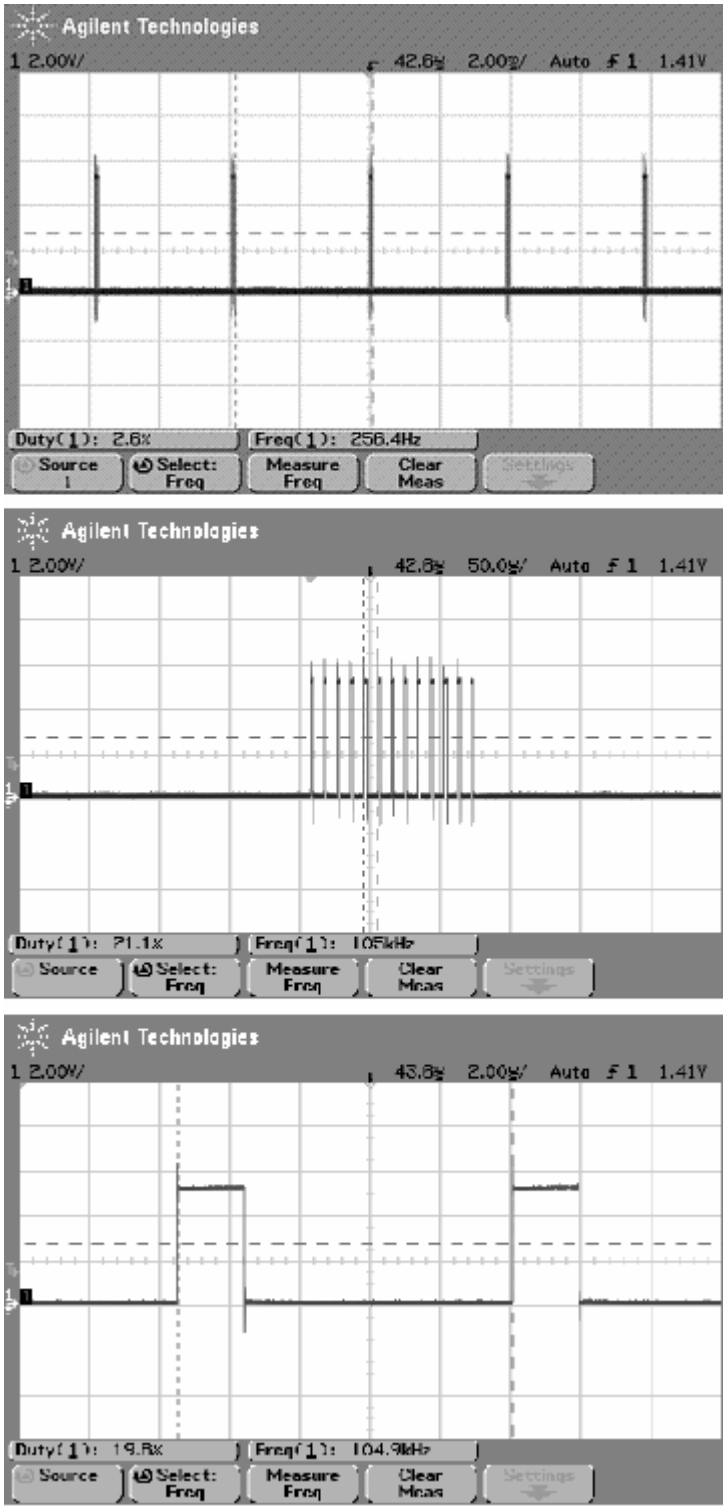


Figure 4.2: PWM Feathering Waveform for a Duty Cycle of "1" with 50kHz PWM (Time scales of 2ms, 50us and 2us are shown from top to bottom respectively)

The 68376 features a Periodic Interrupt Timer(PIT) which was configured for interrupts at its maximum attainable frequency of 8kHz. The PIT was used to compare the *feather_count* with a *counter* that incremented every PIT in the same manner as

Hood's original feathering method. After 32 increments `b1_control()` was called and the *counter* was reset. Figure 4.2 shows a breakdown of a feathered PWM waveform for a duty cycle of 1/1600 using this method. The feathered PWM waveform signal is shown zooming in from top to bottom of Figure 4.2.

Not only did this method still not provide enough resolution but it also made the motors audibly noisy. It is possible that feathering the PWM with a 100kHz interrupt in the same manner as Hood, may improve the resolution and eliminate the audible noise whilst maintaining a high PWM frequency. A high PWM frequency is desirable because ripple current losses are lowered with increasing PWM frequency. However, as PWM frequency is increased, switching losses of the MOSFET's are also increased. A compromise needs to be established through testing. It is also important for PWM frequency to be greater than 25kHz, as this is the approximate audible limit for the human ear. At the present frequency of 15.4kHz, the motor's emit a slightly irritating "whine".

This problem requires rectification. Unfortunately due to time constraints interrupt processing has not yet been fully established in the current software, so feathering at 100kHz could not be tested. This is proposed for future work following this thesis.

4.1.2.2 `read_curr()`

This function simply takes as parameters the motor number for which armature current is desired and returns the armature current in mA. Based upon the motor number from 0 to 2, an ADC queue is setup for the corresponding H-bridge legs and the conversions commence. The larger of the two readings is then selected as the correct armature current conversion. The 10 bit conversion is then multiplied by a calibration factor and returned as an unsigned integer.

At present this function has a calibration multiplier of 5 to convert the 10 bit ADC reading to an amount in mA. This calibration factor is inaccurate. Once calibration of current sensing is completed, a more appropriate calibration factor will be inserted.

4.1.2.3 `read_enc()`

This function is passed as a parameter the motor number from 0 to 2, for the corresponding desired encoder channel and the current encoder count is returned as an unsigned integer. This function also makes use of tpu.c.

4.1.2.4 `transmitShort()` and `transmitChar()`

`transmitShort`() and `transmitChar()` were written for debugging purposes. When passed a *short* parameter, transmitShort(), will output to the SCI the corresponding ASCII character set for its hexadecimal value. `transmitChar()`, will do the same for *character* variables.

The output hexadecimal numbers can be viewed using a program such as Microsoft Hyperterminal as detailed in section 3.4.

4.1.2.5 `outputToLEDs()`

`outputToLEDs()` is a function that can be used to output an entire character to the LEDs for debugging purposes. Because the MOSFET driver lines were re-routed, the upper three LED's are no longer connected to PORT E of the SIM module.

`outputToLEDs()` performs some bit shifting so that the uppers three bits can be output to their newly routed PORT F locations. Board 6 did not require re-routing and hence when this command is used with the DIP switches defining board 6, characters are directly output to PORT E.

4.1.3 Additional Code for the New Processors

In order to purely demonstrate functionality of the boards only a simplified version of can.c was written. In its present state, this source file only contains an initialisation routine for the TouCAN module and a function that checks for incoming CAN messages. Transmission code fragments are spread throughout the demonstration software where CAN transmission is required. Finalisation of can.c is proposed for future work following this thesis. The basic can.c currently in place will provide a good framework for its development.

tpu.c and intr.c were ported from RoboRoo's software and slightly modified for the 2004 boards. They handle the TPU functionality and interrupt processing respectively. Interrupt processing software remains incomplete. Due to time constraints, external interrupts have not yet been implemented. The 2004 boards have been routed so that the push buttons and encoder index lines connect directly to external interrupt lines. At present these lines are only polled, which can lead to them sometimes not being recognised in software. This is also proposed for future work.

mc68376.h is another source file that was adapted from RoboRoo's software. This header file defines addressing for the various registers of the 68376 modules. It has been adapted to specifically suit the 68376. Figure 4.3 shows the register map for the 68376 which mc68376.h defines. Note that figure 3.4 of the Motorola MC68376 datasheet[5] contains an error. It shows the TouCAN and ADC modules listed in the register map in the wrong order. This error has been corrected in Figure 4.3.

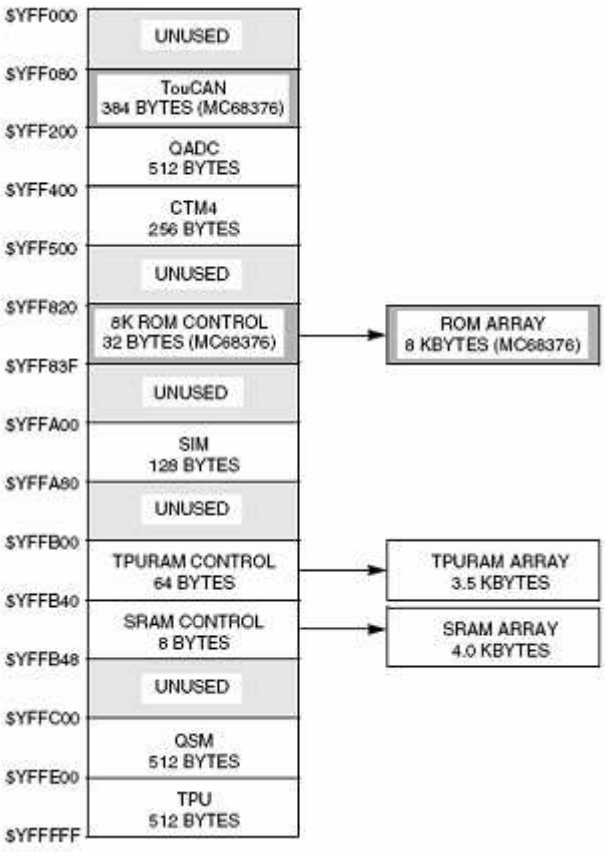


Figure 4.3: Register Addressing – Mapped by mc68376.h. [5]

The defs.h source file was created to define all global constants and new data types.

4.2 Initialisation Code

As discussed prior, GuRoo was in need of an initialisation routine for joint alignment on power up. Code has successfully been written and tested for an initialisation routine that uses the encoder index pulses. This section details how this routine has been implemented.

4.2.1 Index Position Control

The method developed for initialisation basically performs position control of the motors about index encoder pulses. Figure 4.4 is a flowchart describing how this method has been implemented in software. This software utilises a CAN message buffer on each board for commanding the initialisation process. The initialisation routine is contained in startup.c.

On reset, each board runs through the initialisation sequence detailed in section 4.1.1. Flow of control is then passed to a continuous loop in the `main()` function which continuously checks for incoming CAN command messages. Upon reception of a message to commence index positioning, the periodic interrupt timer(PIT) commences to trigger the `posnCtrlIndex()` function at 1kHz.

`posnCtrlIndex()` serves two functions. When this function is first called it slowly increments the PWM duty cycle for each of the motors until they begin to move forward. The forward direction is defined as anticlockwise rotation of motor shafts. Once moving forward the duty cycle ceases to increment and the motors rotate until an index pulse is detected. Index pulses are continuously polled in between calling the `posnCtrlIndex()` function during the main loop. Once detected, the index positions are read from the encoders and `posnCtrlIndex()` switches its functionality to PI position control of motors about their index positions.

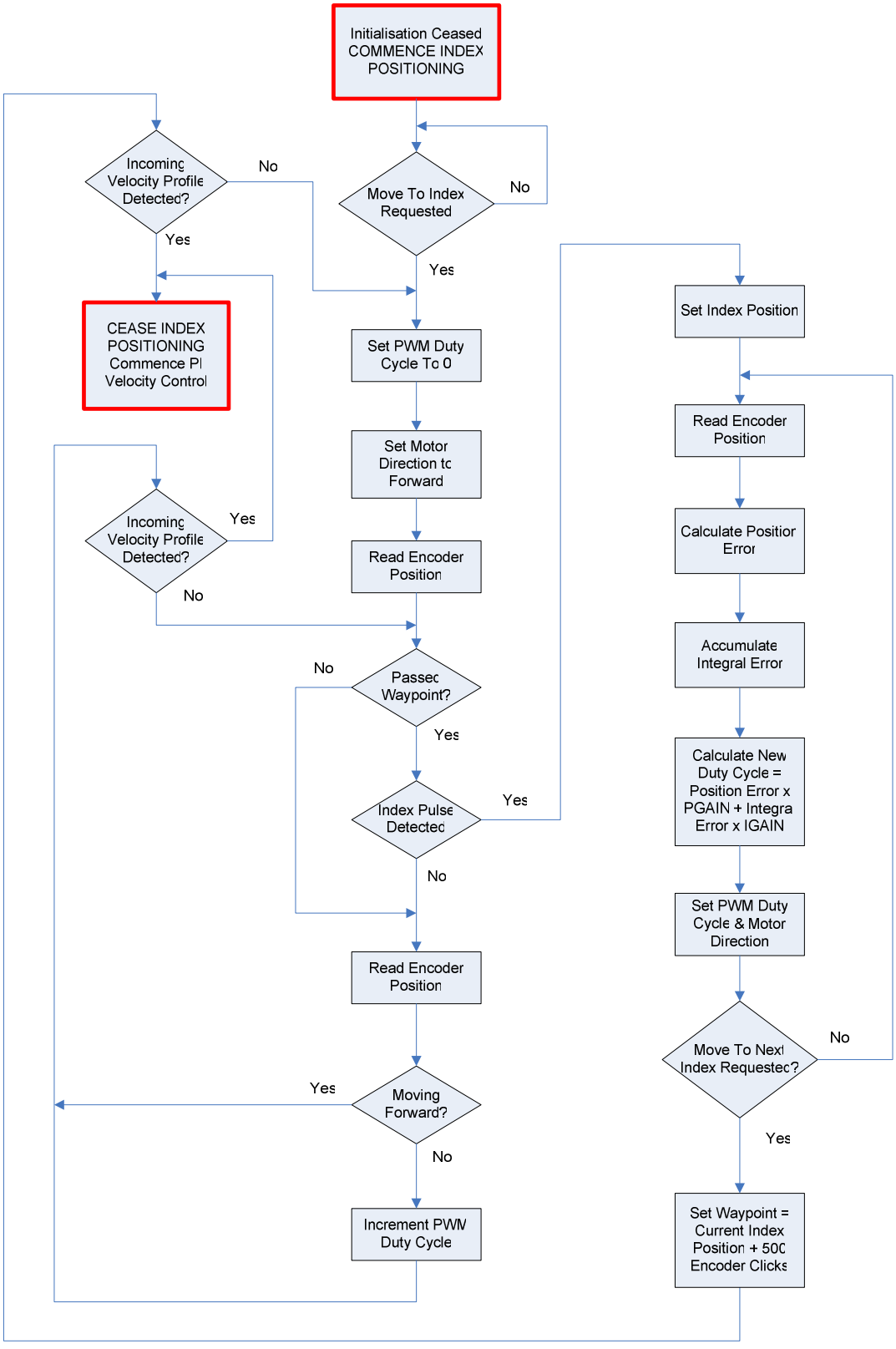


Figure 4.4: Flowchart for Index Positioning Initialisation.

The PI position control loop operates in a similar manner to PI velocity control. For each motor/encoder system, `posnCtrlIndex()` first reads the current encoder position and calculates the error between current position and index position. The accumulation of this error effectively integrates the position error. Multiplication of these errors by suitable gains gives an appropriate duty cycle that is then set by `set_PWM()` to maintain the motors position about its corresponding index pulse. These gains were arbitrarily selected through trial and error. As mentioned prior, the current low PWM frequency requires attention. If a version of feathering is implemented or if the PWM frequency is changed then suitable gains will need to be reselected. This is proposed for future work.

In the event that the processor receives a CAN message to move a particular motor to its next index position `posnCtrlIndex()` breaks out of position control and begins moving to the next forward index pulse. Immediately a waypoint is set 500 encoder clicks forward of the current encoder position to avoid confusion with the current index pulse. Duty cycle is again incremented until the motor is driving forward and `posnCtrlIndex()` continuously checks the current encoder reading. Once the encoders indicate that the waypoint has been passed, searching for the new index pulse commences. Once detected, `posnCtrlIndex()` recommences position control about the new index pulse position.

4.2.2 Index Position Control Performance

The processing time for `posnCtrlIndex()` was measured to ensure that the loop frequency of 1kHz was appropriate. This was measured by turning on and off an LED before and after calling the `posnCtrlIndex()` function. Figure 4.5 shows the oscilloscope output viewing this LED. As can be seen, the position control loop requires 252us of processing time. This proves that running the loop at a frequency of 1kHz is not too fast for the processor to handle.

Testing showed that this index positioning technique would consistently locate the exact index position. This was proved by using an oscilloscope to probe the index input lines to the processor whilst the motors were under index position control.

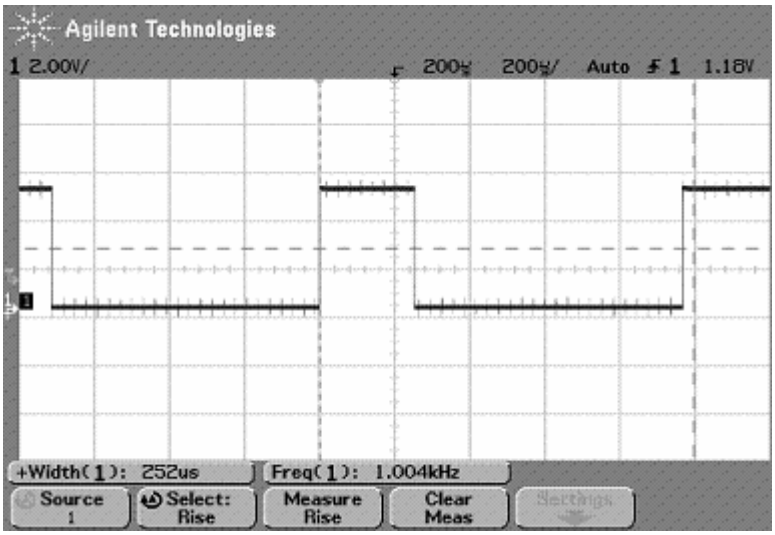


Figure 4.5: Position Control Loop Processing Time.

Repeatability was verified by outputting index position encoder counts for consecutive index positions of GuRoo's left knee joint to an external PC. Table 4.1 lists a sample of the encoder index positions that were transmitted through the SCI. The index encoder counts are consistently 2000 counts apart, which is the exact encoder count distance between index pulses. As indicated by the second sample, because index detection is only polled, this technique has the tendency to sometimes miss the index pulse.

Motor Index Encoder Count (Hexadecimal)	Motor Index Encoder Count (Decimal)	Index Encoder Count Increment (This – Previous)
0217	535	NA
11B7	4535	4000
1987	6535	2000
2157	8535	2000
2927	10535	2000
30F7	12535	2000
38C7	14535	2000
4097	16535	2000
4867	18535	2000
5037	20535	2000

Table 4.1: Repeatability of Index Positioning.

It is anticipated that once external interrupt processing is implemented, software will be written to use interrupt processing to detect index pulses. This will eliminate the current problem where index pulses are sometimes missed.

Verification of index positioning repeatability was also proved visually by using a scribed mark on the ankle joint of GuRoo's left leg. The joint would consistently manoeuvre to the same index position, observed from the parallel scribed marks shown in Figure 4.6.

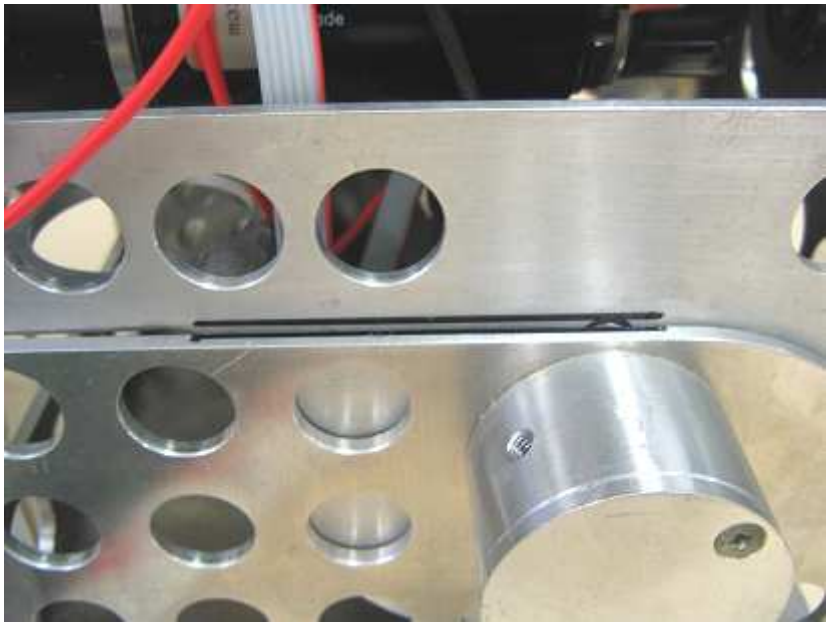


Figure 4.6: Visual Verification of Index Positioning Repeatability with a Scribed Mark.

4.3 Demonstration Software

4.3.1 Control Loop Demonstration Code

In order to demonstrate correct functionality of the new software, a simplified version of board1.c was written to implement PI velocity control. A new 2004 board was installed into a detached leg of GuRoo. Another external board was connected for CAN communication to generate an input velocity profile that was updated every 50Hz, in line with the existing system in GuRoo. This velocity profile consisted of a sin wave trajectory with an amplitude of 10 encoder counts per control loop and period 8s. The board in GuRoo's leg was wired for running three motors

simultaneously. The DIP switches were utilised for CAN message identification and software sharing. The installed board's DIP switches were set to 0x1(board 1), and the external board's switches were set to 0x7(board 7).

Following power up the index positioning routine was initiated by pressing push button one on board 7. This transmitted a command message to board 1 to initialise all three joints to their nearest forward index positions. Push button two on board 7 could be repeatedly pressed to transmit a command message to move the knee joint to its consecutive index positions. Pressing push button one again, commenced transmission of the velocity profile. Upon reception of this incoming velocity profile on board 1 the PI velocity control loop would commence. All three motors were configured to use this velocity profile as the desired velocity for PI control. The knee joint was programmed to run the inverse of the velocity profile, giving a crouching effect to the leg. Figure 4.7 shows a still shot of the leg manoeuvring the three lower motors through this trajectory.

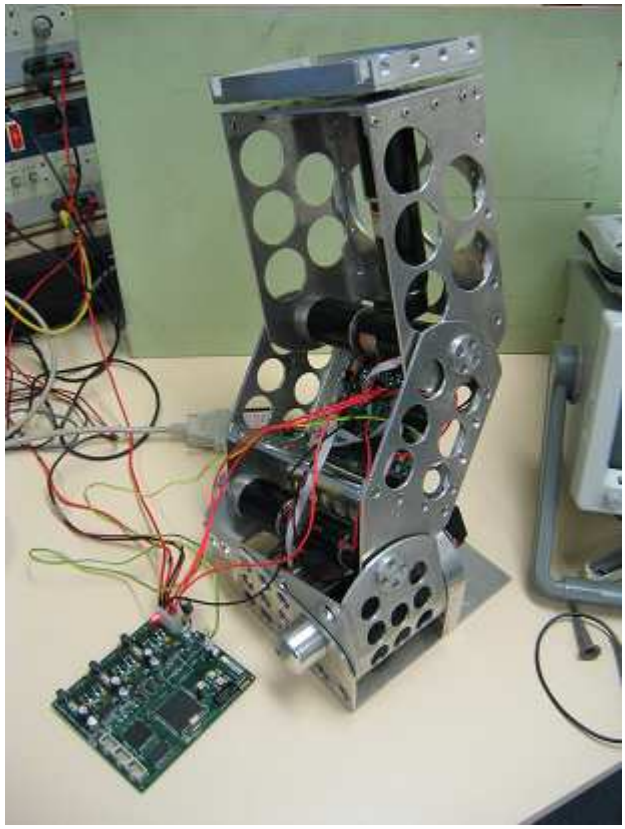


Figure 4.7: Demonstrating the New Software – Initialisation and Velocity Control.

This demonstration software verified correct operation of all of the board functionality achieved in this thesis, except for AD conversions.

4.3.2 PI Velocity Control Loop Processing Time

Section 2.4.1 discussed that a major flaw for the 2001 controllers was their limited control loop speed due to the use of external quadrature decoders. The QD capabilities of the 68376's TPU eliminated this problem. A simplified version of board1.c was written to implement the PI velocity control loop in the demonstration software.

The time taken for processing this velocity control loop was measured on an oscilloscope by turning on and off an LED before and after `b1_control()` was called from the main loop. Figure 4.8 shows that the control loop took 208us to process.

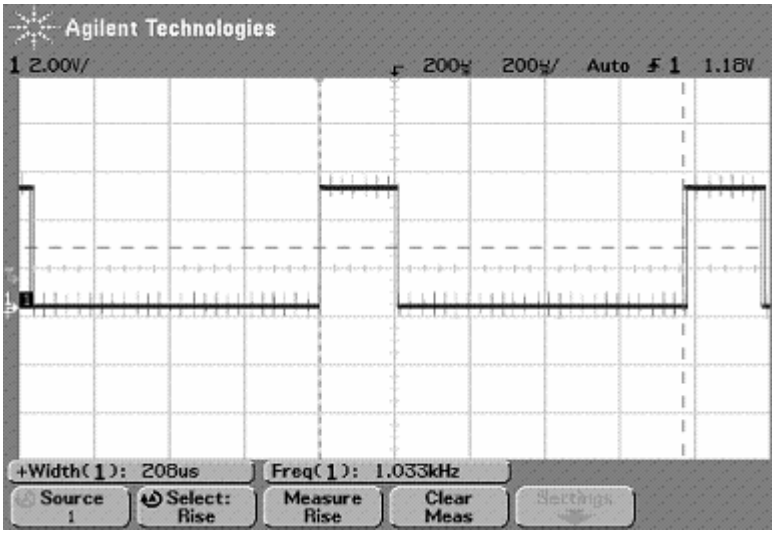


Figure 4.8: 1kHz PI Velocity Control Loop Processing Time(No AD Conversions).

The control loop used in the demonstration software did not however include any AD conversions. These needed to be included in order to fully simulate the actual control loops intention. The software was altered to include `read_curr()` for each motor in `b1_control()`. Figure 4.9 shows the altered processing time of 402us for the control loop, including the six AD conversions required to read the current flowing through all three H-bridges. This is a drastic improvement on the 1.28ms taken for the 2001 controllers to run a control loop. The processing time of 402us proves that the

original target of 2kHz is achievable on the new 2004 controllers, with ample time to spare for CAN transmission of feedback data.

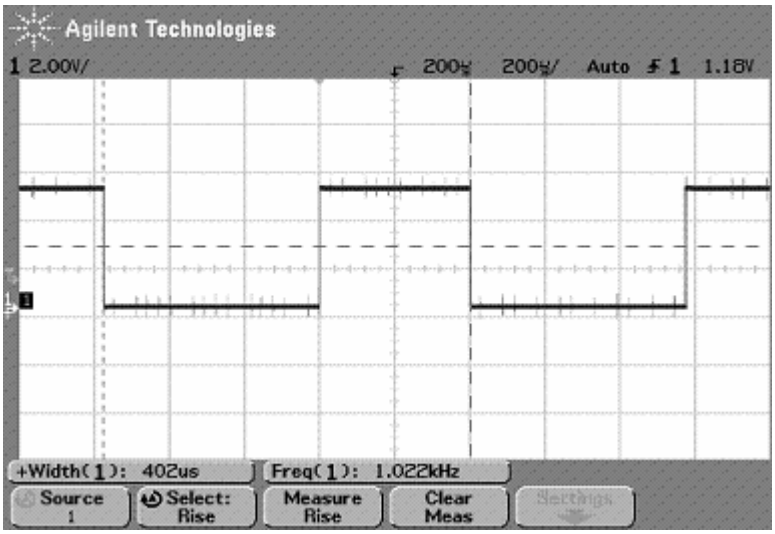


Figure 4.9: 1kHz PI Velocity Control Loop Processing Time(6 AD Conversions Per Loop).

A comparison of the control loop processing time with and without AD conversions shows a difference of 194us. This is almost half of the total control loop processing time, indicating that AD conversions are the bottleneck for required control loop processing time.

4.4 Memory Consumption

The compiler outputs memory utilisation data after the successful compilation and downloading of programs to the 68376. It was observed that the demonstration software only used 9.4kB of flash memory and 1.2kB of SRAM. The new boards are equipped with 256kB of program memory space(flash) and 512kB of variable memory space(SRAM). The memory utilisation of the demonstration software therefore suggests that even after the CAN software for the new 2004 boards is completed, there will still be ample room for further development of software for the new controllers.

5. Thesis Outcomes and Conclusion

Desired functionality for the new controllers was completely achieved by early September. Implementation of this functionality is yet to be finalised for CAN communication and current sensing.

After approximately 150 hours of soldering, five new control boards were completely populated. Figure 5.1 shows one of the new 2004 lower limb controller boards.

By late October, the software was fully updated to suit the Motorola 68376 with the exception of can.c and an intialisation routine was implemented to position the robot's joints to index positions. The intialisation routine was proven to be accurate and repeatable, with the exception of sometimes missing index pulses due to polling their detection.

Demonstration software was developed to verify correct functionality of the new boards and the new software written for them. The demonstration software proved that the new boards were able to run the velocity control loop at a speed of 1kHz. Much faster than the control speed of 250Hz on the 2001 control boards. It was shown that attaining a control loop speed of 2kHz is highly possible.

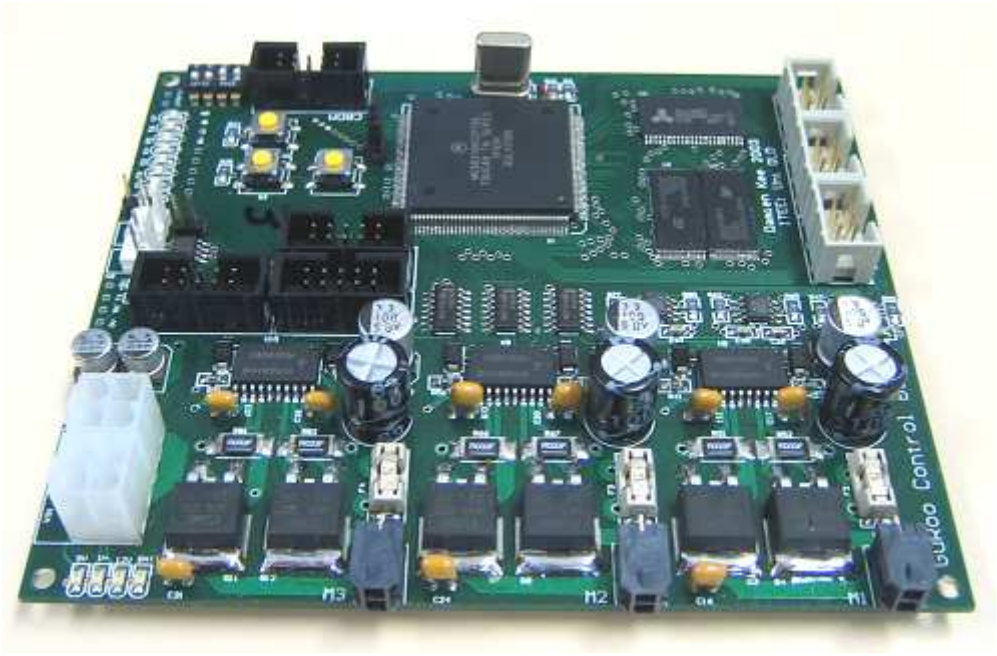


Figure 5.1: The New 2004 Lower Limb Controller Board.

6. The Road Ahead

The new boards only require a minor amount of further work before they may be installed in GuRoo and function with the existing gait systems. There are however further issues that require attention before the full potential of the new controllers can be utilised.

It was discussed that the MOSFET's are triggered into a shoot-through condition if the reset push button is pressed or if motor voltage is turned on before the MOSFET driver's are enabled. This problem requires prompt attention. It is potentially dangerous to GuRoo and those surrounding it if a shoot-through condition is accidently triggered whilst GuRoo is operating. GuRoo's joints could potentially draw full current and rapidly extend or collapse.

It is anticipated that a new current sense resistor value will be chosen to give greater resolution of armature current feedback, closer to the typical operating current range. Alterations to current sensing circuitry need to be finalised before calibration of current sensing can commence. In order to avoid damaging the motors, calibration of current sensing is required so that the current joint control routine can provide adequate feedback of armature current.

Correlation of armature current and torque also requires testing. This will provide a means for the implementation of torque feedback control. An appropriate method for calibration of current sensing and torque correlation was suggested in section 3.6.5.

Due to time constraints, interrupt processing software could not be finalised as part of this thesis. Implementation of an external interrupt routine for index pulse detection will eliminate the problem where index pulses can be missed. The use of external interrupts will also be beneficial for push button functionality and CAN message reception.

Index positioning software has only been written to increment index positioning of GuRoo's joints. Enabling GuRoo's joints to move both forward and backwards between index pulses would be extremely beneficial. It would not be difficult to

further develop the intialisation routine to do so. It is envisaged that the index positioning eventually be controlled from an external PC. Messages could be generated from a GUI controller on the PC and serially transmitted across to board 6 to implement CAN command control of index positioning.

The limited resolution of PWM duty cycle also requires attention. It is recommended that a feathering routine similar to that used by Hood be trialled. This will eliminate the current problem with audible noise and give smoother control where low duty cycles are required.

It is recommended that the control loop speed be maximised once the duty cycle issue is resolved. An analysis of control loop performance can then be performed. It is anticipated that joint control position error will be significantly reduced with an increased control loop speed. New optimum joint gains will need to be calculated when the control loop speed is increased.

References

[1] Kee D., Wyeth G., *Cerebellar Joint Compensation for a Humanoid Robot*, in *Department of Information Technology and Electrical Engineering.* 2004, University of Queensland: Brisbane.

[2] Kee D., *Official GuRoo Website.*
http://www.itee.uq.edu.au/~damien/GuRoo
(Accessed: 21/4/2004)

[3] Hood A., *Distributed Motion Controllers for a Humanoid Robot*, in *Department of Information Technology and Electrical Engineering.* 2002, University of Queensland: Brisbane.

[4] Stirzaker J., *Design of DC Motor Controllers for a Humanoid Robot*, in *Department of Information Technology and Electrical Engineering.* 2001, University of Queensland: Brisbane.

[5] *MC68336/376 User's Manual*, 15 Oct 2000, DATASHEET, Motorola Inc.

[6] Drury A., *Gait Generation & Control Algorithms for a Humanoid Robot*, in *Department of Information Technology and Electrical Engineering.* 2002, University of Queensland: Brisbane.

[7] Matthews-Frederick S., *Embedded Hardware for a Humanoid*, in *Department of Information Technology and Electrical Engineering.* 2004, University of Queensland: Brisbane.

[8] Robocup, *Official Robocup Website*.
http://www.robocup.org
(Accessed: 21/4/04)

[9] Kee D., Wyeth G., Hood A., Drury A., *GuRoo: Autonomous Humanoid Platform for Walking Gait Research*, in *Department of Information Technology and Electrical Engineering.* 2004, University of Queensland: Brisbane.

[10] Darley S., Maiolani M., Melear C., *An Introduction to the MC68331 and MC68332.* 1996, Motorola Inc.

[11] Dobbin A., *Stereo Audio Transmission Over the CAN Bus Using The Motorola 68376 With TOUCAN Module(AN1776/D)*. Rev 1.0, July 10, 1998, Motorola Inc.

[12] Kennedy J., *Design and Implementation of a Distributed Digital Control System in an Industrial Robot*, in *Department of Computer Science and Electrical Engineering.* 1999, University of Queensland: Brisbane.

[13] Turk D., *Mobile Robot Electrical Design*, in *Department of Information Technology and Electrical Engineering*. 2004, University of Queensland: Brisbane.

[14] Alliance Semiconductor, *AS7C4098 Datasheet*, 23/05/02(v1.8), Alliance Semiconductor Corporation.

[15] Cartwright, T., *Design and Implementation of Small Scale Joint Controllers for a Humanoid Robot,*in *Department of Information Technology and Electrical Engineering*. 2001, University of Queensland: Brisbane.

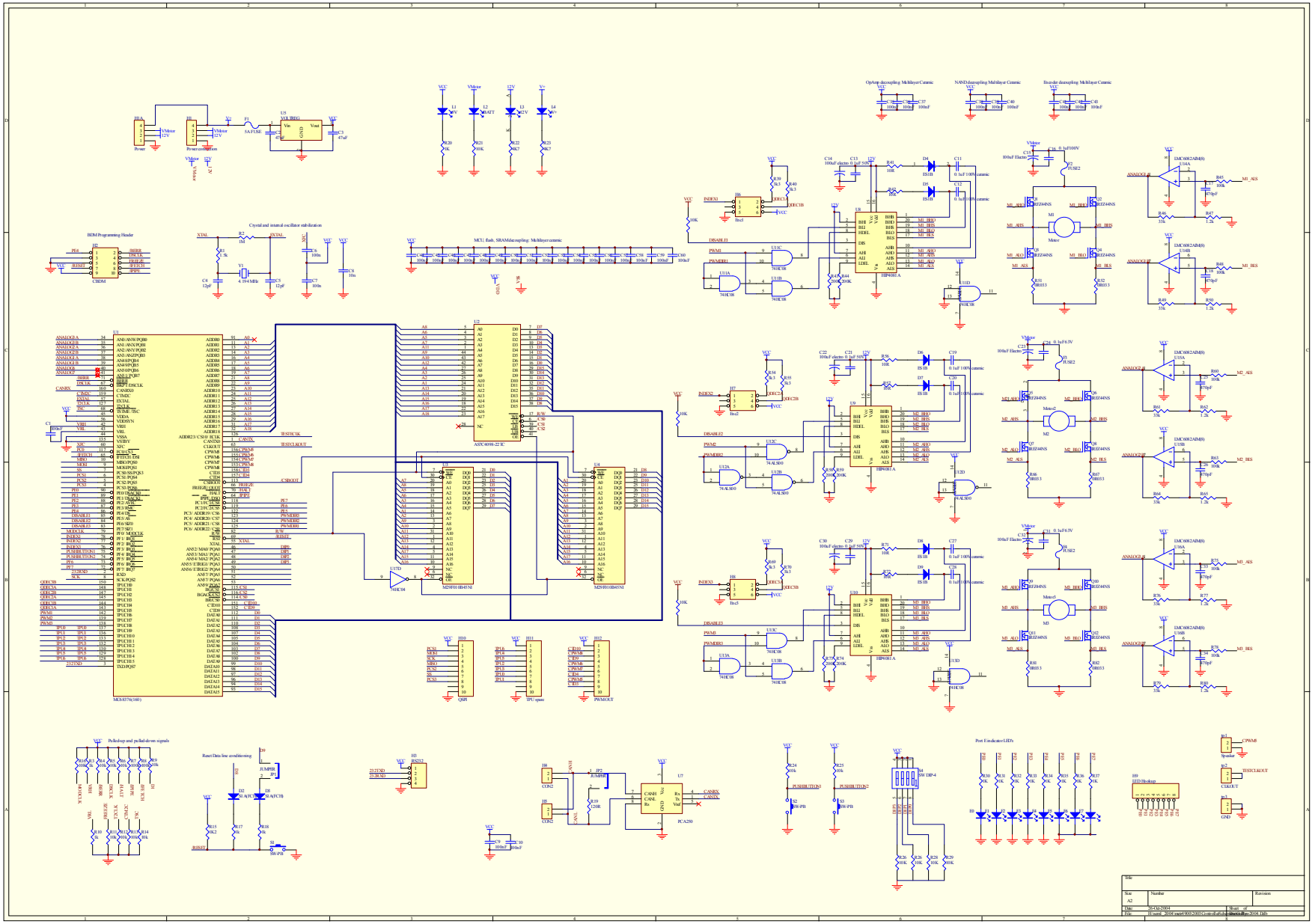
[16] Mohan, Undeland, Robbins, *Power Electronics*, (3rd Edition). 2004, John Wiley & Sons, Inc. NJ, USA.

[17] Kee D., *Drive System Selection and Simulation for a Humanoid Robot,* in *Department of Information Technology and Electrical Engineering*. 2001, University of Queensland: Brisbane.

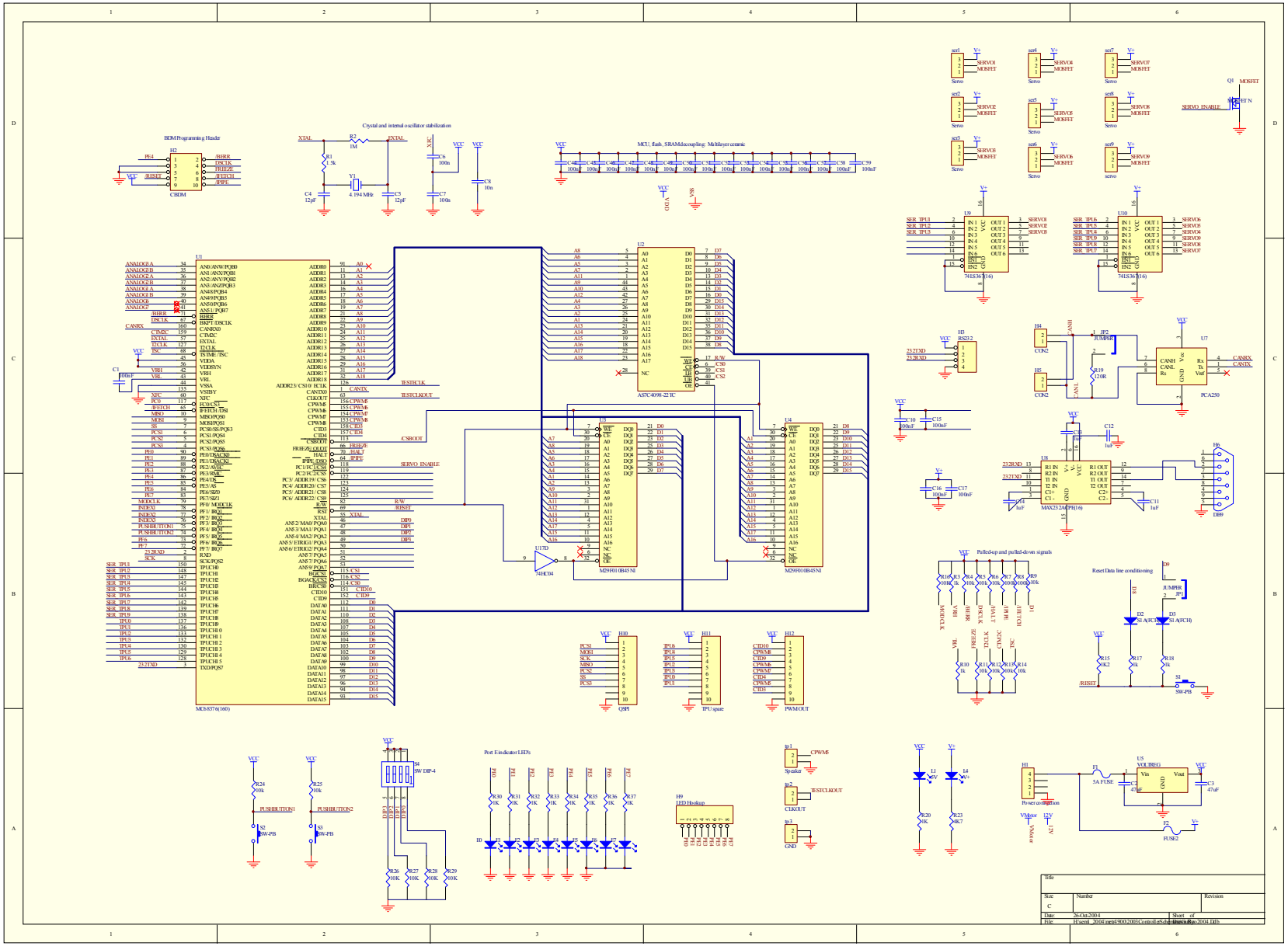
[18] International Rectifier, *IRF530NS Datasheet*, 9/04/02(PD-91352B).

[19] National Semiconductor, *LM2940/LM2940C Datasheet*, March 2000, National Semiconductor Corporation.

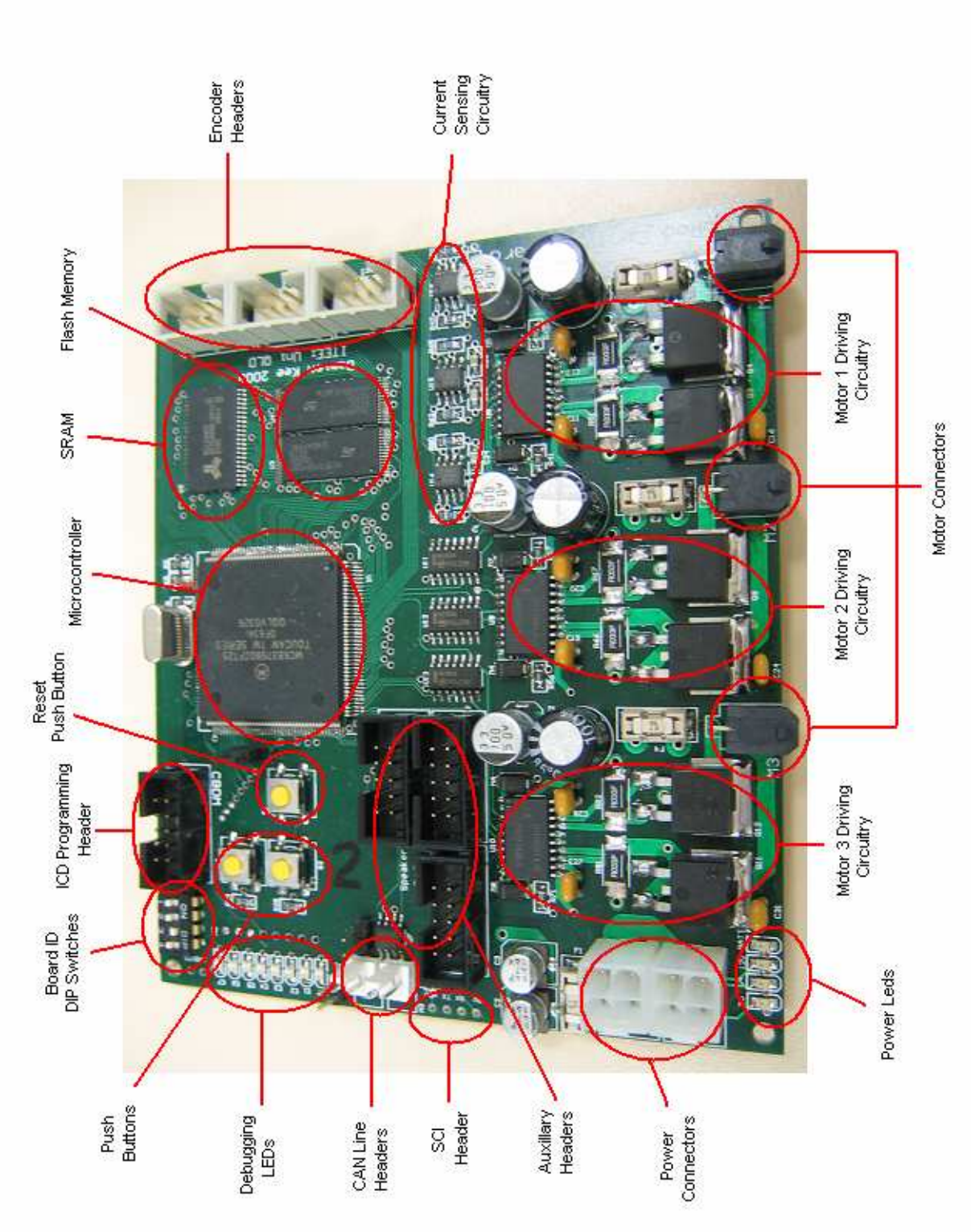
Appendix A - Updated 2004 Controller Board 1-5 Schematic



Appendix B - Updated 2004 Controller Board 6 Schematic



Appendix C – Board Layout & Placement



Appendix D - 2004 Controller Software

```
/*****************************************************************************

  File:    startup.c

  Author:  Simon Hall

  Created: 13/10/04 - Modification of Adam Drury & Andrew Hood's code
           for 2001 lower limb controllers to suit GuRoo's 2004 lower
           limb controller boards(Motorola 68376). Assistance provided
           by Doug Turk for TPU RAM programming routine.

  Summary: For GuRoo's 2004 lower limb controller boards.
           Initialises Motorola 68376 processor for desired
           functionality. Contains "main" loop that calls on other
           files for functionality and control loop code. This startup
           routine is intended for all board switch settings. Currently
           contains code to enable a separate board which when DIP
           switches are set to 0x7 will transmit over the CAN line,
           initialisation to encoder index commands and a sin wave
           velocity profile for PI control. (On Board 7)Press PB1 to
           move to nearest index. Press PB2 to advance to next indexes.
           Press PB1 again to start velocity profile sequence. Pressing
           PB1 finally will set velocity profile permanently to zero.

***************************** INCLUDE *************************************/

#include "startup.h"
#include "mc68376.h"
#include "board1.h"  // This will require updating to reference in "common"
#include "can.h"
#include "intr.h"
#include "tpu.h"
#include "lowlevel.h"


/***************************** CONSTANTS ***********************************/

#define P_POSN_GAIN     10//(pErrorS/10)
#define I_POSN_GAIN     800  //(iErrorS/800)


/***************************** GLOBALS *************************************/

// Board ID set by DIP switches
uchar boardID = 0;

// counter to prescale PIT so b1_control occurs every 256Hz
int countPIT = 0;

// "feathering" duty cycle parameters
//extern int counter1;
//extern int counter2;
//extern int counter3;
//extern int pwm_high_1;
//extern int pwm_low_1;
//extern int pwm_high_2;
//extern int pwm_low_2;
//extern int pwm_high_3;
//extern int pwm_low_3;

// variables used by index position control
uchar foundIndex[3] = {FALSE, FALSE, FALSE};
uchar passedWaypoint[3] = {TRUE, TRUE, TRUE};
int wayPoint[3] = {0, 0, 0};
int oldPosnS[3] = {0, 0, 0};
int indexPosn[3] = {0, 0, 0};
int iErrorS[3] = {0, 0, 0};
int reqdPWMS[3] = {0, 0, 0};
uchar commencePI = FALSE;
uchar findIndex = FALSE;
```

```
int p;


// Variables used by board 7:-
int commandCount = 0;
int profIncr = 0;
int velGen = 0;
uchar stop = FALSE;


// Variables for data retrieval:-
int dataLog[2000] = {0};
int dataLogPointer = 0;
int dp = 0;


/*
 * main:-
 *
 */
void main(void) {

  initCPU();

  // Read board DIP switches & define board number
  boardID = QADC_REGS.PORTQA & 0x0F;
  // Display board ID
  outputToLEDs(boardID);

  setupCAN();

  // board 7 used as external board for velocity profile generation
  if (!(boardID==BOARD_7)) {
    setupADC();
    initTPU();
    setupPWM();
    setupQDEC();
  }
  setupPIT();
  initSCI();

  //initGains(); [after board1.c is updated]


  while (1) {

    if (!(boardID==BOARD_7)) {

      // check for incoming CAN messages
      chkForMsgs();

      // After 32 PIT interrupts b1_control() is processed.
      // This translates to 256Hz to comply with existing gains.
      // Change count condition if using feathering(to 32)
      // (At present feathering not implemented but format remains)
      // (Currently running 1kHz control loops)
      if ((countPIT>=1)&&(commencePI==TRUE)) {

        countPIT = 0;

        b1_control();

      } else if (findIndex==TRUE) {

        int n;
        for(n=0; n<3; n++) {
          // poll for index pulses
          if ((SIM_REGS.PORTF&(1<<(n+1)))&&(foundIndex[n]==FALSE)
                &&(passedWaypoint[n]==TRUE)) {

            // read index position
            indexPosn[n] = (int)(read_enc(n));

            if (n==2) {
              // Transmit index posn to board 7 for serial out
              CAN_REGS.MSGBFR[2].CTRL = (CAN_REGS.MSGBFR[2].CTRL&0xFF0F)+0x0080;
              CAN_REGS.MSGBFR[2].IDHIGH = 0x0000 + (BOARD_7<<5);
```

```
            CAN_REGS.MSGBFR[2].DATA[0] = indexPosn[n];
            CAN_REGS.MSGBFR[2].CTRL = (CAN_REGS.MSGBFR[2].CTRL&0xFF00)+0x00C4;
          }

          set_PWM(n, 0);
          foundIndex[n] = TRUE;
        }
      }

      // Change count condition if using feathering(to 8)
      if (countPIT>=1) {

        countPIT = 0;

        posnCtrlIndex();

      }

    }

  } else {

    // Board 7 Functionality:-

    chkForMsgs();

    // polling routine for PB1
    if (!(SIM_REGS.PORTF & 0x10)) {
      delayPB();
      while (!(SIM_REGS.PORTF & 0x10));
      // PB routine starts here

      commandCount++;

      if ((commandCount>2)||(stop==TRUE)) {

        // begin transmitting 0 velocity
        commandCount = 0;
        stop = TRUE;

        // send serial data output to PC
        for (dp=0; dp<dataLogPointer; dp++) {
          transmitUShort((short)(dataLog[dp]));
        }


      } else if ((commandCount==1)&&(stop==FALSE)) {

        // Transmit command message to board 1 for index positioning
        CAN_REGS.MSGBFR[2].CTRL = (CAN_REGS.MSGBFR[2].CTRL&0xFF0F)+0x0080;
        CAN_REGS.MSGBFR[2].IDHIGH = 0x0000 + ((BOARD_1|COMMAND_CONTROL)<<5);
        CAN_REGS.MSGBFR[2].DATA[0] = MOVE_ALL_TO_INDEX;
        CAN_REGS.MSGBFR[2].CTRL = (CAN_REGS.MSGBFR[2].CTRL&0xFF00)+0x00C4;

      } else if ((commandCount==2)&&(stop==FALSE)) {
        // Commence sending velocity profile(start PIT)
        SIM_REGS.PICR |= 0x0400;
      }

      // PB routine ends here
      delayPB();
    }

    // polling routine for PB2
    if (!(SIM_REGS.PORTF & 0x20)) {
      delayPB();
      while (!(SIM_REGS.PORTF & 0x20));
      // PB routine starts here

      if ((commandCount==1)&&(stop==FALSE)) {
        // Transmit command message to board 1(move motor 3 to next index)
        CAN_REGS.MSGBFR[2].CTRL = (CAN_REGS.MSGBFR[2].CTRL & 0xFF0F)+0x0080;
        CAN_REGS.MSGBFR[2].IDHIGH = 0x0000 + ((BOARD_1|COMMAND_CONTROL)<<5);
        CAN_REGS.MSGBFR[2].DATA[0] = MOVE_TO_NEXT_INDEX3;
        CAN_REGS.MSGBFR[2].CTRL = (CAN_REGS.MSGBFR[2].CTRL&0xFF00)+0x00C4;
      }
```

```
          // PB routine ends here
          delayPB();
        }

      }

    }

}


/*
 * posnCtrlIndex:-
 *
 * PI position control loop. Hones in on index pulse.
 *
 */
void posnCtrlIndex(void) {

  int newPosnS;
  int pErrorS;
  int chkPassedWPVar;

  int m;
  for (m=0; m<3; m++) {

    // index not found, slowly move forward till found
    if (foundIndex[m]==FALSE) {

      // get current posn
      newPosnS = (int)(read_enc(m));

      pErrorS = newPosnS - oldPosnS[m];

      oldPosnS[m] = newPosnS;

      // check for overflow
      if (pErrorS>32767) {
        pErrorS -= 65536;
      } else if (pErrorS<-32767) {
        pErrorS += 65536;
      }

      // if not moving forawrd, rmap up duty cycle
      if (pErrorS<=0) {

        reqdPWMS[m]++;

        // limit PWM
        if (reqdPWMS[m]>1000) {
          reqdPWMS[m] = 1000;
        }

        set_PWM(m, reqdPWMS[m]);
      }

      // check if passed waypoint. If so, start checking for index again
      chkPassedWPVar = newPosnS - wayPoint[m];
      if ((chkPassedWPVar<-32767)||((chkPassedWPVar>0)
            &&(chkPassedWPVar<32767))) {
        passedWaypoint[m] = TRUE;
      }

    // index found commence position control
    } else if (foundIndex[m]==TRUE) {

      pErrorS = indexPosn[m] - (int)(read_enc(m));

      if (pErrorS>32767) {
        pErrorS -= 65536;
      } else if (pErrorS<-32767) {
        pErrorS += 65536;
      }

      iErrorS[m] += pErrorS;

      reqdPWMS[m] = pErrorS/P_POSN_GAIN + iErrorS[m]/I_POSN_GAIN;
```

```
        // limit PWM
        if (reqdPWMS[m]>1000) {
          reqdPWMS[m] = 1000;
        } else if (reqdPWMS[m]<-1000) {
          reqdPWMS[m] = -1000;
        }

        set_PWM(m, reqdPWMS[m]);

      }

    }

}


/*
 * PIT_intr:-
 *
 * Periodic interrupt routine. Executes every 1.024kHz.
 * (~50Hz for board 7 velocity generation)
 *  Does not implemet feathering at present.
 * (Feathering sections hashed out)
 * This can be improved once CTM has been set up.
 * (Beef up to 100kHz and do as before with feathering?)
 *
 */
void interrupt PIT_intr(void) {

   if (boardID==BOARD_1) {

/*   // If using feathering:-
     if (countPIT<counter1) {
        // calls PWM_set function in tpu.c
        tpuPWM_set(PWM1CH, pwm_high_1);
     } else {
        tpuPWM_set(PWM1CH, pwm_low_1);
     }

     if (countPIT<counter2) {
        tpuPWM_set(PWM2CH, pwm_high_2);
     } else {
        tpuPWM_set(PWM2CH, pwm_low_2);
     }

     if (countPIT<counter3) {
        tpuPWM_set(PWM3CH, pwm_high_3);
     } else {
        tpuPWM_set(PWM3CH, pwm_low_3);
     }
*/
     // increment prescaler count
     countPIT++;

     // Board 7 sends sin wave velocity profile
   } else if (boardID==BOARD_7) {

     // segment sine wave period into 400 increments
     if (profIncr>=400) {
        profIncr = 0;
     }

     profIncr++;

     if (commandCount==2) {
        // Calculate new velocity to send
        // period = 8s, omega = 2*pi*50Hz, peak = 10 enc/loop
        velGen = roundValue(10*sin(0.015707963*profIncr));
     } else {
        // stop button pressed on board 7 (PB2)
        velGen = 0;
     }

     // Set buffer 1 as transmit buffer and send new velocity to other board
     CAN_REGS.MSGBFR[2].CTRL = (CAN_REGS.MSGBFR[2].CTRL & 0xFF0F)+0x0080;
     CAN_REGS.MSGBFR[2].IDHIGH = 0x0000 + (BOARD_1<<5);
```

```
    CAN_REGS.MSGBFR[2].DATA[0] = velGen;
    CAN_REGS.MSGBFR[2].CTRL = (CAN_REGS.MSGBFR[2].CTRL & 0xFF00)+0x00C4;

  }

}


/*
 * initCPU():-
 *
 * Initialises boards. Sets up IO registers.
 */
void initCPU(void) {

  // Disable interrupts
  disable();

  // Turn watchdog off
  SIM_REGS.SYPCR = 0x00;

  // Wait while clock stabilises
  #define SLOCK 8
  while (!(SIM_REGS.SYNCR & SLOCK));

  // Initialise interrupt table
  intr_init();

  // Set PORT E (LEDs) DDR to output, I/O mode and init
  // & disable MOSFET drivers
  SIM_REGS.DDRE = 0xFF;
  SIM_REGS.PEPAR = 0x00;
  SIM_REGS.PORTE = 0xE0;

  // Setup PORTC pins 1 to 6 for output function(PWMDIR & LED's 5-7)
  // and clear PORTC pins
  SIM_REGS.CSPAR[0] &= ~(0x3C00);
  SIM_REGS.CSPAR[1] &= ~(0x00FF);
  SIM_REGS.PORTC = 0x00;

  // Set PORT F (Index & PBs) pins to input, I/O mode and init
  SIM_REGS.DDRF = 0x00;
  SIM_REGS.PFPAR = 0x00;
  SIM_REGS.PORTF = 0x00;

  // Setup PORTQA to input for board ID DIP switches
  QADC_REGS.DDRQA &= 0x00FF;  // clear upper 8 bits

  // Enable all interrupts
  enable();

}


/*
 * setupADC:-
 *
 * Initialise ADC operation.
 *
 */
void setupADC(void) {

  // Setup QCLK as appropriate for ADC
  // (TPSH = 739ns, TPSL = 262ns, 0.999MHz QCLK)
  QADC_REGS.QACR0 = 0x00EC;
  // Configure queue 1
  QADC_REGS.QACR1 = 0x0000;
  // Configure queue 2
  QADC_REGS.QACR2 = 0x003F;
  // predefine end of queue
  QADC_REGS.CCW[2] = ENDOFQUEUE;

}


/*
 * initTPU:-
```

```
 *
 * Initialises TPU.
 *
 */
void initTPU(void) {

  // Set base address of TPU RAM to address 0x200000 & enable TPU RAM array
  // Note error in datasheet - bit 0 high = enable!(not disable)
  TPURAM_CTRL.TRAMBAR = 0x2001;

  // Copy microengine code to TPU RAM from flash memory(@30010)
  flashcpy(TPU_MICROCODE_RAM, TPU_MICROCODE_FLASH, (int) 2048/sizeof(ushort));

  // Initialise TPU
  TPU_init();

}


/*
 * flashcpy:-
 *
 * Copies n bytes of data from address src in flash to destination dest in
 * RAM. Used to copy the new TPU function code from FLASH into TPU RAM space.
 *
 */
void flashcpy(ushort *dest, ushort* src, int n) {
  ushort *p;
  for(p = dest; n > 0; n--, p++, src++)
    *p = *src;
}


/*
 *
 * setupPWM:-
 *
 * Initialise PWM.
 *
 */
void setupPWM(void) {

  // Initialse TPU PWM channels
  tpuPWM_init(PWM1CH, PWM_PERIOD, 0);
  tpuPWM_init(PWM2CH, PWM_PERIOD, 0);
  tpuPWM_init(PWM3CH, PWM_PERIOD, 0);

  // Enable all motors after setting all to forward
  SIM_REGS.PORTC &= (PWMDIR1FWD & PWMDIR2FWD & PWMDIR3FWD);
  SIM_REGS.PORTE &= (PWMENABLE1 & PWMENABLE2 & PWMENABLE3);

}


/*
 *
 * setupQDEC:-
 *
 * Initialise quadrature decoding.
 *
 */
void setupQDEC(void) {

  QDEC_init(QDEC1A, QDEC1B);
  QDEC_init(QDEC2A, QDEC2B);
  QDEC_init(QDEC3A, QDEC3B);

}


/*
 *
 * setupPIT:-
 *
 *
 */
void setupPIT(void) {
```

```
  // If using feathering (8kHz)
  //SIM_REGS.PITR = 0x0001;

  // Sets PIT to frequency of 1.024kHz
  SIM_REGS.PITR = 0x0008;

  if (boardID==7) {
    // Set PIT frequency to 49.95Hz for velocity profile transmission
    SIM_REGS.PITR = 0x00A4;
  }

  // Setup the interrupt
  #define PIT_VEC         (2 + INTR_USR_BASE)
  set_vect(PIT_VEC, PIT_intr);

  // Set interrupt vector and keep PIT disabled
  SIM_REGS.PICR = 0x0000 | PIT_VEC;

}


/*
 *
 * initSCI:-
 *
 *
 */
void initSCI(void) {

  QSM_REGS.SCCR0 = 0x0011; // baud rate 38400
  QSM_REGS.SCCR1 = 0x0008; // enable transmitter only

}



/*
 * delayPB:-
 *
 * Simple delay for push buttons so when pressed they don't
 * trigger the PB event twice.
 */
void delayPB(void) {

  uchar x;
  for (x=0; x<100; x++) {}

}


/*
 * roundValue:-
 *
 *  Takes a double quantity and rounds it to the nearest
 *  whole number. That number is returned as an int.
 *
 */
int roundValue(double inputValue) {

  int outputValue;

  if (inputValue<0) {
    inputValue = inputValue*(-1);
    outputValue = (-1)*((int)(inputValue + 0.5));
  } else {
    outputValue = (int)(inputValue + 0.5);
  }

  return outputValue;

}
```

```
/*****************************************************************************

  File:    lowlevel.c

  Author:  Simon Hall

  Created: 13/10/2004 - Modification of Andrew Hood's code for 2001 lower
           limb controllers to suit GuRoo's 2004 lower limb controller
           boards(Motorola 68376).

  Summary: Functionality code for GuRoo's 2004 lower limb controller
           boards.

***************************** INCLUDE ************************************/

#include "lowlevel.h"
#include "mc68376.h"
#include "intr.h"
#include "tpu.h"

/***************************** GLOBALS ******************************/

//int counter1 = 0;
//int counter2 = 0;
//int counter3 = 0;
//int pwm_high_1 = 0;
//int pwm_high_2 = 0;
//int pwm_high_3 = 0;
//int pwm_low_1 = 0;
//int pwm_low_2 = 0;
//int pwm_low_3 = 0;

int emergency_message;

extern uchar boardID;


/*
 * read_curr:-
 *
 * Reads the current sensor flowing through the H-bridge specified
 * by 'k' and returns current in mA.
 *
 */
unsigned int read_curr(int k) {

  unsigned int currentADC;

  // Specify queue of ADC channels relating to 'k'
  switch (k) {
    case 0:
      QADC_REGS.CCW[0] = ANALOG1A;
      QADC_REGS.CCW[1] = ANALOG1B;
      break;
    case 1:
      QADC_REGS.CCW[0] = ANALOG2A;
      QADC_REGS.CCW[1] = ANALOG2B;
      break;
    case 2:
      QADC_REGS.CCW[0] = ANALOG3A;
      QADC_REGS.CCW[1] = ANALOG3B;
      break;
  }

  // initiate single scan of queue 1 channel list
  QADC_REGS.QACR1 = 0x2100;

  // wait for queue of conversions to complete(check flag)
  while(!((QADC_REGS.QASR)&(0x8000)));

  // clear queue 1 completion flag
  QADC_REGS.QASR &= 0x7FFF;

  // whichever H-bridge has highest AD reading has true current
  if (QADC_REGS.RJURR[0]>QADC_REGS.RJURR[1]) {
    currentADC = QADC_REGS.RJURR[0];
  } else {
```

```
    currentADC = QADC_REGS.RJURR[1];
  }

  // Display lower byte of unsigned right justified
  // ADC reading(first one in queue)
  //outputToLEDs(QADC_REGS.RJURR[0]);

  // This is a crude way of scaling currentADC to actual current
  currentADC = currentADC * 5;

  return ((unsigned int)(currentADC));
}


/*
 * read_enc:-
 *
 * Returns the current quadrature decoded encoder count
 * for motor 'k'.
 *
 */
unsigned int read_enc(int k) {

  ushort tpuChannel;

  // Specify quadrature decoded TPU channel relating to 'k'
  switch (k) {
    case 0:
      tpuChannel = QDEC1A;
      break;
    case 1:
      tpuChannel = QDEC2A;
      break;
    case 2:
      tpuChannel = QDEC3A;
      break;
  }

  // use tpu.c for QD reading
  return((unsigned int)(QDEC_read(tpuChannel)));

}


/*
 * set_PWM:-
 *
 * Sets PWM duty cycle for motor k. Value passed ranges between
 * -1600 to 1600 representing 100% backward to 100% foward.
 * (Does not use feathering at present.)
 *
 */
void set_PWM(int k, int pwm_duty) {

  switch(k) {
    case 0:
      if (pwm_duty<0) {
        // make +ve if -ve
        pwm_duty = (-1)*pwm_duty;
        // set direction of motor to reverse
        SIM_REGS.PORTC |= PWMDIR1REV;
      } else {
        // set direction of motor to forward
        SIM_REGS.PORTC &= PWMDIR1FWD;
      }

      tpuPWM_set(PWM1CH, pwm_duty/2);

      /* //If using feathering:-
      // Calculate portion of 0.004s to set PWM to pwm_high
      counter1 = pwm_duty%32;
      // At 100kHz PWM, duty cyle has resolution -50 to 50
      // Therefore dividing 1600/32 = 0 to 50.
      // high time pwm setting
      pwm_high_1 = (pwm_duty/32) + 1;
      // low time pwm setting
      pwm_low_1 = (pwm_duty/32); */
```

```
      break;
    case 1:
      if (pwm_duty<0) {
        pwm_duty = (-1)*pwm_duty;
        SIM_REGS.PORTC |= PWMDIR2REV;
      } else {
        SIM_REGS.PORTC &= PWMDIR2FWD;
      }

      tpuPWM_set(PWM2CH, pwm_duty/2);

      /*
      counter2 = pwm_duty%32;
      pwm_high_2 = (pwm_duty/32) + 1;
      pwm_low_2 = (pwm_duty/32);
      */

      break;
    case 2:
      if (pwm_duty<0) {
        pwm_duty = (-1)*pwm_duty;
        SIM_REGS.PORTC |= PWMDIR3REV;
      } else {
        SIM_REGS.PORTC &= PWMDIR3FWD;
      }

      tpuPWM_set(PWM3CH, pwm_duty/2);

      /*
      counter3 = pwm_duty%32 ;
      pwm_high_3 = (pwm_duty/32) + 1;
      pwm_low_3 = (pwm_duty/32);
      */

      break;
  }

}


/*
 * get_schedule:-
 *
 * Not sure what this is for but was in old software!
 *
 */
int get_schedule(void) {
  return emergency_message ;
}


/*
 * outputToLEDs:-
 *
 * Outputs the decimal value input to the debugging LED's.
 * This function required due to patch for MOSFET drivers.
 *
 */
void outputToLEDs(uchar value) {

  // Board 6 does not have re-routed LED lines
  if (!(boardID==BOARD_6)) {

    SIM_REGS.PORTE = (SIM_REGS.PORTE & 0xE0) + (value & 0x1F);
    SIM_REGS.PORTC = (SIM_REGS.PORTC & 0xF1) + ((value>>6)&(0x02)) +
      ((value>>4)&(0x04)) + ((value>>2)&(0x08));

  } else {
    SIM_REGS.PORTE = value;
  }

}
```

```
/*
 * transmitShort:-
 *
 * Sends a short(16 bit word) to the UART. Use RS232 connector.
 * Hyperterminal will display the binary number
 * in hexadecimal. Baud rate 38400.
 * (Caution when inserting in a control loop - takes time)
 *
 */
void transmitShort(short binNum) {

  uchar lowerHalfByte = (uchar)(binNum & 0x000F);
  uchar lMiddleHalfByte = (uchar)(((binNum & 0x00F0)>>4));
  uchar uMiddleHalfByte = (uchar)(((binNum & 0x0F00)>>8));
  uchar upperHalfByte = (uchar)(((binNum & 0xF000)>>12));

  // Converts each half byte to an ASCII character
  if (lowerHalfByte<10) {
    lowerHalfByte += 48;
  } else {
    lowerHalfByte += 55;
  }

  if (lMiddleHalfByte<10) {
    lMiddleHalfByte += 48;
  } else {
    lMiddleHalfByte += 55;
  }

  if (uMiddleHalfByte<10) {
    uMiddleHalfByte += 48;
  } else {
    uMiddleHalfByte += 55;
  }

  if (upperHalfByte<10) {
    upperHalfByte += 48;
  } else {
    upperHalfByte += 55;
  }

  // Transmit digit
  while(!(QSM_REGS.SCSR&0x00C0)); // wait for transmission
  QSM_REGS.SCDR = upperHalfByte; // the digit

  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = uMiddleHalfByte;

  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = lMiddleHalfByte;

  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = lowerHalfByte;

  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = 0x0A; // new line

  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = 0x0D; // CR

}


/*
 * transmitChar:-
 *
 * Sends a character(8 bit word) to the UART. Use RS232 connector.
 * Hyperterminal will display the binary number
 * in hexadecimal. Baud rate 38400.
 * (Caution when inserting in a control loop - takes time)
 *
 */
void transmitChar(char binNum) {

  uchar lowerHalfByte = (uchar)(binNum & 0x0F);
  uchar upperHalfByte = (uchar)((binNum & 0xF0)>>4);
```

```
  if (lowerHalfByte<10) {
    lowerHalfByte += 48;
  } else {
    lowerHalfByte += 55;
  }

  if (upperHalfByte<10) {
    upperHalfByte += 48;
  } else {
    upperHalfByte += 55;
  }

  // Transmit digit
  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = upperHalfByte; // the digit

  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = lowerHalfByte;

  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = 0x0A; // new line

  while(!(QSM_REGS.SCSR&0x00C0));
  QSM_REGS.SCDR = 0x0D; // CR

}
```

```
/*****************************************************************************

  File:     can.c

  Author:   Simon Hall

  Created:  11/10/04

  Summary:  Handles CAN communication between boards. This requires
            further development. At this stage only includes necessary
            code for demonstration software.

*****************************************************************************/

#include "can.h"
#include "mc68376.h"
#include "board1.h"
#include "lowlevel.h"

extern int velProfile[3];
extern uchar boardID;
extern uchar commencePI;
extern uchar findIndex;
extern uchar passedWaypoint[3];
extern int wayPoint[3];
extern int reqdPWMS[3];
extern uchar foundIndex[3];
extern int oldPosnS[3];

uchar justStarted = TRUE;

// Variables for data retrieval:-
extern int dataLog[30];
extern int dataLogPointer;
//uchar dataLatch = FALSE;


/*
 * setupCAN:-
 *
 * Initialise CAN operation. This will get altered when
 * can.c gets written.
 *
 */
void setupCAN(void) {

  // Setup CAN registers
  CAN_REGS.CANCTRL0 = 0x00;
  CAN_REGS.CANCTRL1 = 0x05;
  CAN_REGS.CANCTRL2 = 0xB7;
  CAN_REGS.PRESDIV = 0x00;

  // Set buffer 0 ID to that of boardID
  // & set to active & empty
  CAN_REGS.MSGBFR[0].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[0].IDHIGH = 0x0000 + (boardID<<5);
  CAN_REGS.MSGBFR[0].CTRL += 0x0040;

  // Set buffer 1 ID to command receiver plus boardID
  CAN_REGS.MSGBFR[1].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[1].IDHIGH = 0x0000 + ((boardID|COMMAND_CONTROL)<<5);
  CAN_REGS.MSGBFR[1].CTRL += 0x0040;

  // Setup remaining CAN buffers as inactive receive buffers
  CAN_REGS.MSGBFR[2].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[3].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[4].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[5].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[6].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[7].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[8].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[9].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[10].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[11].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[12].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[13].CTRL &= 0xFF0F;
  CAN_REGS.MSGBFR[14].CTRL &= 0xFF0F;
```

```
  CAN_REGS.MSGBFR[15].CTRL &= 0xFF0F;

  // CAN Mask register for message filtering (this not really reqd)
  CAN_REGS.RXGMSKHI = (CAN_REGS.RXGMSKHI & 0x001F) + 0xFFE0;

  // Disable interrupts completely
  CAN_REGS.CANICR = 0x0000;
  CAN_REGS.IMASK = 0x0000;

  // Enable CAN
  CAN_REGS.CANMCR = 0x0000;

}


/*
 * chkForMsgs:-
 *
 * Check for new incoming messages.
 *
 */
void chkForMsgs(void) {

  // Check for received messages in receiver buffer 0
  // (transmitted velocity profile)
  if (CAN_REGS.IFLAG & 0x0001) {

    if (boardID==BOARD_7) {

      // Data retreival
      dataLog[dataLogPointer] = CAN_REGS.MSGBFR[0].DATA[0];
      dataLogPointer++;

    } else {

      findIndex = FALSE;

      //dataLatch = TRUE;

      // update velocities
      velProfile[0] = CAN_REGS.MSGBFR[0].DATA[0];
      velProfile[1] = CAN_REGS.MSGBFR[0].DATA[0];
      velProfile[2] = (-1)*(CAN_REGS.MSGBFR[0].DATA[0]);

      // enables motor positions to be initialised
      if (justStarted==TRUE) {
        justStarted = FALSE;
        init_pos();
        commencePI = TRUE;
      }

    }

    // Clear CAN buffer filled flag
    CAN_REGS.IFLAG &= ~(0x0001);

  }

  // Check for received mesages in buffer 1
  // (Command Control)
  if (CAN_REGS.IFLAG & 0x0002) {

    // Move all motors to index positions
    if (CAN_REGS.MSGBFR[1].DATA[0]==MOVE_ALL_TO_INDEX) {

      int w;
      // get new "old positions"
      for (w=0; w<3; w++) {
        oldPosnS[w] = (int)(read_enc(w));
      }

      findIndex = TRUE;
      // enable PIT
      SIM_REGS.PICR |= 0x0400;

    // Move Motor 3 to next index posisiton
    } else if (CAN_REGS.MSGBFR[1].DATA[0]==MOVE_TO_NEXT_INDEX3) {
```

```
        // set waypoint
        wayPoint[2] = (int)(read_enc(2)) + 500;

        // take care of overflow
        if (wayPoint[2]>=65536) {
          wayPoint[2] -= 65536;
        }

        reqdPWMS[2] = 0;
        passedWaypoint[2] = FALSE;
        foundIndex[2] = FALSE;
        oldPosnS[2] = (int)(read_enc(2));
      }

      CAN_REGS.IFLAG &= ~(0x0002);

    }

}
```