Autonomous surf life saving device

Jarod Lam
Supervisor: Matthew Dunbabin

2018-2019 VRES project at Queensland University of Technology

13th February 2019

***Abstract*—Life saving services rescue thousands of people every year and are essential to keeping public beaches safe. However, accidents can still happen and life savers are often required to place themselves at risk. This paper describes the design and testing of the Surf Rescue Boat (SRB), a system intended to supplement the work of life saving services at public beaches. The system comprises a semi-autonomous water vehicle (the "boat"), a communications system, and a computer vision system as an interface for navigation (not developed in this project). The boat uses a life saving rescue board as a base, with two thrusters mounted on the bottom and a waterproof case of electronics on the top. It is intended to roam behind the surf breach and be directed to positions through the communications system from an onshore base station. When a person in distress is identified, the boat is sent by the user to the the GPS coordinates of the person, where the boat provides support while the person is rescued. Over the course of this project, the boat and communications were developed to a stage where they could be tested under controlled conditions in the water; however, time constraints prevented field testing from being completed. Several areas for future development are proposed for this project to move it toward an operational system. The most significant of these proposals is the development of the computer vision component, which will allow GPS positions to be selected from a live onshore camera feed. With more development from its proof of concept stage, the Surf Rescue Boat and similar systems could assist in saving countless lives along public beaches in the future.**

CONTENTS

I Introduction 2

II Boat 3
- II-A Mechanical . . . 3
- II-B Electronics . . . 3
- II-C Software . . . 4

III Communications 5
- III-A Protocol . . . 5
- III-B Hardware . . . 5
- III-C Software . . . 6

IV Testing 6
- IV-A Power consumption . . . 6
- IV-B Forward speed . . . 6
- IV-C Turning speed . . . 6

V Future development 7
- V-A Boat . . . 7
- V-B Communications . . . 7
- V-C Computer vision . . . 7

VI Conclusion 7

References 7

Appendix A: Code 8
- A-A `srb` . . . 8
- A-B `srb-base` . . . 16

Appendix B: Drawings 18
- B-A Electronics housing . . . 18
- B-B Motor mount . . . 21

I. INTRODUCTION

Surf life savers regularly patrol beaches to help those in danger and are essential to keeping public beaches safe. In Queensland alone, over three thousand are rescued and hundreds are resuscitated by life saving services every year [1]. However, whilst saving many lives, from 2008-2018, there was an average of 47 drowning deaths per year at Australian beaches–a tragically high number that many organisations are working to reduce [2]. In addition, surf conditions can be just as dangerous for the rescuer as they are for the rescuee.

To supplement the activities of surf life savers and other services at public beaches, a system has been proposed that will allow timely help to be given to people in distress while waiting to be rescued. The Surf Rescue Boat (SRB) aims to deliver help quickly and reduce the risk to which life savers are exposed.

The basic concept of the design is a remotely-controlled floating water vehicle that sits behind the surf breach away from the shore at a beach. At most times, the vehicle is idle and remains stationary in the water. When a person in distress is seen, a life saver on the shore can remotely direct the vehicle to the person to support them while they wait for help.

A simple water-based robot such as the one proposed can be constructed relatively cheaply and easily with off-the-shelf components. In the future, systems such as these may become widely available and save the lives of many along coastal beaches.

The Surf Rescue Boat is divided into three main systems, shown in Figure 1: the remotely operated water vehicle (the "boat"), an XBee-based radio communications system between the boat and a base station, and a computer vision-based control system. Out of these, only the vehicle and communications were prototyped in this project; the control system has been developed separately in the past and time constraints prevented it from being implemented.

Design considerations taken into account include cost of construction, parts, and maintenance; usability and user-friendliness; and effectiveness as a water-based vehicle.

This report describes these systems in detail, the design and testing methodology, and avenues that can be explored for future development of the Surf Rescue Boat.

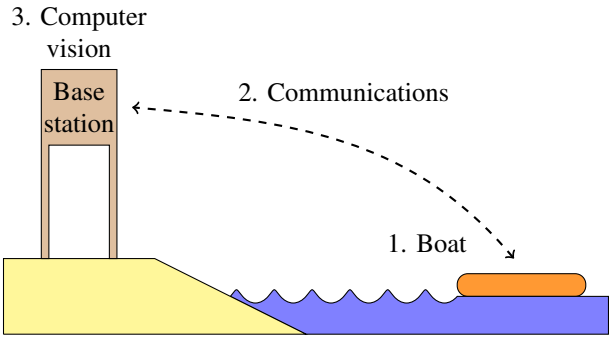


Fig. 1. Overview of the Surf Rescue Boat systems on a cross-section of a beach. The boat is behind the surf break and communicates wirelessly with the computer vision base station on a lifeguard tower.

II. Boat

The boat comprises mechanical, electrical, and software components. The electronic systems are encased in a box strapped to the topside of a rescue board, while two thrusters are mounted to the underside. Photos of both sides of the boat can be seen in Figure 2.

Fig. 2. Photos of the top (left) and bottom (right) sides of the boat. The electronics box is strapped to the top and thrusters are mounted to the bottom, with wires connecting around the sides.

A. Mechanical

The remotely operated boat uses a standard rescue board as a base, and houses electronics in a watertight hard plastic case attached to the top. Two thrusters are mounted to the bottom of the rescue board for movement control.

1) Chassis: Rescue board. Chosen for its stability and familiarity in the surf. The standardness and availability of rescue boards is an advantage to encouraging development of such systems. A custom-built chassis may have been designed, but would have taken more time and money.

Fig. 3. Close-up photo of Blue Robotics T200 thrusters mounted to the underside of the rescue board.

2) Propellors: Two (2) Blue Robotics T200 brushless motor thrusters. A thruster is mounted each to the left and right of the rescue board's middle underside to provide forward and differential steering. Two aluminium mounting plates, one attached to the board with Sikaflex and another that attaches to the thrusters, were designed to distribute force and allow propellors to be detached easily. An image of this setup is shown in Figure 3. Schematics for the mounting plates can be found in Appendix B-B.

3) Electronics housing: Pelican 1120 Case. A laser-cut acrylic frame was made to mount the electronics in the box. Wire glands on the side of the box allow propellor wires to be fed from the thrusters through the box walls.

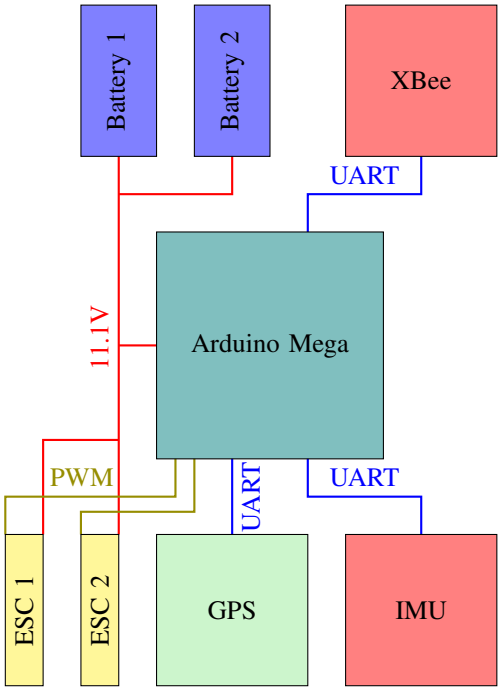


Fig. 4. Connection block diagram for onboard electronics.

B. Electronics

An Arduino Mega 2560 controls the onboard electronics mounted in the case. GPS and IMU modules are used for navigation, and an XBee radio communicates with the base station. Two electronics speed controllers (ESCs) control the two thrusters. A block connection diagram of the electronics setup is shown in Figure 4. A CAD rendering of the physical arrangement of the electronics is shown in Figure 5, and a photo of the electronics assembly is shown in Figure 6.

1) Microcontroller: Arduino Mega 2560 with Seeedstudio Grove Mega Shield breakout board. This development board is powerful enough to handle relatively simple communication and processing tasks required to control the boat's sensors and motors. More powerful ARM-based boards such as the Raspberry Pi are less suited to rugged environments, and more difficult to recover from failures. The shield provides robust headers to the UART functions of the Arduino.

2) Radio: Digi XBee Pro S1 on a SparkFun XBee Explorer Regulated. This connects to the Arduino via UART, and creates a wireless serial connection to the base station. The protocol is defined in section 3.

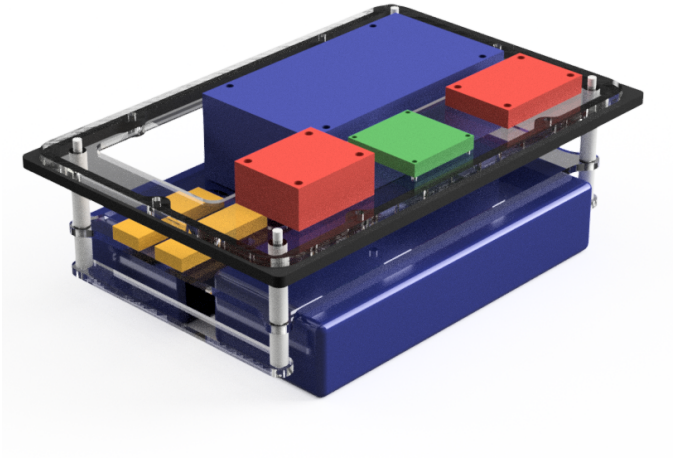


Fig. 5. CAD rendering of internal electronics frame within the Pelican case. Hardware is arranged in three layers and mounted on a clear 3 mm frame separated by plastic PCB standoffs. The first layer contains the XBee radio, GPS receiver, and IMU. The second layer contains the two ESCs and the Arduino Mega 2560. The third layer contains the two batteries.

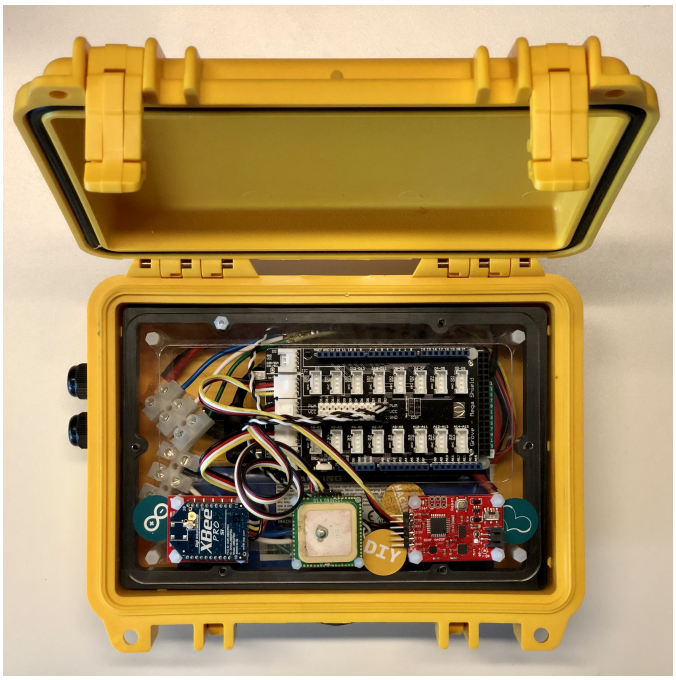


Fig. 6. Top view of the open Pelican case, showing the assembled electronics mounted on the internal frame. Visible from the top are (left to right, top to bottom): Arduino Mega 2560, XBee radio, GPS receiver, and IMU. The two black wire glands on the outer left side of the case allow thruster wires to pass through the case, which connect to the white screw terminals in the top left.

3) GPS: LOCOSYS LS20031 GPS receiver. Sends data to the Arduino via UART using the NMEA 0183 protocol found in section 3.

4) IMU: Sparkfun SEN-10736 9DOF Razor IMU. Sends data to the Arduino via UART. Currently, only the compass value from the sensor is used. Flashed with the Razor AHRS firmware: https://github.com/Razor-AHRS/razor-9dof-ahrs

5) Motor control: Two (2) Flycolor Raptor 390 30A ESC. Firmware modified to allow forward and backward thruster movement. Receives PWM control signals from the Arduino with a duty cycle range of 1000 µs to 2000 µs.

6) Battery: Two (2) Zippy Flightmax Z58003S-30 5800 mAh 3S1P. The two batteries are wired in parallel, with a combined nominal capacity of 11.6 Ah and nominal voltage of 11.1 V.

C. Software

The Arduino Mega 2560 is programmed in C++ on top of Arduino default and custom libraries. Efforts were made to keep the code somewhat portable and reusable.

The code is split up into several modules, each handling a section of robot operation. These are described below, and the full code can be found in appendix A.

1) `srb`: Contains the main loop of the program. Initialises values, classes, etc. Runs update functions for GPS, comms, IMU, nav, and motors. Sends `SRBSM` message at intervals. An AVR watchdog timer is set to reset the microcontroller at the hardware level if the program hangs and reaches a timeout.

2) `nmea`: Defines the `Nmea` class, which contains functions for constructing and parsing NMEA 0183 sentences. This includes generating and validating checksums, counting the number of fields, appending strings and decimal numbers to a sentence, and parsing a sentence by field. All functions use standard C string libraries, so they do not rely on Arduino libraries and will work outside the Arduino environment. Used by `SrbGps` and `SrbComms`.

3) `srb_stats`: Defines the `SrbStats` class, which stores the ID, state, GPS and target location, compass and target heading, and other information related to the current state and navigation of the boat. A pointer to the same instance of this class is passed to most other classes when they are created so that they can read and update this information.

4) `srb_serial`: Defines the `SrbSerial` class, which buffers a hardware serial stream and parses the input when a newline is received. The serial port used is configured when the object is created. This is the base class for `SrbGps`, `SrbComms`, and `SrbImu`.

5) `srb_gps`: Defines the `SrbGps` class, which receives and parses GPS fix data over serial. Latitude, longitude, magnetic variation, and ground speed are parsed from the NMEA `GPRMC` sentence and stored in the `SrbStats` object. Conversions are made from knots to metres per second, and degrees/minutes to decimal degrees.

6) `srb_comms`: Defines the `SrbComms` class, which sends and receives messages to and from the base station via the XBee radio. Contains functions for constructing and parsing the proprietary NMEA sentences defined in section 3. Stores information and instructions received in the `SrbStats` object. Stops the boat if no message is received within a timeout period.

7) `srb_imu`: Defines the `SrbImu` class, which receives data from the Sparkfun IMU over serial. Extracts the compass heading from the serial stream and stores it in the `SrbStats` object.

8) srb_motor: Defines the SrbMotor class, which controls motor movement. Receives motor power ranges from -100 to 100 and sets the corresponding PWM duty cycle. Accelerates motors to the desired speed at a safe pace.

9) srb_nav: Defines the SrbNav class, which controls robot navigation. In manual mode, sets motor speed and orients the boat according to target speed and heading sent from the base station. In auto mode, moves the boat toward a set of coordinates sent from the base station. Motors are controlled with the SrbMotor class.

III. Communications

Communications between the SRB and the base station are achieved using XBee radios. By attaching a pair of XBee modules to the base station computer and the on-board Arduino, a virtual serial connection is created between the two devices.

A. Protocol

NMEA 0183 is a communications specification designed to create a standardised serial interface for GPS devices. Every NMEA 'sentence' begins with a $ and ends with *CS\r\n, where CS is a two-digit hexadecimal checksum of the sentence. Some advantages of using NMEA sentences are that they are standardised, human-readable, robust, and relatively simple to implement.

A common NMEA sentence type is GPRMC, the GPS recommended minimum. This sentence is used to receive information from the onboard GPS module. GPRMC sentences are specified as follows: [3]

```
$GPRMC,<Time>,<Status>,<Lat>,<LatDir>,
<Lon>,<LonDir>,<Speed>,<Angle>,<Date>,
<MagVar>,<MagDir>*CS
```

Fields:

<Time>	UTC timestamp in HHmmss format
<Status>	Status A=active, V=void
<Lat>	Latitude in ddmm.mmm format
<LatDir>	N or S hemisphere
<Lon>	Longitude in dddmm.mmm format
<LonDir>	E or W hemisphere
<Speed>	Ground speed in knots
<Angle>	Track angle in degrees from north
<Date>	Date in DDMMYY format
<MagVar>	Magnetic variation magnitude
<MagDir>	Magnetic variation direction

A NMEA sentence parser and constructor was written in C++ and Python for the boat and the base station, respectively. Specified below is a set of custom NMEA sentence types that was created for communication between the boat and the base station over the XBee radios.

1) SRBSM - Status Message: The SRBSM sentence is sent periodically by the boat to update the base station with status information.

```
$SRBSM,<ID>,<State>,<Lat>,<Lon>,<Speed>,
<Heading>,<BattV>,<FwdPower>,
<TgtHeading>*CS
```

Fields:

<ID>	ID of target SRB
<State>	0=disabled, 1=manual, 2=auto
<Lat>	Latitude in decimal degrees
<Lon>	Longitude in decimal degrees
<Speed>	Speed in metres per second
<Heading>	Compass heading in deg CW from N
<BattV>	Current battery voltage
<FwdPower>	Forward power from -100 to 100
<TgtLat>	Target latitude in decimal degrees
<TgtLon>	Target longitude in decimal degrees
<TgtHeading>	Target heading in deg CW from N

2) SRBJS - Joystick: The SRBJS sentence is sent by the base station for manual control of the boat.

```
$SRBJS,<ID>,<FwdPower>,<Turn>*CS
```

Fields:

<ID>	ID of target SRB
<FwdPower>	Forward power from -100 to 100
<Turn>	Turning dir from -100 left to 100 right

3) SRBWP - Waypoint: The SRBWP sentence is sent by the base station to autonomously direct the boat to a set of coordinates.

```
$SRBJS,<ID>,<TgtLat>,<TgtLon>*CS
```

Fields:

<ID>	ID of target SRB
<TgtLat>	Target latitude in decimal degrees
<TgtLon>	Target longitude in decimal degrees

B. Hardware

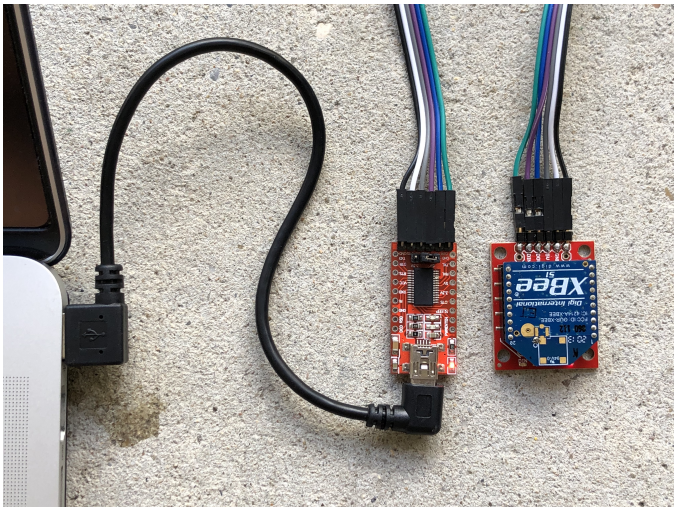


Fig. 7. Communications hardware connected via USB to the base station (left), with an XBee radio (right) and FTDI232 USB to serial UART converter (middle).

The base station uses an XBee S1 radio on a Sparkfun XBee Explorer Regulated to communicate with the boat. An FTDI232 breakout board allows the computer's USB port to interface with the XBee's UART. This system is shown in Figure 7.

C. Software

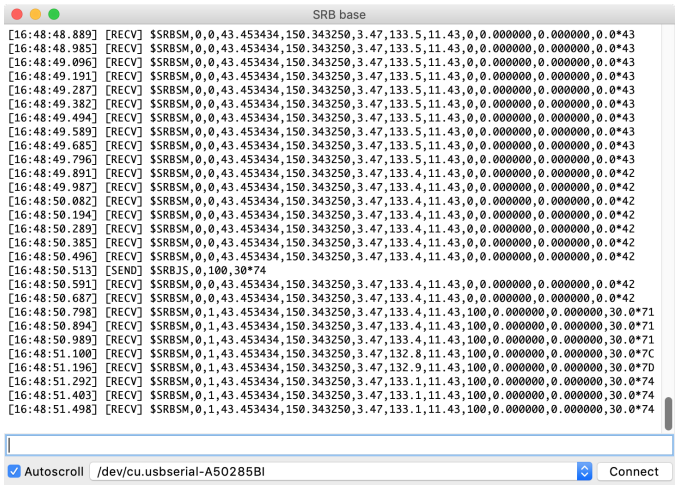


Fig. 8. Screenshot of `srb-base.py` showing NMEA sentences sent and received over the XBee connection.

The Python program `srb-base.py` was developed to provide a rudimentary graphical interface for the wireless serial connection over XBee. A screenshot of this program is shown in Figure 8 with serial input and output from the communications hardware.

The Tkinter interface was modelled on the Arduino IDE's Serial Monitor, but is designed to send NMEA sentences instead of arbitrary text over the serial connection.

"Naked" messages without a leading `$` or trailing `*CS` have these added when they are sent, making it easy to send valid sentences. Incoming messages are validated to ensure the correct syntax and checksum, and invalid sentences are shown in red. The NMEA sentence handler class, `Nmea`, is separated into the file `nmea.py`.

All incoming and outgoing messages are logged to the GUI window, terminal window, and a log file in the working directory. The log file includes a comma-separated timestamp at the beginning of each line, allowing the log data to be processed in a makeshift way as a spreadsheet.

The source code for `srb-base.py` and `nmea.py` can be found in Appendix A. However, this code is unstable and should not be used as-is, as it was written for testing purposes only.

IV. Testing

Testing will be performed on the power consumption of the boat in a lab setting, as well as the forward speed and turning speed of the boat under typical calm conditions.

Time constraints under VRES meant that most testing could not be completed. Field testing is intended to be carried out in the near future.

A. Power consumption

1) Method: The boat was elevated in midair on two stands and oriented to face 0 degrees North according to the onboard magnetometers. Two multimeters were connected to the battery, one in parallel measuring the voltage under load, and one in series measuring current.

Starting from an idle state, the message

`$SRBJS,0,<FwdPower>,0*<CS>`

was sent from the base station to the boat, with `<FwdPower>` as the throttle value being tested. Voltage and current values were recorded once they stabilised. Then, the message

`$SRBJS,0,0,0*46`

was sent from the base station to halt the motors. This was repeated for the throttle levels 0 (idle), 20, 40, 60, 80, and 100.

B. Forward speed

1) Method: The boat was placed in the water at a clear, calm location and oriented along the shoreline. A rope was attached to aid retrieval.

Starting from an idle state, the message

`$SRBJS,0,<FwdPower>,0*<CS>`

was sent from the base station to the boat, with `<FwdPower>` as the throttle value being tested. After the boat travelled roughly 20 metres, the message

`$SRBJS,0,0,0*46`

was sent back from the base station to halt the motors. The boat was moved back to its starting position. This process was repeated twice for each throttle level being tested, and for unloaded and loaded scenarios.

Messages from the boat were captured into a file using `srb-base.py`. All messages except for `SRBSM` were removed by running the terminal command:

```
$grep "SRBSM" [logfilename] > results.csv
```

which saved the results in a `.csv` file.

C. Turning speed

1) Method: The boat was placed in the water at a clear, calm location and oriented along the shoreline. A rope was attached to aid retrieval.

Starting from an idle state, the message

`$SRBJS,0,<FwdPower>,180*<CS>`

was sent from the base station to the boat, with `<FwdPower>` as the throttle value being tested. After a few seconds, the message

`$SRBJS,0,0,0*46`

was sent back from the base station to halt the motors. The boat was moved back to its starting position. This process was repeated twice for each throttle level being tested, and for unloaded and loaded scenarios.

Messages from the boat were captured into a file using `srb-base.py`. All messages except for `SRBSM` were removed by running the terminal command:

```
$grep "SRBSM" [logfilename] > results.csv
```

which saved the results in a `.csv` file.

V. Future development

The Surf Rescue Boat system in its current state is only an early prototype. Significantly more prototyping and testing needs to be done before it can be viably used in a real-world scenario. Some important areas to explore that have been identified are listed below.

A. Boat

1) Power measurement: In testing, external multimeters were used to measure voltage and current consumption of the boat. In the future, the boat would benefit from onboard power measurement hardware to monitor voltage and current usage in real-time. This could be used to automatically shut down or bring ashore the boat when the batteries are low.

2) Number of thrusters: While the two thrusters on the boat are already decently powerful, more thrusters would allow the boat to move faster under load, and reach a distressed person more quickly. Three or more thrusters arranged in different directions would increase manoeuvrability, and possibly enable onmi-directional movement.

3) Battery capacity: According to documentation, the T200 thrusters consume about $15\,\mathrm{A}$ of current at $12\,\mathrm{V}$ with full thrust. Therefore, the two $5.8\,\mathrm{Ah}$ batteries will only last roughly 20 minutes at full thrust. Much larger batteries, possibly in a separate or larger case, are required for the system to be viably used in real conditions.

4) Stability structure: In rough surf conditions, the boat would likely be tipped over by large waves. The structure needs to be modified to lower its centre of gravity below the water, increasing stability and aiding recovery from upside down positions. This could be done by attaching a heavy metal bar on the underside of the boat that extends into the water.

5) Wiring: The internal frame and electronics inside the casing would benefit from better arrangement and accessibility for maintenance. Most of the signal and power wiring could be consolidated onto a PCB which will improve the arrangement and reliability of connections. At the very least, a power switch added to the current design would make testing much easier.

B. Communications

1) Hardware housing: The communications hardware connected to the base station is a jumble of wires and exposed electronics. Ideally, these should be housed in a small box that could be easily laser-cut or 3D printed to fit. This would also allow an antenna to be mounted for the base station, increasing the range and reliability of the communications.

2) Stationary mechanism: In its current state, the boat turns off its thrusters when idle. This is not ideal because without powered movement, it will quickly be swept away by waves and currents. A mechanism should be added for the boat to maintain its position through powered movement when not moving to a new location.

3) Homing mechanism: A homing mechanism would allow the boat to be easily retrieved by a single command, and autonomously return to the base station when communications are disrupted or the battery is low.

C. Computer vision

Computer vision is the third component of the Surf Rescue Boat system, which was not explored in this project. The purpose of this component is to provide an interface for guiding the boat to a desired location in the surf, without knowing coordinates or using manual joystick control.

This is intended to be accomplished with a live camera feed from a tower on the shore, projected onto a computer or tablet screen. By calibrating the camera feed with a set of real-world coordinates, a simple image homography model can be created that translates camera pixel coordinates to GPS coordinates.

In this system, a user will select the target position of the boat on the screen, and this position will be automatically translated into GPS coordinates and sent to the boat. The goal of this is to allow the boat to be directed to a person in distress quickly and efficiently.

The computer vision section has been previously worked on in the past, separately from this project. These may be integrated in the future.

VI. Conclusion

The Surf Rescue Boat system aims to supplement the work of life saving services at public beaches, providing a technological solution that increases the efficiency and reduce the risk involved in life saving duties.

Over the course of this project, two main components of the system were developed: the roaming surface “boat”, and the communications system. Significant amounts of hardware and software were created for both components. These were both developed to a testable stage as first prototypes, but time constraints prevented field testing to be carried out before this report was completed.

The system as it currently stands shows potential to be used in surf rescue situations. However, it is very much a proof-of-concept, and significantly more development and testing is required before it can even be tested in the surf. In particular, the computer vision component needs to be added for the SRB to be viably used in real situations.

With further development, the Surf Rescue Boat could be an invaluable tool for helping people in life saving services help those in distress. In the future, similar systems could help save countless lives in the surf along coastal public beaches.

References

[1] Surf Live Saving Queensland, “Statistics,” accessed February 2019. [Online]. Available: http://lifesaving.com.au/about/statistics/

[2] Royal Life Saving Australia, “National drowning report 2018,” accessed February 2019. [Online]. Available: https://www.royallifesaving.com.au/__data/assets/pdf_file/0004/23197/RLS_NDR2018_ReportLR.pdf

[3] D. DePriest, “Nmea data,” accessed November 2018. [Online]. Available: https://www.gpsinformation.org/dale/nmea.htm

APPENDIX A
CODE

A. srb

1) srb/srb.ino:

```
/*
   srb.ino
   Surf rescue boat control system
   Written by Jarod Lam
*/

#include <avr/wdt.h>
#include "srb_stats.h"
#include "srb_motor.h"
#include "srb_nav.h"
#include "srb_comms.h"
#include "srb_gps.h"
#include "srb_imu.h"

#define LOOP_DELAY 100
#define USE_WATCHDOG

unsigned long prevMillis = 0;
int motorPins[] = {2, 3};
int motorSides[] = {LEFT, RIGHT};

SrbStats stats;
SrbMotor motors;
SrbNav   nav(&stats, &motors);
SrbComms comms(&stats, &Serial1);
SrbGps   gps(&stats, &Serial2);
SrbImu   imu(&stats, &Serial3);

void setup() {

  // Start watchdog
  #ifdef USE_WATCHDOG
  wdt_enable(WDTO_2S);
  #endif

  // Start USB serial
  Serial.begin(9600);

  // Initialise motors
  motors.begin(motorPins, motorSides);

  // Set comms failsafe timeout
  comms.setTimeout(-1);

  // Initialise stats
  stats.ID = 0;
  stats.state = 0;
  stats.lat = 43.4534324;
  stats.lon = 150.3432493;
  stats.speed = 3.4749;
  stats.heading = 174.3;
  stats.battV = 11.434;
  stats.forwardPower = 0;
  stats.targetHeading = 0;

  Serial.println("Initialising");
}

void loop() {

  // Reset watchdog timer
  #ifdef USE_WATCHDOG
  wdt_reset();
  #endif

  // Call update functions for everything!
  gps.update();       // Check GPS serial
  comms.update();     // Check XBee serial
  imu.update();       // Check IMU serial
  nav.update();       // Calculate nav based on new
      ↪ position/target
  motors.update();    // Accelerate motors to speeds set
      ↪ by nav

  // Non-blocking loop delay
  if (millis() >= prevMillis+LOOP_DELAY) {
    prevMillis = millis();

    // Send status message
    comms.sendSRBSM();

  }
}
```

2) srb/nmea.h:

```
/*
   nmea.h
   Library for manipulating NMEA-0183 strings
   Written by Jarod Lam
*/

#ifndef nmea_h
#define nmea_h

#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

#define NMEA_BUFFER_SIZE 1024

class Nmea {
  public:
    Nmea();

    /*
       Returns the sentence.
    */
    const char *read();

    /*
       Clear and set the sentence.
    */
    void write(const char *s);

    /*
       Initialise the sentence with a dollar sign.
    */
    void begin();

    /*
       Append a field to the sentence.
    */
    void append(const char *s);
    void appendInt(int d);
    void appendFloat(float d, int places);

    /*
       Calculate the checksum and append to the sentence
           ↪ .
    */
    void appendChecksum();

    /*
       Checks if an NMEA sentence is valid.

       Needs to:
         - Start with a $
         - End with * and checksum
         - Correct checksum
    */
    int validate();

    /*
       Returns the next argument of the sentence.

       Removes the argument from the string once read.
       Returns NULL if end of the string is reached.
    */
    const char *nextField();

    /*
       Returns the number of fields in the sentence.
    */
    int numFields();

  private:
    /*
       Character buffer working space
    */
    char _buffer[NMEA_BUFFER_SIZE];

    /*
```

```
        Where the sentence is stored
    */
    char _sentence[NMEA_BUFFER_SIZE];

    /*
        Generate the NMEA checksum.

        Automatically skips $ and terminates at *.
    */
    unsigned char generateChecksum(const char *s);

};

#endif
```

3) srb/nmea.cpp:

```
/*
 * nmea.cpp
 * Library for manipulating NMEA-0183 strings
 * Written by Jarod Lam
 */

#include "nmea.h"

Nmea::Nmea(void) {
  memset(&_buffer, 0, sizeof(_buffer));
  _sentence[0] = '$';
}

const char *Nmea::read() {
  return _sentence;
}

void Nmea::write(const char *s) {
  memset(_sentence, 0, sizeof(_sentence));
  strcpy(_sentence, s);
}

void Nmea::begin() {
  memset(_sentence, 0, sizeof(_sentence));
  _sentence[0] = '$';
}

void Nmea::append(const char *s) {
  int fieldLen = strlen(s);
  int sentLen = strlen(_sentence);

  // Abort if the sentence will become too long
  if ((fieldLen + sentLen) > (NMEA_BUFFER_SIZE + 5))
      ↪ return;

  // Add a comma if necessary
  if (_sentence[sentLen-1] != '$') {
    strcat(_sentence, ",");
  }

  // Concatenate
  strcat(_sentence, s);
}

void Nmea::appendInt(int d) {
  char s[16];
  snprintf(s, 16, "%d", d);
  //dtostrf(d, 0, 0, s);
  append(s);
}

void Nmea::appendFloat(float d, int places) {
  char s[16];
  dtostrf(d, 0, places, s);
  append(s);
}

void Nmea::appendChecksum() {
  char cs_string[2];
  sprintf(cs_string, "%.2X", generateChecksum(_sentence)
      ↪ );
  strcat(_sentence, "*");
  strcat(_sentence, cs_string);
}

int Nmea::validate() {
  int sentLen = strlen(_sentence);

  // Check if it starts with a dollar sign
  if (_sentence[0] != '$') return 0;

  // Check if it ends with an asterisk
  if (_sentence[sentLen-3] != '*') return 0;

  // Check the checksum
  char cs_recv = strtol(&_sentence[sentLen-2], NULL, 16)
      ↪ ;
  char cs_calc = generateChecksum(_sentence);
  if (cs_recv != cs_calc) return 0;

  return 1;
}

const char *Nmea::nextField() {
  int startInd = 0;
  int endInd = 0;
  int len = strlen(_sentence);

  // Copy the string into the buffer
  if (len > NMEA_BUFFER_SIZE) return 0;
  memset(&_buffer, 0, sizeof(_buffer));
  strcpy(_buffer, _sentence);

  // Check for end of sentence
  if (_sentence[0] == '*') return 0;

  for (int i = 0; i < len; i++) {
    // Check for dollar sign to start the string
    if (_buffer[i] == '$') {
      startInd = i + 1;
    }
    // Check for comma or asterisk to end the string
    if ((_buffer[i] == ',') || (_buffer[i] == '*')) {
      endInd = i;
      break;
    }
  }
  if (startInd > endInd) return 0;

  // Truncate the string
  strcpy(_sentence, &_buffer[endInd+1]);

  // Return the field
  if (_buffer[0] == ',') {
    return 0;
  } else {
    char returnVal[len];
    memset(returnVal, 0, len);
    strncpy(returnVal, &_buffer[startInd], endInd-
        ↪ startInd);
    strcpy(_buffer, returnVal);
    return _buffer;
  }
}

int Nmea::numFields() {
  int count = 0;
  for (int i = 0; i < strlen(_sentence); i++) {
    if (_sentence[i] == ',') {
      count++;
    }
  }
  return count + 1;
}

unsigned char Nmea::generateChecksum(const char *s) {
  unsigned char checksum = 0;

  for (int i = 0; i < strlen(s); i++) {
    char c = s[i];

    if (c == '$') continue;  // Skip the character if
        ↪ dollar sign
    if (c == '*') break;     // Stop the checksum if
        ↪ asterisk

    checksum ^= c;  // XOR to checksum
  }

  return checksum;
}
```

4) srb/srb_stats.h:

```
/*
```

```
    srb_stats.h
    Surf rescue boat control system
    Written by Jarod Lam
*/

#ifndef srb_h
#define srb_h

// Number of servos
#define NUM_MOTORS 2

// Left and right motor names
#define LEFT 0
#define RIGHT 1

// Class containing current stats of SRB
class SrbStats {

  public:

    /*
     * ID of the SRB, set at start of program
     */
    int ID;

    /*
     * State where 0=disabled, 1=manual, 2=auto
     */
    int state = 0;

    /*
     * Boolean GPS fix flag
     */
    int gpsFix = 0;

    /*
     * Current latitude and longitude in decimal degrees
     */
    float lat = 0;
    float lon = 0;

    /*
     * Current ground speed in m/s
     */
    float speed = 0;

    /*
     * Current compass heading in degrees from north
     */
    float heading = 0;

    /*
     * Current battery voltage
     */
    float battV = 0;

    /*
     * Forward power percentage (-100-100 manual, 0-100 auto)
     */
    int forwardPower = 0;

    /*
     * Target compass heading in degrees from north
     */
    float targetHeading = 0;

    /*
     * Target lat/lon in decimal degrees
     */
    float targetLat = 0;
    float targetLon = 0;

    /*
     * Magnetic variation (declination) in degrees
     */
    float magVar = 0;
};

#endif
```

5) srb/srb_serial.h:

```
/*
    srb_serial.h
    General class for receiving data over serial
    Written by Jarod Lam
*/

#ifndef srb_serial_h
#define srb_serial_h

#define SERIAL_BUFFER_SIZE 1024
#define SERIAL_BUFFER_TIMEOUT 500

#include <Arduino.h>
#include "srb_stats.h"

class SrbSerial {

  public:

    /*
     * Initialise SrbImu with the IMU serial port.
     */
    explicit SrbSerial(SrbStats *stats, HardwareSerial *port, long baud);

    /*
     * Update the values from serial.
     */
    void update();

  protected:

    /*
     * Pointer to the XBee serial port object.
     */
    HardwareSerial *_serial;

    /*
     * Pointer to the stats object given at creation.
     */
    SrbStats *_stats;

    /*
     * Buffers for storing I/O data.
     */
    char _buffer[SERIAL_BUFFER_SIZE];
    unsigned long _bufferClearTime;
    void _clearBuffer();

    /*
     * Update buffer with new serial info.
     */
    void _updateSerial();

    /*
     * Parse the contents of the buffer after a carriage return.
     */
    virtual void _parseBuffer() {};

};

#endif
```

6) srb/srb_serial.cpp:

```
/*
    srb_serial.h
    General class for receiving data over serial
    Written by Jarod Lam
*/

#include "srb_serial.h"

SrbSerial::SrbSerial(SrbStats *stats, HardwareSerial *port, long baud) {
  _serial = port;
  _serial->begin(baud);
  _stats = stats;
  _clearBuffer();
}

void SrbSerial::_updateSerial() {

  while (_serial->available()) {

    // Read a single character from serial
    char c = _serial->read();

```

```
    // Clear the buffer if line feed
    if (c == '\n') {
      _clearBuffer();
    }

    // Parse the sentence if carriage return
    else if (c == '\r') {
      _parseBuffer();
      _clearBuffer();
    }

    // Filter out non printable characters
    else if (!isPrintable(c)) {
      _clearBuffer();
    }

    // Else, add character to the buffer
    else {
      if (strlen(_buffer) >= SERIAL_BUFFER_SIZE) {
        _clearBuffer();
      }
      strncat(_buffer, &c, 1);
    }
  }

  // Timeout and clear the buffer
  if (millis() - _bufferClearTime >
      ↪ SERIAL_BUFFER_TIMEOUT) {
    _clearBuffer();
  }

}

void SrbSerial::_clearBuffer() {
  memset(_buffer, 0, sizeof(*_buffer));
  _bufferClearTime = millis();
}
```

7) `srb/srb_gps.h`:

```
/*
   nmea_gps.h
   Receive and parse GPS NMEA strings
   Written by Jarod Lam
*/

#ifndef nmea_gps_h
#define nmea_gps_h

#include <Arduino.h>
#include <Time.h>
#include "nmea.h"
#include "srb_stats.h"
#include "srb_serial.h"

class SrbGps : public SrbSerial {
  public:

    /*
     * Initialise SrbGps with the GPS serial port.
     */
    SrbGps(SrbStats *stats, HardwareSerial *port);

    /*
     * Update the values from serial.
     */
    void update();

  private:

    /*
     * Parse the sentence in the buffer.
     */
    void _parseBuffer();

    /*
     * Convert NMEA lat/lon string to decimal degrees.
     *
     * "degLen" is number of degrees digits (2 for lat,
         ↪ 3 for lon).
     */
    float _degToDec(const char *s, int degLen);

    /*
     * Convert knots string to m/s.
     */
    float _knotsToMps(const char *s);

    /*
     * Milliseconds since the buffer was last cleared.
     */
    unsigned long _bufferClearTime;
};

#endif
```

8) `srb/srb_gps.cpp`:

```
/*
   nmea_gps.cpp
   Receive and parse GPS NMEA strings
   Written by Jarod Lam
*/

#include "srb_gps.h"

SrbGps::SrbGps(SrbStats *stats, HardwareSerial *port) :
    ↪ SrbSerial::SrbSerial(stats, port, 57600) {

  // Configure to send RMC commands only
  _serial->print("$PMTK314
      ↪ ,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*29\r\n"
      ↪ );
  // Configure to send every 200ms
  _serial->print("$PMTK220,200*2C\r\n");

}

void SrbGps::update() {

  _updateSerial();

}

void SrbGps::_parseBuffer() {

  // Create a new Nmea object and put the sentence in it
  Nmea nmea;
  nmea.write(_buffer);

  // Abort if the sentence is invalid
  if (!nmea.validate()) {
    Serial.println("Invalid GPS sentence");
    return;
  }

  // 1. Type, abort if not GPRMC
  if (strcmp(nmea.nextField(), "GPRMC") != 0) return;

  // 2. Time, discard this
  nmea.nextField();

  // 3. Status, abort if inactive
  if (strcmp(nmea.nextField(), "A") == 0) {
    _stats->gpsFix = 1;
  } else {
    _stats->gpsFix = 0;
    return;
  }

  // 4-5. Latitude
  _stats->lat  = _degToDec(nmea.nextField(), 2);
  _stats->lat *= (strcmp(nmea.nextField(), "N") == 0) ?
      ↪ 1 : -1;

  // 6-7. Longitude
  _stats->lon  = _degToDec(nmea.nextField(), 3);
  _stats->lon *= (strcmp(nmea.nextField(), "E") == 0) ?
      ↪ 1 : -1;

  // 8. Speed (in knots)
  _stats->speed = _knotsToMps(nmea.nextField());

  Serial.print(_stats->lat, 6);
  Serial.print(",");
  Serial.print(_stats->lon, 6);
  Serial.println("");

  // Discard these fields:
  // 9. Track angle
  nmea.nextField();
  // 10. Date
```

```
  nmea.nextField();

  // 11-12. Magnetic variation
  _stats->magVar  = strtod(nmea.nextField(), NULL);
  _stats->magVar *= (strcmp(nmea.nextField(), "E") == 0)
      ↪  ? 1 : -1;

}

float SrbGps::_degToDec(const char *s, int degLen) {

  // Convert degrees portion
  char degStr[degLen];
  strncpy(degStr, s, degLen);
  float deg = strtod(degStr, NULL);

  // Convert minutes portion
  float mins = strtod(&s[degLen], NULL);

  // Minutes to decmial using integer maths
  long minsInt = mins * 10000;
  minsInt = minsInt * 100 / 60;
  mins = minsInt / 1000000.0;

  // Add the two
  return deg + mins;

}

float SrbGps::_knotsToMps(const char *s) {

  float knots = strtod(s, NULL);
  return knots / 1.944;

}
```

9) srb/srb_comms.h:

```
/*
   srb_comms.h
   Communication functions for SRB
   Written by Jarod Lam
*/

#ifndef srb_comms_h
#define srb_comms_h

#include <Arduino.h>
#include "nmea.h"
#include "srb_stats.h"
#include "srb_serial.h"

class SrbComms : public SrbSerial {
  public:

    /*
     * Initialise SrbGps with the XBee serial port.
     */
    SrbComms(SrbStats *stats, HardwareSerial *port);

    /*
     * Set the failsafe timeout in milliseconds.
     * Run during setup.
     */
    void setTimeout(int ms);

    /*
     * Update the values from serial.
     */
    void update();

    /*
     * Send a message over serial
     */
    void sendMessage(const char* s);

    /*
     * General function for parsing SRB sentences.
     */
    void parseSentence(char *s);

    /*
     * Send SRBSM message
     */
    void sendSRBSM();

  private:

    /*
     * Parse contents of buffer.
     */
    void _parseBuffer();

    /*
     * Failsafe timeout length. -1 for disabled.
     */
    int _failsafeTimeout = -1;
    unsigned long _lastRecvMillis = 0;

    /*
     * Functions for parsing sentences.
     */
    void _readSRBJS(Nmea *nmea);
    void _readSRBWP(Nmea *nmea);
};

#endif
```

10) srb/srb_comms.cpp:

```
/*
   srb_comms.cpp
   Communication functions for SRB
   Written by Jarod Lam
*/

#include "srb_comms.h"

SrbComms::SrbComms(SrbStats *stats, HardwareSerial *port
    ↪ ) : SrbSerial(stats, port, 9600) {

}

void SrbComms::setTimeout(int ms) {
  _failsafeTimeout = ms;
}

void SrbComms::update() {

  // Check for failsafe timeout
  if (_failsafeTimeout > 0 &&
      _lastRecvMillis + _failsafeTimeout < millis()) {
    _stats->state = 0;
  }

  _updateSerial();

}

void SrbComms::sendMessage(const char* s) {
  _serial->println(s);
}

void SrbComms::_parseBuffer() {
  parseSentence(_buffer);
}

void SrbComms::parseSentence(char *s) {
  // Create a new Nmea object to parse the sentence
  Nmea nmea;
  nmea.write(s);

  // Check for valid sentence
  if (!nmea.validate()) {
    Serial.print("Invalid sentence received: ");
    Serial.println(s);
    return;
  }

  // 1. Sentence type
  const char *type = nmea.nextField();

  if (strcmp(type, "SRBJS") == 0) {
    _readSRBJS(&nmea);
  }
  else if (strcmp(type, "SRBWP") == 0) {
    _readSRBWP(&nmea);
  }
  else {
    Serial.print("Sentence type '");
    Serial.print(type);
    Serial.println("' not recognised.");
```

```
62      return;
63    }
64  }
65
66  void SrbComms::sendSRBSM() {
67    // Create NMEA object
68    Nmea nmea;
69    nmea.begin();
70
71    // 1. Sentence type
72    nmea.append("SRBSM");
73
74    // 2. SRB ID
75    nmea.appendInt(_stats->ID);
76
77    // 3. State
78    nmea.appendFloat(_stats->state, 0);
79
80    // 4. Latitude
81    nmea.appendFloat(_stats->lat, 6);
82
83    // 5. Longitude
84    nmea.appendFloat(_stats->lon, 6);
85
86    // 6. Speed
87    nmea.appendFloat(_stats->speed, 2);
88
89    // 7. Heading
90    nmea.appendFloat(_stats->heading, 1);
91
92    // 8. Battery voltage
93    nmea.appendFloat(_stats->battV, 2);
94
95    // 9. Forward power
96    nmea.appendInt(_stats->forwardPower);
97
98    // 10. Target latitude
99    nmea.appendFloat(_stats->targetLat, 6);
100
101   // 11. Target longitude
102   nmea.appendFloat(_stats->targetLon, 6);
103
104   // 12. Target heading
105   nmea.appendFloat(_stats->targetHeading, 1);
106
107   // Checksum
108   nmea.appendChecksum();
109
110   // Send the message
111   sendMessage(nmea.read());
112 }
113
114 void SrbComms::_readSRBJS(Nmea *nmea) {
115   // Check number of fields
116   if (nmea->numFields() != 3) return;
117
118   // 2. SRB ID, abort if doesn't match
119   int recvID = strtod(nmea->nextField(), NULL);
120   if (recvID != _stats->ID) return;
121
122   // 3. Forward power
123   _stats->forwardPower = strtod(nmea->nextField(), NULL)
        ↪ ;
124
125   // 4. Heading
126   _stats->targetHeading = strtod(nmea->nextField(), NULL
        ↪ );
127
128   // Set state
129   _stats->state = 1;
130 }
131
132 void SrbComms::_readSRBWP(Nmea *nmea) {
133
134   // Check number of fields
135   if (nmea->numFields() != 3) return;
136
137   // 2. SRB ID, abort if doesn't match
138   int recvID = strtod(nmea->nextField(), NULL);
139   if (recvID != _stats->ID) return;
140
141   // 3. Target latitude
142   _stats->targetLat = strtod(nmea->nextField(), NULL);
143
144   // 4. Target longitude
145   _stats->targetLon = strtod(nmea->nextField(), NULL);
146
147   // Set state
148   _stats->state = 2;
149 }
```

11) srb/srb_imu.h:

```
1  /*
2     srb_imu.h
3     IMU receiver for Sparkfun SEN-10736 using Razor AHRS
          ↪ firmware
4     Written by Jarod Lam
5  */
6
7  #ifndef srb_imu_h
8  #define srb_imu_h
9
10 #define IMU_BUFFER_SIZE 1024
11 #define IMU_BUFFER_TIMEOUT 1000
12
13 #include <Arduino.h>
14 #include "srb_serial.h"
15
16 class SrbImu : public SrbSerial{
17
18   public:
19
20     /*
21      * Initialise SrbImu with the IMU serial port.
22      */
23     SrbImu(SrbStats *stats, HardwareSerial *port);
24
25     /*
26      * Update the values from serial.
27      */
28     void update();
29
30   private:
31
32     /*
33      * Parse the sentence in the buffer.
34      */
35     void _parseBuffer();
36
37 };
38
39 #endif
```

12) srb/srb_imu.cpp:

```
1  /*
2     srb_imu.cpp
3     IMU receiver for Sparkfun SEN-10736 using Razor AHRS
          ↪ firmware
4     Written by Jarod Lam
5  */
6
7  #include "srb_imu.h"
8
9  SrbImu::SrbImu(SrbStats *stats, HardwareSerial *port) :
       ↪ SrbSerial(stats, port, 57600) {
10
11 }
12
13 void SrbImu::update() {
14
15   _updateSerial();
16
17 }
18
19 void SrbImu::_parseBuffer() {
20
21   // Check if first five letters are "#YPR="
22   char s[10];
23   strncpy(s, _buffer, 5);
24   if (strcmp(s, "#YPR=") != 0) {
25     /*Serial.print("Invalid message from IMU: ");
26     Serial.println(_buffer);*/
27     return;
28   }
29
30   // Find the first comma position
31   int endInd = 0;
32   for (int i = 5; i < strlen(_buffer); i++) {
33     if (_buffer[i] == ',') {
34       endInd = i;
```

```
      break;
    }
  }

  // Make sure string length is within reasonable bounds
  int numLen = endInd - 5;
  if (numLen >= 10 || numLen <= 0) {
    Serial.print("Unable to parse number from IMU: ");
    Serial.println(_buffer);
    return;
  }

  // Extract the number from the string
  memset(s, 0, sizeof(s));
  strncpy(s, &_buffer[5], numLen);

  // Decode the number
  float heading = strtod(s, NULL);

  // Wrap around negative values
  if (heading < 0) {
    heading += 360;
  }
  heading = constrain(heading, 0, 360);

  // Add magnetic variation (declination)
  //heading += _stats->magVar;

  // Update stats value
  _stats->heading = heading;

}
```

13) `srb/srb_motor.h:`

```
/*
   srb_motor.h
   Motor functions for SRB
   Written by Jarod Lam
*/

#ifndef srb_motor_h
#define srb_motor_h

#include <Arduino.h>
#include <Servo.h>
#include <stdlib.h>
#include "srb_stats.h"

// Pulse width for idle motor signal
#define SRB_SERVO_IDLE 1480

// Motor acceleration in power percentage points per second
#define MOTOR_ACCEL 100

class SrbMotor {

  public:

    /*
     * Initialise the servo objects, call in setup()
     */
    void begin(int pins[], int sides[]);

    /*
     * Update motors every tick
     */
    void update();

    /*
     * Set power for specified motor
     */
    void setPower(int motorNo, float power);

    /*
     * Set power for specified side of motors
     */
    void setSidePower(int side, float power);

    /*
     * Set power for all left/right motors
     */
    void setLeft(int power);
    void setRight(int power);

    /*
     * Stop all motors
     */
    void stopAll();

    /*
     * Return number of motors
     */
    int count();

    /*
     * Return the side of the SRB the motor is on (0=left, 1=right)
     */
    int side(int motorNo);

  private:

    /*
     * Servos
     */
    Servo _motors[NUM_MOTORS];

    /*
     * Motor sides
     */
    int _motorSides[NUM_MOTORS];

    /*
     * Motor target powers
     */
    float _currentPowers[NUM_MOTORS];

    /*
     * Milliseconds since last update
     */
    unsigned long _lastMillis;

    /*
     * Directly set speed of motor
     */
    void _setMotor(int motorNo, float power);

};

#endif
```

14) `srb/srb_motor.cpp:`

```
/*
   srb_motor.cpp
   Motor functions for SRB
   Written by Jarod Lam
*/

#include "srb_motor.h"

void SrbMotor::begin(int pins[], int sides[]) {

  // Attach motors to their pins
  for (int i = 0; i < NUM_MOTORS; i++) {
    _motors[i].attach(pins[i], SRB_SERVO_IDLE-430, SRB_SERVO_IDLE+430);
    _motors[i].writeMicroseconds(SRB_SERVO_IDLE);
    _motorSides[i] = sides[i];
  }

  delay(500);

  _motors[0].write(90);
  _motors[1].write(90);

  //_motors[0].writeMicroseconds(SRB_SERVO_IDLE);
  //_motors[1].writeMicroseconds(SRB_SERVO_IDLE);

}

void SrbMotor::update() {

  // Debug
  Serial.print("L=");
  Serial.print(_currentPowers[0]);
  Serial.print(" R=");
  Serial.println(_currentPowers[1]);

}
```

```
37
38 void SrbMotor::setPower(int motorNo, float power) {
39
40   // Abort if motor number not in range
41   if (motorNo < 0 || motorNo >= NUM_MOTORS) return;
42
43   // Constrain to range
44   power = constrain(power, -100, 100);
45
46   // Set motor
47   _setMotor(motorNo, power);
48
49 }
50
51 void SrbMotor::setSidePower(int side, float power) {
52
53   for (int i = 0; i < NUM_MOTORS; i++) {
54     if (_motorSides[i] == side) {
55       setPower(i, power);
56     }
57   }
58
59 }
60
61 void SrbMotor::stopAll() {
62
63   for (int i = 0; i < NUM_MOTORS; i++) {
64     setPower(i, 0);
65   }
66
67 }
68
69 int SrbMotor::count() {
70
71   return NUM_MOTORS;
72
73 }
74
75 int SrbMotor::side(int motorNo) {
76
77   return _motorSides[motorNo];
78
79 }
80
81 void SrbMotor::_setMotor(int motorNo, float power) {
82
83   // Abort if motor number not in range
84   if (motorNo < 0 || motorNo >= NUM_MOTORS) return;
85
86   // Map value from power domain to angle domain
87   int val = map(power, -100, 100, 0, 180);
88   Serial.println(val);
89
90   // Update stored power value and servo value
91   _currentPowers[motorNo] = power;
92   _motors[motorNo].write(val);
93
94 }
```

15) srb/srb_nav.h:

```
1  /*
2     srb_nav.h
3     Navigatoin system for SRB
4     Written by Jarod Lam
5  */
6
7  #ifndef srb_nav_h
8  #define srb_nav_h
9
10 #include <Arduino.h>
11 #include <math.h>
12 #include "srb_stats.h"
13 #include "srb_motor.h"
14
15 class SrbNav {
16
17   public:
18
19     /*
20      * Intialise navigation with stats object.
21      */
22     SrbNav(SrbStats *stats, SrbMotor *motors);
23
24     /*
25      * Update function called every loop.
26      */
27     void update();
28
29   private:
30
31     /*
32      * Pointer to the stats object given at creation.
33      */
34     SrbStats *_stats;
35
36     /*
37      * Pointer to the motors object given at creation.
38      */
39     SrbMotor *_motors;
40
41     /*
42      * Navigation functions.
43      */
44     void _navDisabled();
45     void _navAuto();
46     void _navManual();
47
48     /*
49      * Get the difference between target and current
            ↪ heading
50      */
51     int _headingDiff(int goalDirection, int
            ↪ currentHeading);
52
53     /*
54      * Send the SRB in a particular direction with a
            ↪ particular power
55      */
56     void _moveTo(int power, int tHeading);
57
58     /*
59      * Calculate a multiplier used to scale motor power
            ↪ according to heading offset
60      */
61     float _turnMultiplier(int dHead);
62
63 };
64
65
66 #endif
```

16) srb/srb_nav.cpp:

```
1  /*
2     srb_nav.cpp
3     Navigation system for SRB
4     Written by Jarod Lam
5  */
6
7  #include "srb_nav.h"
8
9  SrbNav::SrbNav(SrbStats *stats, SrbMotor *motors) {
10   _stats = stats;
11   _motors = motors;
12 }
13
14 void SrbNav::update() {
15
16   // Check the SRB status and run corresponding function
17   switch(_stats->state) {
18     case 1: _navManual(); break;
19     case 2: _navAuto();   break;
20     default: _navDisabled();
21   }
22
23 }
24
25 void SrbNav::_navDisabled() {
26
27   _motors->stopAll();
28
29 }
30
31 void SrbNav::_navManual() {
32
33   _moveTo(_stats->forwardPower, _stats->targetHeading);
34
35 }
36
37 void SrbNav::_navAuto() {
38
```

```
39    // Calculate distance from goal
40    float dLat = _stats->targetLat - _stats->lat;
41    float dLon = _stats->targetLon - _stats->lon;
42
43    // Get target heading of goal (convert to deg)
44    int tHead = atan2(dLat, dLon) * 57.29578;
45
46    // Move
47    _moveTo(_stats->forwardPower, tHead);
48
49  }
50
51  int SrbNav::_headingDiff(int goalDirection, int
        ↪ currentHeading) {
52
53    int maxHeading = max(goalDirection, currentHeading);
54    int minHeading = min(goalDirection, currentHeading);
55
56    int clockDiff = maxHeading - minHeading;
57    int antiClockDiff = minHeading + 360 - maxHeading;
58    int smallest = min(clockDiff, antiClockDiff);
59
60    if (((currentHeading + smallest) % 360) ==
          ↪ goalDirection) {
61      return smallest * -1; //clockwise
62    } else {
63      return smallest;
64    }
65
66  }
67
68  void SrbNav::_moveTo(int power, int tHeading) {
69
70    // Calculate distance from heading
71    int dHead = _headingDiff(tHeading, _stats->heading);
72
73    // Set motor speeds
74    int powerL = power;
75    int powerR = power;
76
77    // Scale motors by heading rotation
78    if (dHead < 0) {
79      powerR *= _turnMultiplier(dHead);
80    } else {
81      powerL *= _turnMultiplier(dHead);
82    }
83
84    // Turn motors
85    _motors->setSidePower(LEFT,  powerL);
86    _motors->setSidePower(RIGHT, -powerR);
87
88  }
89
90  float SrbNav::_turnMultiplier(int dHead) {
91
92    if (dHead < 0) dHead *= -1;
93    dHead = constrain(dHead, 0, 180);
94
95    // Scale according to the function y = (1/90)x + 1
96    float multiplier = -(1.0 / 90.0) * dHead + 1;
97
98    return multiplier;
99
100 }
```

B. srb-base

1) srb-base/srb-base.py:

```
1   #!/usr/bin/env python3
2
3   import serial, sys, threading
4   import tkinter as tk
5   import tkinter.ttk as ttk
6   from tkinter import BOTH, END, LEFT, RIGHT, DISABLED,
        ↪ NORMAL, N, S, E, W, X, Y
7   from serial.tools.list_ports import *
8   from datetime import *
9   from nmea import *
10
11  class Application(ttk.Frame):
12
13      def __init__(self, parent, *args, **kwargs):
14          tk.Frame.__init__(self, parent, *args, **kwargs)
15
16          self.master.protocol("WM_DELETE_WINDOW", self.
                ↪ exitHandler)
17
18          self.logFileOpen()
19          self.setupGUI()
20          self.setupThread()
21
22      def setupGUI(self):
23          self.pack(fill=BOTH, expand=True)
24          self.master.title("SRB base")
25
26          self.master.bind('<Return>', self.sendSentence)
27
28          # Log box
29          textScrollFrame = ttk.Frame(self)
30          textScrollFrame.pack(fill=BOTH, expand=True)
31          textScrollFrame.grid_columnconfigure(0,weight=1)
32          textScrollFrame.grid_columnconfigure(0,weight=1)
33          textScrollFrame.grid_rowconfigure(0,weight=1)
34
35          scrollbar = ttk.Scrollbar(textScrollFrame)
36          scrollbar.grid(column=1,row=0,sticky=N+S)
37          self.textBox = tk.Text(textScrollFrame,
                ↪ yscrollcommand=scrollbar.set)
38          self.textBox.grid(column=0,row=0,sticky=N+S+E+W)
39          self.textBox.config(state=DISABLED)
40          self.textBox.tag_configure("error", foreground="
                ↪ red")
41          scrollbar.config(command=self.textBox.yview)
42
43          # Command entry
44          self.textEntry = ttk.Entry(self)
45          self.textEntry.pack(fill=X)
46
47          # Bottom controls
48          controls = ttk.Frame(self)
49          controls.pack(fill=X,ipady=3)
50
51          self.autoscroll = tk.IntVar()
52          scrollCheck = ttk.Checkbutton(controls, text="
                ↪ Autoscroll",
53                                        variable=self.
                                              ↪ autoscroll
                                              ↪ )
54          scrollCheck.pack(side=LEFT)
55          self.autoscroll.set(1)
56
57          serialPorts = [item.device for item in comports
                ↪ ()]
58          self.selectedPort = tk.StringVar()
59          self.connectButton = ttk.Button(controls, text="
                ↪ Connect",
60                                          command=self.
                                                ↪ openPort
                                                ↪ )
61          self.connectButton.pack(side=RIGHT)
62          portMenu = ttk.OptionMenu(controls, self.
                ↪ selectedPort, *serialPorts)
63          portMenu.pack(side=RIGHT,fill=X, expand=True)
64
65      def setupThread(self):
66          self.ser = serial.Serial()
67          self.serialThread = threading.Thread(target=self
                ↪ .readSerial)
68          self.serialThread.daemon = True
69
70      def log(self, text, error=False):
71          # Get current time string
72          timeStr = datetime.now().strftime("%H:%M:%S.%f")
73          timeStr = timeStr[:-3]  # Truncate microseconds
74
75          # Console
76          print("[%s] %s" % (timeStr, text))
77
78          # Log file
79          self.logFile.write(timeStr + "," + text + "\n")
80          self.logFile.flush()
81
82          # GUI window
83          self.textBox.config(state=NORMAL)
84          if error:
85              self.textBox.insert(END, "[%s] %s\n" % (
                    ↪ timeStr, text), "error")
86          else:
87              self.textBox.insert(END, "[%s] %s\n" % (
```

```
                ↪ timeStr, text))
88          if self.autoscroll.get():
89              self.textBox.yview_moveto(1)
90          self.textBox.config(state=DISABLED)
91
92      def openPort(self):
93          port = self.selectedPort.get()
94          try:
95              self.ser = serial.Serial(port, 9600, timeout
                    ↪ =1)
96              self.serialThread.start()
97              self.log("Serial port opened at %s" % self.
                    ↪ ser.name)
98          except Exception as err:
99              self.log("Error opening port %s: %s" % (port
                    ↪ , str(err)), error=True)
100             self.closePort()
101
102     def closePort(self):
103         self.ser.close()
104
105     def readSerial(self):
106         while True:
107             if self.ser.isOpen():
108                 self.connectButton.config(state=NORMAL)
109                 line = self.ser.readline()
110                 if len(line) > 0:
111                     self.parseSentence(line)
112             else:
113                 self.connectButton.config(state=DISABLED
                        ↪ )
114
115     def parseSentence(self, sent):
116         try:
117             # Create a Nmea object and validate
118             s = Nmea()
119             s.sentence = sent.decode().rstrip('\r\n')
120             if s.validate():
121                 self.log("[RECV] %s" % s.sentence)
122             else:
123                 self.log("[RECV] %s" % s.sentence, error
                        ↪ =True)
124         except Exception as err:
125             self.log("Error parsing sentence: %s" % str(
                    ↪ err), error=True)
126
127     def sendSentence(self, *args):
128         if not self.ser.isOpen(): return
129
130         # Convert to valid NMEA sentence
131         text = self.textEntry.get()
132         s = Nmea()
133         s.append(text)
134         s.appendChecksum()
135         s.sentence = s.sentence + "\r\n"
136
137         # Send over serial
138         try:
139             self.ser.write(s.sentence.encode())
140             self.textEntry.delete(0, 'end')
141             self.log("[SEND] %s" % s.sentence.rstrip('\r
                    ↪ \n'))
142         except Exception as err:
143             self.log("Error sending sentence: " + str(
                    ↪ err), error=True)
144
145     def logFileOpen(self):
146         now = datetime.now()
147         fileName = now.strftime("srb_%Y-%m-%d_%H-%M-%S.
                ↪ log")
148         self.logFile = open(fileName, "w+")
149
150     def logFileClose(self):
151         self.logFile.close()
152
153     def exitHandler(self):
154         self.closePort()
155         self.master.destroy()
156         self.logFileClose()
157
158 if __name__ == "__main__":
159     root = tk.Tk()
160     app = Application(root)
161     root.mainloop()
```

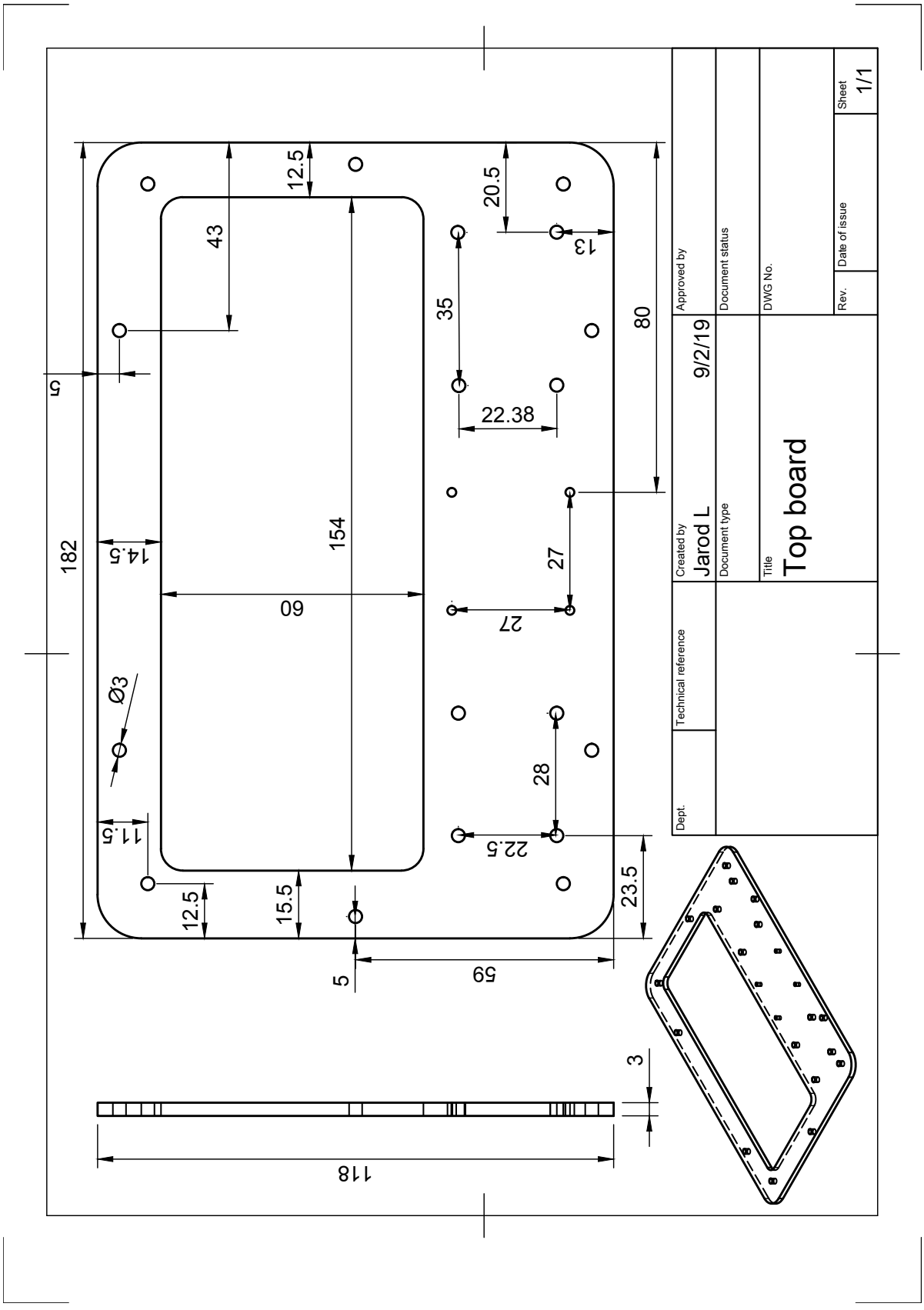
2) srb-base/nmea.py:

```
1  #!/usr/bin/env python3
2
3  class Nmea:
4
5      def __init__(self):
6          self.sentence = "$"
7
8      # Append string to end of sentence.
9      def append(self, s):
10         if len(self.sentence) > 0:
11             if self.sentence[-1] != '$':
12                 self.sentence = self.sentence + ','
13
14         self.sentence = self.sentence + s
15
16     # Generate checksum for the sentence.
17     def generateChecksum(self):
18         checksum = 0
19
20         for c in self.sentence.encode():
21             if c == '$'.encode()[0]: continue
22             if c == '*'.encode()[0]: break
23             checksum ^= c
24
25         checksumString = "%02X" % checksum
26         return checksumString
27
28     # Append checksum to the sentence.
29     def appendChecksum(self):
30         checksum = self.generateChecksum()
31         self.sentence = self.sentence + '*' + checksum
32
33     # Check if the sentence is valid.
34     def validate(self):
35         if self.sentence[0]  != '$': return False
36         if self.sentence[-3] != '*': return False
37
38         cs_recv = self.sentence[-2:]
39         cs_calc = self.generateChecksum()
40         if (cs_recv != cs_calc): return False
41
42         return True
43
44     # Return arguments as a list.
45     def listify(self):
46         sentenceStripped = self.sentence[self.sentence.
                ↪ find("$")+1 :
47                                          self.sentence.
                                                ↪ find("*
                                                ↪ ")]
48         listified = sentenceStripped.split(',')
49         return listified
```

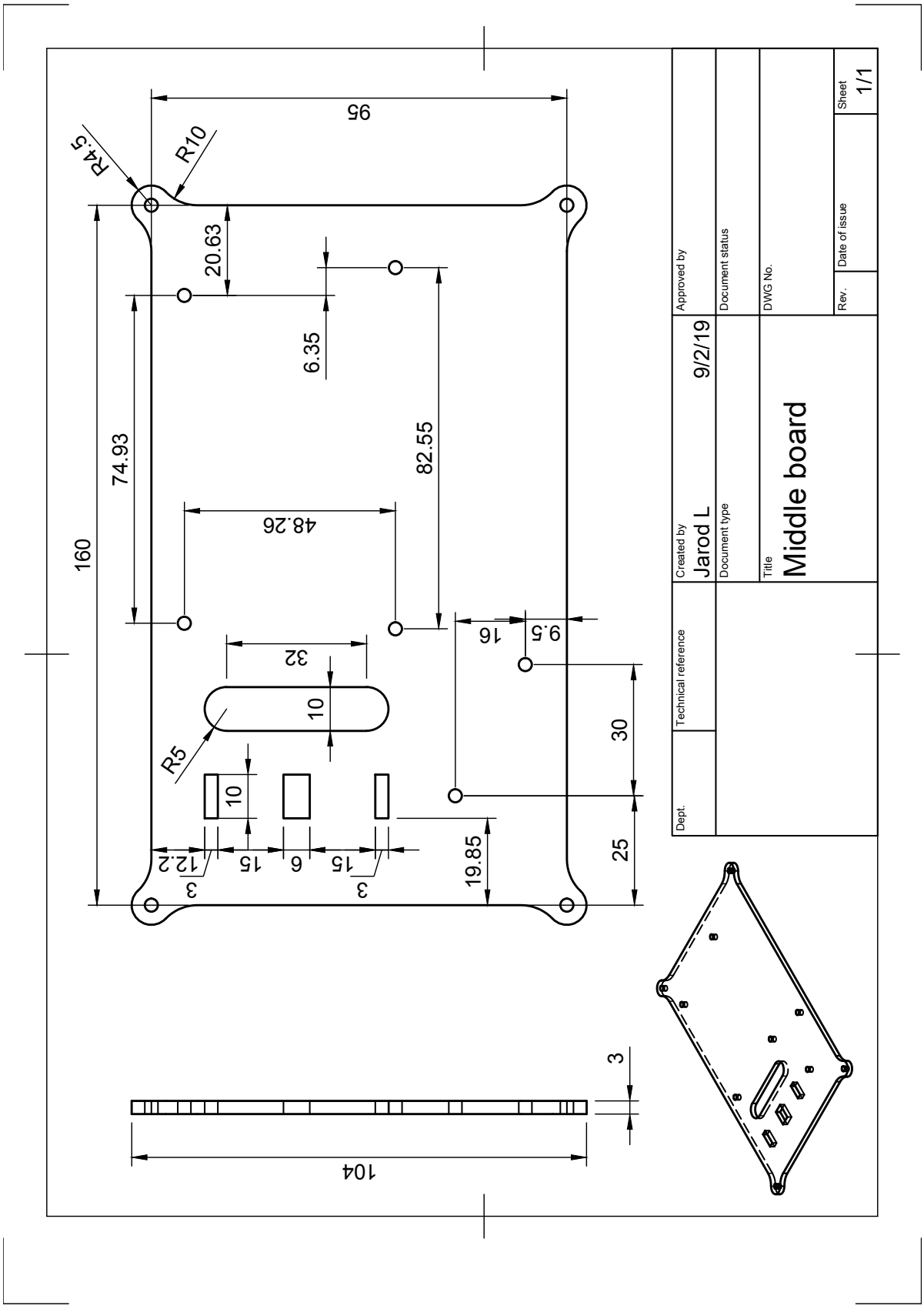
APPENDIX B
DRAWINGS

A. Electronics housing

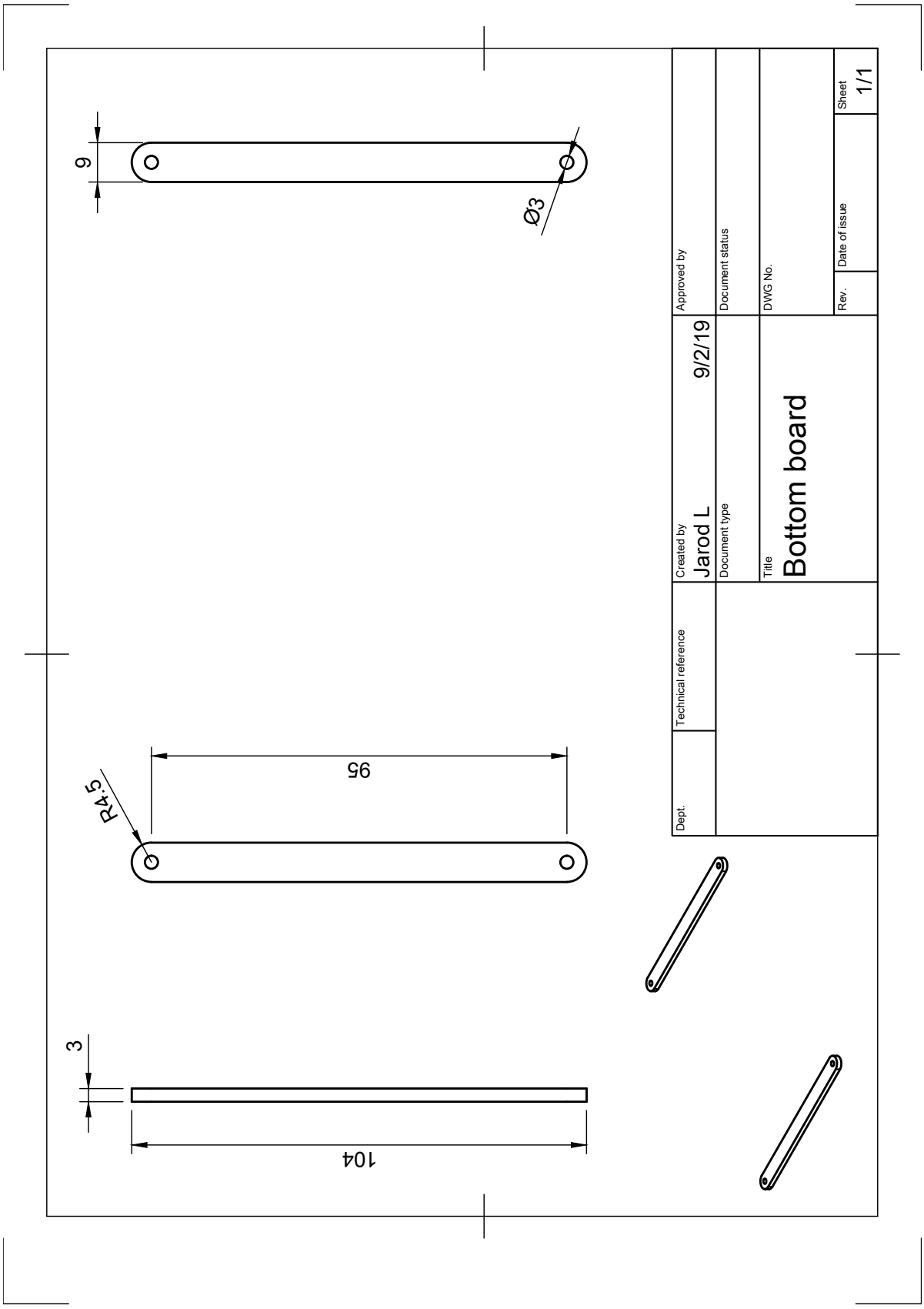
1) Top board: 3 mm clear acrylic. All dimensions in mm.



2) Middle board: 3 mm clear acrylic. All dimensions in mm.

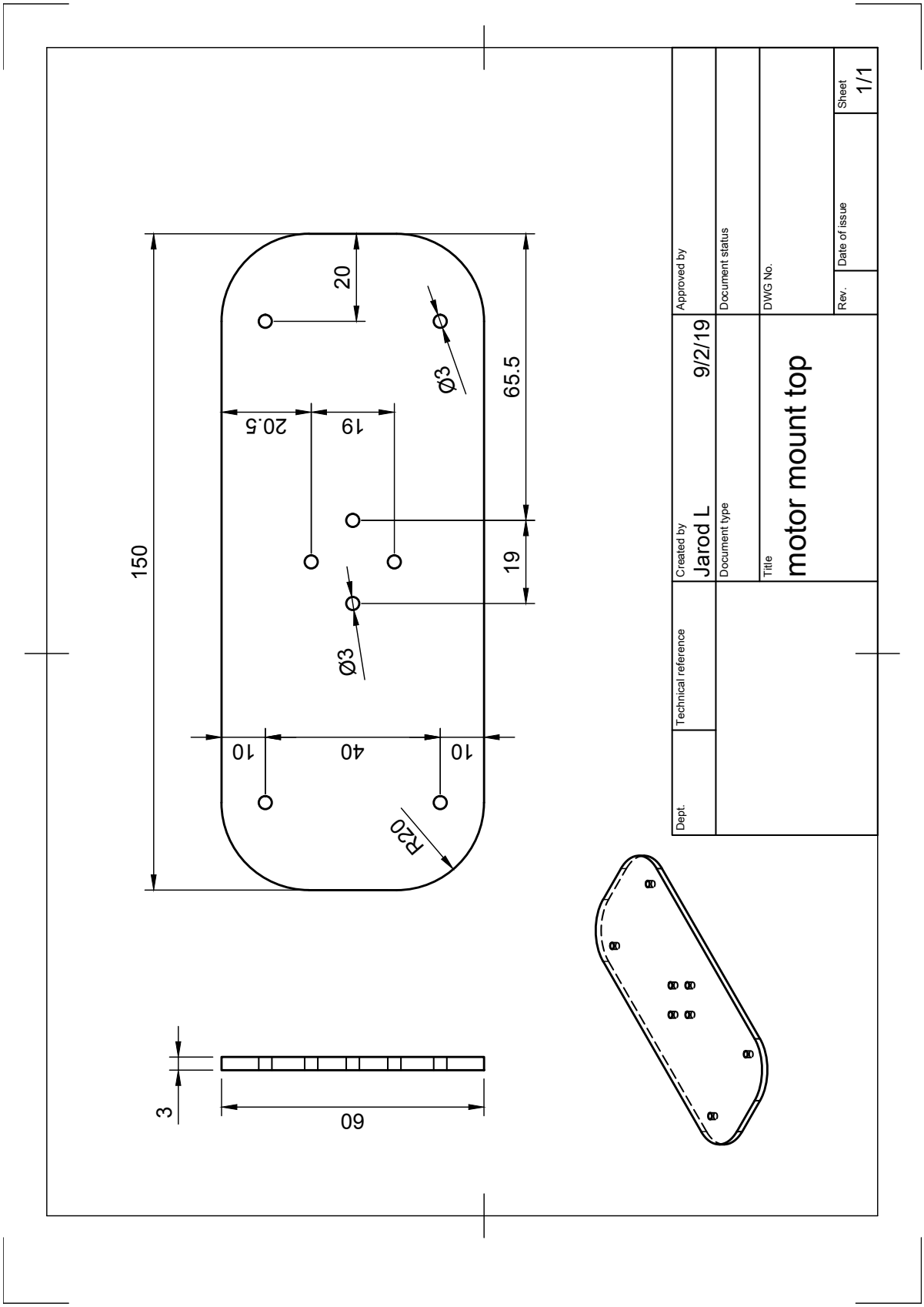


3) Bottom board: 3 mm clear acrylic. All dimensions in mm.



B. Motor mount

1) Motor mount top: 3 mm aluminium. All dimensions in mm.



2) Motor mount bottom: 8 mm aluminium. All dimensions in mm.