Autonomous surf life saving device

Jarod Lam
Supervisor: Matthew Dunbabin

2018-2019 VRES project at Queensland University of Technology

CONTENTS

***Abstract*—build boat that saves people**

I. INTRODUCTION

Surf life savers regularly patrol popular beaches to help those in danger, but there is a limit to the speed and ability of a human swimmer.

To supplement the activities of surf life savers at public beaches, a system has been proposed that will allow timely help to be given to people in danger while they wait to be rescued. The surf rescue boat (SRB) aims to deliver help quickly and reduce the risk to which life savers are exposed.

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 our coastal beaches. A prototype of one such robot was designed and built over the course of this project.

The system is divided into three main sections: a 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.

This report describes these systems in detail, the design methodology, and avenues that can be explored for future development of the surf rescue boat.

II. BOAT

A. Mechanical

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

1) Chassis: Surfboard. Chosen for its stability and familiarity in the surf. The standardness and availability of surfboards 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.

2) Propellors: 2 Blue Robotics T100 Thrusters. A propeller is mounted each to the left and right of the surfboard's middle underside. Aluminium mounting plates, attached to the board with Sikaflex, were designed to distribute force and allow propellors to be detached easily.

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 through the box walls.

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 each of the two propellers. A block connection diagram of the electronics is shown in Figure 1.

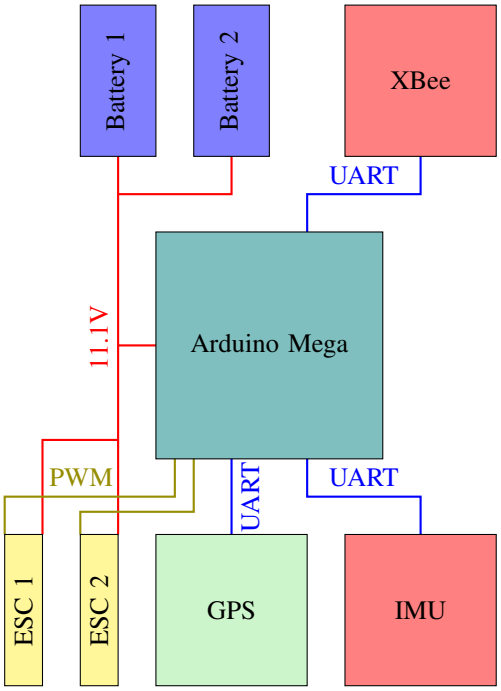

Fig. 1. Connection block diagram for onboard electronics.

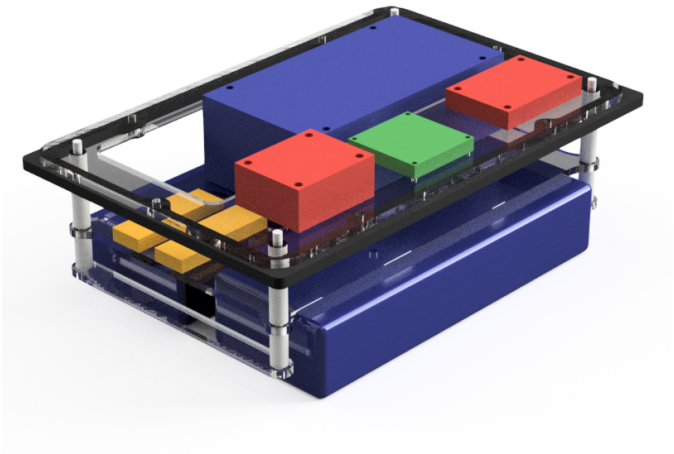

Fig. 2. Rendering of internal electronics frame

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.

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

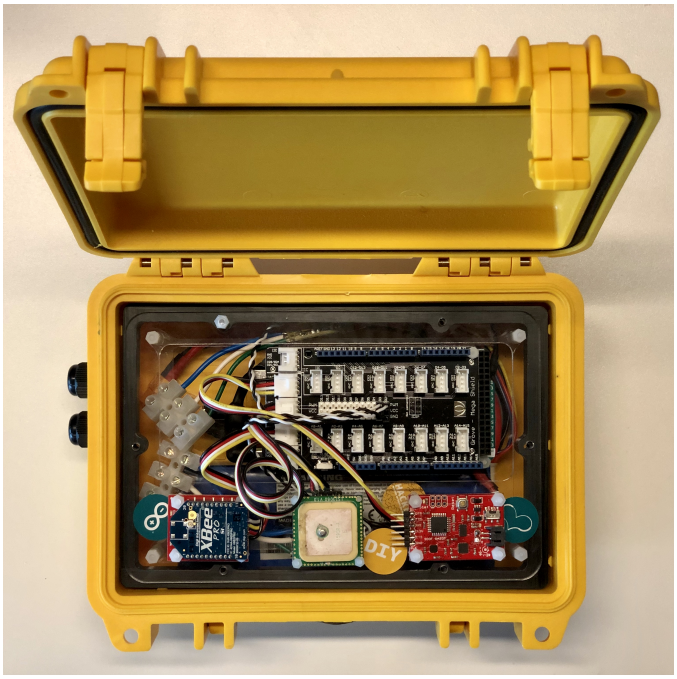

Fig. 3. Boat electronics in Pelican case housing

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

6) Battery: 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 performed with 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. NMEA 0183 sentences

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: [1]

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

Where:

<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
```

Where:

<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 degrees CW from north
<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 degrees CW from north

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

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

Where:

<ID>	ID of target SRB
<FwdPower>	Forward power from -100 to 100
<TgtHeading>	Target heading in degrees CW from north

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
```

Where:

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

B. Hardware

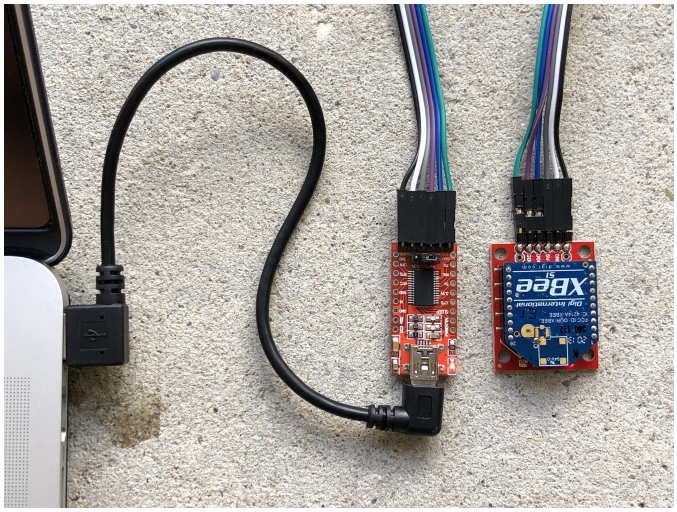

Fig. 4. Communications hardware

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 4.

C. Software

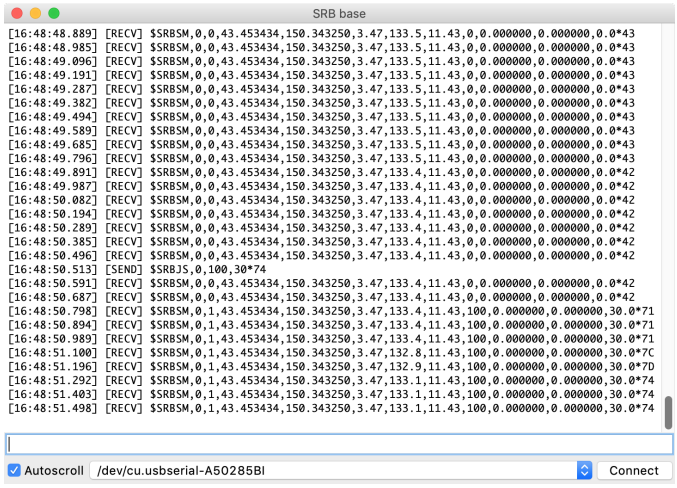

Fig. 5. Screenshot of `srb-base.py`

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 5.

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

"Naked" messages without a leading `$` or trailing `*CS` have these added when these are sent, making it easy to send valid arbitrary 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 was 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.

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.

Throttle (%)	Voltage (V)	Current (A)
0		
20		
40		
60		
80		
100		

TABLE I
Power consumption test results

2) Analysis and discussion:

B. Forward speed

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

Throttle (%)	Power (W)	11.6 Ah runtime (min)
0		
20		
40		
60		
80		
100		

TABLE II
POWER CONSUMPTION ANALYSIS

Starting from an idle state, the message

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

was sent from the base station to the boat, with `<FwdPower>` as the throttle value being tested and `<TgtHeading>` as the boat's current orientation. 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.

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.

Throttle (%)	Accel time (s)			Max speed ($\mathrm{m\,s^{-1}}$)		
	1	2	avg.	1	2	avg.
25						
50						
75						
100						

TABLE III
FORWARD SPEED TEST RESULTS

Fig. 6. Forward speed vs. time

2) Analysis and discussion:

V. FUTURE DEVELOPMENT

A. Boat

B. Communications

C. Computer vision

VI. CONCLUSION

REFERENCES

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

APPENDIX A
CODE

A. srb-base.py

```
#!/usr/bin/env python3

import serial, sys, threading
import tkinter as tk
import tkinter.ttk as ttk
from tkinter import BOTH, END, LEFT, RIGHT, DISABLED, NORMAL, N, S, E, W, X, Y
from serial.tools.list_ports import *
from datetime import *
from nmea import *

# Tkinter
class Application(ttk.Frame):

    def __init__(self, parent, *args, **kwargs):
        tk.Frame.__init__(self, parent, *args, **kwargs)

        self.master.protocol("WM_DELETE_WINDOW", self.exitHandler)

        self.logFileOpen()
        self.setupGUI()
        self.setupThread()

    # Set up GUI
    def setupGUI(self):
        self.pack(fill=BOTH, expand=True)
        self.master.title("SRB base")

        self.master.bind('<Return>', self.sendSentence)

        # Log box
        textScrollFrame = ttk.Frame(self)
        textScrollFrame.pack(fill=BOTH, expand=True)
        textScrollFrame.grid_columnconfigure(0,weight=1)
        textScrollFrame.grid_columnconfigure(0,weight=1)
        textScrollFrame.grid_rowconfigure(0,weight=1)

        scrollbar = ttk.Scrollbar(textScrollFrame)
        scrollbar.grid(column=1,row=0,sticky=N+S)
        self.textBox = tk.Text(textScrollFrame, yscrollcommand=scrollbar.set)
        self.textBox.grid(column=0,row=0,sticky=N+S+E+W)
        self.textBox.config(state=DISABLED)
        self.textBox.tag_configure("error", foreground="red")
        scrollbar.config(command=self.textBox.yview)

        # Command entry
        self.textEntry = ttk.Entry(self)
        self.textEntry.pack(fill=X)

        # Bottom controls
        controls = ttk.Frame(self)
        controls.pack(fill=X,ipady=3)

        self.autoscroll = tk.IntVar()
        scrollCheck = ttk.Checkbutton(controls, text="Autoscroll", variable=self.autoscroll)
```

```
        scrollCheck.pack(side=LEFT)
        self.autoscroll.set(1)

        serialPorts = [item.device for item in comports()]
        self.selectedPort = tk.StringVar()
        self.connectButton = ttk.Button(controls, text="Connect", command=self.openPort)
        self.connectButton.pack(side=RIGHT)
        portMenu = ttk.OptionMenu(controls, self.selectedPort, *serialPorts)
        portMenu.pack(side=RIGHT, fill=X, expand=True)

    # Threads
    def setupThread(self):
        self.ser = serial.Serial()
        self.serialThread = threading.Thread(target=self.readSerial)
        self.serialThread.daemon = True

    # Logging to Tkinter widget
    def log(self, text, error=False):
        # Get current time string
        timeStr = datetime.now().strftime("%H:%M:%S.%f")
        timeStr = timeStr[:-3]  # Truncate microseconds

        # Console
        print("[%s] %s" % (timeStr, text))

        # Log file
        self.logFile.write(timeStr + "," + text + "\n")
        self.logFile.flush()

        # GUI window
        self.textBox.config(state=NORMAL)
        if error:
            self.textBox.insert(END, "[%s] %s\n" % (timeStr, text), "error")
        else:
            self.textBox.insert(END, "[%s] %s\n" % (timeStr, text))
        if self.autoscroll.get():
            self.textBox.yview_moveto(1)
        self.textBox.config(state=DISABLED)

    # Open a serial port
    def openPort(self):
        port = self.selectedPort.get()
        try:
            self.ser = serial.Serial(port, 9600, timeout=1)
            self.serialThread.start()
            self.log("Serial port opened at %s" % self.ser.name)
        except Exception as err:
            self.log("Error opening port %s: %s" % (port, str(err)), error=True)
            self.closePort()

    # Close a serial port
    def closePort(self):
        self.ser.close()

    # Serial input
    def readSerial(self):
        while True:
            if self.ser.isOpen():
```

```
                self.connectButton.config(state=NORMAL)
                line = self.ser.readline()
                if len(line) > 0:
                    self.parseSentence(line)
            else:
                self.connectButton.config(state=DISABLED)

    # Sentence parsing
    def parseSentence(self, sent):
        try:
            # Create a Nmea object and validate
            s = Nmea()
            s.sentence = sent.decode().rstrip('\r\n')
            if s.validate():
                self.log("[RECV] %s" % s.sentence)
            else:
                self.log("[RECV] %s" % s.sentence, error=True)
        except Exception as err:
            self.log("Error parsing sentence: %s" % str(err), error=True)

    # Send a sentence and add checksum
    def sendSentence(self, *args):
        if not self.ser.isOpen(): return

        # Convert to valid NMEA sentence
        text = self.textEntry.get()
        s = Nmea()
        s.append(text)
        s.appendChecksum()
        s.sentence = s.sentence + "\r\n"

        # Send over serial
        try:
            self.ser.write(s.sentence.encode())
            self.textEntry.delete(0, 'end')
            self.log("[SEND] %s" % s.sentence.rstrip('\r\n'))
        except Exception as err:
            self.log("Error sending sentence: " + str(err), error=True)

    # Exit handler
    def exitHandler(self):
        self.closePort()
        self.master.destroy()
        self.logFileClose()

    # Open log file
    def logFileOpen(self):
        now = datetime.now()
        fileName = now.strftime("srb_%Y-%m-%d_%H-%M-%S.log")
        self.logFile = open(fileName, "w+")

    # Close log file
    def logFileClose(self):
        self.logFile.close()

# Main loop
if __name__ == "__main__":
    root = tk.Tk()
```

```
    app = Application(root)
    root.mainloop()
```

B. nmea.py

APPENDIX B
DRAWINGS

A. the drawings