Technical Manual
BarrelBeast

Group 6

MTRX3700 Mechatronics 3
Major Project

Name	SID
Jared Silbermann	480270915
Rhys Tirado	470421303
Andrew Munchenberg	460385167
Gabriel Ralph	470205736
Nicolina Bagayatkar	480220910
Vishant Prasad	470416309
Zara Caldwell	470385289
Zheng Li	460418627
Bharath Santhosh	480546670

Date : 24/11/2020
Revision : 0
Prepared by : Group 6: Jared Silbermann, Rhys Tirado, Andrew Munchenberg, Gabriel Ralph, Nicolina Bagayatkar, Vishant Prasad, Zara Caldwell, Zheng Li, Bharath Santhosh
Distribution : MTRX3700 Lecturer (David Rye)

1. Introduction

1 Document Identification

This document describes the design of the BarrelBeast. This document is prepared by Group 6 for assessment in MTRX3700 in 2020.

2 System Overview

The BarrelBeast is a system that comprises of two main subsystems: a Mobile Robot and a Controller. The Controller will be able to acquire user input using two Altronics Z6363 joysticks and buttons. It will also be able to display certain readings to the user on the Altronics Z7011 LCD display on its surface. The Mobile Robot is a motor driven vehicular robot that is controlled by the controller. Data is transferred between the robot and the controller through the use of XBee RF Modules. The Motor used is the Polulu 707 Motor Driver. The Mobile Robot should operate in two modes: Manual and Assist. Manual mode is used to allow the user to freely drive the Robot. Assist mode uses PID control along with two Sharp GP2Y0A41SK0F Infrared Sensors on either end of the robot to move it automatically around cylindrical surfaces.The main objective of the BarrelBeast is to create a Robot that can mimic and compete in a to-scale barrel race.

1.3 Reference Documents

The present document is prepared on the basis of the following reference documents and should be read in conjunction with them.

- Microchip Technology Inc., “High Performance, Enhanced Flash Microcontrollers with 10-Bit A/D,” PIC18FXX2 Data Sheet, July 2007.
- STMicroelectronics, “Automotive fully integrated H-bridge motor driver,” VNH3SP30-E datasheet, Sep. 2013.
- Digi International Inc., *XBee®/XBee-PRO® RF Modules,* 2009.
- Sharp Electronic Components, “Distance Measuring Sensor Unit Measuring Distance : 4 to 30 cm Analog Output Type,” GP2Y0A41SK0F datasheet.
- Altronics, “16x2 Wide Angle Green LED Backlit Alphanumeric LCD,” Z7011 datasheet.
- Sitronix, “Dot Matrix LCD Controller/Driver,” ST7066U datasheet, March 2001.
- Altronics, *Joystick Module,* Z6363 datasheet

1.4 Acronyms & Abbreviations

Table 1-1 lists the acronyms and abbreviations used in this document.

Table 1-1: Acronyms and Abbreviations.

Acronym	Meaning
PID	Proportional-Integral-Derivative
IR	Infrared
LCD	Liquid Crystal Display
RF	Radio Frequency
RPM	Revolutions per minute
ADC	Analog-to-Digital Conversion
PWM	Pulse Width Modulated signal
CCP	Capture/Compare/PWM
GND	Ground
VCC	Voltage Common Collector

2. System Description

This section is intended to give a general overview of the basis for the BarrelBeast system design, of its division into hardware and software modules, and of its development and implementation.

2.1 Introduction

BarrelBeast is robotic system that has been designed to complete a robot barrel racing competition, adhering to the product deliverable specifications outlined in 'Major Project Rev B'. It consists of two major components: the mobile robot and the commander. The commander is a handheld battery powered device that allows the user to control the direction and velocity of the mobile robot via radio communication. The functionality of each major component, the mobile robot and commander, has been separated into a series of modules outlined in Table 1.

Table 1.2 BarrelBeast Modules

Module	**Module Description**
Mobile Robot Modules	
Encoders	Software and hardware module that interfaces with 2 64 count-per-revolution magnetic encoders fitted to the mobile robot motor shafts. The module measures the rotation of the motors.
Motor Drivers	Software and hardware module that interfaces with a Pololu Dual VNH3SP30 Motor Driver Carrier MD03A. The module sets the rotational speed of the left and right mobile robot wheels.
PID Control	Software module that allows the proportional, integral, and derivative gains related to the mobile robot motion to be adjusted and set.
IR Sensors	Software and hardware module that interfaces with 2 Sharp GP2Y0A41SK0F Infrared Distance Sensors located on the left and right side of the mobile robot. The module obtains the current infrared sensor measurements.
XBee Communication	Software and hardware module that interfaces with a Digi XBee 1 mW Radio and Sparkfun "Xbee Explorer Regulated" Circuit, located on the top plate of the mobile robot. The module sends and receives values from the XBee module on the commander component.
Commander Modules	
Joysticks	Software and hardware module that interfaces with two THB001P joysticks on the commander. The module obtains the x and y-coordinate values of the joysticks controlled by the user.

LCD and Menu Interface	Software and hardware module that interfaces with a 2-line x 16-character compact alphanumeric dot matrix LCD. The module controls the current state of the LCD screen based on the user input
XBee Communication	Software and hardware module that interfaces with a Digi XBee 1 mW Radio and Sparkfun "Xbee Explorer Regulated" Circuit, located on the top commander component. The module sends and receives values from the XBee module on the mobile robot component.

2.2 Operational Scenarios

The operation of BarrelBeast is determined by the class of the user and the selected driving mode.

User Classes:
Those who operate the BarrelBeast robotic system can be classified into two streams of users: **privileged users** and **non-privileged users** (see Fig. 1).

- Privileged Users: Privileged users are individuals authorised to participate in the testing, adjustment, and calibration of the robotic system during its development. Privileged users can access a set of calibration settings including setting yaw rate, PID gains, IR samples per distance estimate, IR sample rate and displaying IR measurements to the screen. These commands are available in addition to those available to non-privileged users.

  To limit the accessibility of factory mode commands to privileged users only, knowledge of accessing these commands through the right pushbutton on the commander is only divulged to privileged users. Pressing the right pushbutton will only call on factory mode to be entered when the user is concurrently in the configuration submenu.

- Non-Privileged Users: Non-privileged users refer to individuals who are not involved in the development of the robotic system and use the robotic system purely for entertainment purposes. These users have access to three commands: setting the maximum speed of the mobile robot, operating the robot in manual mode, and operating the robot in assist mode.

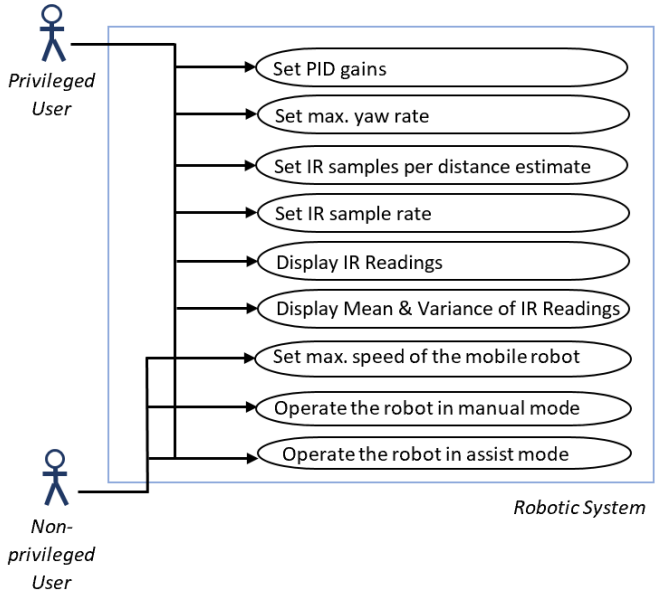

Figure 1: Use Case Diagram of BarrelBeast Robotic System

Driving Modes:
As noted above, both streams of users have access to controlling the robot in two different driving modes: manual mode and assist mode.

- Manual Mode: In manual mode, the user can control the robot using the two joysticks on the console on the left and right side. The right joystick controls the speed and forwards/backwards motion of the robot while the left joystick controls the rotation of the robot. This mode can be entered assist mode menu item in the opening menu and press down the right pushbutton

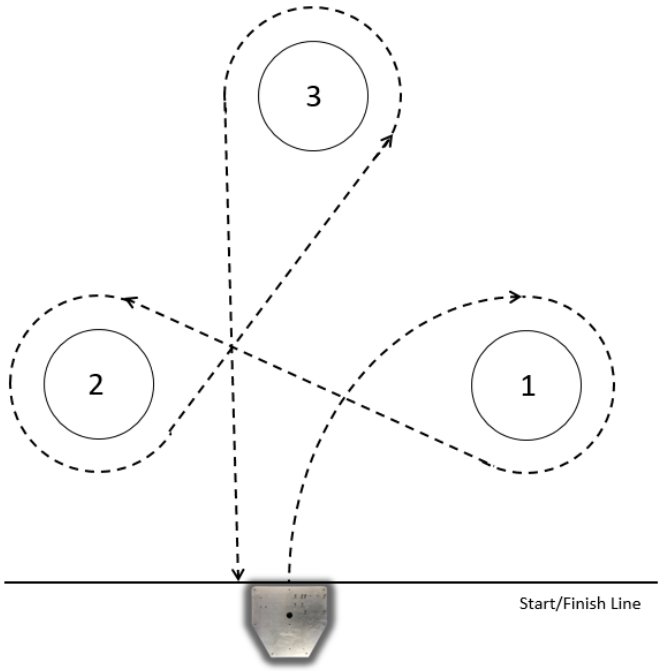

Figure 2: Example of Manual-Mode Driving Operation in a 3-Barrel Course

- Assist Mode: In assist mode, sensors located on the robot are used to detect if the mobile robot unit is in close-range of the barrels. The sensors will detect which side that the barrel appears on - as soon as the barrel detected, the robot will automatically enter into a circular path around the barrel maintaining an 12cm distance between detected barrel edge and mobile robot. To get out of this loop and regain manual control, the user may press down the right joystick pushbutton (indicated by green circles in Fig. 3).

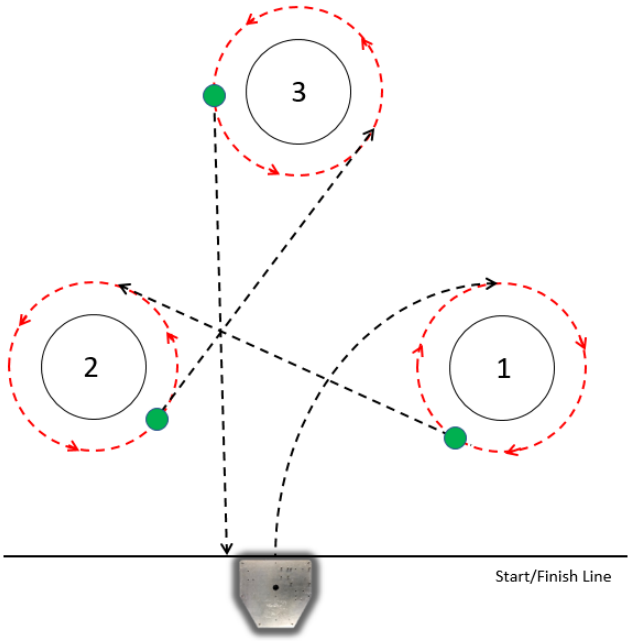

Figure 3 : Example of Assist-Mode Driving Operation in a 3-Barrel Course

Improper use and care for BarrelBeast within the described operational modes may, however, result in system failure. The failure modes of the robotic system are detailed below.

TABLE 2.1 BarrelBeast Failure Modes

Failure Mode	**Consequence**
Low Battery	If the batteries used on the commander or mobile robot are low, the modules may exhibit unexpected behaviour such as slower wheel speed than specified by the user.
Use on an uneven Surface	If driven in an unsuitable environment for operation such as on an uneven surface, the mobile robot wheels may lose grip with the pathway surface. The mobile robot may be at risk of flipping if the wheels are on different levels and variable friction may also affect the individual speeds of the wheels. Subsequently, the user may have less control of the robot speed and direction and damage to the mobile robot module is likely to ensue.
Interrupted Radio Signal	If radio signal is interrupted or disturbed as a result of the XBee communicating modules being used outside of the specified performance range (30m), the XBee module on the mobile robot may stop receiving data/commands from the user and the user may lose control of the mobile robot resulting in potential damage to the robot or infliction of damage to obstacles in the surrounding environment.
Jammed Joysticks	If the joysticks become jammed during operation of the mobile robot, the mobile robot may become locked into a particular direction or speed causing the user to have less control over the mobile robot unit and potentially resulting in damage to the robot if crashing into surrounding obstacles ensues.
Jammed Wheels	If the wheels of the robot become jammed as a result of external friction from an unexpected environmental element such as dust the

	robot mobile unit's direction and speed may be affected and if both wheels are jammed the mobile unit may be halted altogether.
Exposure to Heat or Water	If the user operates the commander or mobile robot unit outside the operational temperature ranges outlined in the user manual (5 - 35°) or exposes either module to water, the electronic components may cease to operate (due to IC damage or corrosion) resulting in the commander or mobile robot units operating in unexpected ways or not at all.
Inadequate User Familiarity & Mobile Robot Crashes	If the user is unfamiliar with the motion-control of the mobile robot via the commander, they may drive the mobile robot into nearby obstacles or barrels in the course resulting in damage to the mobile robot unit.
IR Sensor Disturbances	If there are unexpected obstacles in the barrel racing course or the mobile robot unit is driven within a range of closer than 3cm to a barrel, unexpected behaviour such as intermittent wheel direction changes may occur in assist mode. The failure mode may also result in incorrect IR sensor measurements being output to the LCD screen if this display option is used.

2.3 System Requirements

The operational scenarios considered place certain requirements overall BarrelBeast system, and on the modules that comprise it. The system requirements sourced from 'Major Project Rev B' are detailed below:

- The robotic system shall be capable of accurately and rapidly manoeuvring around a flat barrel racing course which has a width and length that does not exceed 30m
- The robotic system shall operate within a temperature range of 5 - 35°
- The robotic system's manual mode shall be executable and accessible to all users
- The robotic system's assist mode shall be executable and accessible to all users
- Transitions between operating modes shall be bump less
- The robotic system shall power up and power down in a safe and well-defined way
- The robotic system shall operate whenever both the mobile robot and the robot commander are powered
- The robotic system's factory commands shall be executable and accessible to authorised users only
- The mobile robot shall be inhibited from moving until the motor drives (amplifiers) are energised through a deliberate user action
- Operation of the commander module shall be intuitive and display appropriate command prompts, status, and error messages for the user
- The robotic system shall be implemented using two MNML·PIC·18 Minimal Boards (v. 3) fitted with Microchip PIC18F452-I/P microcontrollers. Other circuits shall be fabricated on 'Veroboard' stripboard or equivalent prototyping board.
- A modular crimp terminal wiring system based on the Dupont crimp system shall be used in the robotic system wiring
- All integrated circuits that have a retail price of $5.00 or more shall be socketed

3. Joystick Module Design

3.1 Module Requirements: Joysticks

The operational scenarios considered place certain requirements on the joysticks, and on the modules that comprise it. Two THB001P joysticks have been implemented on the commander component. The x-coordinate, y-coordinate and pushbutton statuses of these joysticks are used both to control the motion of the mobile robot unit and to navigate between menu items displayed on the LCD module and allow intuitive control of the user visual interface.

3.1.1 Functional Requirements

This section describes the functional requirements of Joysticks – those requirements that must be met if the module (and system) is to function correctly.
The Joystick module shall:

- Respond to mechanical manipulation of the thumb-handle components
- Output an appropriate value corresponding to the x-coordinate, y-coordinate and pushbutton status able to be used by both the LCD module and XBee module
- Be intuitive to control

3.1.2 Inputs

The joysticks receive inputs in the form of mechanical manipulation of the thumb-handle hardware by the user. The thumb-handle sits in shafts along the x-axis and the y-axis. As the thumb-handle is tilted, 2 10kΩ potentiometers attached to these shafts are used to translate the coordinates of the handle by outputting a corresponding voltage between 0-5V. The maximum potential voltage is not expected to exceed 20% for first 10,000 cycles of joystick use. Both joysticks were interfaced with a PIC18F452 microcontroller, using three pins each on PORTA corresponding to their x-coordinate (0-5V), y-coordinate (0-5V) and pushbutton statuses (0/5V).

3.1.3 Process

The six 0-5V inputs on PORT A's ADC channels are converted to digital values using a 10-bit ADC module implemented in the software. The output is written to ASRESH and ADRESL. However, only the value in ASRESH is stored, sacrificing 2 bits of resolution in the process. The decision to obtain only 8-bits was made in the interest of simplifying the joystick data processing for the XBee module which is only able to send and receive 8-bits of data at a time. The timing requirements of the ADC conversion are described in below.

3.1.4 Timing

The minimum required acquisition time for ADC is the sum of the amplifier settling time, holding capacitor charging time and temperature coefficient. The capacitor must be allowed to fully charge to the input channel voltage level. Refer to Fig. 4 for minimum acquisition timing calculation. The acquisition time can be accounted for in the software by using the delay peripheral library. The ADC module then waits for the A/D conversion to complete by polling for the GO/DONE bit to be cleared. Interrupts were not necessary for implementation.

T_{ACQ} = Amplifier Settling Time + Holding Capacitor Charging Time + Temperature Coefficient

$T_{ACQ} = T_{AMP} + T_C + T_{COFF}$

Temperature coefficient is only required for temperatures > 25°C.

$T_{ACQ} = 2\,\mu s + T_C + [(Temp - 25°C)(0.05\,\mu s/°C)]$

$T_C = -C_{HOLD}(R_{IC} + R_{SS} + R_S)\ln(1/2048)$
$-120\text{ pF}\,(1\text{ k}\Omega + 7\text{ k}\Omega + 2.5\text{ k}\Omega)\ln(0.0004883)$
$-120\text{ pF}\,(10.5\text{ k}\Omega)\ln(0.0004883)$
$-1.26\,\mu s\,(-7.6246)$
$9.61\,\mu s$

$T_{ACQ} = 2\,\mu s + 9.61\,\mu s + [(50°C - 25°C)(0.05\,\mu s/°C)]$
$11.61\,\mu s + 1.25\,\mu s$
$12.86\,\mu s$

Figure 4: Minimum Acquisition Time Calculation for ADC Module

3.1.5 Non-Functional (Quality of Service) Requirements

3.1.6 Performance

No additional performance requirements were necessary for the implementation of the Joysticks. Polling of the ADC proved to provide accurate joystick readings in sufficient timing for use by other modules.

3.1.7 Interfaces

To interface with the commander, spacers for the joysticks were implemented, ensuring ease of control, and providing user comfort.

3.2 Conceptual Design: Joysticks

The joysticks are integrated with both the XBee module and the LCD Module via a PIC18F452 microcontroller. The inputs and outputs of the module interface are detailed in Table 1.

TABLE 3.1 Inputs and Outputs of Joystick Module

Input	**Port**	**Value (V)**	**Format**	**Value**
Left Joystick x coordinate	AN0	0 – 5	Signed Char	0 – 255
Left Joystick y coordinate	AN1	0 – 5	Signed Char	0 – 255
Left Joystick pushbutton state	AN2	0 – 5	Char	[0,1]
Right Joystick x coordinate	AN3	0 – 5	Signed Char	0 – 255
Right Joystick y coordinate	AN4	0 – 5	Signed Char	0 – 255
Right Joystick pushbutton state	AN5	0 – 5	Char	[0,1]

XBee Module: In their integration with the XBee module, the Joystick module is to produce an 8-bit number corresponding to the appropriate output being read (x-coordinate/y-coordinate/z-pushbutton status) off the appropriate joystick which is then mapped to a velocity and direction of the robot motors using algorithms described in Section 8.2.

When the mobile robot is being driven by the user in manual or assist mode, the y-coordinate of the right joystick is being read and used to control the speed of the robot while the x-coordinate of the left joystick is used to control the rotation of the robot (see Fig. 2). The pushbutton status of the right joystick is only relevant in assist mode where the user may press the pushbutton to return from an assisted circular path around a detected barrel and regain complete manual control.

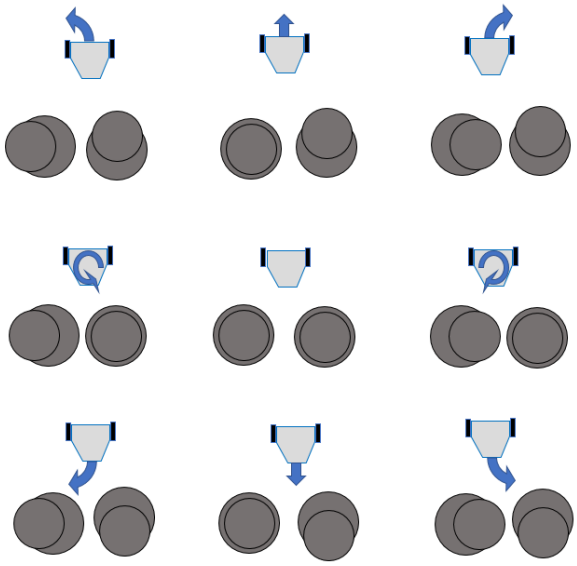

Figure 5: Illustration of Joystick Movement Control

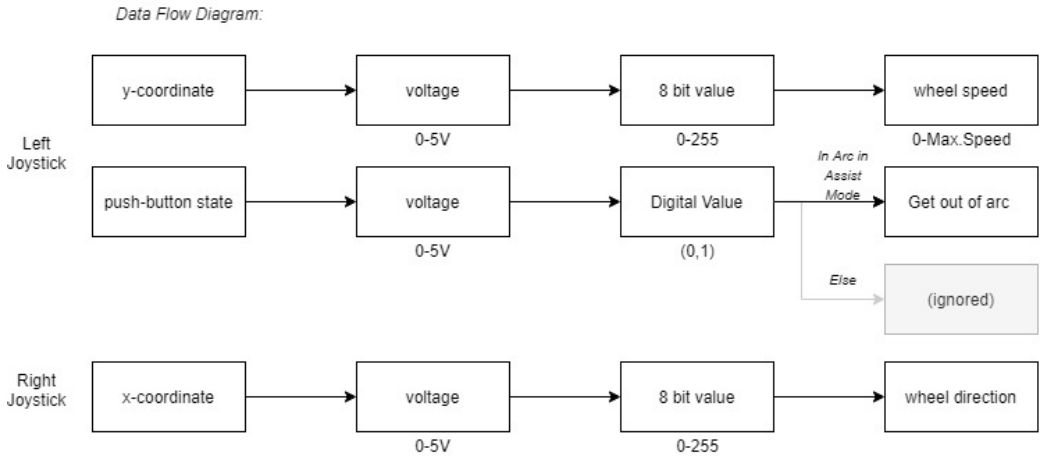

Figure 6: Data flow diagram of joystick module integration with XBee

LCD Module: In their integration with the LCD module, the Joystick module is also to produce an 8-bit number corresponding to the appropriate output being read (x-coordinate/y-coordinate/z-pushbutton status). The joysticks are used to intuitively control the LCD screen status and navigate between menu items/robotic system commands. With the left joystick, the user shall scroll up and down through the menu items. The y-coordinate of the left joystick is read resulting in the following outcomes: Pulling the left joystick downwards will show the next menu item. Pushing the left joystick upwards will show the previous menu item. To enter a menu item, the user is also able to use the left joystick. The x-coordinate of the joystick is read resulting the following outcomes: When the user reached the desired menu item, they may push the left joystick to the right enter its sub-menu. Pushing the left joystick left will show the previous menu. Pressing the left joystick inwards is used to select the menu item.

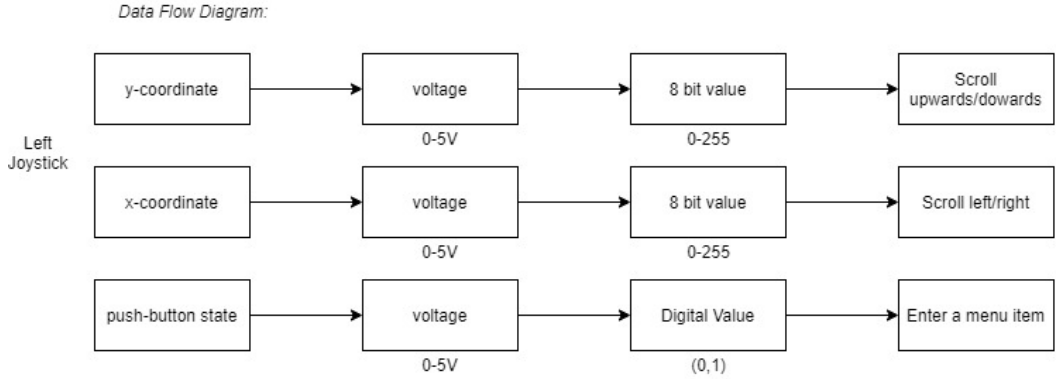

Figure 7: Data flow diagram of joystick integration with LCD

The use of the joysticks in either mobile robot driving or in the navigation of the LCD menus is determined by the status of the mobile robot motors. If the motors are powered on which occurs when the user selects either manual or assist mode operation, the joystick values will be called by the XBee module for driving the robot. However, when the user has finished racing or testing the robot in manual or assist mode, they may press down on the button below the left joystick on the console to regain control of the console screen so the joystick values are then called upon to scroll between menus.

3.3 Constraints on Joystick Performance

Friction between the joysticks and the surrounding components on the commander may result in the joysticks becoming locked into a particular position, preventing the design from satisfying its requirements.

4. XBee Module Design

4.1 Module Requirements: XBee

4.1.1 Functional Requirements

4.1.2 Inputs

The XBee on the Robot and the Controller both take the same class of input being strings which are then transmitted wirelessly between the two chips. Whilst the types of inputs are similar the form in which they are sent varies. On the Controller the XBee takes two different string inputs being joystick readings and configuration values. The inputs for these strings to be formed are the user inputs from the configuration menu and the joystick analogue readings. This data is sent to the XBee Robot as inputs on the receive side of the duplex system.

There are inputs as well for string to be sent from the XBee Robot to the XBee controller. These strings are sent back to the XBee Controller to be displayed on the LCD. Four pieces of information are sent from the robot to the controller as per the requirements. These are the mean and variance of the IR scanners and the left and right motor power. Hence all 4 pieces of data are inputs to the module as well on the other side of the duplex.

4.1.3 Process

The processes involved in the XBee module all revolved around the saving of data in the correct place once it has been sent from one side of the wireless transfer to the other. To facilitate this saving two slightly different processes were used for joystick commands and configuration. Further details on the sending protocol for each type of message can be found in the **Conceptual Design** section for this module.

4.1.4 Outputs

The outputs of this module overall involve specific variables in the program being updated based on the strings sent between the XBee's. Therefore, the outputted variables are the x,y,z joystick positions and all elements in the configuration union shown above. For further information on input and output data formats please see the **Conceptual Design** section for this module.

4.1.1.4 Timing

A small delay was included after transmission of each byte sequence of 100ms. This was implemented during testing and integration as it had positive effects on the reception of strings on the other side of the wireless XBee transmission,

4.1.1.5 Failure modes

Due to the use of start bytes failure modes from the XBee module have been deliberately programmed out. If there is an issue with the order in which bytes are sent or an inaccurate transmission it will not be saved into the output variables. Hence faulty serial string transmission has no effect on the program adding to the robustness of the XBee module. Additionally, if the XBee travels outside the communication range the last motor power value sent to the robot will remain and the robot will travel in a continuous direction and speed. Programming of a kill switch is necessary for future development of the robot to stop the motors when this occurs.

4.1.2 Non-Functional (Quality of Service) Requirements

4.1.2.1 Performance

XBee performance is highly dependent on how it is configured using dedicated XBee software. This was done externally to the project by unit supervisors.

4.1.2.2 Interfaces

For ease of interface between inputs and outputs as a standard all variables were converted to chars before being used by the XBee module.

4.1.2.3 Design Constraints

XBee performance constraints are dictated by configuration set up as mentioned in performance.

4.2 Conceptual Design: XBee Module

BRGH Calculation

As per the project information the XBee's were configured to operate in transparent mode at a baud rate of 9600. Therefore, a calculation was required to ensure that the USART baud rate of the PIC18f452's matched the XBee baud rate as close as possible. This would ensure any issues involving baud mismatch were avoided. The following calculation was used to calculate the value that was required to be written into the SPBRG register.

$$BR = \frac{FOSC}{16(X+1)}$$

$$X = \frac{10 * 10^6}{16 * 9600} - 1 = 64.1 = 64$$

Equation 1 – Formulation BRGH Calculation

Hence the final baud rate with a BRGH value at 64 was 9615.38. The error rate between 9600 desired baud rate and this value is shown below.

$$BR_{error} = \frac{9615.38 - 9600}{9600} * 100 = 0.16\% \; Error$$

Equation 2 – BRGH Error Calculation

Please note that BRGH was chosen as 1 (high speed) as in low speed the error was 1.7%.

Input and Output Data and Message Formats

Controller Sending Protocols

To allow for error handling in string transmission a sending protocol has been developed. There is a slightly different protocol for sending configuration data and joystick data. This ensure only the correct strings are received and translated into variables on the robot side. **Configuration data** is sent using the following protocol:

C Index Value 0

Where:
C = Configuration Start Bit
Index = Position to be saved into config union array as a character
Value = Value to be saved in the index position as a character
0 = Filler value

For example, if the max speed was changed by the user to 50% the following string will input into the XBee Controller and sent to the XBee Robot, where 'ENQ' is the character for 5 and '5' is the character for 50.

'CENQ50'

Joystick data is sent using the following protocol:

V x y z

Where:
V = Velocity Start Bit
x = x coordinate from joystick 1
y = y coordinate from joystick 2

z = Push from joystick 1
For example, if the x coordinate of joystick 1 was 50, the y coordinate of joystick 2 was 100 and the z value of joystick 1 was 0 the following string will input into the XBee Robot and sent to the XBee Controller, where '5 is the character for 50 and 'd' is the character for 100 and 'NULL' is the character for 0.

'V5dNNULL'

Controller Sending Protocols
Unfortunately, due to issues with the receiving of data on the control side of the XBee module the team was not able to implement the sending protocol for wirelessly transmitting information back to the controller. The following was the planned sending protocol:

D M V PR PL

Where:
D = Data Start Bit
M = Mean
V = Variance
PR = Power Right
PL = Power Left

Overall Module Operation
The overall process for correctly interpreting messages sent from the Controller to the Robot was quite similar. Both processes started with only beginning to read inputs once either the 'V' or 'C' start byte was read. Once this byte was read the following 3 entries were saved into different places depending on the start bit. This process was facilitated by the design choice to ensure within our sending protocol that both velocity and configuration strings were 4 bytes long. This allowed us to read for 3 bytes after the start byte was read before resetting the process and waiting for a new piece of information to be sent.

The process for saving joystick values after the start bit was read was quite simple. As per the sending protocol shown in the input section above the first entry after the start byte was saved as x, the second as y and the third as z within a **Velocity** struct. The process for saving configuration values involved a similar process. Instead of saving values directly to variables an array within a union was used. Based on the index sent in the string values were placed in a particular place in the array. This array position always correlated to a specific configuration setting.

Joystick Information Diagram

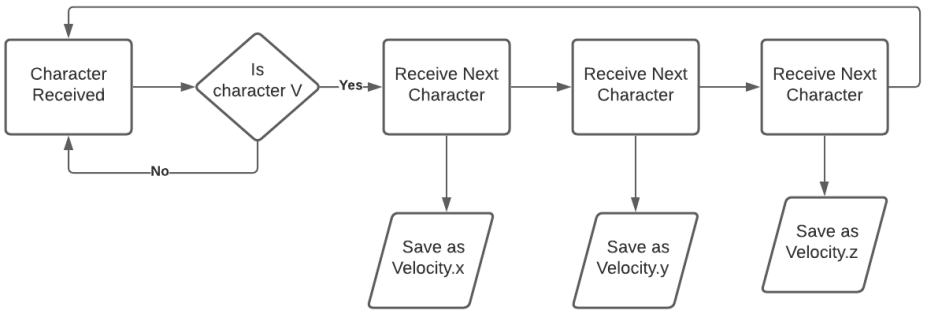

Figure 8: Joystick Sending Diagram

Config Information Diagram

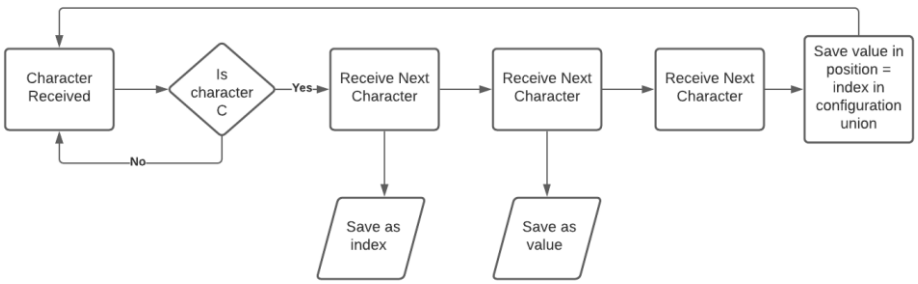

Figure 9: Config Sending Diagram

Input and Output Data Formats

Please note the format of the inputs and outputs below are listed before they are converted to chars to be sent over USART.

Table 4.1 – Input & Output Data Formats

Input	Format	Explanation	Output	Format
Controller XBee Inputs			**XBee Robot Output Variables**	
x	Signed Char	x Reading from joystick 1	Velocity.x	Signed Char
y	Signed Char	y reading from joystick 2	Velocity.y	Signed char
z	Char	z push from joystick 1	Velocity.z	Signed Char
Robot XBee Inputs			Pid_p	Unsigned Char
Mean	Float	Mean from number of samples by IR scanner	Pid_i	Unsigned Char
Variance	Float	Variance from number of samples by IR scanner	Pid_d	Unsigned Char
Motor Power Left	Signed Char	Power sent to left wheel	Max yaw	Unsigned Char
Motor Power Right	Signed Char	Power sent to right wheel	Max speed	Unsigned Char
			Display	Unsigned Char
			IR Samples	Unsigned Char
			IR Rate	Unsigned Char
			Mode	Unsigned Char

Response to Failure and Error Conditions

For the XBee module failure or error conditions occur when data is sent incorrectly and not in the format of the sending protocol indicated above. However, due to deliberate programming in how variables are sent they will not influence the robot or controller in anyway. In our testing any unwanted strings sent due to errors or slight USART failure did not have the correct start bit associated with the transmission. Hence when error or failure conditions occur with USART sending the output variables are not affected and therefore there is no impact on system performance. This highlights robust module operation to failure conditions and error in the string and data transmission.

Constraints on Module Performance

The speed of data being transmitted and received over USART and XBee communication is limited by the Baud Rate. Since this was set at 9600 this is the only constraint on the speed and performance of the XBee Module.

5. Menu Module Design

5.1 Module Requirements: Menu

5.1.1 Functional Requirements

5.1.1.1 Inputs

The inputs to the menu model all come from dedicated user input into the respective menu options. When these values are entered, they are saved into the menu item union struct.

5.1.1.2 Process

The menu is generated via a pre-defined union which stores the headings of each menu item or configuration option.

5.1.1.3 Outputs

The outputs to this module are 2 different strings as mentioned in the XBee module. These are the joystick and configuration value strings formatted as per the sending protocol detailed in the XBee conceptual design. These strings are sent each time the main loop runs and each time a configuration value is entered into the menu.

5.1.1.5 Failure modes

If an illegal input was placed in the value of a setting there was an error message saying, “Invalid input”. Upon failure of the LCD there was no contingency planned for displaying the menu as it was solely designed for this display.

5.2 Conceptual Design: Menu Module

To create a dynamic and easily customizable menu a 4 way linked list data structure was implemented. The individual node, MenuItem is a union struct that contains pointers to menuItems above, bellow to the left and to the right of itself. A MenuItem also contains a display string (i.e. a character array of length 16) that can be printed to the LCD. Index, value and setterMax are elds used by menuItems directly responsible for setting configuration variables on the robot. When a configuration menuItem is set, the value is compared to the setterMax and if the value is larger an error message is printed, and the value is left unchanged. If the new value is within the valid range, then the value is set and a message containing the index and value is sent to the robot via the XBee (see Configuration). Implementation of this method required more storage and so the pragma command udata was used.

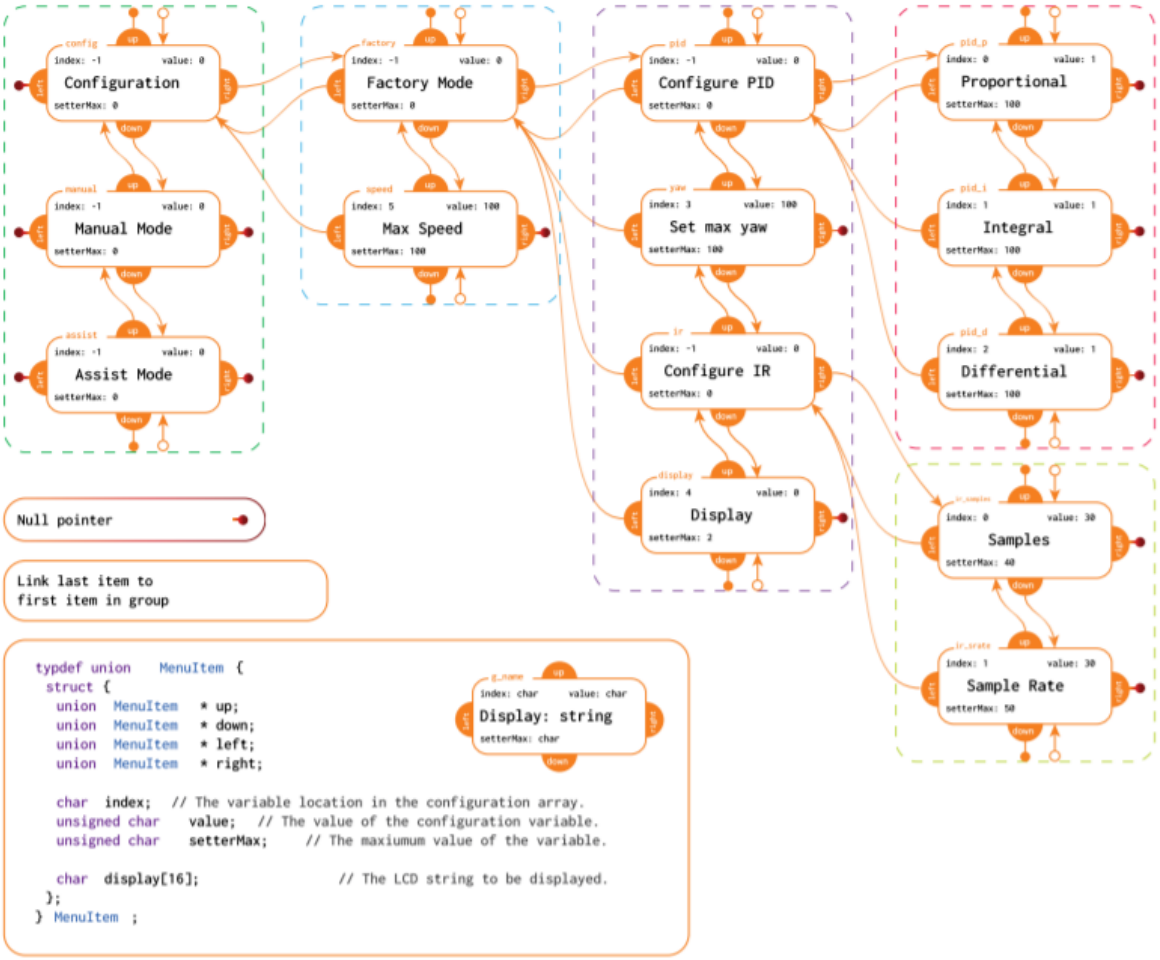

Figure 9 – Menu Breakdown

User control

The menu is controlled within the mainMenu function, where a single pointer to the current menuItem is stored. The current menuItem and item bellow it (if applicable) are printed to the LCD when the user moves the joystick in a distinct direction the moveMenu function will be called. If there exists a menuItem in the direction given, then the moveMenu function will animate a transition to that menuItem on the LCD and return the menuItem transitioned to, if no menuItem exists the function will return the old menuItem item. If moveMenu is called with the current menuItem, "factory" and direction right the menu will only move if the right button is high.

6. Motor Driver Module Design

6. 1 Module Requirements: Motor Drivers

6.1.1 Functional Requirements

6.1.2 Inputs

From the Xbee module the values for the left and right joysticks were received as signed chars, allowing for the direction required for each wheel to be received. Whilst in User_Assist mode the motor driver module received these values from the PID function and the IR motor driver functions to account for the distance from the barrel, allowing for tracking and auto-pathing around it. These values are taken from two separate global power structs, one for the User_Assist motion and another for manual driving to allow for control to be taken back by the user when in User_Assist mode.

6.1.3 Process

The x and y values were then converted into separate values for each wheel, being scaled between 0-512, using the signed negative values from the initial data to determine the directions of the wheels and the magnitude for the speed. The mapping allows for the rate of turning to be determined directly by the left analogy stick while the right manages the average velocity, as such at full lock left or right on the left stick the robot will turn on the spot with full magnitude on each wheel in opposite directions. Anywhere below this the drivers will output a speed on each that turns at a relative rate compared to where the left stick is but will maintain speed and direction from the right stick. The motor drivers also had to consider the use of a max speed and max yaw rate input from the Factory_Mode settings, this was done by scaling to a ratio of the whole, if the max speed was set to 80* the corresponding value of 127 forward to the speed input was scales to the value that would be received at 80% before mapping. The same process was applied to the x value from, the left joystick to slow the turning rate at full lock to the same ratio of the fastest possible speed. To speed up processing and minimised memory wastage chars were used when possible instead of ints and the scaling processes were done before the mapping to allow for operations on only one value instead of on both of the motor powers. The PWM signal was also set to directly take the output values from the xbee such that scaling would not have to take place during normal driving.

6.1.4 Outputs

The main outputs from this mode were to PORTD 1:4 to determine the direction of the motors and to CCP1/2CON and CCPR1/2L to determine the duty cycles that were sent to the pololu Motor driver board to determine the speed the motors would run. The PWM signal was required by the datasheet to be set at below 10kHz to prevent damage but was decided to be set at 4.88KHz as this worked best with the timer pre-scale values available for timer2. The directions also were necessary to set before the duty cycle or the robot could potentially start to accelerate in previous directions before this was changed.

6.1.5 Timing

Apart from the duty cycle remaining below 10kHz the only requirement was for ‘bump less’ movements in all modes, requiring quicker refresh rates than the 0.1s reaction time of the average person. This was achieved by quick running efficient code throughout the project.

6.1.6 Failure modes

A failure in most cases related to the motor drivers would include a cut-off of power to the board, engaging the brakes on the motors through setting output pins to 0. Additional breaking was intended to be implemented after a certain amount of time without communication from the controller but was not implemented in the final build.

6.1.7 Non-Functional (Quality of Service) Requirements

6.1.8 Performance

The main loop and motor computation was required to be fast enough that the transition between the duty cycles was “bump less” as such all conversions were attempted to be minimalised in operations and PID was added to the system to slow the power spikes that would have been felt by the motors if the PWM signals were allowed to jump suddenly.

6.1.9 Design Constraints

The PWM chosen meant that the motor drivers would have at most 512 bits of resolution but due to the constraint of the joysticks only reaching values of +-127 they were already constrained such that a greater resolution would have been unnecessary as it only would have been scaled up from the initial values with no way to access those between.

6.2 Conceptual Design: Motor Drivers

PWM Setup

During initialisation of the PWM for the two possible CCP registers, both using the same timer and duty cycle, the equation below was used

$$PWMperiod = (PR2 + 1) * 4 * Tosc * TMR2prescale$$

From this equation and the duty cycle equation the values for the input resolution can be seen to be (PR2 +1)*4 as this replaces the CCPRXL and CCPXCON bits. Therefore PR2 +1 was set to a power of two such that the value could be directly scaled up by writing to the MSB in CCPRXL and the LSB from this byte could be written into CCPXCON, giving an error of 0.8%, to save time by bit shifting and scaling the input values, this 0.8% value was assumed to be greater than the A/D conversion error of the joysticks and as such would not affect the overall uncertainty of the device. With the possible prescalers of 1,4 and 16 and the 10MHz clock cycle the options for PWM period were therefore between 614Hz to 19.5kHz. with the assumption that a higher frequency would mean a faster reaction rate of the motors and better performance the centre of the 10kHz band was chosen to ensure good sampling time and to be far enough below the maximum so as to not risk blowing up the motor driver board. From this a 4 prescaler and 127-bit PR2 was chosen, giving:

$$PWMPeriod = (127 + 1) * 4 * 4 * \frac{1}{10 * 10^6} = 4.88kHz$$

Using these values and the equation for the input resolution gives

$$resolution = (127 + 1) * 4 = 512$$

The duty cycle could then be calculated using the equation

$$PWM\ duty\ cycle = (CCPRXL{:}CCPXCON < 5{:}4 >) * Tosc * TMR2Prescalevalue$$

The duty cycle being the same equation as the period except this resolution ($CCPRXL{:}CCPXCON < 5{:}4 >$) being four times greater as seen above.

This system gave slightly worse performance at the absolute lowest possible values, but these were difficult to maintain manually by hand and were relatively insignificant to the time save that they caused.

Pin placement

For the PWM generation the inbuilt PWM feature of the CCP module was used as this was inbuilt in the chip and the most efficient way to keep the PWM running in the periphery without constant interrupts. Multiplexing of the PWM2 port was attempted to change the output from PORTB<3> to PORTC<1> although even with power cycling this output port was not changing so the decision was made to use the factory settings for port output.

The outputs for the INA/INB on the motor driver was determined to use PORTD as this port does not have any extra features that might have been required should the project be extended, PORTB has interrupts, both external and interrupt on change, PORTA has the analogue to digital conversions and PORTC has UART protocols and the PWM output, leaving PORTD as the best option.

Memory decisions

The use of chars throughout the motor driver program from the power struct to the motor struct allowed for less memory to be used when it would not be required. Due to the way that the duty cycle was being handled with 128 bits representing the 512 value this variable could be stored as an unsigned char after the direction of the motor was taken into consideration.

Table 5.1: Inputs and Outputs from the Motor Driver Module

Input	Format	Explanation	Output	Format
Motor driver Inputs			**Motor Driver outputs**	
Velocity.x	Signed Char	x Reading from joystick 1	Motor Power Left	Unsigned Char
Velocity.y	Signed char	y reading from joystick 2	Motor Power Right	Unsigned Char
Velocity.z	Signed Char	z push from joystick 1	INA/B left	PIN high/low
Max yaw	Unsigned Char	Max yaw	INA/B right	PIN high/low
Max speed	Unsigned Char	Max speed		

Joystick mapping

The format used for the joystick conversion to motor speeds based on the x-y system described above was used to emulate what most of the group was used to, driving via controller on a games system uses this double joystick system. As such this would not require a process of relearning and allowed for easier and intuitive control.

Calculation of Wheel Speeds

The calculations for the motor speeds were mapped such that at any full lock of the left joystick the robot would have one wheel full forward and the other full back to allow for higher manoeuvrability, this mapping also allowed for turns to be scaled to the speed that they were required, with the y direction of the right joystick being the average velocity of the robot, cancelling out the slowing effect that differential turning can have on the velocity of the robot when the values are directly added/subtracted to the wheels. This meant that one wheel would have to be scaled down and the other up to compensate for the speed lost. The drivability of the robot was also increased as the direct correlation would have made slight turns very difficult. Two equations were also used for if x was greater than y and if y was greater than x as there were some potential speed values that were lost when this was not implemented. Altogether this gave the effect of consideration of the ratio of difference between the two joysticks with a greater difference having a greater effect upon the wheel difference for a high x and less difference for a high y.

The equations used to convert the joystick values, x from the left joystick and y from the right joystick can be seen below for x>y first and y>=x after, if x=0 then y was directly output to the wheels.

$if\ x > y$,outx = (|x| * x) / (|x| + |y|)
outy = (|x| * y) / (|x| + |y|)

$if\ y \geq x$ outx = (|y| * x) / (|x| + |y|)
outy = (|y| * y) / (|x| + |y|)

Derivation

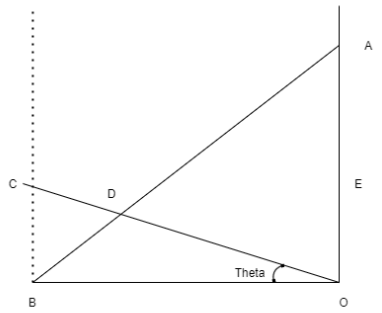

Figure 10: Scaling from C (x from joystick, y from joystick) to achievable motor speed D

For values which are out of range such as C would be as the differential speed does not allow for full speed turning the value must be scaled back to calculate the motor speeds required to move in the expected fashion. To derive the equation which allows for a trade-off between the speed and the turning velocity both the average speed of the wheels and the difference must be considered, this can be done as seen in the vector calculations below.

$$D(x,y) = \frac{OD}{CO}C(x,y)$$

$$\frac{OD}{CD} = \frac{AO}{CB}$$

$$\frac{OD}{OC} = \frac{AO}{CB + AO}$$

$$CB = OBtan(Theta) = \frac{yc}{xc} * OB, where\ A) = 100, OB = 100$$

$$\frac{OD}{OC} = \frac{100}{\frac{yc}{xc} * 100 + 100}$$

$$\frac{OD}{OC} = \frac{1}{\frac{yc}{xc} + 1} = \frac{xc}{yc + xc}$$

$$Therefore\ D(x,y) = \frac{|xc|}{|yc| + |xc|}C(x,y)$$

$$xo = \frac{xc * |xc|}{|yc| + |xc|}$$

$$yo = \frac{yc * |xc|}{|yc| + |xc|}$$

This equation weas then tested in Matlab for the full range of possible values with the outputs graphed as is seen below.

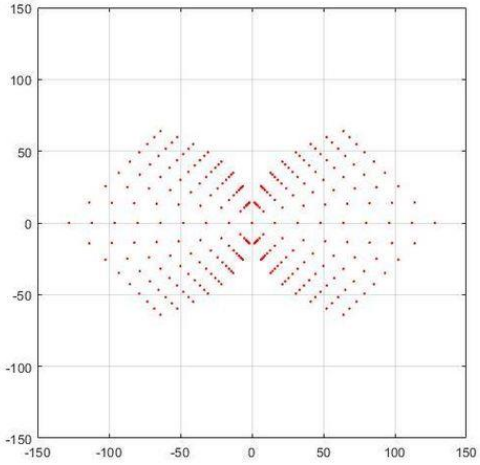

Figure 11: Value map for initial single equation

To solve the issue of when y>x and to fill out the map the x and y coordinates were flipped for this value range, giving full coverage for the possible values as can be seen below.

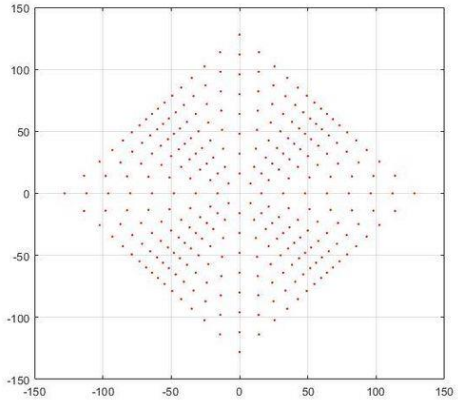

Figure 12: Value map with two equations for x<y and x>y

This completed equation set gave much more robust behaviour to the system and allowed for all possible values of the joysticks to be mapped effectively.

The driving direction for the robot was decided to be drive wheels front as this appeared to give the best performance for speed and turning after testing, when run drive wheels back there was a tendency for the robot to lift slightly due to the torque change when going from full speed forwards to full speed back and this was taken to indicate a slowing of the robot.

6.3 Assumptions Made

A maximum performance speed of 80RPM was assumed for the motors as seen on the datasheet. An assumption was made that the most common joystick control method from home games systems would be the most effective way to drive the robot. There was an assumption that the A/D conversion module of the Pic18 Minimal board would be greater than 0.8%.

6.4 Constraints on the Motor Driver Performance

The motor driver resolution was constrained to the +-127 value range from the joysticks and had physical limitations such as the robot having to use differential speeds of motors to turn, meaning that a turn could not be affected at full speed on both wheels, indicating that turning will always be slower than going straight. Upon insertion of the joystick into the controller case it was found that the shell interfered with the joysticks full range of motion, capping joystick and driver input values to +-117/ 127, this was fixed through scaling the values by 1.1*, giving the full possibly range withing these joystick values.

7. Module Design Encoders

7.1 Module Requirements: Encoders

7.1.1 Functional Requirements

This section describes the functional requirements of the Encoder module – those requirements that must be met if the module (and system) is to function correctly.

7.1.2 Inputs

External inputs to the encoder module are taken from the encoder outputs on the robot hardware. The connections were implemented as shown in Table 6.1 below, with all encoder outputs connected to a 10k Ω pull-down resistor to remove noise to prevent input error.

Table 6.1. Encoder inputs from robot hardware to PIC18F452 Minimal board.

Encoder Output	PIC Input	Variable
X2-6 (A1)	RD0	pinAR
X2-7 (B1)	RD5	pinBR
X6-6 (A2)	RD7	pinAL
X6-7 (B2)	RD6	pinBL

7.1.3 Process

The encoder maximum frequency was slightly above 3.0 kHz, so the sampling frequency was taken as approximately 7.0 kHz to allow some margin for higher frequency depending on surface friction and other sources of error. By setting Timer1 to be close to 7.0 kHz, a sampling rate of 7.1 kHz was achieved based on Timer1 overflows. Every overflow, the status of both the left and right pin A was checked and the value saved as the previous value for the next sample. By checking the current and previous pin status a rising edge could be detected, and an edge counter incremented. After 10 ms of Timer1 overflows had occurred the RPM was calculated by dividing the edge counter by 10 ms and multiplying by an empirically determined constant of 0.0213. This RPM value was then converted to a value between 0 and 127 for use as motor power in the motor PID control.

7.1.4 Outputs

The encoder module outputs a value from 0-127 for the right and left encoders as the variables encoderPowerR and encoderPowerR, respectively. These outputs can be used to determine the PID values that adjust the motor output.

7.1.5 Timing

As detailed above in Section 2.12.1.2, Timer1 is as a sample timer by taking a sample each time it overflows. Timer1 was set up in 16-bit mode with a 1:8 prescaler and was loaded with a hexadecimal value of FFD3 each overflow for a sample timer of 0.1408 ms, to generate a sample rate of 7.1 kHz.

7.1.6 Non-Functional (Quality of Service) Requirements

7.1.7 Interfaces

Several output variables are set to aid in debugging the module and understanding what is happening, rather than performing all computations from input to output in one calculation. RPM and edge count values are useful in debugging to understand exactly what the motors are doing, without being required for functionality.

7.1.8 Design Constraints

The software for the encoder module was programmed in C using the PIC18F452 with C18 compiler and was designed and debugged using an ICD3 and MPLAB X IDE v5.20.

7.2 Conceptual Design: Encoders

The encoders work in quadrature with two pins A and B being out of phase by 90°, and several points on the encoders that connect these contacts (16 per edge for each pin). By determining how many of these contacts occur

in each time period, i.e. the frequency of the encoders, the RPM of the encoders can be determined, and thus the motor RPM.

As stated previously the sampling frequency is taken at slightly over double the maximum encoder frequency to allow for some error. This utilises Nyquist theorem for sampling to avoid signal aliasing by sampling at least twice as fast as the maximum input frequency:

$$Nyquist\ Rate = 2 \times Highest\ Signal\ Frequency$$

The highest signal frequency was determined empirically by analysing the encoders with an oscilloscope while the motors were set to run at full speed.

The encoder frequency was determined by checking the status of the encoder output pins at regular intervals to check if they were a HIGH or a LOW. By determining the number of LOW to HIGH transitions (rising edge) that occur over a set interval, the encoder frequency can be calculated and then converted to an RPM (revolutions per minute) value for the motor PID control.

A faster sampling frequency could be used to determine exactly more accurately when a rising edge occurs, however this would slow down the computation as more samples would need to be taken in a given time period. By taking a sample speed of 7.1kHz, the exact time when a rising edge occurs is not known, and multiple samples may be taken before a rising edge occurs, however the frequency is slow enough to reduce computation time and remove aliasing from the encoder signal. Reducing computation time while maintaining an accurate representation of the signal was the most important design consideration and given that the signal is a regular square wave, a minimal sampling frequency is acceptable for locating edge changes.

The Timer1 initial values TMR1H:TMR1L was calculated using the following formula:

$$delay = \frac{1}{(\frac{Fosc}{4}) \div prescalar} \times no.of\ timer\ counts$$

$$\frac{1}{7000\ Hz} = \frac{1}{(\frac{10 \times 10^6\ Hz}{4}) \div 8} \times no.of\ timer\ counts$$

$$no.of\ timer\ counts = 44.64 \cong 44$$

This gave a desired number of timers counts of 44 for a sample rate of 7000 Hz. This corresponded with setting TMR1H:TMR1L as 0xFFD3 for the desired sample rate.

7.3 Assumptions Made

It was assumed that the CS1 and CS2 pins from the robot hardware were connected to a voltage high for the encoder output to be determined.

It was assumed that when the motors were set to maximum speed that they would run at 80 RPM, and the empirically determined constant of 0.0213 used to convert the edges per second to an RPM was based on this assumption.

7.4 Constraints on Encoder Module Performance

As mentioned in Section 2.13.1, the maximum speed of the motors was assumed to be 80 RPM, and the empirically determined constant based off this assumption limits the software from determining the actual RPM. This has no effect on motor performance, however, as the RPM calculated by the encoders will still be in ratio even if the calibration constant is incorrect.

8. LCD Module Design

8.1 Module Requirements: LCD

8.1.1 Functional Requirements

This section describes the functional requirements of the LCD module – those requirements that must be met if the module (and system) is to function correctly.

8.1.2 Inputs

The LCD module utilizes external inputs from the joystick to permit menu navigation. This communication is sent through a buffer which contains a string with the joystick direction via the XBee. Furthermore, the modules employ global functions and C header files to call the inputs into the LCD module. Input errors are therefore, minimized by the utilization of the joysticks allowing scrolling on the LCD. The menus that can be accessed and options/configurations are limited by the scrolling and repeating of the menus when scrolling up or down. This allows for greater usability.

8.1.3 Process

The LCD module requires an initialization of the display followed by the additional functionalities. These include:

- Writing a String
- Displaying the Scrollable Effect
- Displaying Character Positions/Line Positions
- Error Messages and Welcome Screen
- Mode Switching and Toggling Input from Joysticks

Incorporating the interrupt-driven serial receive function is a functionality that was not able to be implemented by transmitting serial data to the *PIC18F452* from the IR Sensor readings and have it displayed at a designated location on the LCD module. However, the joysticks readings were successfully displayed to the LCD via serial receive function and transmission of data.

The first step of the setup involves configuration of the LCD to 8-bit, two-line interface. Initially external libraries were utilized to interface the LCD however, the functionalities are more customizable utilizing self-written functions.

8.1.4 Outputs

The LCD outputs produced for the modules functionality include most importantly the initialization of the LCD display, the LCD commands which define the display position and line as well as the LCD display which defines the character to be displayed.

8.1.5 Timing

The LCD has a variety of specified delay ranges when utilizing the LCD command and LCD display functionalities written in the program. These values are evident from the datasheet of the LCD controller/driver *ST7066U*. The delay functionality is evident within the LCD module and is called to have a 1ms delay upon command, 2ms upon display and a 50ms prior to a loop iteration (end of function). Scrolling also has its own delay customized to have the most suitable and visually pleasing effect.

8.1.6 Failure Modes

If an illegal integer value is selected beyond ranges or tolerance an error message is outputted with a delay and furthermore, the selection is reset, returning to the previous screen.

If robot and commander communication is cut off, the previous selections are saved and displayed as such on the LCD, unless the controller is fully powered down and turned back on/rest.

8.1.7 Non-Functional (Quality of Service) Requirements

8.1.8 Performance

The LCD has been integrated with the menu to perform very low delay between the user's joystick input and the display. Therefore, the delays are close to the limits defined on the controller datasheet to increase usability and computational loop time. Furthermore, the display is exactly accurate to the user interface when operating the joysticks as well as the robot in displaying the positions defined as and positions.

8.1.9 Interfaces

The LCD for high quality interfacing was separated into functionalities, definitions, and macros to reduce software requirements and repeated programming. Furthermore, reducing memory and delay times for display.

8.1.10 Design Constraints

The LCD is restricted to having 16 characters by 2 lines and therefore, only a certain amount of data can be displayed. Therefore, there is a limitation in the options and names that define each configuration the user may select, and the readability of the messages displayed on the screen.
Operation in 8-bit module utilizes more pins from the *PIC18F452* restricting the pins that can be used by other modules.
There are restrictions on the displays on the LCD such as the configurable integers to reduce the input errors from the user and furthermore, limit their selections.

8.2 Conceptual Design: LCD

The LCD utilizes outputs as characters which are displayed via their storage in strings:

1. Alphabets
2. Integers
3. Symbols

The message and data formats are highlighted more extensively in the menu section of this report.
To reduce error, the LCD has scrollable integer and menu configuration which are looped to restrict the range of selection.
In case of failure with the error inputted still being illegal, an error screen is employed.

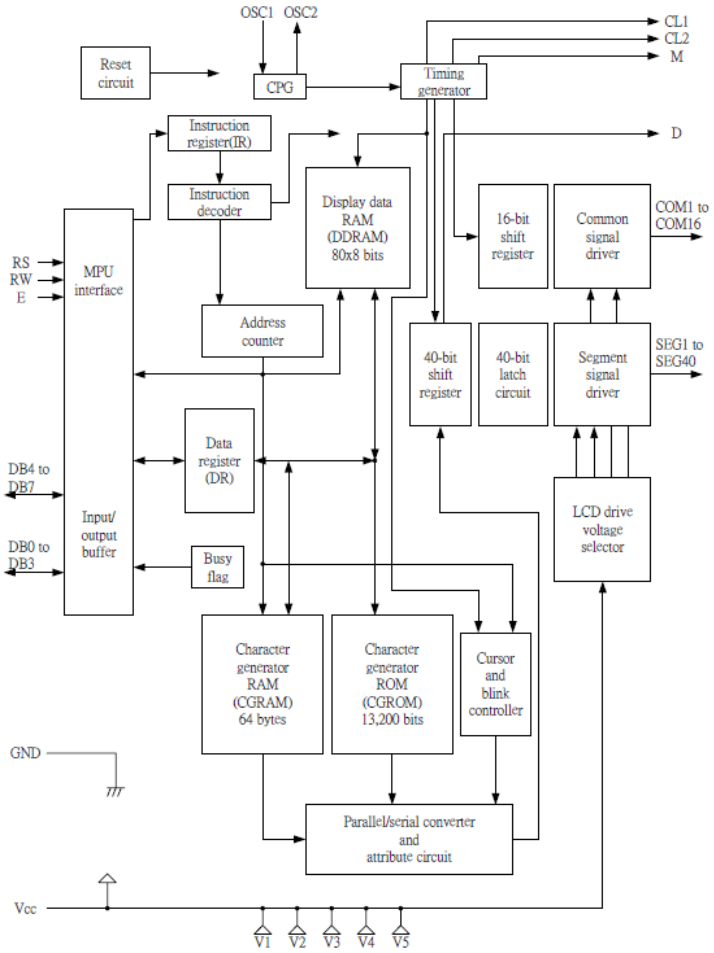

Figure 13: Block Diagram

Reference - Sitronix ST7066U Controller Driver

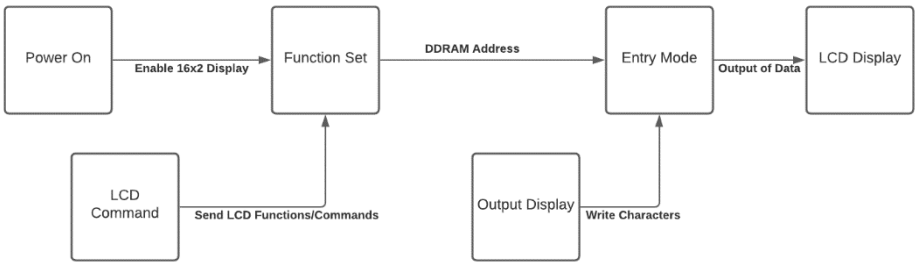

Figure 14: Basic Dataflow Initial Diagram

8.3 Assumptions Made

Assumptions made include that the options the user can configure is limited and hence the LCD will only write out and not take input as well as only

8.4 Constraints on Module LCD Performance

The delays offer a form of constraints as it is a requirement for the operation of the LCD and hence, there will always exist some delay between the user input and the display on screen. Furthermore, there will be a delay between the robot's readings and the display on the LCD screen.

9. PID Controller Module Design

9.1 Module Requirements: PID Controller

9.1.1 Functional Requirements

A PID controller was used during User Assist and whilst controlling the speed at which the device was driven. For User Assist, the PID controller ensured that the device was at a constant radius away from the barrel and adjusted as such, if it deviated from its path. A simple proportional controller worked best. A PID controller was also used to ensure that joystick values corresponded to the speed at which the motors were driven at.

9.1.2 Inputs

The inputs to PID included a struct that consisted of the desired value that was needed, the current value of the motors, the gains, and the final controller value.

9.1.3 Process

The PID function simply calculated the offset needed to adjust the motors, to achieve the desired path for User Assist and the correspondence of joystick to motor values. The proportional controller was implemented by determining the difference between the current value and the desired value (error) and multiplying by the gain. The integral controller summed the error for each iteration and the derivative controller was the difference between the current error and previous errors. After multiplying by each respective gain, the controllers were added up and then converted into respective motor power outputs.

9.1.4 Outputs

The output of the PID controller was sum of all the controllers, corresponding to the motor offset. This motor offset was then converted into a power output, to be sent to the motors, to change the wheels as needed.

9.1.5 Failure modes

The integral controller can be subject to failure, due to the summation of all error over time, therefore a failure mode was implemented to ensure there was no summation of large error over time. This was implemented by keeping the integral gain 0.

9.1.6 Non-Functional (Quality of Service) Requirements

9.1.7 Performance

A performance aspect that must be considered for PID controller is that an infinite loop would take up computing power and may halt all other operations, until the controller reaches a desirable value. Instead, the controller was used for each reading to reach desired value.

9.1.8 Design Constraints

A software design constraint is not using an infinite loop to reach the desired value.

9.2 Conceptual Design: PID

The design of PID was made such that the gains used were rather small. This is because the error obtained for both User Assist implementation and encoder-joystick implementation were rather small and therefore, no excessive gains were needed to correct these values. This resulted in no integral and derivative controllers implemented for non-privileged users, however, it can be altered for privileged users. The process of PID is such:

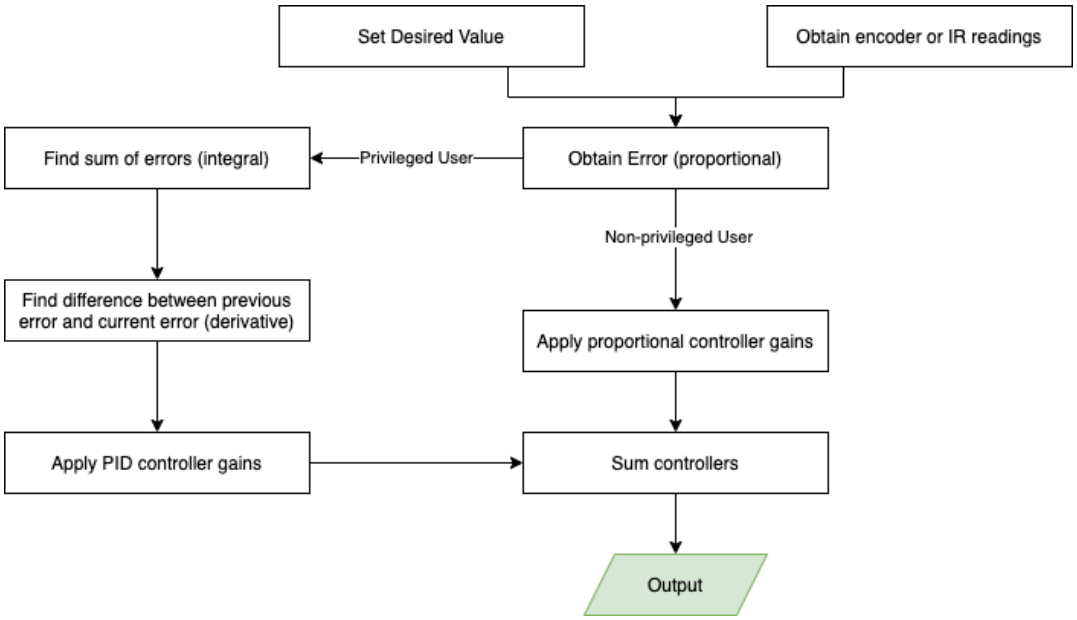

Figure 15: PID Data Flow Diagram

10. User Assist Module Design

10.1 Module Requirements: User Assist

10.1.1 Functional Requirements

10.1.2 Inputs

The inputs needed for User Assist to function properly, include the global struct PIDVariables IR_PID, where PIDVariables is the name of the struct. This stores the calculations needed for PID, including the desired distance from the barrel, the current distance from the barrel, error, summation of error for integral control and previous error for derivative control. User assist mode also uses another struct that stores the IR readings from both left and right sensors into a struct, in type unsigned chars, called IRResult. A global struct was made for this, called readings.

User Assist also uses the global variable mode, type char, to determine whether the robot is in manual mode (mode = 0) or in Assist mode (mode = 1). Another global variable used is UAFlag, which determines whether user assist is conducting a left or right turn.

10.1.3 Process

The purpose of User Assist is to allow the robot to take control and assist in completing a turn around the barrel. User Assist will automatically begin, when a barrel is detected within 15cm, and exit when the right joystick is pressed. The functions used to make this possible include:

Function	Description
TMR0Setup	Sets up the timer0 to meet the IR sample rate
userAssist	Polls until timer0 flag is set, and resets timer. Calls irCalc function.
irReadValues	Obtains IR readings from both left and right IR/
irGetMean	Converts analog reading into millimetres and outputs the mean
irCalc	Determines if IR readings are within a range of 15cm and if the left or right barrels are closest. Also implements the PID function and irToMotor function.
irPID	Calculates the controller value, using the current IR readings and desired output from the barrel.
irToMotor	Converts IR readings into turning power for motors.

The process in which this occurs, includes first obtaining IR readings from both left and right sensors, and then obtaining a mean value for both sensors. After obtaining a mean IR reading, determine which barrel is within a 15cm range and what turn to proceed with, determined by setting the UAFlag. When UAFlag is 1, a right turn is desired and when UAFlag is 0, a left turn is desired. Then the PID is implemented to determine how much closer the robot must get to, to achieve a set turning radius of 22cm from the centre of the barrel. The controller output is then converted proportionally to adjust to power sent to both motors. Once the right joystick is pressed down, the User Assist mode flag is set off to disable the User Assist functionality.

10.1.4 Outputs

The main output of User Assist is from the irToMotor function, which converts the desired change for motors into a power output. In the format of a struct, discussed in section 2.11. Another output that is essential is the output of the timer flag, as this ensures IR readings are taken at a certain sample rate.

10.1.5 Timing

The IR sample readings must be done at the sample rate entered in factory mode or done at a set rate for standard non-privileged users (section 2.2), therefore User Assist must also ensure correct IR readings are updated, each time.

10.1.6 Non-Functional (Quality of Service) Requirements

10.1.7 Interfaces

Desired interface was to display IR variance to controller, which would not impact the overall functionality of User Assist. The variance was obtained, however, displaying on controller via XBee was not possible, refer to section 2.7.

10.1.8 Design Constraints

Design considerations that took place initially were associated with hardware, and the potential inaccuracy of IR readings. However, IR readings were within a tolerance of 1cm of the actual range and no software adjustments had to be made while undergoing assist.

10.2 Conceptual Design: User Assist

The software design of User Assist can be modelled:

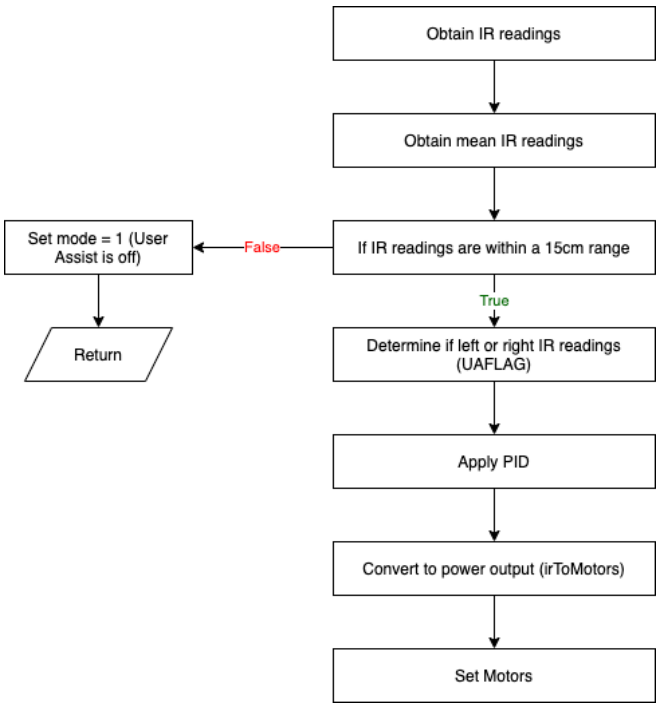

Figure 16: User Assist Data Flow Diagram

The format of all inputs and outputs were of type char, to reduce the potential of a stack overflow, and unnecessary use of memory. Exceptions were made for PID calculations to obtain more accurate outputs. Structures were primarily used as outputs, their formats include:

Structure	**Variable Description**	**Global Structure used**
typedef struct { char desired_value; char IR; char last_error; float current_desired_variable; char error; float integral; float derivative; }PIDVariables;	desired_variable – Desired value wanted	PIDVariables IR_PID;
	IR – Stores current IR reading	
	last_error – Stored the previous error for derivative controller	
	current_desired_variable – The calculated sum of all controllers	
	error – The difference between the desired variable and the read variable	
	integral – Stores integral calculation	
	derivative - Stores integral derivative	
typedef struct{ unsigned char left; unsigned char right; } IRResult;	left – Stores left IR reading	IRResult readings;
	right - Stores right IR reading	
typedef struct{ char left; char right; } Motor;	left – Stores left motor power reading	Motor IR_power;
	right - Stores right motor power reading	

Global variables used for user assist include:

Variable	Format	Explanation
UAFlag	Char	Determines the direction of turn. UAFlag = 1, is a right turn and UAFlag = 0, is a left turn.
Mode	Char	Determines if user assist if ongoing or has ended. mode = 0, is user assist is ongoing, whereas mode = 1, means user assist if off.

10.3 Assumptions Made

Assumptions made were that the barrel used for racing would have a radius of 10cm, which were taken into account while calculating the IR readings to motor power.

10.4 Constraints on User Assist Performance

Constraints made to improve performance including, limiting the use of floats and integers to reduce the amount of memory used.

11. IR Sensor Module Design

11.1 Module Requirements: IR Sensors

11.1.1 Functional Requirements

11.1.2 Inputs

The main inputs required for this Module are values for the number of samples per estimate and the estimation frequency. These values will have to be set from the Factory Mode by the user. The last bit are the readings that are acquired by the IR Sensor, which will be accessed through the pins Port A.2 and A.3. These values will be accessed through the Analog to Digital Converter.

11.1.3 Outputs

The IR Module calculates and acquires the mean distance between a specific IR Sensor and any obstacle in its way. This result is sent back to the motors, in order to let it know if it should modify its angle or not. This is to assist the PID Control.

Also, in one version of Assist mode that can be set in Factory, the controller will have to display readings of the mean and variance calculated by the IR sensor. The results will, thus have to be sent back to the controller through the XBee once this mode is set. However, this couldn't be done because the XBee Modules aren't able to suitably send data back from the motor to the controller.

11.1.4 Timing

The time taken between each estimate from the sensors will be determined by a user input through Factory Mode. This is set in terms of a scale from 1-100. However, the span of the frequency that can be set is 284 to 28409 Hz.

11.1.5 Non-Functional (Quality of Service) Requirements

11.1.6 Performance

The results are fairly accurate, normally giving results within an error of ±0.5 cm. The computational rate of the IR readings will differ, mainly because of the possibility of the user changing the number of samples per estimate and the sample rate.

11.1.7 Interfaces

A feature that can be considered is the reception of the mean and variance values from the IR Sensors by the controller for display. This is a specific mode set in the Factory Mode, which doesn't assist the operation of the robot in any way expect to return the read values from the IR in Assist Mode. This couldn't be implemented because of a fault in the XBee Module, being unable to send data across both directions.

11.2 Conceptual Design: IR Sensors

The readings from the IR Sensor were calculated through the A/D Converter, through pins on Port A.2 and A.3. An oscillator frequency scale of 8 was set, and all pins were set to analog with Vdd and Vcc as the reference. The final readings were left justified so that the most significant 8 bits are acquired. With subsequent delays to facilitate the process, the A/D Converter is activated and the raw reading of the converter is acquired in an 8-bit format. This is all done in the analogRead() function. This function also required a character that contains either 2 or 3, depending on which IR the values have to be computed for.

The function irReadValues() is responsible for the gathering of the samples using the analogRead() function. At a moment, samples from both IRs are taken. Using a buffer variable as a counter, it stores each individual sample in the space in an unsigned char array as denoted by the buffer, after an increment in the buffer. The buffer variable will also reset to zero once the set samples per estimate is reached, so that the results are found from the beginning for a new estimate. This function is looped for the duration of the samples per estimate to allow all the samples to be collected.

The function irGetMean() takes the arrays set in the previous function and sums the samples up to find the final mean estimate. The calculation for the estimate is as follows

$$Mean = 180 - \frac{Sum\ of\ Samples}{Samples\ per\ Estimate}$$

This particular method was used from analysis of the raw samples from the previous functions. Normally, the maximum value that would be acquired is from around 2.8 to 2.9 V, if the obstacle is 3 cm away from the sensor. Upon scaling for the 8-bit results, it was found that the result should be subtracted from 180 to lead to an accurate result. Note that the result will be in millimetres.

The method mentioned in the Data Sheet for the sensor, using the distance-voltage graphs, was attempted. However, the result from this was not as optimal, and lead to several errors at times. This method was far more reliable than the use of the graph.

The timing was set through the operation of Timer 0. Since the system receives the Sample Rate in Hz, it is necessary for it to convert the frequency into cycles that can be processed by the timer.

Setting the timer to a 16-bit with a prescaler of 1:2 we get a scale of the timer from the user. This scale is set into timer iterations using the following formula

$$Timer = 65535 - Sample\ Rate * 655$$

This formula was used mainly so that the timer could span from the least possible value to the most possible value. The prescaler was set as 1:2 to allow for a good range of resultant frequency values. After this, the computed value is split into its higher and lower bytes and inserted into the respective timer sets.

11.3 Constraints on Module X Performance

The IR sensor has a span of 4-30 cm. Anything closer than 4 cm will result in erratic readings in the sensor, making it return a distance far greater than it should. Beyond 30 cm, no results will be picked up, and the values will be read accordingly. Note that these values are given a factor of safety, and in reality are about 3 to 40 cm in span. Therefore, meeting this span can work, but it isn't recommended due to the lack of optimisation in the extraneous areas.

12. User Interface Design

The user primarily interfaces with the device through the controller, including entering all configurations and navigating the robot. To power the motors of the device, the robot must be powered on, using a switch located on the power board, lighting green. The controller can be powered on, using the switch located at the centre of the console. A welcome message will be displayed on the LCD before the configuration menu will appear. To navigate through the items on the menu, use the left joystick to scroll up and down, revealing previous and next menu items, respectively. At this menu, three options will appear, including Configuration, Manual Mode and Assist Mode.

Manual mode allows the user to start driving the robot, using the right joystick to control the speed of the robot and the left joystick to control the direction. To enter manual mode, push down the left joystick and all menu items will disappear, allowing the user to start navigating. Pushing the right joystick forward will speed up the robot forward and pushing it down will move the robot backwards. Pushing the left joystick, left and right, will navigate the robot left and right. The robot can turn on the spot by simply using only the left joystick.

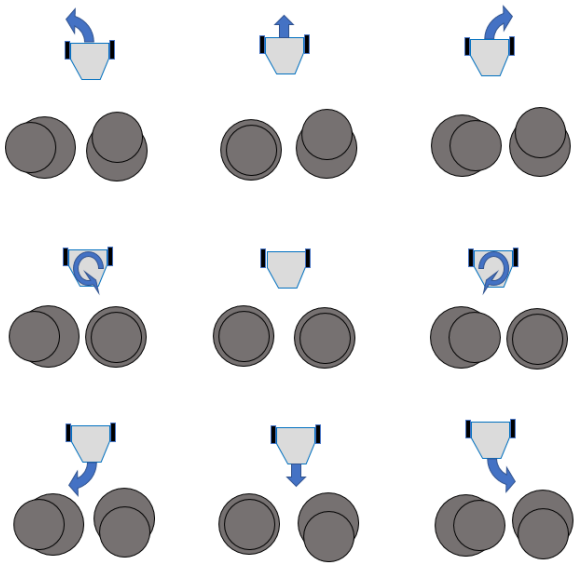

Figure 17. Illustration of Joystick Movement Control

The second option is assist mode, that allows the user to drive in manual mode, until a barrel is detected. In which the robot will take control and continue a circular path around the barrel, until the user pushes down the right joystick to exit out of the circular path. It will then enter back into manual mode, for the user to control, until another barrel is detected.

Configuration allows the user to enter a maximum speed, that the robot will be able to navigate at. However, to navigate through the sub-menu for configuration mode, use the left joystick and scroll left and right, moving to the previous menu and next sub-menu, respectively. Once, in configuration mode, using the left joystick, scroll right once again to enter in numerical values in RPM. The numerical values can be set by pushing the joystick up and down for each blank character box, incrementing or decrementing values between 0-9. The maximum speed that can be set is 80RPM. If a value greater than 80 is entered an error message will occur After entering the correct digit into the first character box, push down the left joystick and scroll right to enter the next digit.

Figure 18. Controller Illustration

12.1 Classes of User

As stated previously in Section 2.2, those who operate the BarrelBeast robotic system can be classified into two streams of users: **privileged users** and **non-privileged users**.

- Privileged Users: Are users who have access to all modes, as well as factory mode, used primarily for the development of the robot.

- Non-Privileged Users: Non-privileged users refer to individuals who are not involved in the development of the robotic system and can only access standard features, the 3 basic modes.

12.2 Interface Design: Non-Privileged User

12.2.1 User Inputs and Outputs

The user inputs into the system while starting the controller, entering the desired modes. The configuration mode allows the user to set the highest RPM. Manual mode allows the user to navigate the robot with full control, adjusting speed and turning proportional to the movement with the joysticks. In assist mode, the user will have partial control over the robot, allowing it to take control around barrels. The user can change all modes at any point, by pressing the manual/ assist button, allowing the user to re-enter any menu items.

12.2.3 Input Validation and Error Trapping

An error message is displayed if an incorrect value, out of the range is entered.

12.3 Interface Design: Privileged User

12.3.1 User Inputs and Outputs

Developers have the option of entering factory mode by holding down the manual/ assist button and pushing the left joystick, right at the configuration menu. Upon entering factory mode, developers can scroll through more menu items, including Configure PID, Set Max Yaw, Configure IR and Display. In Configure PID, a sub-menu will allow the developer to enter individual the Proportional, Integral and Derivative gains, in the same way the max speed was set. The Configure IR menu allows the user to enter Samples and Sample Rate. Samples determining the amount of IR readings that should be taken at once and Sample Rate, indicating at what rate the

IR readings should be taken. Set Max Yaw, allows the user to enter the maximum yaw rate and Display allows the developer to display infrared reading on the LCD.

12.3.2 Input Validation and Error Trapping

An error message is displayed the same way if an incorrect value is entered

13. Hardware Design

13.1 Controller

The shell and PCB of the controller are designed by the group. The shell was printed by FDM 3d printer, and the PCB was out sourced to overseas company for production. Figure below shows the engineering drawing of the controller.

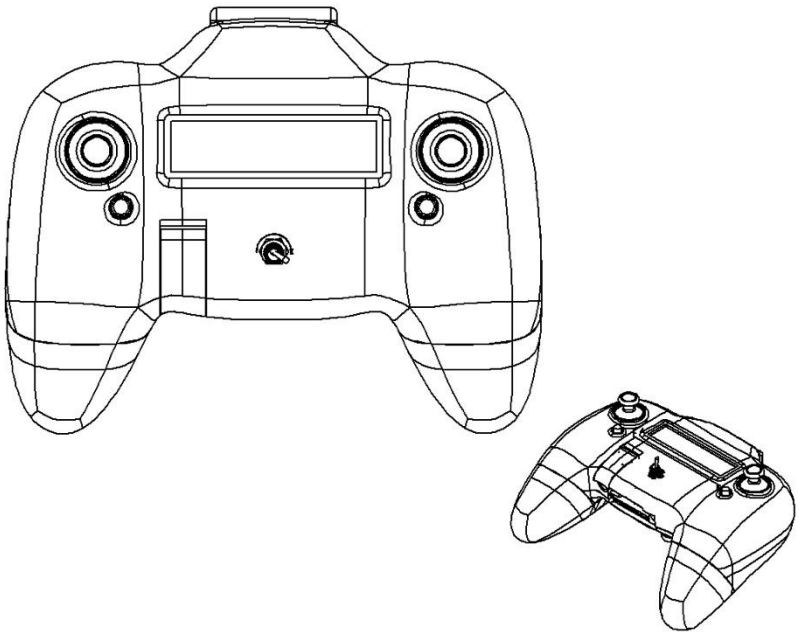

Figure 19. Controller Mock-up

13.1.2 Power supply

The power supply of the controller is a 9v battery. The battery powers a voltage regulator on the PCB designed by the team, and the voltage is regulated to 5v. The voltage regulator powers the minimal board. Although minimal board can be powered directly via 9v battery, the power supply connector on the minimal board will significantly increase the volume of the controller. Instead power the minimal board via power supply connector, it is powered by 5v port.

13.1.3 Hardware quality assurance

Since the controller shell was printed using FDM style 3D printer, the overhangs require supports during the printing process. The sliding software Cura can only generate printing support in one setting. However, due to the accuracy requirement is different for different part of the shell, some support is designed in Solidworks. The figure below shows the support for over hangs. The black arrow is pointing designed support, and the light blue mesh is the generated support.

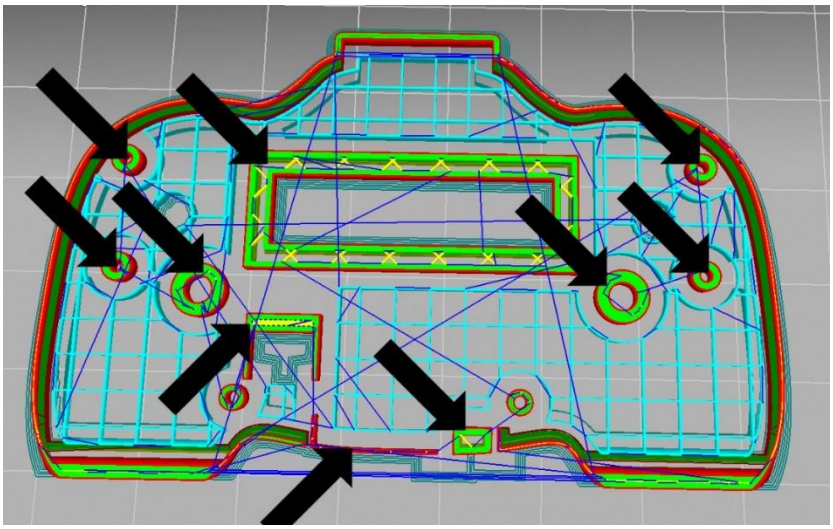

Figure 20: Shell Schematic

Due to the characteristic of the FDM 3D printer, the top surface has more smooth finish than the overhang surface. Some parts of the controller are sliced into multiple pieces and glued together by CA glue. Figure below shows all parts of the controller and how they are laid on the printing bed.

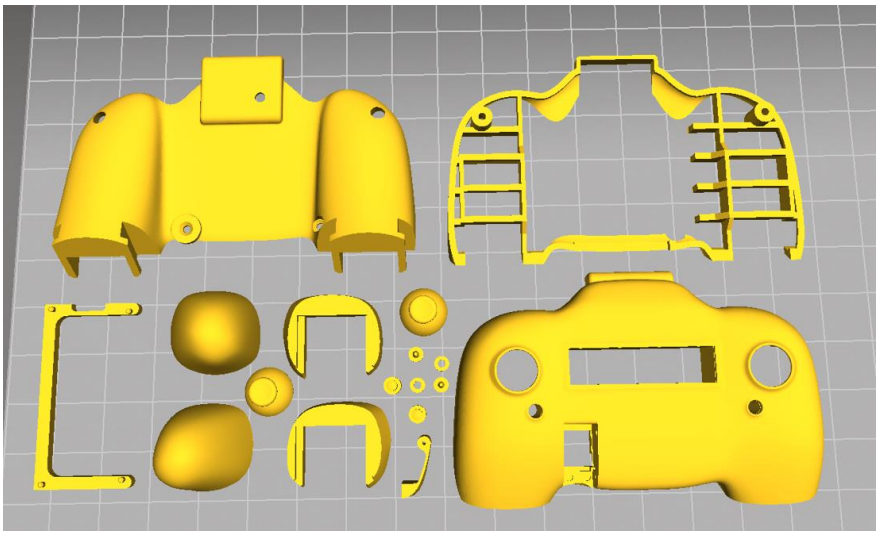

Figure 21: 3D Printing layout

13.1.4 Hardware validation

Since the mount hole for PCB is printed as overhang sitting on support, after remove the support error may occurs. To ensure correct fitment, the distance from the PCB mounting hole to bottom surface was measured, and extra material was removed.

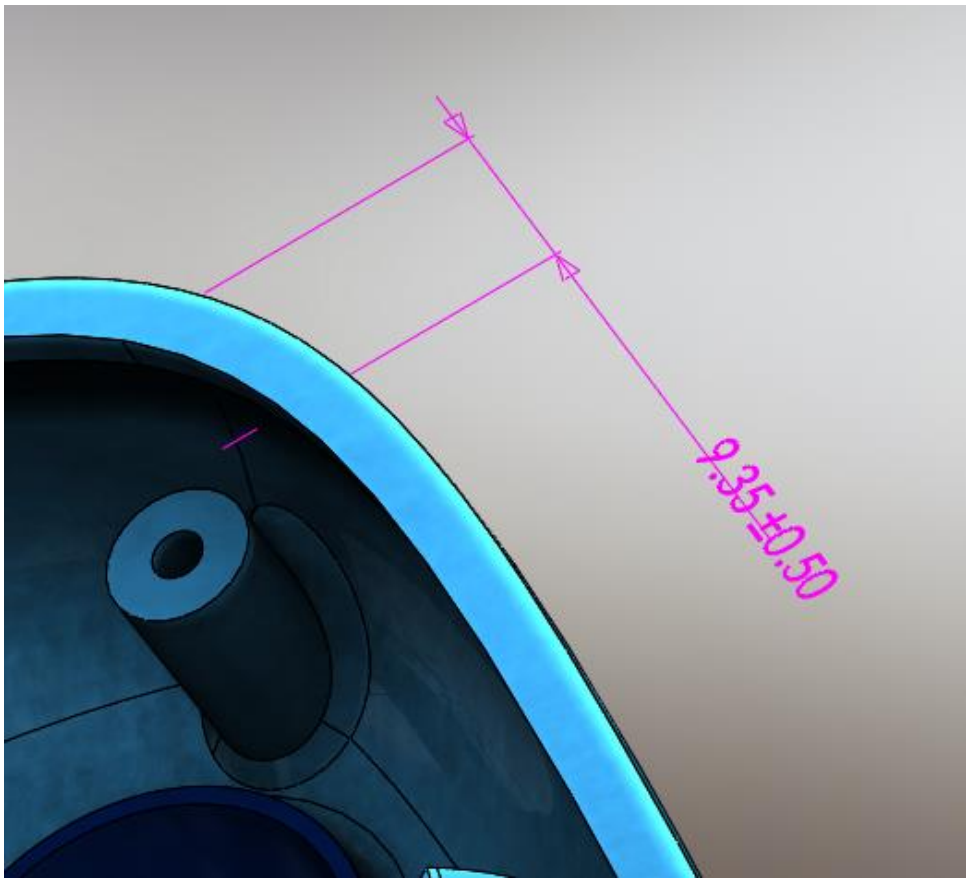

Figure 22: Tolerancing Check

Before the PCB is connected to the minimal board, the power supply voltage was measured to ensure the system will have correct voltage supply.

13.1.5 Hardware Maintenance and adjustment

The right handle of the controller can slide off for changing battery. The left handle can also slide off, a spare 9V battery can be fitted into the spare battery housing. The spare battering can counter the weight of the main battery to improve the user experience. However, the spare battery is not necessary for the controller to operate.

13.2 Robot

The IR sensor mount, circuit board mount and wiring loom was designed by the team. Figure below shows the engineering drawing of both mounts.

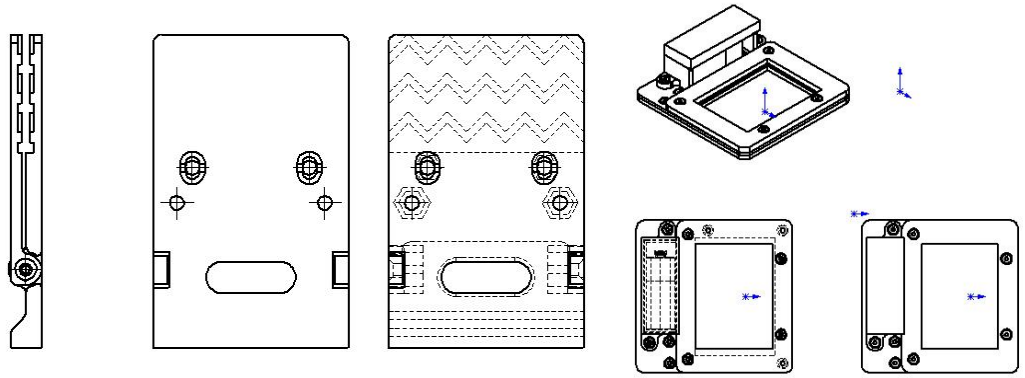

Figure 23: IR Mounting 3D Printing Layout

13.2.1 Power supply

The robot has 2 source of power supply, 12V battery for motors, and 9V battery for control module.

Circuit Design

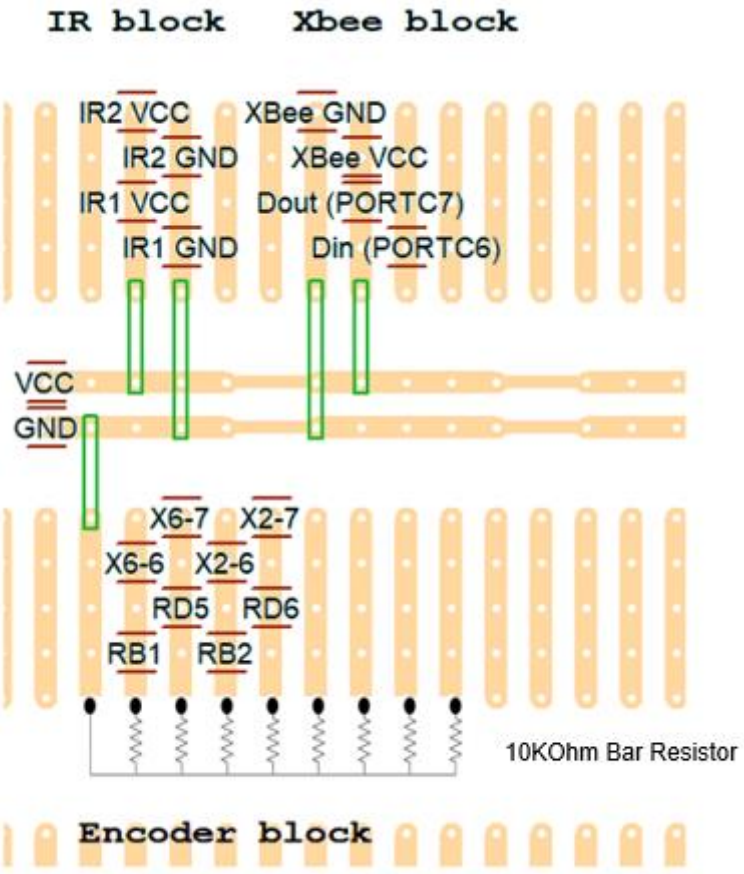

Figure 24: Veraboard layout

13.2.3 Wiring Assignment

Table: Pin assignments on the PIC18F452 on the robot

PIC pin	Pin use	Connections	Pull down on Veroboard?
RA2	IR_R	Right IR	No
RA3	IR_L	Left IR	No
RB3	PWM2	X2-4	No
RC2	PWM1	X6-4	No
RC6	Din	Xbee Din	No
RC7	Dout	Xbee Dout	No
RD0	Encoder _RA	X2-6	Yes
RD1	INB2	X6-3	No
RD2	INA2	X2-3	No
RD3	INB1	X6-2	No
RD4	INA1	X2-2	No
RD5	Encoder_RB	X2-7	Yes
RD6	Encoder_LB	X6-7	Yes
RD7	Encoder_LA	X6-6	Yes

13.2.4 PCB Schematic

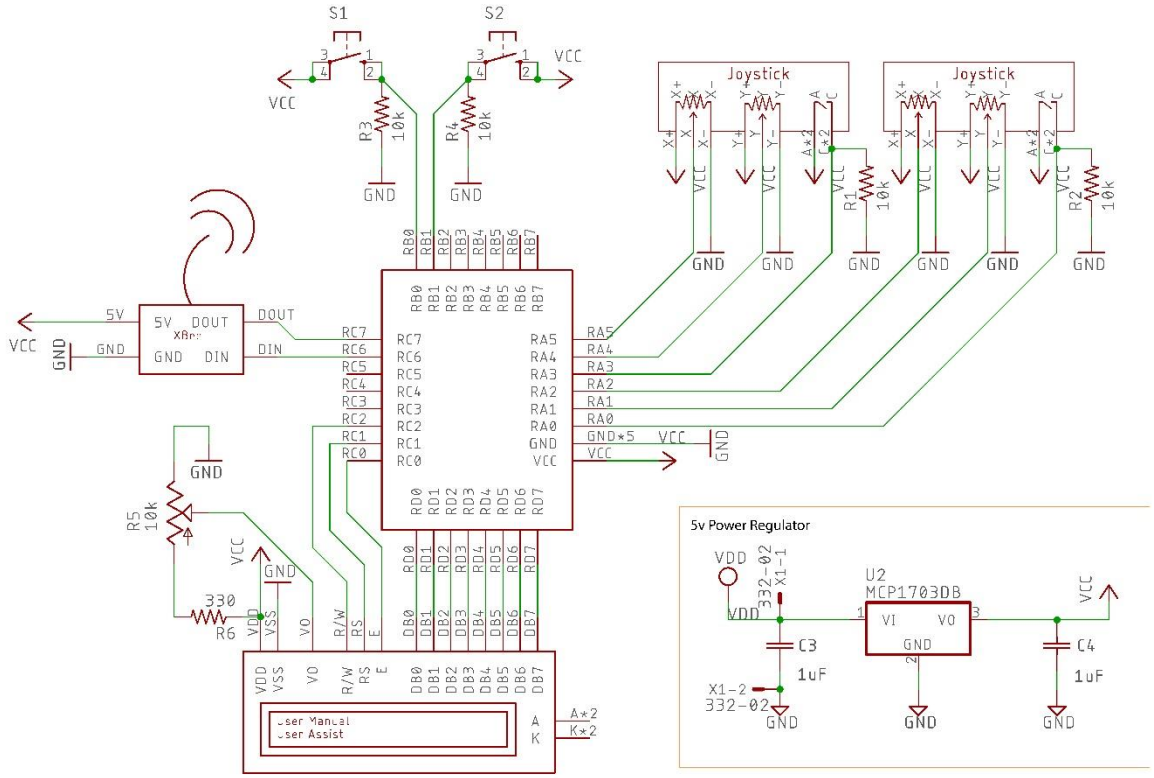

Figure 25: PCB Schematic layout

13.2.5 Wiring diagram

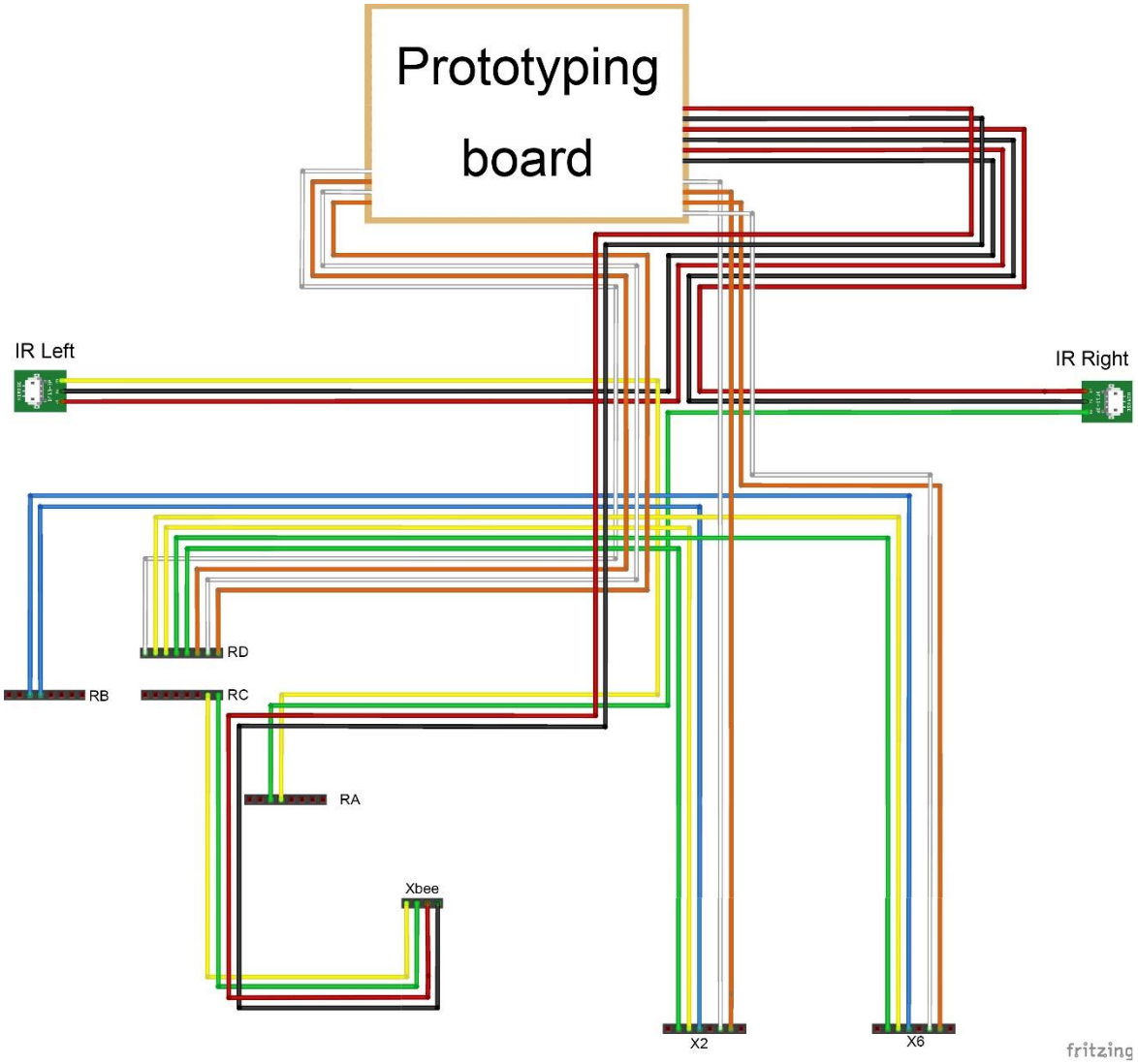

Figure 24: Wiring to Prototyping board

13.2.6 Hardware quality assurance

Both IR mount and circuit mount are printed by FDM 3d printer. To ensure the quality, the base surface of every part is designed to be flat. This will increase the ability for printing parts to stick on to printing bed, hence increase the quality of finished parts. The base of circuit mount is sliced into to piece to eliminate overhang during the printing process. The battery housing is also sliced into two pieces. The result of it is to increase printing quality and decrease printing time and material waste. All printed parts and the layout are shown in the figure below.

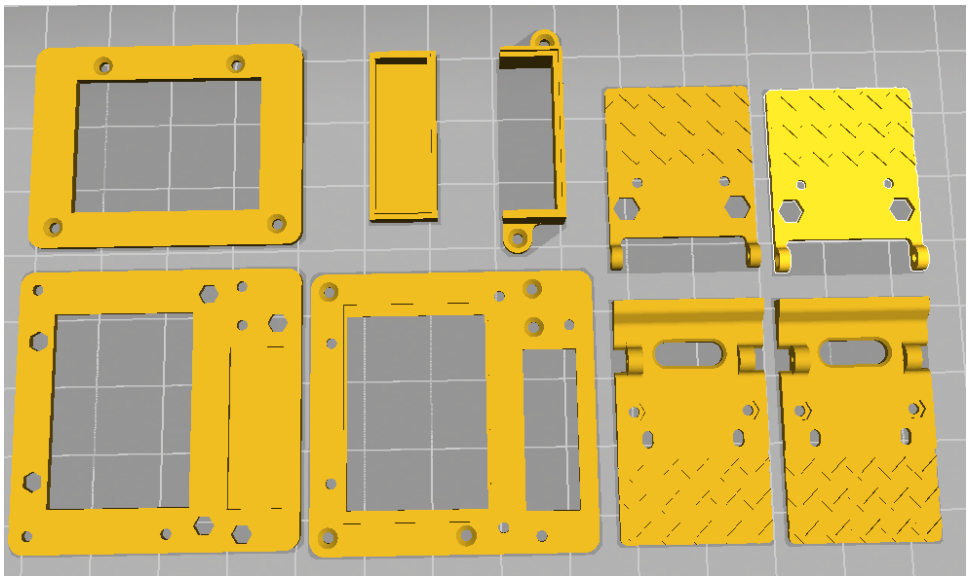

Figure 25: IR Mounting

13.2.7 Hardware Calibration Procedure

The IR sensor mount are designed to allow adjustment along y axis. The IR sensor mount are clamped to the robot body. To adjust IR sensor position, untighten the two flat head screws on the IR sensor mount. After adjusting it to the preferred location, tighten these two screws to secure the mount.

13.2.8 Hardware Maintenance and adjustment

To replace the battery, remove the two screws holding the battery housing. After changed the battery, place the battery housing to its original location and tighten the two screws.

14. Software Design

The software requirements and overview have been dealt with elsewhere in this document. The present section addresses the design and implementation of the software that forms the Barrel Beast.

14.1 Software Design Process

The software design process involved separation of each component and therefore, engineering each component individually. Hence, the software design involves Modular Programming with a Top-Down approach to get an executable prototype. The individual components would require to either take in or initiate data and output data that would be handled by another component. Upon completion of development with each individual component, integration would begin. Integration involved firstly; the creation of specified functions that would allow for data to be transferred from one module to another. From this point the process of testing would take place alongside integration which would involve tuning the software in order to gain the desired system performance.

14.2 Software Development Environment

The tools employed within the software environment include the *MLAP X IDE v5.20*, *C18 Compiler* was utilized for the module's design and *Solidworks* which was employed for the 3D software design. Furthermore, for the printing of the initial designs of the controller and robot support aspects employed Cura, Rhinoceros with T – Splines software.

14.3 Software Implementation Stages and Test Plans

The implementation stage of the system design involved a combination of software pseudocode with functions planned out prior to testing that would handle data flow between modules. Employing a modular programming software design technique allows for the following components to be engineered separately and be divided into more manageable sections:

- LCD
  - Write Out
  - Scrolling
- Motor Drive
  - Drive Left
  - Drive Right
- Encoders
  - Set RPM
- IR Sensor
  - Mean Calculation
  - Variance Calculation
- Joysticks
  - Define Direction
  - Navigation through Menue
- XBee Communication
  - Read/Save
  - Send

Ideally, these components would all have a set of necessary functions that would be called from main. The tasks were divided out to allow for efficient software module creation.

An incremental development approach was also employed with the following stages:

1. Intra-Module Testing
   a. MPLAB Simulator, Hardware Testing, Unit Testing.
2. Inter-Module Testing
   a. Integration between Modules
      i. Ensuring modules work together as intended.
3. System Integration Testing
   a. Whole system testing, integration between robot and commander.

Testing of each module was performed firstly individually via unit testing and upon implementation repeated unit testing would take place allowing for the desired inputs and outputs to be gained from each module. Therefore, such a design was accurate in relation to the system requirements as each component was made certain to be correlating, handling and transferring data as expected prior to moving to another component for implementation.

14.4 Software Quality Assurance

The software design quality has been upheld through placement of documentation standards such as indentation and regular commenting. Furthermore, the employment of unions, linked lists and structs, macros and definitions along a set of the modules allowed for simplification in the code design. The employment of functions related to each module has permitted further simplification of the software and in turn has reduced response delays with less repetitive code design. In order to keep code versions consistent, GitHub and a password secured website was utilized for code sharing. (https://mtrx3700.galetora.com) For the documentation of code Doxygen was employed for a consistent styling. Furthermore, this method of sharing and updating of code is reliable allowing for rollbacks in case of unforeseen errors or failures.

14.5 Software Design Description

14.5.1 Architecture

The software is planned to be designed as a program which is triggered and calls multiple functions which interact with each module.

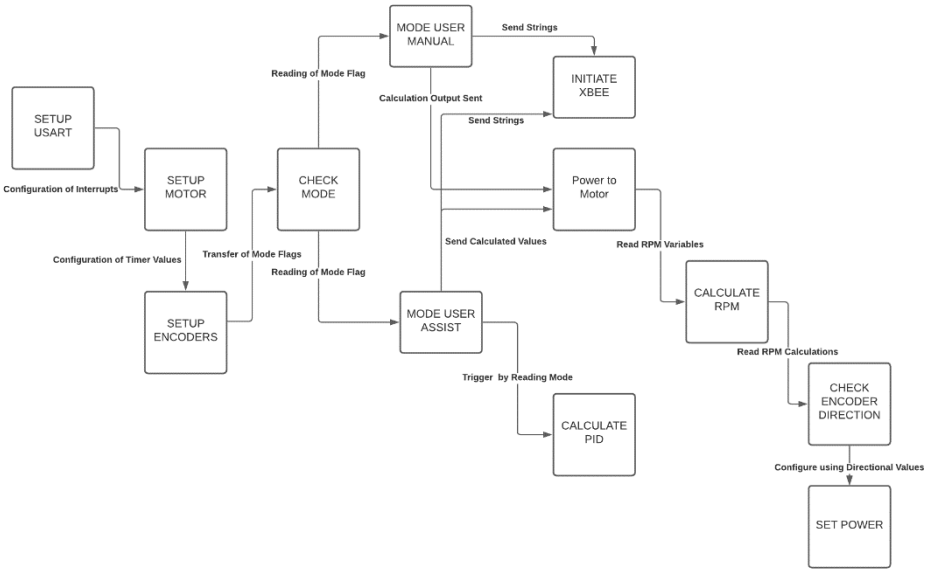

Figure 26: Program Data Flow Diagram

14.5.2 Software Interface

The public interface of the robotic system involves all the following details for each module:

- **LCD**:
  - The LCD public interface involves the menu navigation that contains scrollable options and configurations for the robot control.
- **Motor Drive**:
  - The public interface of the motor drive involves the movement/rotation of each driver and furthermore, the toggling on and off as well as rotational velocity control for turns to be performed.
- **Encoders**:
  - The encoders contain public interfacing aspects which communicate with the motor drive and the XBee communication when RPM calculations are completed.
  - Furthermore, has function which works around user assist mode to run necessary IR calculations communicating with the IR Sensor module.
- **IR Sensor**:
  - The IR Sensor was planned to have a public interface that would deliver readings outputted onto the LCD in the form of a variance reading and a mean value.
- **Joysticks**:
  - The joysticks readings are further displayed on the LCD as an aspect of the public interface.
  - Furthermore, the user interface allows the joystick to handle controller input and configurations and display such changes publicly.
  - The joysticks also publicly communicate with the motor drivers to allow user defined movement of the robot.
- **XBee Communication**:
  - The XBee contains public entities that send strings from the robot to the controller wirelessly over USART and XBee transmission. Furthermore, the software has been designed to receive and save joystick values as well as configuration values selected by the user.

14.5.3 Software Components

- **LCD**:
  - The LCD functionality involves a menu defined through a union which can be navigated with the integration of the joysticks which send a direction through the XBee component. This direction is sent in the form of a message packet of data to be read.
- **Motor Drive**:
  - The motor drive module is designed to manipulate the motor power sent to each wheel by reading the joystick values communicated by the XBee communication. More specifically, this involves calculations for the power values and RPM which works with the encoder module and furthermore, has power set according to PID calculations.
- **Encoders**:
  - The encoders module calculates the RPM of each encoder utilizing prioritized interrupts via timers. Furthermore, this PRM is converted to a power output suitable for the motors. When the power conversions are completed, the values are employed in the PID calculation function.
  - Proportional control is implemented for the adjustment of power that is outputted to the motors via a function.
- **IR Sensor**:
  - The IR sensor utilizes structs to store PID variables for use in calculation of IR and PID. Furthermore, the module incorporates reading voltage from a specified channel and returns the voltage in 8-bit resolution.
  - The IR values rea values from right and left IR sensors.
  - A function, furthermore, stores IR values over the sample period and calculates the mean and returns the value to a struct.
  - Contains functionality to set default values of configuration settings for normal and factory modes.
- **Joysticks**:
  - Joysticks functionality involves reading voltage from joysticks and potentiometers and its push button switch.
  - A vector struct will store fields of, and values of the joystick (including the push button stage) respectively.
  - Joysticks will also return the direction vector value and restrains the given vector within a tolerance from the default center positions.
- **XBee Communication**:
  - The XBee declares the USART receive interrupt and hence is called for the high priority interrupt.
  - The XBee is utilized to send strings from the robot controller wirelessly over USART and XBee transmission.
  - The XBee will save joystick values, and joystick values into the velocity struct if the appropriate start bit has been read.
  - Furthermore, the program will save configuration values into the correct index position of the configuration array defined in the overall configuration union.
  - The configuration function sets the default values of the configuration settings for normal and factory modes.

14.6 Preconditions for Software

14.6.1 Preconditions for System Start-up

For system start-up to perform as per expected, there is no required order of powering up between the controller and robot, however system operation takes place when both the robot and commander are powered.

14.6.2 Preconditions for System Shutdown

For an effective power down of the system, it is recommended that the controller is not in manual or assist mode and the robot is not in mid movement when the power in the controller is turned off. There are no required conditions for the robot to be shut down. There is no order required to power down between the controller and robot.

15. System Performance

15.1 Performance Testing

There were no errors found in Manual Mode. The only errors could be found through occasional drift in the motors when the controller wasn't adjusted or even the occasional humming from the robot when left on for a long period of time.

For Assist Mode, however, there were errors found regarding the application of the 'closest IR is best' method. This method ended up making the robot jerk in an opposite direction whenever there is an extraneous obstacle closer than the barrel. The interference between the two causes for erratic movement on the motors.

15.2 State of the System as Delivered

The Robot was generally operational for Manual Mode, while it wasn't as effective in Assist Mode as mentioned in the previous section.

One major issue is the inability of the XBee in sending results back to the controller. Because of this, neither the Factory mode displaying the current value nor the display of the mean and variance from the IRs could be implemented.

One last issue would be one that we could not find any alternative from, and that is the use of a Factory Menu. It was said that the Factory Mode should not be seen by the user at all, which could not be implemented properly for our design due to the uncertainty of angle that the use of joysticks can offer.

15.3 Future Improvements

A high priority addition would be changing the User_Assist Mode to accommodate for a semblance of close proximity to the barrel. This is done to optimize the traversal of the robot and reduce time when set in Assist Mode, by bringing the robot closer to the barrel while rotating around it instead of keeping it at a constant distance set from the beginning of the mode. This addition will drastically improve the time it takes for the robot to complete the race once Assist Mode is used. This will also reduce the risk of interacting with external distractions, because once the robot gets closer to the real radius it should circumvent, all other obstacles will be further away from it.

This is not as high priority as the one above, but it is quite significant regardless. This would be the correct alignment of the IR Sensors. The User Guide had suggested that the Sensors be placed perpendicular to the ground to optimize its use, but design miscommunication had led to the sensors being placed parallel instead. This hadn't hampered the IR readings drastically, which is why it was left in. However, it would be a source of minor error to correct in the future.

Finally, a useful addition would be the correction of the XBee Module, to allow it to send data in both directions. This does not improve the operation of the system in any way, but it does allow for the completion of two features that require this: the Factory Mode displaying the current readings and the mean and variance return from the IR sensors.

16. Safety Implications

Foreseeable safety hazards associated with the robotic system have been assessed and are detailed in Table 2. The risks were assessed by severity and likelihood, assigning a value between 0 and 10, values closer to 0 representing low severity and likelihood and values closer to 10, representing high severity and likelihood. Appropriate mitigation methods have been implemented and have resulted in reduced risk. They have been identified as acceptable risks in the design of the robotic system.

TABLE. Safety Implications

	Risk	S	L	Risk	Mitigation Method	S	L	Risk
RSK 1	Choking on commander or mobile robot device components by children	9	5	45	Promote the user to keep the commander and mobile robot unit out of reach of children under the age of 5 years old in user manual. Ensure small electrical components are soldered and fastened by commander casing and that the electrical components underdo regular inspection.	9	2	18
RSK 2	User or other individuals are caused injury by tripping over commander or mobile robot unit due to inappropriate storage of these devices	5	6	30	In the user manual, instruct the user to store the device in a well-ventilated closed off area which is where it does not have potential to cause people to trip over the device. When in operation, instruct the user to operate device from a metre away in the user manual.	5	**3**	**15**
RSK 3	Live internal electrical components in the commander or mobile robot unit become exposed causing electric shock and burns to user when in contact	7	4	28	Promote cleaning of the devices with a dry cloth and instruct the user to operate device and store in a well-ventilated area. Ask user to avoid exposure to moisture, heat, vibration, mechanical damage, corrosive chemicals or dust. System must be regularly inspected to obvious damage, defects or modifications to the electrical equipment.	7	3	21
RSK 4	Electrical components on the commander or mobile robot unit overheat causing burns to the user when touched	7	4	28	Include user manual instructions which warn the user to put down commander down or avoid picking the commander up if any heat is detected and avoid prolonged use of system. Use insulative materials for commander shell.	7	3	21
RSK 5	User drops the commander or mobile robot unit causing physical damage to hardware	5	5	25	Design the commander to be lightweight with a shape and material that is easily-gripped and provides user comfort. Include user manual instructions which promote the user to take extra care of the commander or mobile robot unit when lifting and transporting these devices.	5	3	15
RSK 6	Mobile robot crashes into user or obstacles in surrounding environment causing injury or damage due to lack of familiarity of user with mobile unit motion control or failure modes	5	5	25	Implement an emergency stop button on the commander that can be used to immediately deenergise the motors if the user perceives potential risk of injury or damage to the mobile robot device whilst in motion. Instruct the user to operate the robot on dry, flat surfaces within 5 - 35° temperature range.	5	2	10
RSK 7	Lithium Iron Phosphate Batteries cause projectile injuries or burns to the user or nearby individuals due to potential current discharge of 168 A	8	3	24	Ensure the user does not leave Lithium Iron Phosphate Batteries unattended when charging by including appropriate instructions in the user manual. Instruct the user to turn off the charging device immediately if high heat or strange smells are detected emanating from the battery	8	2	16
RSK 8	Electrical components of the commander or mobile robot unit are short circuited causing device failure	6	4	24	Encourage the user to check for faults and defects before user operation of the robotic system. Discourage electrical usage during storms in user manual. Provide regular inspection and maintenance services for electrical components.	6	1	6
RSK 9	User experiences headaches or sore eyes from overexposure to the bright LCD screen	7	3	21	User is advised not to use the system in the user manual if they have previous medical conditions which prohibit them from observing bright screens for prolonged periods of time. Warnings are included in the user manual which warn the user to stop using the device immediately if at any time the user experiences headaches or sore eyes while operating the robot. Prolonged usage of the device is also discouraged in the user manual.	7	2	14
RSK 10	Commander or mobile robot unit are inappropriately operated	5	4	20	Include detailed instructions on how to operate commander in user manual and implement user-intuitive design of user interface.	5	2	10
RSK 11	Non-privileged user accessing factory mode commands	4	5	20	Protect the factory mode from being accidentally entered by implementing unintuitive access method to the factory mode menu in commander menu design.	4	3	12
RSK 12	Settings are not saved to device due to user turning off power during storage process	2	7	14	Include user manual instructions which promote the user to keep the commander and mobile robot device powered on when changing the settings.	2	3	6

17. Conclusions

Group 6 has successfully implemented a robotic system able to manoeuvre around a flat barrel racing course accurately and rapidly. BarrelBeast is an intuitive system which can both be executed in manual and assist mode. The product is also able to provide extended functionality for authorised privileged users, enabling them to manipulate and calibrate settings related to the motion control of the robot such as yaw rate and maximum speed. The project allowed members of Group 6 to gain a stronger understanding of the PIC18F452 microcontroller and develop the skills related to large group projects where both software and hardware modules are integrated. The teamwork skills, organisation skills and technical understanding gained throughout the project are highly valuable and may be applied by the individual members in future group projects.

```
undefined#define TOL 12
#define MTL 50
#define CTL 30

/**
   Vector is struct used to represent a 3d vector in 8bit resolution.
*/
typedef struct{
    char x;
    char y;
    char z;
}Vector;


/**
    initAdc sets port A as inputs. Must be called first in order
    to use the analogRead function.

        @see analogRead
*/
void initAdc(void){
    TRISA  = 0xff;            // PORT A Set as Input Port
}

/**
    analogRead reads a voltage from a specified channel and returns the
    value in 8 bit resolution.

        @param  channel, the pin on port A to be read (i.e. 0 - 7).

        @return The voltage on the pin in 8 bit resolution.
*/
unsigned char analogRead(char channel){
    ADCON0bits.ADCS = 1;       // Fosc/8 clock conversion
    ADCON1bits.ADCS2 = 0;       // Fosc/8 clock conversion

    ADCON1bits.ADFM = 0;       // Left Justified
    ADCON1bits.PCFG = 0b1001; // Use all 6 analog pins and Vdd and Vcc as ref

    ADCON0bits.CHS = channel; // Set channel

    ADCON0bits.ADON = 1;       // Power up ADC

    Delay10TCYx(150);          // wait acquisition time

    ADCON0bits.GO = 1;         // Start AD Conversion

    while(ADCON0bits.GO == 1);// Wait for ADC Conversion to complete

    return ADRESH;             // Output ADC Result
}

/**
    getJoystick reads the voltage from a joysticks x and y potentiometers and
    its push button switch.

        @param  dir, the direction of the joystick 'l' | 'r'.

        @return A vector struct who's x, y and z fields are the joysticks x, y
                and push button state respectively.
*/
Vector getJoystick(char dir){
    Vector v1;
    unsigned char z;
    v1.z = 0;

    if (dir == 'l'){
        v1.x = analogRead(1) - 128;
        v1.y = analogRead(2) - 128;
        z = analogRead(0);
        if (z > 50) v1.z = 1;
    }else if (dir == 'r'){
        v1.x = analogRead(4) - 128;
        v1.y = analogRead(5) - 128;
        z = analogRead(3);
        if (z > 50) v1.z = 1;
    }
    return v1;
}


/**
    getDirection returns the direction of a Vector. For a Vector
    to have a defined direction, the value of one axis must be
    within a tolerance from the center (CTL), and the other extend
    past a tolerance (MTL).

        @param vector to get direction of.

        @return A char corresponding to direction 'u' up, 'd' 'down', 'r'
                right, 'l' left 0 if no defined direction.
*/
char getVDirection(Vector vector){
    if (vector.x > MTL && vector.y > -CTL && vector.y < CTL) return 'l';
    if (vector.x < -MTL && vector.y > -CTL && vector.y < CTL) return 'r';
    if (vector.y > MTL && vector.x > -CTL && vector.x < CTL) return 'u';
    if (vector.y < -MTL && vector.x > -CTL && vector.x < CTL) return 'd';
    return 0;
}

/**
    sanitizeV restrains a given vector to ±127 and sets it to a zero vector
    if the x and y components are within a tolerance (TOL) from the center.

        @param vector to sanitize.

        @return A vector who's components have been sanitized.
*/
Vector sanitizeV(Vector v){
    if (v.x == -128) v.x = -127;
    if (v.y == -128) v.y = -127;
    if (v.x < TOL && v.x > -TOL) v.x = 0;
    if (v.y < TOL && v.y > -TOL) v.y = 0;
    return v;
}
#define DISPLAY     0b00001000
#define DDRAM       0b10000000
#define FUNCTION    0b00100000
#define CLEAR       0b00000001
#define HOME        0b00000010
#define ENTRYMODE   0b00000100
#define SHIFT       0b00010000
#define CGRAM       0b01000000

#define FONT_5x8 0
#define FONT_5x11 1
#define RIGHT 1
#define LEFT 0

#define E(x) PORTCbits.RC0 = x;
#define RW(x) PORTCbits.RC2 = x;
#define RS(x) PORTCbits.RC1 = x;

/**
  LcdCmd is a bitfield that provides meaningful names for the bits of
  a command to be sent to the lcd. The cmd field will return the
  entire byte and can be set to a defined value to set unused
  bits that are required by the lcd for specific commands as described
  in the lcd controller datasheet.
 */
typedef union{
    unsigned char cmd;

    struct {
        unsigned blinkOn:1;
        unsigned cursorOn:1;
        unsigned displayOn:1;
        unsigned :5;
    };
    struct {
        unsigned addressDD: 7;
        unsigned :1;
    };

    struct {
        unsigned addressCG: 6;
        unsigned :2;
    };

    struct {
        unsigned :2;
        unsigned fontType:1;
        unsigned twoLine:1;
        unsigned eightBit:1;
        unsigned :3;
    };

    struct{
        unsigned shiftDisplay: 1;
        unsigned direction: 1;
        unsigned :6;
    };
    struct{
        unsigned :2;
        unsigned dir: 1;
        unsigned dispOrCursor: 1;
        unsigned :4;
    };
} LcdCmd;

/**
  initLcd, when called sets port D and port C RC0 - 2 as outputs,
```

```
  initializes the Lcd's mode and function settings and then turns it on.
 */
void initLcd(void){
    LcdCmd lcdSet;

    TRISD = 0;
    TRISC = 0;
    PORTC = 0;

    delay(40);

    lcdSet.cmd = FUNCTION;
    lcdSet.eightBit = 1;
    lcdSet.twoLine = 1;
    lcdSet.fontType = FONT_5x8;
    lcdRunCmd(lcdSet);
    lcdRunCmd(lcdSet);


    lcdSet.cmd = DISPLAY;
    lcdSet.displayOn = 1;
    lcdSet.cursorOn = 0;
    lcdSet.blinkOn = 0;
    lcdRunCmd(lcdSet);

    lcdSet.cmd = CLEAR;
    lcdRunCmd(lcdSet);

    lcdSet.cmd = ENTRYMODE;
    lcdSet.shiftDisplay = 0;
    lcdSet.direction = RIGHT;
    lcdRunCmd(lcdSet);
}

/**
  lcdSendData sends one byte of data to the lcd.

    @param value, the character value to send.
 */
void lcdSendData(char value){
    RS(1);
    RW(0);

    E(0);
    E(1);

    delayMicros(1);

    PORTD = value;

    delayMicros(1);

    E(0);

    delayMicros(37);
}

/**
  lcdRunCmd takes an lcdCmd bitfield and sends it to the lcd as a byte.

    @param lcdCmd, an LcdCmd to send to the lcd.
 */
void lcdRunCmd(LcdCmd lcdSet){
    RW(0);
    RS(0);

    E(0);
    E(1);

    delayMicros(1);

    PORTD = lcdSet.cmd;

    delayMicros(1);

    E(0);

    delayMicros(37);
    if (lcdSet.cmd < 4) delay(2);
}

/**
  delayMicros when called will halt the code for the specified number of
  microseconds.

    @param micros, the number of microseconds to halt for.
 */
void delayMicros(unsigned char micros){
    Delay10TCYx(micros);
}

/**
  delay when called will halt the code for specified number of milliseconds.
  If 0 is given the function will return immediately.

    @param millis, the number of milliseconds to halt for.
 */
void delay(unsigned int millis){
    unsigned int reps;
    unsigned int i;

    if (millis == 0) return;

    reps = millis/255;
    millis -= 255 * reps;
    for (i = 0; i < reps; i++) Delay10KTCYx(255);
    Delay10KTCYx(millis);
}

/**
  lcdSetCursorPosition sets the position of the cursor on the lcd.

    @param col, the column position to set the cursor [0, 40].
    @param row, the row to set the cursor 0|1.
 */
void lcdSetCursorPosition(unsigned char col, unsigned char row){
    LcdCmd lcdSet;


    //Set DDRAM Address
    lcdSet.cmd = DDRAM;
    lcdSet.addressDD = col + 64*row;
    lcdRunCmd(lcdSet);
}

/**
  lcdClear clears the lcd of all characters.
 */
void lcdClear(void){
    LcdCmd lcdSet;


    //Set DDRAM Address
    lcdSet.cmd = CLEAR;
    lcdRunCmd(lcdSet);
}

/**
  lcdPrintRandom, prints a 16 character string by randomly selecting one
  character from the array that has not already been printed, printing
  it to the lcd and then repeating until all characters have been printed.

    @param arr, the character array to print randomly.
    @param row, the row to print the string on.
 */
 void lcdPrintRandom(char arr[], char row){
     int r;
     int rndm;
     int index;

     srand(123832423);
     for (r = 15; r >= 0; r--){
         rndm = rand() % r;
         index = 0;
         while(rndm){
             if (arr[index]) rndm--;
             index ++;
         }

         while (!arr[index] && index < 16) index++;

         if (!arr[index]) return;

         lcdPrintChar(arr[index], index, row);
         arr[index] = 0;
         delay(35);
     }
 }

/**
  lcdPrintString, prints a character array (string) to the lcd
  at a postion specified.

  @param arr, the character array to print.
  @param col, the column to begin printing the string.
  @param row, the row to print the string 0|1.
 */
void lcdPrintString(char arr[], unsigned char col, unsigned char row){
    char c;

    lcdSetCursorPosition(col, row);

    while(c = * (arr ++)){
        lcdSendData(c);
```

```
    }
}

/**
  lcdShiftDisplay, shifts the lcd display in a given direction until
  all 16 characters are outside of the display.

    @param delayTime the time in milliseconds to delay between a single shift.
    @param direction, the direction to shift the display LEFT|RIGHT.
 */
void lcdShiftDisplay(unsigned char delayTime, char direction){
    LcdCmd lcdSet;

    lcdSet.cmd = SHIFT;
    lcdSet.dir = direction;
    lcdSet.dispOrCursor = 1;

    for (direction = 0; direction < 16; direction++){
        lcdRunCmd(lcdSet);
        if (delayTime != 0)  delay(delayTime);
    }
}

/**
  lcdPrintChar, prints a single character to the lcd at a specified position.

    @param character to print to the lcd.
    @param col, the column to print the char at.
    @param row, the row to print the char on 0|1.
 */
void lcdPrintChar(char character, char col, char row){
    lcdSetCursorPosition(col, row);
    lcdSendData(character);
}

/**
  lcdSetCursorOn, sets the lcd cursor's state.

    @param cursorOn, 1: cursor is set on | 0: cursor is set off.
    @param blinkOn, 1: cursor will blink | 0: cursor will not blink.
 */
void lcdSetCursorOn(char cursorOn, char blinkOn){
    LcdCmd lcdSet;
    lcdSet.cmd = DISPLAY;
    lcdSet.displayOn = 1;
    lcdSet.cursorOn = cursorOn;
    lcdSet.blinkOn = blinkOn;
    lcdRunCmd(lcdSet);
}

/**
  lcdPrintMenuItem will print a menuItem node and the menuItem bellow it
  to the lcd at a specified location.

    @param menuItem, the menuItem to display on the lcd
    @param loc, 0: prints from the first column | 1: prints from the 16th column

    @see MenuItem
 */
void lcdPrintMenuItem(MenuItem menuItem, char loc){
    if (loc == 1) loc = 16;
    lcdPrintString(menuItem.display, loc, 0);
    lcdPrintString(menuItem.down->display, loc, 1);
}

/**
  lcdPrintVector prints a vector to the lcd, showing its x, y, z components.

    @param vector, the vector to print to the lcd.
    @param row, the row to print the vector on.

    @see Vector
 */
void lcdPrintVector(Vector vector, char row){
   char line[16];
   sprintf(line, "x%-4d y%-4d z%d", vector.x,  vector.y, vector.z, getVDirection(vector));
   lcdPrintString(line, 0, row);
}

/**
  lcdPrintSetterMaxError, prints an error message to lcd explaining that
  the given menuItem can be set to a value less than its specified max
  setter value.

    @param current, a pointer to the menuItem that has raised the error.
    @param newValue, value the user has attemped to set.

    @see MenuItem
 */
void lcdPrintSetterMaxError(MenuItem * current, unsigned char newValue){
    char error[16];
    char error2[16];

    sprintf(error, "Error: max %3d", current->setterMax);
    sprintf(error2, "but value %3d", newValue);

    lcdClear();
    lcdPrintString(error, 0, 0);
    lcdPrintString(error2, 0, 1);

    delay(800);

    lcdClear();
    lcdSetCursorOn(0, 0);
    lcdPrintString(current->display, 0, 0);
    lcdPrintString(current->down->display, 0, 1);
}
#include "ConfigRegsPIC18F452.h"
#include
#include
#include
#include "delays.h"

/**
  main, displays a welcome message, initiates ADC, UART, menu and the LCD then
  enters a loop where the menu is displayed and when a drive mode is selected
  then the controller enters into drive mode.
*/
void main(void) {
    char ff[] =  "  hey,  David,  ";
    char ff2[] = " Avie and Yiwei ";
    TRISB = 3;


    initAdc();
    configureUART();
    initMenu();
    initLcd();
    lcdPrintRandom(ff, 0);
    lcdPrintRandom(ff2, 1);

    delay(100);
    lcdClear();

    while(1){
        lcdClear();
        mainMenu();
        drive();
    }
}

/**
  drive, when called enters a loop where the user can send joystick vectors
  to the robot to control it.
*/
void drive(){
    char testy[16];
    char rec;

    Vector vLeft;
    Vector vRight;
    Vector velocity;

    char driving = 1;
    unsigned int overflow = 0;

    while(driving){

        //Get velocity vector from joysticks
        vLeft = getJoystick('l');
        vRight = getJoystick('r');

        velocity.z = vLeft.z;
        velocity.x = vLeft.x;
        velocity.y = vRight.y;

        //If exit button pressed
        if (PORTBbits.RB1 != 0){
            velocity.x = 0;
            velocity.y = 0;
            driving = 0;
        }else{
            delay(50);

        }

        //Sanitize and transmit
        xBeeCmd.vector = sanitizeV(velocity);
        xBeeCmd.V = 'V';
        xBeeTransmitCmd();
        lcdClear();

        sprintf(testy, "%c: %4d, %c: %4d", 0b11100100, 20 + (rand() %50), 0b11100101, 20 + (rand() % 12));
        lcdPrintVector(xBeeCmd.vector, 0);
```

```
        lcdPrintString(testy, 0, 1);
    }
}
#define PID_P_DEFAULT 1;
#define PID_I_DEFAULT 1;
#define PID_D_DEFAULT 1;

#pragma udata udata1
MenuItem config;
MenuItem factory;
MenuItem pid;
MenuItem pidP;
MenuItem pidI;
MenuItem pidD;
#pragma udata udata2
MenuItem yaw;
MenuItem display;
MenuItem ir;
MenuItem ir_samples;
MenuItem ir_srate;
MenuItem speed;
MenuItem manual;
MenuItem assist;
#pragma udata

/**
  MenuItem is a union struct that represents a node in a 4 way linked list.
  A menuItem contains pointers to other menuItems located above, bellow, to
  the left and to the right of itself. MenuItem's hold a character
  array that can be printed to an lcd and a value, index and setterMax
  that can be used for robot configuration.
*/
typedef union MenuItem{
    struct{

        union MenuItem * up;
        union MenuItem * down;
        union MenuItem * left;
        union MenuItem * right;

        char index;
        unsigned char value;
        unsigned char setterMax;

        char display[16];

    };
} MenuItem;

/**
  initMenu, initializes the 4 way linked list of menuItems and sets their
  default values.
*/
void initMenu(void){
    char none[] = "None";
    strcpypgm2ram(config.display, "Configuration");
    config.index = -1;

    strcpypgm2ram(factory.display, "Factory mode");
    factory.index = -1;

        strcpypgm2ram(pid.display, "Configure PID");
        pid.index = -1;

            //PID setters
            pid_p.setterMax = 100;
            pid_p.value = PID_P_DEFAULT;
            sprintf(pid_p.display, "Proportional %3d", pid_p.value);
            pid_p.index = 0;

            pid_i.setterMax = 100;
            pid_i.value = PID_I_DEFAULT;
            sprintf(pid_i.display, "Integral     %3d", pid_i.value);
            pid_i.index = 1;

            pid_d.setterMax = 100;
            pid_d.value = PID_D_DEFAULT;
            sprintf(pid_d.display, "Differential  %d", pid_d.value);
            pid_d.index = 2;

        yaw.setterMax = 100;
        yaw.value = 100;
        sprintf(yaw.display, "Set max yaw  %3d", yaw.value);
        yaw.index = 3;

        ir.setterMax = 0;
        ir.value = 0;
        strcpypgm2ram(ir.display, "Configure IR");
        ir.index = -1;

        ir_srate.setterMax = 100;
        ir_srate.value = 23;
        sprintf(ir_srate.display, "Samples%9d", ir_srate.value);
        ir_srate.index = 7;

        ir_samples.setterMax = 40;
        ir_samples.value = 40;
        sprintf(ir_samples.display, "Sample Rate%5d", ir_samples.value);
        ir_samples.index = 6;

        display.setterMax = 3;
        display.value = 0;
        sprintf(display.display, "Display   %6s", none);
        display.index = 4;


    speed.value = 100;
    speed.setterMax = 120;
    sprintf(speed.display, "Max speed    %3d", speed.value);
    speed.index = 5;

    strcpypgm2ram(manual.display, "Manual mode");
    manual.index = -1;

    strcpypgm2ram(assist.display, "Assist mode");
    assist.index = -1;

    config.up = &assist;
    config.left = NULL;
    config.down = &manual;

    config.right = &factory;
    factory.left = &config;
    speed.left = &config;

        factory.up = &speed;
        factory.down = &speed;

        factory.right = &pid;
        pid.left = &factory;
        yaw.left = &factory;
        display.left = &factory;
        ir.left = &factory;


            pid.up = &display;
            pid.down = &yaw;

            pid.right = &pid_p;
            pid_p.left = &pid;
            pid_i.left = &pid;
            pid_d.left = &pid;

                pid_p.up = &pid_d;
                pid_p.down = &pid_i;
                pid_p.right = NULL;

                pid_i.up = &pid_p;
                pid_i.down = &pid_d;
                pid_i.right = NULL;

                pid_d.up = &pid_i;
                pid_d.down = &pid_p;
                pid_d.right = NULL;

            yaw.up = &pid;
            yaw.down = &ir;
            yaw.right = NULL;

            ir.up = &yaw;
            ir.down = &display;
            ir.right = &ir_samples;

            ir_samples.left = &ir;
            ir_samples.right = NULL;
            ir_samples.up = &ir_srate;
            ir_samples.down = &ir_srate;

            ir_srate.left = &ir;
            ir_srate.right = NULL;
            ir_srate.down = &ir_samples;
            ir_srate.up = &ir_samples;

            display.up = &ir;
            display.down = &pid;
            display.right = NULL;

        speed.up = &factory;
        speed.down = &factory;
        speed.right = NULL;

    manual.up = &config;
    manual.down = &assist;
    manual.right = NULL;
```

```
    manual.left = NULL;

    assist.up = &manual;
    assist.down = &config;
    assist.right = NULL;
    assist.left = NULL;

}


/**
  moveMenu, will attempt to move the menuItem currently displayed to
  a menuItem in a given direction. If there is no menuItem in the given
  direction or the move is not permitted then the method will return
  the current menuItem and no move will be made. If the menuItem is
  successfully moved then it will be displayed graphically on lcd.

    @param current, a pointer to the current menuItem displayed.
    @param dir, the direction to move in.

    @return The a pointer to the menuItem moved to.
*/
MenuItem * moveMenu(MenuItem * current, char dir){
    MenuItem * newItem = current;
    unsigned int delayTime = 0;

    switch(dir){
        case 'd':
            newItem = current->down;
            delayTime = 100;
            break;

        case 'u':
            newItem = current->up;
            delayTime = 100;
            break;

        case 'r':
            if (current->right == NULL) return current;
            newItem = current->right;
            if (current == &factory && PORTBbits.RB0 == 0) return current;
            lcdPrintMenuItem(* newItem, 1);
            lcdShiftDisplay(5, LEFT);
            break;

        case 'l':
            if (current->left == NULL) return current;
            newItem = current->left;
            lcdShiftDisplay(0, LEFT);
            lcdPrintMenuItem(* newItem, 0);
            lcdPrintMenuItem(* current, 1);
            lcdShiftDisplay(5, RIGHT);
            break;

        default: return newItem;

    }
    lcdClear();
    lcdPrintMenuItem(* newItem, 0);
    delay(delayTime);
    return newItem;
}

/**
  mainMenu when called enters a loop where the user can move the left joystick
  to navigate between menuItem's, and set their values where applicable. When
  the user sets a value it will be sent to the robot via the xBee. When the
  user enters either manual or assist mode the robot will notified and the
  menu will exit its loop.

    @see xBeeTransmitCmd
*/
void mainMenu(void){
    Vector joystickL;
    Vector joystickR;
    char direction = 0;

    MenuItem * current = &config;
    MenuItem * oldMenu = current;


    char inMenu = 1;


    lcdPrintMenuItem(* current, 0);

    while(inMenu){
        joystickL = getJoystick('l');
        joystickR = getJoystick('r');

        direction = getVDirection(joystickL);

        if (joystickL.z && current->index >= 0){
            if (current == &display){
                editDisplay(current);
            }else{
                editMenuItemValue(current);
            }
        }
        if (joystickL.z && current == &manual){
            xBeeCmd.C = 'C';
            xBeeCmd.index = 8;
            xBeeCmd.value = 1;
            inMenu = 0;
        }
        if (joystickL.z && current == &assist){
            xBeeCmd.C = 'C';
            xBeeCmd.index = 8;
            xBeeCmd.value = 2;
            inMenu = 0;
        }

        current = moveMenu(current, direction);
        delay(20);
    }
    xBeeTransmitCmd();
}

/**
  editMenuItemValue, when called enters a loop where the user can move the left
  joystick to set the value of a given menuItem. To set a value the user can
  change one of three digits by pushing the joystick in an upwards or downwards
  direction. Moving the joystick in either the left or right direction will
  change the digit being set.

  When the joystick's push button is pressed the
  value will be checked against the menuItem's max value, if the value is valid
  the menuItem will be changed and the update sent to the robot via the xBee.
  If the new value is not valid an error message will be displayed.

    @param current, the menuItem who's value is to be set.

    @see xBeeTransmitCmd
    @see lcdPrintSetterMaxError
*/
void editMenuItemValue(MenuItem * current){
  unsigned char digits[3];
  unsigned char indx = 0;
  unsigned int newValue;
  char direction;
  char dig_max = 10;
  Vector v;

  //Turn cursor on and set its position to
  lcdSetCursorOn(1, 0);
  lcdSetCursorPosition(15 - indx, 0);

  v = getJoystick('r');
  while (v.z) v = getJoystick('r');
  delay(100);


  digits[2] = current->value/100;
  digits[1] = (current->value - digits[2]*100)/10;
  digits[0] = (current->value - digits[1]*10 - digits[2]*100);

  while(v.z == 0){
    direction = getVDirection(v);
    dig_max = 10;

    switch (direction){
      case 'u':
        digits[indx] = (digits[indx] + 1)%dig_max;
        break;

      case 'd':
        digits[indx] = (dig_max - 1 + digits[indx])%dig_max;
        break;

      case 'l':
        indx = (indx + 1)%3;
        break;

      case 'r':
        indx = (2 + indx)%3;
        break;
    }
    if (direction != 0){
      lcdPrintChar(digits[indx] + 48, 15 - indx, 0);
      lcdSetCursorPosition(15 - indx, 0);
      delay(100);

    }
    v = getJoystick('l');
```

```
  }
  lcdSetCursorOn(0, 0);

  //Calculate the value
  newValue = digits[2]*100 + digits[1]*10 + digits[0];
  if (newValue > current->setterMax){
      lcdPrintSetterMaxError(current, newValue);
      return;
  }

  //Set the menu item value and update the display string
  current->value = newValue;
  current->display[13] = 0;
  sprintf(current->display, "%s%3d", current->display, current->value);

  //Send value through xBee
   xBeeCmd.C = 'C';
   xBeeCmd.index = current->index;
   xBeeCmd.value = current->value;
   xBeeTransmitCmd();


  //Print menu and remove cursor
  lcdPrintMenuItem(* current, 0);
  delay(300);
}

/**
  editDisplay, similarly to editMenuItemValue enters a loop where the user
  can change the value of a menuItem using the left joystick. In the
  editDisplay loop a user can choose one of three values to set the
  display menuItem by moving the joystick up or down to cycle through
  them. The options are displayed as string's to make it easy to understand.

  0: 'None' displays nothing when in drive mode (manual or assist).
  1: 'Stats' displays ir sensor statistics in drive mode.
  2: 'Motors' displays the power that each motor is set to in drive mode.

  Once changed the update is sent to the robot via the xBee.

    @param displayItem, a pointer to the display menuItem.

    @see xBeeTransmitCmd
*/
void editDisplay(MenuItem * displayItem){
    char options[3][7] = {"None", "Stats", "Motors"};
    char indx = 0;
    char direction;
    Vector v;

    v = getJoystick('l');
    while (v.z){
        v = getJoystick('l');
    }
    delay(100);

    while (v.z == 0){

        switch(direction){
            case 'u':
                indx = (indx + 1) % 3;
                break;
            case 'd':
                indx = (indx + 2) % 3;
                break;
        }
        if (direction == 'u' || direction == 'd'){
            sprintf(displayItem->display, "Display   %6s", options[indx]);
            lcdPrintString(displayItem->display, 0, 0);
            displayItem->value = indx;

            delay(100);
        }
        v = getJoystick('l');
        direction = getVDirection(v);
    }
    xBeeCmd.C = 'C';
    xBeeCmd.index = displayItem->index;
    xBeeCmd.value = displayItem->value;
    xBeeTransmitCmd();
    delay(100);
}

/**
    XBeeCmd is a union of 5 bytes used to store cmds to be sent to the robot
    wirelessly via the xBee. The first byte defines the header of the message
    to be sent, and the remaining 4 bits are the data to send. The data can
    be in format of a vector or the index and value of a configuration setting.
*/
typedef union{
    struct{
        char V;
        Vector vector;
        char z2;
    };
    struct{
        char C;
        unsigned char index;
        unsigned char value;
        char spr3;
        char spr4;
    };
    char array[5];
} XBeeCmd;

XBeeCmd xBeeCmd;

/**
    configureUART sets the appropriate configuration bits inside the TXSTA and
    RCSTAbits registers in order for USART to function correctly as per
    project requirements. PORTC6 is set as an output to enable Transmission
    and PORTC7 is set as an input to enable receiving. This function must be
    called before receiving or transmitting strings via USART using xBeeSend
    and xBeeOnReceive.
*/
void configureUART(void){
    TXSTAbits.BRGH = 1; // high speed mode (y=16)
    SPBRG = 64;          // X = (10^7/9600*16) -1   = 64.1
    TXSTAbits.SYNC = 0; // Asynchronous mode
    RCSTAbits.SPEN = 1; // Enable serial port

    TXSTAbits.TX9 = 0;  // Enable 8 bit transmission
    TXSTAbits.TXEN = 1; // Enable transmit
    RCSTAbits.RX9 = 0;  // 8 bit receive
    RCSTAbits.CREN = 1; // receive enabled

    TRISCbits.RC6 = 0;  // Enable Pin 6 as output to transmit serial output from XBee
    TRISCbits.RC7 = 1;  // Enable pin 7 as input to receive to XBee
}

/**
xBeeWrite is used to send a byte from the robot to controller wirelessly
over USART and xBee transmission.

    @param val, the byte to send.
*/
void xBeeWrite(char val){
  while(PIR1bits.TXIF == 0);
  TXREG = val;
}

/**
xBeeTransmitCmd is used to send the global XbeeCmd to the robot wirelessly
over USART and xBee transmission.
*/
void xBeeTransmitCmd(void){
    int i;
    for (i = 0; i < 4; i++){
        xBeeWrite(xBeeCmd.array[i]);
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include "ConfigRegsPIC18F452.h"
#include <p18f452.h>
#include <string.h>
#include "delays.h"
#include <usart.h>
#include <math.h>
#define TRUE 1
#define FALSE 0


////////////////////////////////////////////////////////////////////////////
// XBee variables and functions
/**
    configValues is a struct used to store configuration values entered by
    user on the controller menue interface.
*/
typedef union{
    struct{
        unsigned char pid_p;
        unsigned char pid_i;
        unsigned char pid_d;
        unsigned char yaw;
        unsigned char display;
        unsigned char max_speed;
        unsigned char IR_samples;
        unsigned char IR_rate;
        unsigned char mode;
    };
    unsigned char array[8];
}ConfigValues;
ConfigValues config;

/**
    Vector is a struct used to store joystick values once transmitted over
    USART and xBee communication.
*/
typedef struct{
    signed char x;
    signed char y;
    char z;
}Vector;
Vector velocity;

// XBee Global Variables
int i = 0;                  // Index counter for receiving strings
char readValues[4];         // Receive Character Buffer
int startReading = FALSE;   // Start Reading flag

/**
    configureUART sets the appropriate configuration bits inside the TXSTA
    and RCSTAbits registers in order for USART to function correctly as per
    project requirements. PORTC6 is set as an output to enable Transmission
    and PORTC7 is set as an input to enable receiving. This function must be
    called before receiving or transmitting strings via USART using xBeeSend
    and xBeeOnReceive.

        @see xBeeOnReceive & xBeeSend
*/
void configureUART(void){

    TXSTAbits.BRGH = 1; // High speed mode (y=16)
    SPBRG = 64;         // X = (10^7/9600*16) - 1 = 64.1
    TXSTAbits.SYNC = 0; // Asynchronous mode
    RCSTAbits.SPEN = 1; // Enable serial port

    TXSTAbits.TX9  = 0;  // Enable 8 bit transmission
    TXSTAbits.TXEN = 1; // Enable transmit
    RCSTAbits.RX9  = 0;  // 9 bit receive
    RCSTAbits.CREN = 1; // receive enabled

    TRISCbits.RC6  = 0;  // Enable Pin 6 as Tx
    TRISCbits.RC7  = 1;  // Enable pin 7 as Rx

}

/**
    configureUARTInterrupts sets the appropriate configuration bits in order
    to enable interrupt on receive for the system. This function must be
    called before xBeeOnReceive.
        @see xBeeOnReceive
*/
void configureUARTInterrupts(void){

    INTCONbits.GIEH = 0; // Disable Interrupts
    INTCONbits.GIEL = 0; // Disable Peripheral interrupts
    RCONbits.IPEN   = 1;   // Enable Priority
    PIE1bits.RCIE   = 1;   // Enable USART Receive Interrupt
    IPR1bits.RCIP   = 1;   // Enable High Priority on receive interrupt *
    INTCONbits.GIEH = 1;
    INTCONbits.GIEL = 1;
    INTCONbits.PEIE = 1; // Re-enable interrupts

}

/**
    xBeeSend is used to send strings from the robot to controller wirelessly
    over USART and xBee transmission.

        @param string, the string to be sent
*/
void xBeeSend(char * string){
    char c;
```

```
    // Loop through all characters in string
    while(c = *(string ++)){
        while(PIR1bits.TXIF == 0);     // Wait till transmit buffer is clear
        TXREG = c;                     // Transmit character over USART
    }

}

/**
    saveJoyStickValues saves the x,y and z joystick values into the velocity
    struct if the appropriate start bit has been read.
*/
void saveJoyStickValues(void){
    velocity.x = readValues[1]; // Save buffer position 1 as x
    velocity.y = readValues[2]; // Save buffer position 2 as y
    velocity.z = readValues[3]; // Save buffer position 3 as z
}

/**
    saveConfigValues saves configuration values into the correct index
    position of the configuration array defined in the config union.
*/
void saveConfigValues(void){
    // Save value in buffer position 2 into array position dictated by buffer
    // position 1.
    config.array[readValues[1]] = readValues[2];
}

/**
    xBeeOnReceive is used to declare the USART receive interrupt and declare
    the ISR in the correct place in memory. This function must be called for
    the highPriorityISR to function correcty.

        @see highPriorityISR
*/
void highPriorityISR(void);    // Function declaration for highPriorityISR
#pragma code highPriAddr = 0x08
void xBeeOnReceive(void){
    _asm goto highPriorityISR _endasm
}
#pragma code

/**
    highPriorityISR is the Interrupt Service Routine involved with recieving
    configuration values and storing them in the correct variables. This
    function uses saveConfigValues & saveConfigValues to store received
    strings correctly.

        @see saveJoyStickValues & saveConfigValues
*/
#pragma interrupt highPriorityISR
void highPriorityISR(void){

    // Read character from receive register
    char currentChar = RCREG;

    // Check if one of the start bits as been read
    if((currentChar == 'V'||currentChar == 'C') && startReading == FALSE){

        // Begin reading if start bit found
        i = 0;
        startReading = TRUE;
        readValues[i] = currentChar;
        i++;
        return;
    }

    // If start reading flag is off dont read character.
    // This stops errors in transmission effecting output variables
    if(startReading == FALSE)return;

    // Save current character into read buffer
    readValues[i] = currentChar;
    i++;

    // If 3 characters have been read after stop bit
    if(i == 3){

        // If start bit was V joystick values were sent and save buffer into
        // velocity struct
        if(readValues[0] == 'V'){
            saveJoyStickValues();

        }

        // If start bit was C config values were sent and save buffer into
        // config array
        if(readValues[0] == 'C'){
            saveConfigValues();
        }

        // Reset start reading flag and index
        startReading = FALSE;
        i = 0;
    }
}
#pragma code
// End XBee variables and functions
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// Motor variables and functions
/**
    Struct to store left and right power values for motor
```

```
*/
typedef struct
{
    char left;
    char right;
} Motor;

Motor power;                // Global struct for power form joysticks
Motor IR_power;             // Global struct for power from ir_sensors

// Global variables
float speed;
float speedscale;
float max_speed;
unsigned int max_yaw = 100;
float yawscale;

/**
    vectorToMotor converts a vector received from the joysticks to a power
    value that can be used in further calculations or sent to the motors as
    output with the setMotors function.

        @see setMotors

        @param v, the vector from the joysticks

        @return The left and right power values from 0-127 to be sent to the
        motors.
*/
Motor vectorToMotor(Vector v){
    Vector output;
    Motor motor_out;
    int xadjust;
    int yadjust;


    // scale v.x to a percentage according to max yaw/ 100
    xadjust = v.x;
    xadjust = xadjust * config.yaw / 100;
    v.x = -xadjust;

    // scale v.y to a percentage according to max speed/100
    yadjust = v.y;
    yadjust = yadjust * config.max_speed / 100;
    v.y = yadjust;


    // if there is no x componet there is no need to scale
     if (v.x == 0)
    {
        output.x = v.x;
        output.y = v.y;
    }
    else
    {
         // two different equations to allow for greater control in fringe
         // cases obtain x offset from centres effect on each wheel and
         //calculate y for average speed of wheels
        if((fabs(v.x)/fabs(v.y))>1){

            output.x = (fabs(v.x) * v.x) / (fabs(v.x) + fabs(v.y));
            output.y = (fabs(v.x) * v.y) / (fabs(v.x) + fabs(v.y));

        }
        else{
            output.x = (fabs(v.y) * v.x) / (fabs(v.x) + fabs(v.y));
            output.y = (fabs(v.y) * v.y) / (fabs(v.x) + fabs(v.y));
        }
    }
    // offset speeds values by the calculated x offset
    motor_out.left = output.y + output.x;
    motor_out.right = output.y - output.x;

    return motor_out;
}

/**
    setMotors takes a Motor struct with values for the left and right power
    and sends them to the appropriate pins to output power to the motors.
*/
void setMotors(Motor power){
    char left;
    char right;
    // set pins based on direction required and make values positive if
    // required for each wheel
    if (power.left < 0){

        // make positive
        left = power.left * -1;

       //set INA to 0 and INB to 1 to reverse
       PORTDbits.RD3 = 0;
       PORTDbits.RD4 = 1;


    }else{


        left = power.left;
        //set INA to 0 and INB to 1 to reverse
       PORTDbits.RD3 = 1;
       PORTDbits.RD4 = 0;
```

```
    }if (power.right < 0){

        // make positive
        right = power.right * -1;

       //set INA to 0 and INB to 1 to reverse
       PORTDbits.RD1 = 0;
       PORTDbits.RD2 = 1;


    }else{


        right = power.right;
        //set INA to 0 and INB to 1 to reverse
        PORTDbits.RD1 = 1;
        PORTDbits.RD2 = 0;
    }

    //left=left/4;
    //right=right/4;

    //add least significant bits to the low duty registers
    CCP1CONbits.DC1B = left;
    CCP2CONbits.DC2B = right;

    //left = left >> 2;
    //right = right >> 2;

    CCPR1L = left; //left motor output to RC1 for 8 MSB

    CCPR2L = right; //Right motor output to RB3 for 8 MSB
}

/**
    motorsSetup sets up the necessary pins as input and output, and sets
    up the CCP module to output a PWM signal to the motors.
*/
void motorsSetup(){
    TRISD = 0x00; //set D all outputs for LED

    TRISCbits.RC1 = 0;// set RC1 to output
    TRISCbits.RC2 = 0;// set RC2 to output
    TRISCbits.CCP2 = 1;
    TRISBbits.CCP2 = 1;
    TRISBbits.RB3 = 0;// set RB3 to output for PWM


    //timer 2 on and prescale 4
    T2CONbits.TMR2ON = 1;
    T2CONbits.T2CKPS0 = 1;
    T2CONbits.T2CKPS1 =0;

    // PWM setup
    CCP1CONbits.CCP1M3 = 1;
    CCP1CONbits.CCP1M2 = 1;

    CCP2CONbits.CCP2M3 = 1;
    CCP2CONbits.CCP2M2 = 1;
    CCP2CONbits.CCP2M1 = 1;
    CCP2CONbits.CCP2M0 = 1;


    // make PR2 to 127 for 4.8khz pwm duty to fit announcement
    PR2 = 0x7F;
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// IR variables and functions
// IR Macros
#define IR_BUFFER_MAX 40            // Maximum buffer size for IR samples
#define WHEELDIST 200               // Wheelbase in mm
#define TURNING_R 220               // Turning radius
#define TURNING_S 50                // Turning speed
#define IR_MIN_SAMPLE 655           // Minimum number of IR samples

// IR global variables
unsigned char sample_multiplier = 1;    // Multiplier for configuring sample
                                        // rate
unsigned short int sample_rate = 0;         // IR sample rate
unsigned char sample_rate_H = 0;            // For high byte TMR0H
unsigned char sample_rate_L = 0;            // For low byte TMR0L

unsigned char irBufferLeft[IR_BUFFER_MAX];  // Buffer for left IR values
unsigned char irBufferRight[IR_BUFFER_MAX]; // Buffer for right IR values

unsigned char irBufferLeftHead = 0;     // Buffer for reading left IR value
unsigned char irBufferRightHead = 0;    // Buffer for reading right IR value

// User assist globals
char UAFlag = 0;                // User assist flag for which side IR sensors
                                // have detected an object
char mode = 1;              // Flag for current mode of operation

/**
    PIDVariables is struct used for storing the PID variables for use in
    IR PID calculation.
*/
typedef struct {
    char desired_value;
    char IR;
    char last_error;
```

```
    float current_desired_variable;
    char error;
    float integral;
    float derivative;
}PIDVariables;
PIDVariables IR_PID;

/**
    IRResult is struct used to store left and right IR values.
*/
typedef struct{
    unsigned char left;
    unsigned char right;
} IRResult;

IRResult readings;

/**
    initADC sets PORTA as input for analog input.
*/
void initADC(void){
    TRISA  = 0xff;            // PORT A Set as Input Port
}

/**
    analogRead reads a voltage form a specified channel and returns
    the voltage in 8 bit resolution.

        @param  channel, the pin on port A to be read (i.e. 0 - 7).

        @return The voltage on the pin in 8 bit resolution.
*/
unsigned char analogRead(char channel){
    ADCON0bits.ADCS = 1;      // Fosc/8 clock conversion
    ADCON1bits.ADCS2 = 0;      // Fosc/8 clock conversion

    ADCON1bits.ADFM = 0;      // Left Justified
    ADCON1bits.PCFG = 0b1001; // Use all 6 analog pins and Vdd and Vcc as ref

    ADCON0bits.CHS = channel; // Set channel

    ADCON0bits.ADON = 1;      // Power up ADC

    Delay10TCYx(15);         // wait acquisition time

    ADCON0bits.GO = 1;        // Start AD Conversion

    while(ADCON0bits.GO == 1);// Wait for ADC Conversion to complete

    return ADRESH;            // Output ADC Result
}
```

```
/**
    TMR0Setup sets up Timer1 as a 16-bit timer with a prescale of 1:2 and an
    initial delay dependent on the input IR sample rate from the commander.
    Timer0 is started at the end of TMR0Setup to begin sampling.
*/
void TMR0Setup(void) {
    T0CONbits.TMR0ON = 0;       // Start timer off
    T0CONbits.T08BIT = 0;       // 16-bit
    T0CONbits.T0CS = 0;         // internal clock
    T0CONbits.PSA = 0;          // use prescaler
    T0CONbits.T0PS2 = 0;        // 1:2 prescaler
    T0CONbits.T0PS1 = 0;
    T0CONbits.T0PS0 = 0;

    sample_multiplier = config.IR_rate;
    sample_rate = 65535 - sample_multiplier * IR_MIN_SAMPLE;
    sample_rate_H = sample_rate >> 8;
    sample_rate_L = sample_rate & 0xFF;

    TMR0H = sample_rate_H;
    TMR0L = sample_rate_L;

    T0CONbits.TMR0ON = 1;        // Turn timer on
}

/**
    irReadValues reads values from the right and left IR sensors on PORTA pins
    2 and 3 using the function analogRead.

        @see analogRead
*/
void irReadValues(void){
    unsigned char right = analogRead(2);
    unsigned char left = analogRead(3);

    irBufferLeftHead =  ( irBufferLeftHead +  1 ) % config.IR_samples;
    irBufferRightHead = ( irBufferRightHead + 1 ) % config.IR_samples;

    irBufferLeft[irBufferLeftHead] = left;
    irBufferRight[irBufferRightHead] = right;
}

/**
    irGetMean takes the stored IR values over the sample period and calculates
    the mean and returns the value as an IRResult struct.

        @return IRResult struct of left and right IR values.
*/
IRResult irGetMean(void){
    IRResult result;
    unsigned int sumLeft = 0;
    unsigned int sumRight = 0;
```

```
    unsigned char i;
    for (i = 0; i < config.IR_samples; i++) {
        sumLeft += irBufferLeft[i];
        sumRight += irBufferRight[i];
    }
    result.left = 180 - sumLeft/(config.IR_samples);
    result.right = 180 - sumRight/(config.IR_samples);
    return result;
}

/**
    irPID uses proportional control to adjust the power that is output to the
    motors by the setMotors function.

        @see setMotors
*/
void irPID(void){
    float kp_ir = 0.1;
    float ki_ir = 0;
    int kd_ir = 0;

   // Find error
      IR_PID.error = IR_PID.desired_value - IR_PID.IR;
   // Calculate integral controller
      IR_PID.integral = IR_PID.integral + IR_PID.error;

   // Calculate derivative controller
      IR_PID.derivative = IR_PID.error - IR_PID.last_error;

   // Calculate value closer to the desired variable
      IR_PID.current_desired_variable = (kp_ir*IR_PID.error) +
      (ki_ir*IR_PID.integral) + (kd_ir*IR_PID.derivative);

   // Store previous error
      IR_PID.last_error = IR_PID.error;
}

/**
    irToMotor converts the PID output from the irPID function to a power
    output suitable for the motors to be used in the setMotors function if in
    user assist mode.

        @see irPID
        @see setMotors
*/
void irToMotor(void) {

    if(!UAFlag) {
        IR_power.left = 3 * (TURNING_S*(TURNING_R-WHEELDIST/2) /
            (TURNING_R+WHEELDIST/2) - IR_PID.current_desired_variable);
        IR_power.right = 2.5 * (TURNING_S + IR_PID.current_desired_variable);
    }
    else {
        IR_power.right = 3 * (TURNING_S*(TURNING_R-WHEELDIST/2) /
            (TURNING_R+WHEELDIST/2) - IR_PID.current_desired_variable);
        IR_power.left = 2.5 * (TURNING_S + IR_PID.current_desired_variable);
    }
}

/**
    irCalc acquires IR readings with the irGetMean function, determines
    which IR sensor has detected an object and calculates the appropriate IR
    PID values in the irPID function. The values are then converted to a power
    output suitable for the motors in the irToMotor function.

        @see irGetMean
        @see irPID
        @see irToMotor
*/
void irCalc(void){
    irReadValues();
    if(!irBufferRightHead){
        readings = irGetMean();
        if(readings.left > readings.right && readings.right<150){
            UAFlag = 1;     // turn right = 1
            mode = 0;
            IR_PID.IR = readings.right;
            irPID();
        } else if(readings.right>readings.left && readings.left<150) {
            UAFlag = 0;     // turn left = 0
            mode = 0;
            IR_PID.IR = readings.left;
            irPID();
        } else if(readings.right>100 && readings.left>100){
            mode = 1;
        }
        irToMotor();
    }
}

/**
    userAssist checks to see if running in user assist mode, runs the
    necessary IR calculations in the irCalc function and resets Timer1.

        @see irCalc
*/
void userAssist(void) {
    if(INTCONbits.TMR0IF){
        if(config.mode == 2){
            irCalc();
            T0CONbits.TMR0ON = 0;       // Turn off timer

            sample_multiplier = config.IR_rate;
```

```
            sample_rate = 65535 - sample_multiplier * IR_MIN_SAMPLE;
            sample_rate_H = sample_rate >> 8;
            sample_rate_L = sample_rate & 0xFF;

            TMR0H = sample_rate_H;
            TMR0L = sample_rate_L;

            T0CONbits.TMR0ON = 1;       // Turn timer on

            INTCONbits.TMR0IF = 0;
        }
    }
}

/**
    configInit sets the default values of the configuration settings for
    normal and factory modes.
*/
void configInit(void) {
    config.pid_p = 1;
    config.pid_i = 1;
    config.pid_d = 1;
    config.yaw = 50;
    config.IR_rate = 23;
    config.IR_samples = 10;
    config.max_speed = 100;
    IR_PID.desired_value = 80;
    config.IR_samples = 40;
}
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Encoder variables and functions
// Macros for encoders
#define ENCODER_SAMPLE_RATE 0.0001408   // Based on Timer1
#define CONVERT_TO_RPM 0.0213           // Convert edges/second to RPM
#define NO_OF_SAMPLES 710               // Number of samples per RPM calc
#define MAX_RPM 80                      // Maximum RPM of motors

// Encoder global variables
unsigned char pinAR = 0;        // pinA right encoder
unsigned char pinBR = 0;        // pinB right encoder
unsigned char pinAL = 0;        // pinA left encoder
unsigned char pinBL = 0;        // pinB left encoder

unsigned char pinAROld = 0;     // previous pinA value right encoder
unsigned char pinALOld = 0;     // previous pinA value left encoder

unsigned short int edgeCountR = 0;      // Edges counted on right encoder
unsigned short int edgeCountL = 0;      // Edges counted on left encoder

unsigned short int overflowCount = 0;   // Timer overflow counter

float dt = ENCODER_SAMPLE_RATE * NO_OF_SAMPLES;     // Time for RPM calc

unsigned char rpmR = 0;         // RPM right encoder
unsigned char rpmL = 0;         // RPM left encoder

// Motor PID global variables
char encoderPowerR = 0;       // Encoder power for motor input
char encoderPowerL = 0;       // Encoder power for motor input

float powerErrorR = 0;        // Error between encoder and motor power
float powerErrorL = 0;

float powerPIDR = 0;        // Output from PID control
float powerPIDL = 0;

float kp = 0.001;             // Initial motor PID proportional gain

/**
    lowVectorInterrupt is used to declare the lowPriorityISR Interrupt Service
    Routine and declare the ISR in the correct place in memory. This function
    must be called for the lowPriorityISR to function correctly.

        @see lowPriorityISR
*/
void lowPriorityISR(void);
#pragma code lowPriAddr=0x18        // Low priority vector at 0x18
void lowVectorInterrupt(void){
    _asm GOTO lowPriorityISR _endasm
}
#pragma code

/**
    lowPriorityISR is the Interrupt Service Routine that triggers whenever
    Timer1 overflows at a sampling frequency that is twice the fastest
    encoder rising edge frequency. The values from the encoder pins are read
    and if a rising edge has occured since the last sample the edge count for
    the appropriate encoder is increased. The current pinA status is saved as
    the previous pinA status for the next sample and the overflow counter is
    incremented so that the calculateRPM function can calculate the RPM after
    the total sample time has occurred. Timer1 is also reset to the initial
    values.

        @see calculateRPM
*/
#pragma interrupt lowPriorityISR
void lowPriorityISR(void) {
    INTCONbits.GIEL = 0;        // Disable low priority interrupts

    if (PIR1bits.TMR1IF) {
        //// When timer 1 overflows, store values from encoder
        // Right motor
```

```
        pinAR = PORTDbits.RD0;
        pinBR = PORTDbits.RD5;

        // Left motor
        pinAL = PORTDbits.RD7;
        pinBL = PORTDbits.RD6;

        // Compare old pinA values to new values, if LOW changes to
        // HIGH (rising edge) increment edgeCount variables
        // Right motor
        // Check for rising edge
        if (pinAR == 1 && pinAROld == 0) {
            edgeCountR++;
        }

        // Left motor
        // Check for rising edge
        if (pinAL == 1 && pinALOld == 0) {
            edgeCountL++;
        }

        // Save current variables as old variables for next overflow
        pinAROld = pinAR;
        pinALOld = pinAL;

        // Increment overflow counter
        overflowCount++;

        // Reset Timer1
        TMR1H = 0xFF;
        TMR1L = 0xD3;

        // Reset Timer1 Overflow interrupt bit
        PIR1bits.TMR1IF = 0;
    }

    INTCONbits.GIEL = 1;        // Enable low priority interrupts
}
#pragma code

/**
    TMR1Setup sets up Timer1 as a 16-bit timer with a prescale of 1:8 and an
    intial delay of 0.0001408 seconds as a sample rate. Timer1 is started at
    the end of TMR1Setup to begin sampling.
*/
void TMR1Setup() {
    T1CONbits.RD16 = 1;         // Timer1 as 16-bit timer
    T1CONbits.T1CKPS1 = 1;      // Timer1 prescale of 8:1
    T1CONbits.T1CKPS0 = 1;
    T1CONbits.T1OSCEN = 0;      // Timer1 oscillator shut off
    T1CONbits.TMR1CS = 0;       // Use internal clock Fosc/4
    T1CONbits.TMR1ON = 0;       // Timer off

    // Sample rate
    TMR1H = 0xFF;               // 0xFFD3 gives a delay of 0.0001408 seconds
    TMR1L = 0xD3;               // at 8:1 prescale to use as a sample rate

    T1CONbits.TMR1ON = 1;       // Start the timer
    return;
}

/**
    PORTDSetup sets up RD0 and RD7 as input for the encoder A pins for the
    right and left encoders, respectively. RD5 and RD6 are set up as input
    for encoder B pins on right and left encoders, respectively. The ports
    are cleared to start with.
*/
void PORTDSetup() {
    TRISDbits.RD0 = 1;          // Input pinAR on RD0
    TRISDbits.RD7 = 1;          // Input pinAL on RD7

    TRISDbits.RD5 = 1;          // Input pinBR on RD5
    TRISDbits.RD6 = 1;          // Input pinBL on RD6

    PORTDbits.RD0 = 0;          // Clear ports to start with
    PORTDbits.RD7 = 0;
    PORTDbits.RD5 = 0;
    PORTDbits.RD6 = 0;


    return;
}

/**
    interruptSetup configures Timer1 as a low priority interrupt for the
    TMR1IF to be used in the lowPriorityISR function when Timer1 overflows.

        @see lowPriorityISR
*/
void interruptSetup() {
    INTCONbits.GIEH = 0;        // Disables all interrupts

    RCONbits.IPEN = 1;          // Enable priority interrupts

    PIE1bits.TMR1IE = 1;        // Enable TMR1 overflow interrupt
    IPR1bits.TMR1IP = 0;        // Low priority interrupt

    INTCONbits.GIEH = 1;        // Enable high priority interrupts
    INTCONbits.GIEL = 1;        // Enable low priority interrupts

    return;
}

/**
```

```
    calculateRPM calculates the RPM of each encoder by dividing the encoder
    edge count determined in the lowPriorityISR function by the amount of time
    that has passed and multiplying by a constant CONVERT_TO_RPM to convert
    the edge per second reading to RPM.
    The RPM is converted to a power output suitable for the motors to be used
    in the encoderPIDCalculation function.
    The overflow and edge count variables used are also reset.

        @see lowPriorityISR
        @see CONVERT_TO_RPM
        @see encoderPIDCalculation
*/
void calculateRPM(void) {
    if (overflowCount >= NO_OF_SAMPLES) {
        // Right motor
        edgeCountR = edgeCountR * 1.067;
        edgeCountL = edgeCountL * 1.067;

        // Calculate RPM from edgeCount and dt
        rpmR = edgeCountR / dt * CONVERT_TO_RPM;
        if (rpmR > 80) {
            rpmR = 80;
        }

        // Left motor
        rpmL = edgeCountL / dt * CONVERT_TO_RPM;
        if (rpmL > 80) {
            rpmL = 80;
        }

        // Convert to power_reading for PID control
        //
        encoderPowerR = rpmR * 1.5875;
        encoderPowerL = rpmL * 1.5875;

        // Reset count variables
        overflowCount = 0;
        edgeCountR = 0;
        edgeCountL = 0;
    }
    // End of encoder variables
}

/**
    correctEncoderDirection corrects the encoder direction based on the
    power input from the motors.
*/
void correctEncoderDirection(void) {
    if (power.right < 0) {
        encoderPowerR = encoderPowerR * -1;
    }
    if (power.left < 0) {
        encoderPowerL = encoderPowerL * -1;
    }
}

/**
    encoderPIDCalculation uses proportional control to adjust the power that
    is output to the motors by the setMotors function.

        @see setMotors
*/
void encoderPIDCalculation(void) {
    // Set pid gains from controller
    kp = config.pid_p * 0.001;

    powerErrorR = power.right - encoderPowerR;
    powerErrorL = power.left - encoderPowerL;

    powerPIDR = kp * powerErrorR;
    powerPIDL = kp * powerErrorL;


    power.right = power.right + (char)powerPIDR;
    power.left = power.left + (char)powerPIDL;

    return;
}
////////////////////////////////////////////////////////////////////////////////

void main(void){

    // Setup for USART
    configureUART();
    configureUARTInterrupts();

         motorsSetup();
    max_speed = 80;
    max_yaw = 300;
    configInit();
    mode = 1;

    // Setup for encoders
    TMR1Setup();
    PORTDSetup();
    interruptSetup();
    // End setup for encoders

    // Setup for IR timer
    TMR0Setup();
    initADC();
```

```
    for(;;){

       // Send string back to controller
        xBeeSend("SSending");
        power = vectorToMotor(velocity);// output x,y value

        userAssist();

        // Start Encoder RPM calcualtion
        // Calculate rpm from encoders
        calculateRPM();

        if(velocity.z == 1){
            mode = 1;
        }
        // Correct encoder direction
        correctEncoderDirection();

        // End encoder RPM calculations

        // Encoder PID calculation
        encoderPIDCalculation();

        // Decipher what mode system is in and set power accordingly
        if(mode)setMotors(power);
        if(!mode)setMotors(IR_power);

    }
}
```