Sailbot Report
ENGS 147

Peter Ankeny, Chris Bodine, Chris Hoder, Darren Reis

May 30, 2013

Contents

1 Project Overview

The goal of this project was to create a model sailboat capable of remote and autonomous control. The task we sought to complete was a traverse to a windward-lured mark and back (e.g, to sail upwind to a predetermined point, then sail back to the starting point). To do this, we developed a large model trimaran housing a GPS sensor, an anemometer, and a weathervane as control inputs, along with two servos controlling the rudder and sail positions; the control outputs. The mechanical and control design was implemented as described below, and the boat was tested on Occom Pond, in Hanover, NH, to moderate success.

Figure 1: CAD model of boat

2 Mechanical Design

To optimize the design of the boat, we used Solidworks extensively to create a full 3D model of the boat and all of its components. By working with a virtual model we were able to troubleshoot many design issues before the construction phase began. Figure 1 shows the overall design of the boat. Additional CAD models are shown in Appendix B.

2.1 Boat Type

The first consideration when building this boat was the overall type of sailboat to be built. We elected to design a catamaran, leveraging the advantages provided by the multihull design to simplify design and improve performance. The first advantage with a twin-hulled design was space - a catamaran design gave us ample room for all of our electronics without the issue of overcrowding. Making a catamaran allowed us to place all of the electronics on the deck spanning the hulls (known as the trampoline) between the two hulls, keeping everything dry and above the water line. Second, a catamaran allowed us to design foam-filled pontoon hulls (as opposed to sealed hulls for carrying electronics) without worrying about creating leaks in the hulls. This design ensured that we did not have to spend time ensuring that any penetrations through the hull (i.e. rudders, centerboard) were perfectly sealed, as the foam prevents the boat from sinking. Finally, a catamaran design made for a very stable platform when compared to single hulled boat. The wide base paired with enough ballast from the electronics and batteries provided our boat with sufficient righting moment to prevent capsizing. Due to high winds during testing, a fixed keel was added to the boat. The two 36-inch hulls were created by cutting a positive mold from insulation foam on a Prototrak 3-axis mills and wrapping the cutouts in carbon fiber. Another carbon fiber mat on the top seals the foam in-

side the hull, providing a very tough and buoyant hull.

After initial testing, however, it was discovered that the boat pitched too far forward when under wind load. To solve this issue we fabricated a large third hull, making our boat a trimaran. The third hull provided increased stabilization and buoyancy to keep the bow above the waterline when under large wind loads. This also allowed for additional stabilization of the mast because the third hull provided a location to attach a back stay, holding the mast from pitching forward when sailing down wind.

The trampoline was constructed in a similar manner to the hulls. An 1/8-inch acrylic board was secured to the top of a curved piece of foam, and the resulting sandwich was wrapped in carbon fiber to provide a light deck with a secure mounting point for the electronics and mast. The deck is 18-inches wide, 1/2-inch thick, and approximately 14-inches long. The deck has several pins that fit into the hull's foam body, securing it. Additional carbon fiber strips were attached to smooth out the transition and secure the hull to the decks.

2.2 Sails

The next design consideration was the sail. We elected to build a stiff wing sail for the mainsail and a traditional cloth sail for the jib. Using a wing sail gave us the same (if not improved) performance while avoiding the complexities of making a cloth mainsail by hand. The mainsail was constructed from a series of wooden ribs cut out of lightweight balsa-wood using a laser cutter. The ribs were spaced 6-inches apart and covered in a Monokote®, a material that shrinks when heated and provides excellent rigidity. The symmetrical wing sail was designed following a NACA 0010 formula. The jib is a simple triangular sail sewn from sport nylon, a tough but light cloth. Due to the design of the boat, a bow sprit was added off the front of the deck which holds the fore end of the jib boom in place.

The mast was constructed from 3/8-inch poly stainless steel tubing with a 0.028-inch wall thickness thus providing ample stiffness while remaining lightweight. The mast was fixed to the deck with clamps machined from aluminum and attached to either side of the deck. The mast is held in place with stays attached to a spreader over the sail: one extending to the bow, six going to the hulls just aft of the mast and one to the rear of the center pontoon. The sail has a sleeve that fits over the main mast, giving a low-friction rotation.

Both the main sail and the jib were trimmed and eased using a single servo with double purchase rigging. The rigging was done in such a way to allow the length of sheet to be individually set for both the main sail and jib so that the single control servo could trim them the appropriate amount. Due to the double purchase setup, the sail servo could bring the main sail and jib from fully eased to fully trimmed by rotating through 90°.

2.3 Steering and stabilization

The centerboards and rudders were 3D printed to capture the optimal wing shape of the control surfaces. The centerboard is 2 inches wide and extends 8 inches into the water. They are curved inward, a design that increases lift when the boat heels, decreasing the wetted area of the hull in the water and therefore increasing speed. The rudders are simpler, extending about 6 inches from the bottom of the hull. The rudder shafts pierce the hull just forward of the stern, a de-

sign that simplifies the construction and helps keep the rudders vertical by inserting a plug into the hull to support the shaft. The foam in the hull helps keep the plug in place and vertical; the only downside of this design is that it is necessary to pierce the hulls, which is a non-issue for pontoons.

2.4 Servos

The sail boat is controlled by a pair of servos, one controlling the rudders and one controlling the sails. The rudder servo is recessed into the starboard hull, and moves a transfer arm connected to the starboard rudder. Another arm connects the two rudders, allowing the one servo to control both. In order to choose the correct servo, the maximum torque on the rudders was calculated from the drag equation, $F_d = 1/2 \cdot \rho \cdot v^2 \cdot C_d \cdot A$. ρ is the density of water, 1000 kg/m^2, V is the velocity of the water stream, estimated at 2.8 m/s, and C_d is the drag coefficient, given a worst-case value of 2. Each rudder was estimated as having an in-water area of 2" by 4" (the rudders were eventually made larger, but the conservative nature of the analysis ensured that the servos were still more than adequate to move them), and can rotate 45 degrees off the centerline, giving them a a forward-facing area of 0.000365 m^2. These values give a rudder force of 32 N per rudder. Assuming that the forces face evenly across the entire rudder, the force results in a total torque equivalent to $F \cdot 1" \cdot 2 \cdot \sqrt{2}/2$ (not all the force acts perpendicular to the rudder), which evaluates to 5.5 $kg \cdot cm$ of torque per rudder, or 11 $kg \cdot cm$ for both. The HiTec Hs-5646WP servo was selected for this task, as it could output the torque required, and had the advantage of being waterproof. The waterproofing was crucial, as the hulls sometimes dip below the water on turns.

The sail servo was chosen using a similar method, save with the lift equation in place of the drag equation. The sail turned out to require a similar amount of torque to the rudder, for wind speeds under 15 knots, so the HS-5646WP was also chosen for this task. Waterproofing was less crucial, since the sail servo was mounted on the deck rather than the hulls, but splash-proofing was appreciated, and being able to interchange the servos made maintenance and testing much easier.

2.5 Waterproofing

A crucial design aspect of the electrical systems on the boat was waterproofing. While the GPS and servos could withstand immersion, most of the electronics could not. Thus, a sealed dry box was necessary to protect the electrical system. While thermoformed plastic was explored as an construction method, a plastic storage box proved to be the cheapest and sturdiest solution. Two holed drilled in the side and sealed with rubber sealant and duct tape provided communication with the servos and GPS, while the lid was sealed with weather stripping. The entire enclosure box is mounted to the top of the deck (the original plan of mounting the board under the deck was scrapped when it was realized that the board would extend below the waterline) using industrial Velcro®. The mounting point for the box is just behind the mast, to keep the weight slightly behind the center of the boat, counteracting the tendency of multihull boats to tip forward.

As a backup, we included fast-blow fuses on all of the electronics in the case of a water-short. We also discovered that, despite the waterproof ratings, care should be made to keep the servos

as dry as possible since we did experience leaking into our rudder servo.

2.6 Sensors

The wind sensors are mounted atop the mast, to avoid as much distortion from the surface as possible. The wind speed is measured from a three-cup anemometer about 6-inch wide, designed to be lightweight while still being large enough to capture wind that moves the sail. 3D printing turned out to be the easiest method for creating such a device. The anemometer is mounted on a short 1/4-inch brass shaft, which passes through (and is supported by) a bearing mounted into a plastic plate extending out from the top of the mast. The anemometer is placed above this plate, allowing its arms to pass over the mast, while an encoder sits below, reading the position of the anemometer. The weathervane, which measures wind direction, is constructed in a similar manner save that it hangs below the plate and encoder, to stay out of the way of the anemometer. The vane itself is 3D printed using the same method as the anemometer. The main driving design problem was that of friction reduction. This is accommodated by placing the encoders below the mounting plate, thus eliminating high-friction interaction between the shaft collars and the encoder cases. Said shaft collars also sit on thin metal rings sandwiched between them and the bearings. These ensure that the weight of the shaft collar rests only on the inner, moving ring of the bearing, further reducing friction. The wires for the encoder run down the mast and into the electronics box.

Placement of the GPS sensor was not as intensive. Early testing indicated that the higher the sensor was placed, the better, placement on the stay platform at the top of the mast provided excellent GPS signal.

By combining our knowledge of the apparent wind speed and direction with the GPS measurements of velocity and direction, we were able to back out the true wind direction. Figure 2, below, lays out the geometry needed to calculate the true wind from our sensor measurements of headwind speed and angle and boat speed and angle. We referenced wikipedia for the following equations and diagram but their derivation is fairly straightforward.

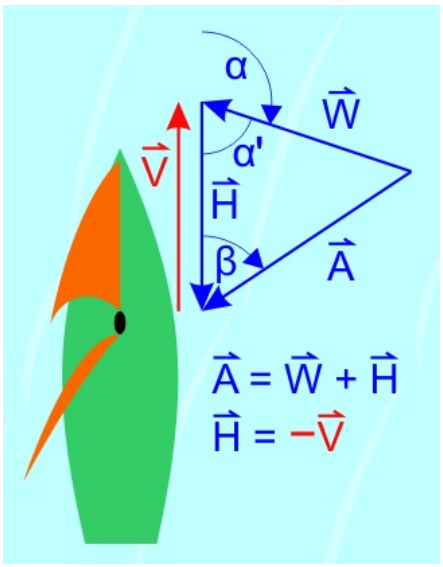

Figure 2: V = boat speed, H = head wind, W = true wind, A = apparent wind, α = pointing angle, β = angle of apparent wind. Source: Wikipedia

$$A = \sqrt{W^2 + V^2 + 2WVcos\,(\alpha)}$$
$$\alpha = arccos\left(\frac{Acos\beta - V}{W}\right) \quad (1)$$

where V = boat speed, H = head wind, W = true wind, A = apparent wind, α = pointing angle, β = angle of apparent wind.

Our implementation of these equations is via a costate and a rolling average of the last 20 measurements. The rolling average is to reduce the high frequency noise caused by small movements in the wind and GPS. This information is transferred back to the user via the radio.

3 Control Design

3.1 Micro-controller

The micro-controller for the project was the Digi BL2600 using a rabbit 3000 microprocessor at 44MHz.The major advantage to this board is the ability to do low level multi-tasking through their costate interface. This proves to be an incredibly useful tool when it comes to maintaining a constant time control loop without blocking other essential calculations.

3.2 Control Implementation

The overall code structure is a set of three while loops: an outer loop encompassing an initialization loop and a main loop, which acts to control most of the sailboat's functions. The initialization loop sets up the encoders, the radio, and the GPS,and then waits for a radio command to begin. The main loop is a series of costates, running pseudo-concurrently, that handle the various jobs needed to run the sailboat. The most important of these is a state setter, which turn the other costates on and off based on the control state. For example, the autonomous control state has no need for costates handling user control, and vice versa. Additional always-on costates handle radio communication and data collection from the sensors (see below), as well as detecting and implementing initialization, stop, and abort commands.

3.3 Sensor Inputs

As indicated above, inputs to the control system come from three sensors: two encoders connected to an anemometer and a weathervane, respectively, and a Garmin GPS sensor. The encoders communicate through a 24-bit quadrature encoder chip (LSI/CSI LS7166) mounted on a separate board below the Digi-Key BL2600. The chip maintains a continuous record on encoder position, accessible via an 8-bit I/O bus. Each time the appropriate costate is called, the microprocessor retrieves the positions and converts the number into degrees off 0 (forward) in the case of the weathervane, or in the case of the anemometer compares the position to that of the previous time-step to find the angular velocity of the anemometer. Using a linear function based on empirical testing, this value is converted to a wind speed in knots.

The GPS data is read in every time there is a message in the buffer, about every 1/5th of a second, and then parses the string to retrieve the location, heading, and speed of the boat. The code assumes that the boat will not cross a GPS degree line, so only the degree minutes (expressed as a floating point number) are stored to save memory. The success or failure of the GPS connection is also stored, both for use inside the program and for sending out over the radio link.

3.4 Control Outputs

To control the servos, we needed to learn Pulse Width Modulation communication. Pulse Width Modulation, PWM, is a common method to digitally control the root mean squared voltage on a load. This is useful in motor controllers and sending data. Remote Control servos use the width on a PWM signal to control the angle at

which the horn should be set. The RC PWM signal is a 50 Hz square wave whose duty cycle controls an 180° range of motion. By setting the pulse width to 0.9, 1.5, and 2.1 ms wide, the servo horn is set at $-90°$, 0°, and 90°, respectively.

The challenge with our servos was in setting the signal frequency at 50 Hz while setting the pulse width with the acceptable resolution. Controlling the servo with 5 degree steps requires resolution of 30 μs, or 30 kHz. This period is shorter than our control loop period (40 μs). Even with built in PWM code, we could not control the servos directly. Instead, we purchased a serial servo controller (Sparkfun ROB-08897) to send commands over RS232. The commands for the board are shown in **??**. The micro-controller can, for instance, send a command of ± 60 deg and the serial board translates it to a pulse width of 1.897 ms, corresponding to the appropriate horn orientation.

To actually power the servo, the power rails all connect to the 7.4V LiPo battery. The control signal from the serial controller board controls a comparator that rails to 7.4V as data from the servo board exceeds 4V.

3.5 Serial Communication

To debug our boat while on the water, we clearly needed a wireless data connection. We decided to use Radio Frequency communication with an XBee Pro (Sparkfun WRL-11216). This module creates easy-to-use multipoint network communication. Using IEEE 802.15.4 standard, these units communicate amongst themselves and can be ready over serial. We used the Xbib-R development board to talk over RS232 to the Bl2600 on the boat, and set up the Xbib-U development board to talk to the computer via USB, as seen in Figure 3. These RF modules claim to have 1

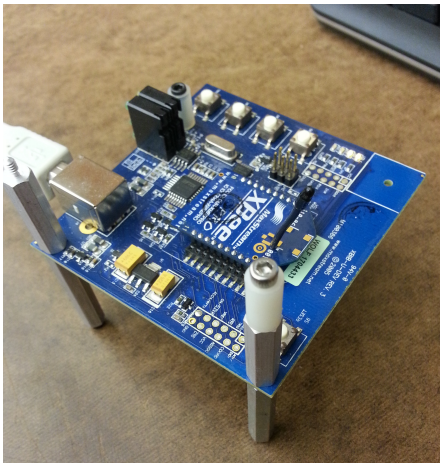

Figure 3: Our wireless communications all use XBee, an RF transciever over which we sent serial commands to read and control the boat.

mile range, but we seem to get closer to 200 or 300 feet. While not optimal, this is fine as we expect to sail on rivers and ponds, not the open ocean. Distance could be improved by purchasing external antennas and attaching them to the boat.

Communicating with the dock computer then amounted to defining a language of commands to send and receive, then building a User Interface to control the boat. The first portion was an easy process of declaring verbs and actions. The XBee would be used to transmit strings of characters delimited by commas. Each command is initiated with a $ followed by a single letter verb and then a variable number of data values. Each field is, as mentioned, delimited by a ',' . Refer to Appendix A for a complete listing of the communication syntax.

To handle the language, Figure 4 shows the GUI and Appendix C shows the code that we designed in MATLAB to run a serial command terminal. This GUI allows actions such as user

control of the boat with arrow keys, establishing new GPS waypoints, reading the status of the boat's servos, gathering wind data, and more.

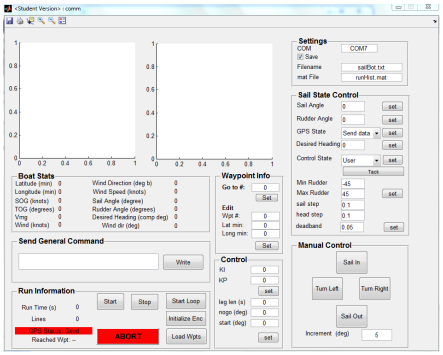

Figure 4: Screen grab of our computer side GUI that allows the user to visualize the state of the sailboat in real time. The longitude and latitude as well as velocity and direction are visualized in the two graphs. Boat state information is displayed below with buttons to adjust the controls and state of the boat located on the right.

3.6 Boat States

The control of the sailboat is broken down according to the mode it is in. The setting of the actions is executed differently for different modes. The control actions, themselves are run off the same action costates. This allows modular development of logical control, with set code for the control actions.

3.6.1 User Mode

For debugging and calibration, we designed a user control mode where the rudder and sail orientation could be incrementally controlled with arrow keys from the dock computer. The state of the rudder and sail angle is changed directly and updated in the next iteration. This makes the entire boat act just as it would were it controlled by a radio control transceiver. User control allowed us to drive the boat and determine the characteristics of the sail and its balancing (**??**). From testing, we found the boat maintained heading best when the sail had an angle of attack of $45^(\circ)$.

When under user control, it can be difficult to run upwind towards a waypoint, as that requires tacking (which is difficult with the low rate of wind data from the boat to the dock computer. To alleviate this problem, an autotack system was implemented. When the boat is ordered to tack, it compares the current bearing to the destination waypoint to the true wind direction. If the waypoint bearing is left of the true wind, the boat moves to tack at 45 degrees left of the true wind (45 degrees being the experimentally-derived angle off the wind that gives the best performance) Empirical testing revealed that the the boat could travel no less than 30 degrees off the wind, and that a bearing 45 degrees off the wind provided excellent boat performance for upwind travel. This control is used as a supplement to user control, and only by direct user command.

3.6.2 Heading Control Mode

As an intermediate goal toward full autonomy, we created a heading control mode where we implemented a PI loop. The boat is told a desired heading and preset gain parameters tune the negative feedback loop. This code is in the *headingControl* costate. The boat reads in its heading and controls until the desired and actual angles match. As expected, the effort drops to zero when the error is zero.

3.6.3 GPS Waypoint Mode

With the ability to drive straight, we next sought to navigate a course. A typical course has marks the boat must sail around. We added in waypoint setting to define the marks on the course, but needed higher level controllers for automation.

3.7 V_{mg} Maximization

When a way point is set, the boat attempts to head straight toward the mark. This works when the boat travels downwind, but not if the heading is into the wind. Upwind the boat must tach and jibe toward the goal. This, however slows the boat's *velocity made good*, or the the speed toward the mark. The fastest path through a course will maximize the v_{mg}.

Figure 2 shows how the true and apparent wind and boat velocity are related. The boat actually heads α off the wind. The best boat would minimize the difference between the optimal alpha and the bearing to the mark. That is, it will want to sail as much toward the mark as possible. This is accomplished autonomously by a costate executing a simple gradient comparison of the current versus previous v_{mg}. If the change is negative, the boat has lost velocity toward the mark. This difference, scaled by the loop time, is an approximate derivative. The heading control is updated by a scaling of this slope.

The v_{mg} is controlled by setting a new heading slightly off the current one. This value is then a command sent to the rudder servo.

3.8 Velocity Maximization

We can additionally maximize the speed of the boat using the other boat actuator: a sail servo. This is a somewhat separate degree of freedom. While the sail force causes heading change just like the rudder, we decided to simplify the problem by handling them separate. That is, we ignored the coupling of sail effort and rudder effort. This allowed us to add a speed maximizer on top of our velocity made good maximizer. It can be thought of as the sail finding the place where it best drives the boat.

The maximizer is another simplified gradient search where the microprocessor compares the current and previous GPS-reported Speed Over Ground. The slope of the boat's change in speed determines how the directive is updated. This has a tuning parameter to protect against excessive changes. When the slope nears a maxima, the sail is no longer adjusted.

In the present code, the maximization directly updates the sail position, but it may be better in the future to separate the calculation and action. This would be easy if we create an additional field in the sailState structure like the in heading process. Separating the maximizer and controller would allow for the sail actions to be independent of the model, if we later decide to fuse the sail and rudder controllers together.

3.9 Waypoint checking

Our implementation of waypoint checking mirrors that of the Yeti code in the Ray research lab. The waypoints are stored in an array of GPS Locations structures. An initial set of waypoints are hardcoded but they can be modified via radio communication.

The boat determines if it should advance to the next waypoint by checking it's distance from the current waypoint at a set rate. If this distance is within a certain radius then our current waypoint is incremented to the next waypoint. Currently

the code is set to continue to loop between the two waypoints that have been set.

4 Troubleshooting

Throughout the construction of the boat we had to troubleshoot several design and control issues in order to make the boat operational. The purpose of this section is to allow future groups to learn from our errors to streamline construction of a new boat.

4.1 Buoyancy and Stability

On the mechanical side the main issue was buoyancy with the catamaran design. With two hulls there was not enough buoyancy force in the bow of the boat given how the hulls taper forward of the mast, causing the boat to pitch forward when sailing under high wind loads. To solve this problem, we added an additional hull 25% larger than the outer hulls, mounted in the middle of the boat. This dramatically increased the stabilization and improved the hydrodynamics of the hulls traveling through the water.

The next big issue we encountered was the boat's tendency to heel too aggressively in high winds, risking capsizing. As with buoyancy, the two small hulls did not provide enough moment to resist tipping when the sail caught the full force of strong wind gusts. The solution was to add a heavy keel below the center hull. As the boat tips, the keel swings out to the side, providing a restoring moment, just as the similar keels on monohull boats do. The keel is a 1kg bullet of tool steel mounted 18" below the center hull on a threaded rod, and in high wind testing performed admirably, preventing the boat from heeling more than 45 degrees, compared to 60 degree heels before the change.

4.2 Encoders

The main sensor problem we encountered was with the encoders. This particular encoder setup (US Digital E4P encoders with a LFLS7083 encoder counter chip) has an unusual behavior when down-stepping from zero ticks: the board will load the value of the preset buffer onto the output buffer, and resume the count from there. Thus, when going down to zero ticks from the positive side, the encoder chip will count to zero, jump to 256 (the value on out preset buffer), and continue down from there. When counting up, the opposite problem happened: this chip would count to 256 and then reset to zero. This posed a significant problem for the wind direction indicator, as the jump amounted to a 60-degree error in wind direction. Loading additional values onto the preset buffer did not solve this problem (indeed, no values could be made to stick in the buffer). The problem was mitigated if not solved by spinning the wind indicator around several times, then determining which value indicated the wind was blowing from directly astern, and using it as an offset for all the counts. This ensured that we would still get accurate data, since the counter would always remain positive or negative, never overcoming the initial spin to reach zero.

4.3 Servos and PWM

As mentioned before, we had issue running the servos. Using the BL2600, we attempted to control the servos at 50 Hz, but were unable to obtain satisfactory resolution as the board can only control duty cycle in terms of integer percentages. At 50 Hz, a 1% change resulted in a $30(\circ)$ change in servo position. Using a higher frequency, 450Hz, provided better resolution, but

burned out the servos, as they were only built to withstand frequencies of 270Hz. In the end, as discussed above, we purchased a PWM driving board and ran the signals through an amplifier to generate the appropriate PWM commands.

4.4 IMU

We also purchased a 9 degrees of freedom IMU (Sparkfun SEN-10736) for further sensor control of our sailboat. Due to setup complications and time constraints we were unable to implement it into our final product. The board should be ready to use and future users should be aware of the different voltage levels of RS-232. The computer and BL2600 serial ports are ± 10 V but the IMU has ± 3.3 V TX and RX pins. We purchased an RS-232 level shifter to solve this issue.

Implementation of this IMU into our sailboat would require daisy-chaining one of our serial ports since we only have access to three. We were also unable to get the IMU properly setup for easy communication and some effort will be needed to get the correct firmware installed.

Figure 5: Group shot after successful day of sailing

Appendices

A Language Definitions

Verbs	Determine the update case, control the action taken
Data	Follow a verb. Data items separated by a delimiter (,)
Syntax	All words separated by a delimiter (,)
	All commands begin with a starter ($)

Verb	Action	Data
A	Abort all	-
B	Reached Waypoint	wayNum
D	set speed control variables	step size, dead band
F	Rudder max/min values	-
R	Set rudder angle	angle
S	Set sail angle	angle
G	Set new GPS state	GPSsentence
E	Edit waypoint	wayNum, latmin, longmin
W	Set Goal	wayNum
C	Set User Control	-
O	Init wind offset	-
K	Set Controller Parameter	kri, krp
H	Set new desiredHeading	desiredHeading
T	Set Tacking parameters	tack length, angle off of wind, no sail angle
Z	Set tack flag	-

Variables	Interpretation
angle	Angle to set (max $\pm$ 90, norm $\pm$ 45)
GPSsentence	refer to GPS.lib
wayNum	Waypoint number
latmin	Waypoint latitude in min
longmin	Waypoint latitude in min
desiredHeading	Heading to set the boat on

B Solidworks Images

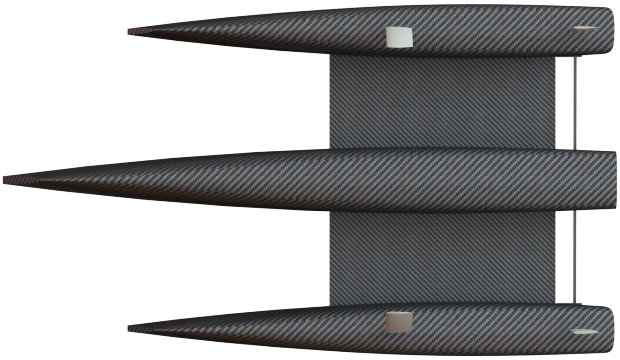

Figure 6: Bottom of boat

Figure 7: Complete boat

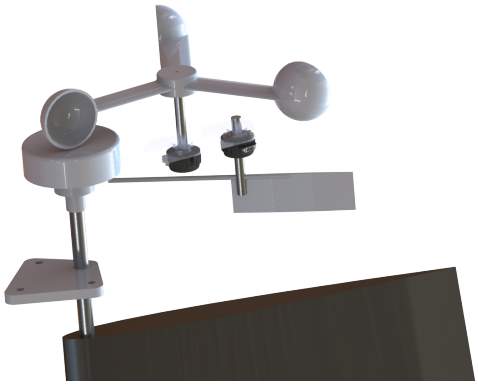

Figure 8: Wind vane and anemometer detail

Figure 9: Detail of hulls, centerboards and rudders

C MATLAB Comm Code

```
function varargout = comm(varargin)
% COMM MATLAB code for comm.fig
%      COMM, by itself, creates a new COMM or raises the existing
%      singleton*.
%
%      H = COMM returns the handle to a new COMM or the handle to
%      the existing singleton*.
%
%      COMM('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in COMM.M with the given input arguments.
%
%      COMM('Property','Value',...) creates a new COMM or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before comm_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to comm_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help comm

% Last Modified by GUIDE v2.5 02-Jun-2013 21:49:33

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @comm_OpeningFcn, ...
                   'gui_OutputFcn',  @comm_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

```

```
% --- Executes just before comm is made visible.
function comm_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to comm (see VARARGIN)

% Choose default command line output for comm
handles.output = hObject;


%global stop;
setappdata(0,'hMainGui',gcf);
hMainGui = getappdata(0,'hMainGui');

%global stopping code for ending the loop
setappdata(hMainGui,'STOP',1);
setappdata(hMainGui,'STOPPING',1);

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes comm wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = comm_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in start.
function start_Callback(hObject, eventdata, handles)
% hObject    handle to start (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)%
%value = get(handles.minX,'String')
    clear locationHist Run_Information

    % load data from figure
    hMainGui = getappdata(0,'hMainGui');
```

```
95      stopVal = getappdata(hMainGui,'STOP');
        stoppedVal = getappdata(hMainGui,'STOPPING');
97

99      %%%% CHECK OUR CURRENT STATE %%%%

101     if( stopVal == 1 && stoppedVal == 1 ),
            % this is the state we want to be in to start. do nothing
103
        elseif( stopVal == 0 && stoppedVal == 0),
105         return; % we are running the code currently
        elseif( stopVal == 0 && stoppedVal == 1),
107         %should never get here
            disp('ERROR');
109         warndlg('Reached Unreachable state!'); %displays warning
            return
111     elseif( stopVal == 1 && stoppedVal == 0),
            % we have requested a stop on the while loop and it is yet to
113         % execute
            return;
115     end

117
        %%%% SAVE DATA CHECK %%%%
119     %check to see if we want to save the data
        saveVal = get(handles.saveCheck,'Value');
121     if( saveVal),
            filename = get(handles.filename,'String');
123         filename2 = get(handles.filename2,'String');
            %filename = './GPS_DATA';
125         fp = fopen(filename,'w');
        end
127

129     %%% COMPORT USER WANTS TO OPEN. THIS IS THE PORT THE XBEE RADIO IS
        %%% CONNECTED TO
131     comport = get(handles.comPort,'String');
        %try to connect to the comport.
133     try
            s = xBeeInit(comport); % open/configure com port
135     catch %#ok<CTCH>
            disp('failed to connect to com port');
137         return
        end
139
        %%% RESET time, lines, GPS status %%%
141     set(handles.runTime,'String',0);
        set(handles.LinesRead,'String',0);
143     set(handles.gpsStatus,'BackgroundColor','red');
```

```
    set(handles.gpsStatus,'String','GPS Status: No Connection');

    %%%% INITIALIZE DATA STRUCTURES %%%%
    MAX_ITERATIONS = 100000;% maximum number of iterations we expect to run
    locationHist = zeros(2,MAX_ITERATIONS); %history
    Run_Information  = zeros(MAX_ITERATIONS,13);
    GPS_Location= struct('timestamp',0,'lat',0,'long',0,'latD','N',...
                                                'longD','W','sog',0,'tog',0);
    setappdata(hMainGui,'GPS_Location',GPS_Location);
    boat_State = struct('rAngle',0,'sAngle',0,'desiredHeading',0,...
                                        'control',0,'windDir',0,'windVel',0,...
                                        'trueDir',0,'trueVel',0);

    setappdata(hMainGui,'boat_State',boat_State);
    setappdata(hMainGui,'Run_Information',Run_Information);



    %%%% LOAD WAYPOINTS %%%%
    %%%%%%%%%%%%%%%%%%%%%%%%
    load('waypoints.mat'); %load the waypoints data

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% INTIALIZATION OF VALUES %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %%% ITERATION COUNTS %%%%
    count = 0;       % number of lines read in since begining of loop
    locationNum = 1; % number of valid GPS lines read
    setappdata(hMainGui,'xBee',s); % save the serial com port

    startTime = clock();
    GPS_data_count = 0;

    %set loop stopping information
    setappdata(hMainGui,'STOPPING',0);
    setappdata(hMainGui,'STOP',0);

    %%% START WHILE LOOP %%%
    while(1),

        %% update time
        secTime = etime(clock(),startTime);
        set(handles.runTime,'String',sprintf('%d',round(secTime)));
        drawnow();

        %%% STOP LOOP CHECK %%%
```

```
193          if( getappdata(hMainGui,'STOP') == 1),
                 break;
195          end

197         %%% read data %%%
            if( s.BytesAvailable == 0),
199            %pause(0.1);
               continue;
201         else
               if( s.BytesAvailable < 3), % wait for more info
203                pause(0.2);
               end
205         end

207         %%% READ IN LINE %%%
            line = fgetl(s); % read in line
209
            disp(line);
211
            % double message check. want to make sure there is only 1 complete
213         % message
            dollarLocations = strfind(line,'$');
215         % more than one message
            if( length(dollarLocations) > 1),
217            % take only the first line
               % NOTE we are throwing away useful information here
219            line = line(dollarLocations(1):dollarLocations(2)-1);
            end
221
            % remove white space
223         line = strtrim(line);


225
            %msg recieved
227         if(~isempty(line)),
                %update count
229             count = count + 1;
                set(handles.LinesRead,'String',sprintf('%d',count));
231
                %cut out the begining of the message:
233
                if(saveVal),
235                  fprintf(fp,line); %save raw message
                     fprintf(fp,'\n');
237             end


239
                % ignore short messages (incomplete)
241             if( length(line) < 3),
```

```
            continue;
        % error in gps
        elseif( strcmp(line(1:2),'$E')),
            %error in the gps connection
            set(handles.gpsStatus,'BackgroundColor','red');
            set(handles.gpsStatus,'String','GPS Status: No Connection');
        elseif( strcmp(line(1:3),'$B')),

            breakInd = strfind(line,',');
            wpt = line(breakInd(1)+1:end);
            set(handles.reachedWpt,'String',sprintf('Reached Wpt: %s',wpt));
            %%% GPS DATA %%%
        elseif( strcmp(line(1:4),'$GPS')),
            %parse string GPS data
            [out, GPS_Location] = parseGPS_custom(line,GPS_Location);
            setappdata(hMainGui,'GPS_Location',GPS_Location); %UPDATE SAVED
                STATE

            %GPS_Location
            if( out ~= 0),
                disp('Error parsing GPS string');
                disp(line);
                disp('END string');
                continue;
            end

            %update GUI
            %lat
            set(handles.lat,'String',num2str(GPS_Location.lat));
            %long
            set(handles.long,'String',num2str(GPS_Location.long));
            %sog
            set(handles.sog,'String',num2str(GPS_Location.sog));
            %tog
            set(handles.tog,'String',num2str(GPS_Location.tog));

            %long-lat plot
            locationHist(:,locationNum) = [GPS_Location.long; GPS_Location.lat
                ];
            axes(handles.longLat);
            %plot the history of all the points

            plot(locationHist(1,1:locationNum), locationHist(2,1:locationNum));
            %plot the waypoints
            hold on;
            scatter(waypoints(:,2),waypoints(:,1)); %plot the waypoints
            xlabel(sprintf('Longitude, %s',GPS_Location.longD));
            ylabel(sprintf('Latitude, %s',GPS_Location.latD));
            hold off;
```

```
289                 locationNum = locationNum + 1;

291                 %plot compass plot
                    axes(handles.sogTog);
293                 rdir = GPS_Location.tog * pi/180; %convert to radians
                    [x,y] = pol2cart(rdir,GPS_Location.sog);
295                 compass(x,y);
                    guidata(hObject, handles);
297                 set(handles.gpsStatus,'BackgroundColor','green');
                    set(handles.gpsStatus,'String','GPS Status: Good');
299                 drawnow();

301                 % save the data to our matrix of information
                    if( saveVal ),
303                      GPS_data_count = GPS_data_count + 1;
                         Run_Information(GPS_data_count,1:5) = [...
305                              GPS_Location.timestamp,...
                                 GPS_Location.long,...
307                              GPS_Location.lat,...
                                 GPS_Location.sog,...
309                              GPS_Location.tog];
                         save(filename2,'Run_Information');
311                      setappdata(hMainGui,'Run_Information',Run_Information);
                    end
313
                elseif(strcmp(line(1:3),'$ST')),
315             %status update
                     [out, boat_State] = parseBoatState(line,boat_State);
317                  set(handles.sA_r,'String',num2str(boat_State.sAngle));
                     set(handles.rA_r,'String',num2str(boat_State.rAngle));
319                  set(handles.desiredHeading,'String',...
                                        num2str(boat_State.desiredHeading));
321                 %wind direction
                    set(handles.wD,'String',num2str(boat_State.windDir));
323                 %wind velocity
                    %boat_State
325                 set(handles.wS,'String',num2str(boat_State.windVel));
                    set(handles.trueVel,'String',num2str(boat_State.trueVel));
327                 set(handles.trueDir,'String',num2str(boat_State.trueDir));
                    drawnow();
329                 setappdata(hMainGui,'boat_State',boat_State);

331                 %save information
                    if( GPS_data_count ~= 0 && saveVal == 1),
333                     Run_Information(GPS_data_count,6:13) = [ ...
                                        boat_State.rAngle,...
335                                     boat_State.sAngle,...
                                        boat_State.desiredHeading,...
337                                     boat_State.windDir,...
```

```
                                    boat_State.windVel,...
339                                 boat_State.trueDir,...
                                    boat_State.trueVel,...
341                                 boat_State.control];
                        save(filename2,'Run_Information');
343                     setappdata(hMainGui,'Run_Information',Run_Information);

345                 end

347             else
                    disp('Unknown data recieved:');
349                 %disp(line);
                end
351


353
            end
355
        end
357     % we have stopped.
        setappdata(hMainGui,'STOPPING',1);
359
        fclose(s); % close serial port
361


363 % --- Executes on button press in stop.
    function stop_Callback(hObject, eventdata, handles)
365 % hObject    handle to stop (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
367 % handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
369 stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
371 setappdata(hMainGui,'STOP',1);
    disp('Stopping Loop');
373


375 % --- Executes on button press in checkbox1.
    function checkbox1_Callback(hObject, eventdata, handles)
377 % hObject    handle to checkbox1 (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
379 % handles    structure with handles and user data (see GUIDATA)

381 % Hint: get(hObject,'Value') returns toggle state of checkbox1


383


385 function sA_s_Callback(hObject, eventdata, handles)
    % hObject    handle to sA_s (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of sA_s as text
%        str2double(get(hObject,'String')) returns contents of sA_s as a double


% --- Executes during object creation, after setting all properties.
function sA_s_CreateFcn(hObject, eventdata, handles)
% hObject    handle to sA_s (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function rA_s_Callback(hObject, eventdata, handles)
% hObject    handle to rA_s (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of rA_s as text
%        str2double(get(hObject,'String')) returns contents of rA_s as a double

% --- Executes during object creation, after setting all properties.
function rA_s_CreateFcn(hObject, eventdata, handles)
% hObject    handle to rA_s (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function bot_state_Callback(hObject, eventdata, handles)
% hObject    handle to bot_state (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of bot_state as text
%        str2double(get(hObject,'String')) returns contents of bot_state as a double


% --- Executes during object creation, after setting all properties.
function bot_state_CreateFcn(hObject, eventdata, handles)
% hObject    handle to bot_state (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in sA_button.
function sA_button_Callback(hObject, eventdata, handles)
% hObject    handle to sA_button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
     if( stopVal ~= 0 || stoppedVal ~= 0)
         return; % do nothing
     end
     sA_s = str2num(get(handles.sA_s,'String'));
     boat_State = getappdata(hMainGui,'boat_State');
     fprintf(s,sprintf('$S,%d',round(sA_s)));


% --- Executes on button press in rA_button.
function rA_button_Callback(hObject, eventdata, handles)
% hObject    handle to rA_button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
     if( stopVal ~= 0 || stoppedVal ~= 0)
         return; % do nothing
```

```
481       end
          rA_s = str2num(get(handles.rA_s,'String'));
483       boat_State = getappdata(hMainGui,'boat_State');
          fprintf(s,sprintf('$R,%d',round(rA_s)));
485

487 % --- Executes on button press in bot_state_button.
    function bot_state_button_Callback(hObject, eventdata, handles)
489 % hObject    handle to bot_state_button (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
491 % handles    structure with handles and user data (see GUIDATA)

493
    % --- Executes on button press in saveCheck.
495 function saveCheck_Callback(hObject, eventdata, handles)
    % hObject    handle to saveCheck (see GCBO)
497 % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
499
    % Hint: get(hObject,'Value') returns toggle state of saveCheck
501


503
    function comPort_Callback(hObject, eventdata, handles)
505 % hObject    handle to comPort (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
507 % handles    structure with handles and user data (see GUIDATA)

509 % Hints: get(hObject,'String') returns contents of comPort as text
    %        str2double(get(hObject,'String')) returns contents of comPort as a double
511

513 % --- Executes during object creation, after setting all properties.
    function comPort_CreateFcn(hObject, eventdata, handles)
515 % hObject    handle to comPort (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
517 % handles    empty - handles not created until after all CreateFcns called

519 % Hint: edit controls usually have a white background on Windows.
    %       See ISPC and COMPUTER.
521 if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
        defaultUicontrolBackgroundColor'))
        set(hObject,'BackgroundColor','white');
523 end

525

527 function filename_Callback(hObject, eventdata, handles)
    % hObject    handle to filename (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of filename as text
%        str2double(get(hObject,'String')) returns contents of filename as a double


% --- Executes during object creation, after setting all properties.
function filename_CreateFcn(hObject, eventdata, handles)
% hObject    handle to filename (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in set_userControl.
function set_userControl_Callback(hObject, eventdata, handles)
% hObject    handle to set_userControl (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end
    control_state = get(handles.controlState,'Value');
    switch control_state
        case 1
            %user control
            fprintf(s,sprintf('$C, 0'));
            %break;
        case 2
            %heading control
            fprintf(s,sprintf('$C,3'));
            %break;
        case 3
            %GPS control
            fprintf(s,sprintf('$C,1'));
        case 4
            %MAX_VELOCITY_MODE
```

```
            fprintf(s,sprintf('$C,4'));
        case 5
            %HEADING_CONTROL_MODE
            fprintf(s,sprintf('$C,5'));
    end
    disp(control_state);
    %sA_s = str2num(get(handles.sA_s,'String'));
    %boat_State = getappdata(hMainGui,'boat_State');
    %fprintf(s,sprintf('$S,%d',round(sA_s)));


% --- Executes on selection change in controlState.
function controlState_Callback(hObject, eventdata, handles)
% hObject    handle to controlState (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns controlState contents as cell array
%        contents{get(hObject,'Value')} returns selected item from controlState


% --- Executes during object creation, after setting all properties.
function controlState_CreateFcn(hObject, eventdata, handles)
% hObject    handle to controlState (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function writeCustomString_Callback(hObject, eventdata, handles)
% hObject    handle to writeCustomString (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of writeCustomString as text
%        str2double(get(hObject,'String')) returns contents of writeCustomString as a double


% --- Executes during object creation, after setting all properties.
function writeCustomString_CreateFcn(hObject, eventdata, handles)
```

```
% hObject    handle to writeCustomString (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in writeString.
function writeString_Callback(hObject, eventdata, handles)
% hObject    handle to writeString (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
hMainGui = getappdata(0,'hMainGui');
s = getappdata(hMainGui,'xBee');
xBeeString = get(handles.writeCustomString,'String');
xBeeString
stopVal = getappdata(hMainGui,'STOP');
stoppedVal = getappdata(hMainGui,'STOPPING');
if( stopVal == 0 && stoppedVal == 0)
    fprintf(s,char(xBeeString));
end



% --- Executes on button press in startLoop.
function startLoop_Callback(hObject, eventdata, handles)
% hObject    handle to startLoop (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
hMainGui = getappdata(0,'hMainGui');
s = getappdata(hMainGui,'xBee');
stopVal = getappdata(hMainGui,'STOP');
stoppedVal = getappdata(hMainGui,'STOPPING');
if( stopVal == 0 && stoppedVal == 0)

    fprintf(s,'1');
    disp('loop started');
end


function dH_s_Callback(hObject, eventdata, handles)
% hObject    handle to dH_s (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of dH_s as text
%        str2double(get(hObject,'String')) returns contents of dH_s as a double


% --- Executes during object creation, after setting all properties.
function dH_s_CreateFcn(hObject, eventdata, handles)
% hObject    handle to dH_s (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in GPS_state.
function GPS_state_Callback(hObject, eventdata, handles)
% hObject    handle to GPS_state (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns GPS_state contents as
    cell array
%        contents{get(hObject,'Value')} returns selected item from GPS_state


% --- Executes during object creation, after setting all properties.
function GPS_state_CreateFcn(hObject, eventdata, handles)
% hObject    handle to GPS_state (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in wind_Encoder.
function wind_Encoder_Callback(hObject, eventdata, handles)
% hObject    handle to wind_Encoder (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)
hMainGui = getappdata(0,'hMainGui');
s = getappdata(hMainGui,'xBee');
stopVal = getappdata(hMainGui,'STOP');
stoppedVal = getappdata(hMainGui,'STOPPING');
if( stopVal == 0 && stoppedVal == 0)
    fprintf(s,'$O');
    disp('Sent Initialization Command');
end


% --- Executes on button press in leftTurn.
function leftTurn_Callback(hObject, eventdata, handles)
% hObject    handle to leftTurn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
disp('leftTurn');
hMainGui = getappdata(0,'hMainGui');
s = getappdata(hMainGui,'xBee');
stopVal = getappdata(hMainGui,'STOP');
stoppedVal = getappdata(hMainGui,'STOPPING');
incAngle = str2num(get(handles.incAmount,'String'));
boat_State = getappdata(hMainGui,'boat_State');
 if( stopVal ~= 0 || stoppedVal ~= 0)
     return; % do nothing
 end

% not in user control
if(boat_State.control ~= 0)
    return
end

fprintf(s,sprintf('$R,%d',boat_State.rAngle-incAngle));
boat_State.rAngle = boat_State.rAngle - incAngle;
setappdata(hMainGui,'boat_State',boat_State);

% --- Executes on button press in rightTurn.
function rightTurn_Callback(hObject, eventdata, handles)
% hObject    handle to rightTurn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
disp('rightTurn');
hMainGui = getappdata(0,'hMainGui');
s = getappdata(hMainGui,'xBee');
stopVal = getappdata(hMainGui,'STOP');
stoppedVal = getappdata(hMainGui,'STOPPING');
boat_State = getappdata(hMainGui,'boat_State');
incAngle = str2num(get(handles.incAmount,'String'));
if( stopVal ~= 0 || stoppedVal ~= 0)
```

```
765       return; % do nothing
    end
767
    % not in user control
769 if(boat_State.control ~= 0)
        return
771 end

773 fprintf(s,sprintf('$R,%d',boat_State.rAngle+incAngle));
    boat_State.rAngle = boat_State.rAngle + incAngle;
775 setappdata(hMainGui,'boat_State',boat_State);


777
    % --- Executes on button press in sailOut.
779 function sailOut_Callback(hObject, eventdata, handles)
    % hObject    handle to sailOut (see GCBO)
781 % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
783 disp('sailOut');
    hMainGui = getappdata(0,'hMainGui');
785 s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
787 stoppedVal = getappdata(hMainGui,'STOPPING');
    boat_State = getappdata(hMainGui,'boat_State');
789 incAngle = str2num(get(handles.incAmount,'String'));
     if( stopVal ~= 0 || stoppedVal ~= 0)
791      return; % do nothing
     end
793
    % not in user control
795 if(boat_State.control ~= 0)
        return
797 end

799 fprintf(s,sprintf('$S,%d',boat_State.sAngle-incAngle));
    boat_State.sAngle = boat_State.sAngle - incAngle;
801 setappdata(hMainGui,'boat_State',boat_State);

803 % --- Executes on button press in sailIn.
    function sailIn_Callback(hObject, eventdata, handles)
805 % hObject    handle to sailIn (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
807 % handles    structure with handles and user data (see GUIDATA)
    disp('sailIn');
809 hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
811 stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
813 boat_State = getappdata(hMainGui,'boat_State');
```

```
incAngle = str2num(get(handles.incAmount,'String'));
if( stopVal ~= 0 || stoppedVal ~= 0)
     return; % do nothing
end

% not in user control
if(boat_State.control ~= 0)
    return
end

fprintf(s,sprintf('$S,%d',boat_State.sAngle+incAngle));
boat_State.sAngle = boat_State.sAngle + incAngle;
setappdata(hMainGui,'boat_State',boat_State);

% --- If Enable == 'on', executes on mouse press in 5 pixel border.
% --- Otherwise, executes on mouse press in 5 pixel border or over sailIn.
function sailIn_ButtonDownFcn(hObject, eventdata, handles)
% hObject    handle to sailIn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)



function KI_Callback(hObject, eventdata, handles)
% hObject    handle to KI (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of KI as text
%        str2double(get(hObject,'String')) returns contents of KI as a double


% --- Executes during object creation, after setting all properties.
function KI_CreateFcn(hObject, eventdata, handles)
% hObject    handle to KI (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function KP_Callback(hObject, eventdata, handles)
% hObject    handle to KP (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of KP as text
%        str2double(get(hObject,'String')) returns contents of KP as a double


% --- Executes during object creation, after setting all properties.
function KP_CreateFcn(hObject, eventdata, handles)
% hObject    handle to KP (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in setControls.
function setControls_Callback(hObject, eventdata, handles)
% hObject    handle to setControls (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end
    Kri = str2num(get(handles.KI,'String'));
    Krp = str2num(get(handles.KP,'String'));
    fprintf(s,sprintf('$K,%f,%f',Kri,Krp));

% --- Executes on key press with focus on sailIn and none of its controls.
function sailIn_KeyPressFcn(hObject, eventdata, handles)
% hObject    handle to sailIn (see GCBO)
% eventdata  structure with the following fields (see UICONTROL)
%       Key: name of the key that was pressed, in lower case
%       Character: character interpretation of the key(s) that was pressed
%       Modifier: name(s) of the modifier key(s) (i.e., control, shift) pressed
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on key press with focus on figure1 and none of its controls.
function figure1_KeyPressFcn(hObject, eventdata, handles)
```

```
% hObject    handle to figure1 (see GCBO)
% eventdata  structure with the following fields (see FIGURE)
%       Key: name of the key that was pressed, in lower case
%       Character: character interpretation of the key(s) that was pressed
%       Modifier: name(s) of the modifier key(s) (i.e., control, shift) pressed
% handles    structure with handles and user data (see GUIDATA)
hMainGui = getappdata(0,'hMainGui');
s = getappdata(hMainGui,'xBee');
stopVal = getappdata(hMainGui,'STOP');
stoppedVal = getappdata(hMainGui,'STOPPING');
if( stopVal ~= 0 || stoppedVal ~= 0)
     return; % do nothing
end
char_val = lower(get(handles.figure1,'currentcharacter'));
disp(get(handles.figure1,'currentcharacter'));

   %execute these instructions
if( char_val == 'w' )
    disp('w');
    sailIn_Callback(hObject, eventdata, handles);
elseif( char_val == 'a' )
    disp('a');
    leftTurn_Callback(hObject, eventdata, handles);
elseif( char_val == 'd' )
    disp('d');
    rightTurn_Callback(hObject, eventdata, handles);
elseif( char_val == 's')
    disp('s')
    sailOut_Callback(hObject, eventdata, handles);
elseif( get(handles.figure1,'currentcharacter') == 'h'),
    disp('h');


end


% --- Executes on button press in GPS_State_Set.
function GPS_State_Set_Callback(hObject, eventdata, handles)
% hObject    handle to GPS_State_Set (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end
    GPS_state = get(handles.GPS_state,'Value');
```

```
959     switch GPS_state
            case 1
961             %send data
                fprintf(s,sprintf('$G, 0'));
963             %break;
            case 2
965             %don't send data
                fprintf(s,sprintf('$G,1'));
967             %break;
        end
969     disp(GPS_state);
        %sA_s = str2num(get(handles.sA_s,'String'));
971     %boat_State = getappdata(hMainGui,'boat_State');
        %fprintf(s,sprintf('$S,%d',round(sA_s)));
973

975 % --- Executes on button press in setHeading.
    function setHeading_Callback(hObject, eventdata, handles)
977 % hObject    handle to setHeading (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
979 % handles    structure with handles and user data (see GUIDATA)
        hMainGui = getappdata(0,'hMainGui');
981     s = getappdata(hMainGui,'xBee');
        stopVal = getappdata(hMainGui,'STOP');
983     stoppedVal = getappdata(hMainGui,'STOPPING');
        if( stopVal ~= 0 || stoppedVal ~= 0)
985         return; % do nothing
        end
987     tempdata=get(handles.dH_s,'String');
        heading = str2num(char(tempdata));
989     fprintf(s,sprintf('$H,%f',heading));
        disp(heading);
991     %sA_s = str2num(get(handles.sA_s,'String'));
        %boat_State = getappdata(hMainGui,'boat_State');
993     %fprintf(s,sprintf('$S,%d',round(sA_s)));

995
    % --- Executes on button press in button_Reset.
997 function button_Reset_Callback(hObject, eventdata, handles)
    % hObject    handle to button_Reset (see GCBO)
999 % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
1001    hMainGui = getappdata(0,'hMainGui');
        s = getappdata(hMainGui,'xBee');
1003    stopVal = getappdata(hMainGui,'STOP');
        stoppedVal = getappdata(hMainGui,'STOPPING');
1005    if( stopVal ~= 0 || stoppedVal ~= 0)
            return; % do nothing
1007    end
```

```
        disp('Resetting to initial state loop');
1009    fprintf(s,'$A');

1011
    % --- Executes on button press in button1.
1013 function button1_Callback(hObject, eventdata, handles)
    % hObject    handle to button1 (see GCBO)
1015 % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
1017


1019
    function incAmount_Callback(hObject, eventdata, handles)
1021 % hObject    handle to incAmount (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
1023 % handles    structure with handles and user data (see GUIDATA)

1025 % Hints: get(hObject,'String') returns contents of incAmount as text
    %        str2double(get(hObject,'String')) returns contents of incAmount as a
        double
1027

1029 % --- Executes during object creation, after setting all properties.
    function incAmount_CreateFcn(hObject, eventdata, handles)
1031 % hObject    handle to incAmount (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
1033 % handles    empty - handles not created until after all CreateFcns called

1035 % Hint: edit controls usually have a white background on Windows.
    %       See ISPC and COMPUTER.
1037 if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
        defaultUicontrolBackgroundColor'))
        set(hObject,'BackgroundColor','white');
1039 end

1041


1043 function goToWay_Callback(hObject, eventdata, handles)
    % hObject    handle to goToWay (see GCBO)
1045 % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
1047
    % Hints: get(hObject,'String') returns contents of goToWay as text
1049 %        str2double(get(hObject,'String')) returns contents of goToWay as a double


1051
    % --- Executes during object creation, after setting all properties.
1053 function goToWay_CreateFcn(hObject, eventdata, handles)
    % hObject    handle to goToWay (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in goToWaypoint.
function goToWaypoint_Callback(hObject, eventdata, handles)
% hObject    handle to goToWaypoint (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end
    tempdata=get(handles.goToWay,'String');
    wayPoint = str2double(tempdata);
    fprintf(s,sprintf('$W,%d',wayPoint));
    disp(wayPoint);


function wayNum_Callback(hObject, eventdata, handles)
% hObject    handle to wayNum (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of wayNum as text
%        str2double(get(hObject,'String')) returns contents of wayNum as a double


% --- Executes during object creation, after setting all properties.
function wayNum_CreateFcn(hObject, eventdata, handles)
% hObject    handle to wayNum (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
```

```
end


function wayLatMin_Callback(hObject, eventdata, handles)
% hObject    handle to wayLatMin (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of wayLatMin as text
%        str2double(get(hObject,'String')) returns contents of wayLatMin as a
    double


% --- Executes during object creation, after setting all properties.
function wayLatMin_CreateFcn(hObject, eventdata, handles)
% hObject    handle to wayLatMin (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function wayLongMin_Callback(hObject, eventdata, handles)
% hObject    handle to wayLongMin (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of wayLongMin as text
%        str2double(get(hObject,'String')) returns contents of wayLongMin as a
    double


% --- Executes during object creation, after setting all properties.
function wayLongMin_CreateFcn(hObject, eventdata, handles)
% hObject    handle to wayLongMin (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
```

```
1147      set(hObject,'BackgroundColor','white');
    end
1149
1151  % --- Executes on button press in editWaypoint.
    function editWaypoint_Callback(hObject, eventdata, handles)
1153  % hObject    handle to editWaypoint (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
1155  % handles    structure with handles and user data (see GUIDATA)
      hMainGui = getappdata(0,'hMainGui');
1157      s = getappdata(hMainGui,'xBee');
      stopVal = getappdata(hMainGui,'STOP');
1159      stoppedVal = getappdata(hMainGui,'STOPPING');
      if( stopVal ~= 0 || stoppedVal ~= 0)
1161          return; % do nothing
      end
1163      wayPoint=str2double(get(handles.goToWay,'String'));
      latMin = str2num(get(handles.wayLatMin,'String'));
1165      longMin = str2num(get(handles.wayLongMin,'String'));
      fprintf(s,sprintf('$E,%d,%f,%f',wayPoint,latMin,longMin));
1167      disp(wayPoint);

1169

1171  function filename2_Callback(hObject, eventdata, handles)
    % hObject    handle to filename2 (see GCBO)
1173  % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
1175
    % Hints: get(hObject,'String') returns contents of filename2 as text
1177  %        str2double(get(hObject,'String')) returns contents of filename2 as a
      double

1179
    % --- Executes during object creation, after setting all properties.
1181  function filename2_CreateFcn(hObject, eventdata, handles)
    % hObject    handle to filename2 (see GCBO)
1183  % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    empty - handles not created until after all CreateFcns called
1185
    % Hint: edit controls usually have a white background on Windows.
1187  %       See ISPC and COMPUTER.
    if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
      defaultUicontrolBackgroundColor'))
1189      set(hObject,'BackgroundColor','white');
    end
1191

1193  % --- Executes on button press in load_Waypoints.
```

```
function load_Waypoints_Callback(hObject, eventdata, handles)
% hObject    handle to load_Waypoints (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end

    %load waypoint data from the computer
    [FileName, PathName, ~] = uigetfile();
    load([PathName,FileName]);

    %send waypoint data to the boat
    %build the message with all the waypoints
    xBeeMsg = sprintf('$F,%d',size(waypoints,1));
    for i = 1:size(waypoints,1),
         xBeeMsg = [xBeeMsg, sprintf(',%d,%f,%f',i,waypoints(i,1),waypoints(i,2))
             ];
    end
    %send the message
    fprintf(s,xBeeMsg);



function legLength_Callback(hObject, eventdata, handles)
% hObject    handle to legLength (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of legLength as text
%        str2double(get(hObject,'String')) returns contents of legLength as a
    double


% --- Executes during object creation, after setting all properties.
function legLength_CreateFcn(hObject, eventdata, handles)
% hObject    handle to legLength (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
```

```
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function noGoZone_Callback(hObject, eventdata, handles)
% hObject    handle to noGoZone (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of noGoZone as text
%        str2double(get(hObject,'String')) returns contents of noGoZone as a
    double


% --- Executes during object creation, after setting all properties.
function noGoZone_CreateFcn(hObject, eventdata, handles)
% hObject    handle to noGoZone (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function anlgeOffDegrees_Callback(hObject, eventdata, handles)
% hObject    handle to anlgeOffDegrees (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of anlgeOffDegrees as text
%        str2double(get(hObject,'String')) returns contents of anlgeOffDegrees as
    a double


% --- Executes during object creation, after setting all properties.
function anlgeOffDegrees_CreateFcn(hObject, eventdata, handles)
% hObject    handle to anlgeOffDegrees (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
```

```
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in setTackInfo.
function setTackInfo_Callback(hObject, eventdata, handles)
% hObject    handle to setTackInfo (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end
    anlgeOffDegrees = str2num(get(handles.anlgeOffDegrees,'String'));
    noGoZone = str2num(get(handles.noGoZone,'String'));
    legLength = str2num(get(handles.legLength,'String'));
    fprintf(s,sprintf('$T,%f,%f,%f',legLength,noGoZone,anlgeOffDegrees));


% --- Executes during object deletion, before destroying properties.
function figure1_DeleteFcn(hObject, eventdata, handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
hMainGui = getappdata(0,'hMainGui');
s = getappdata(hMainGui,'xBee');
if(~isempty(s))
    fclose(s);
end
Run_Information = getappdata(hMainGui,'Run_Information');
filename = getappdata(handles.filename2,'String');
saveVal = get(handles.saveCheck,'Value');
%save a second backup copy  -- this may be redundant.
if( saveVal ),
   save(sprintf('%S_2.mat',filename(1:end-4)),'Run_Information');
end


% --- Executes on button press in start_Tack.
function start_Tack_Callback(hObject, eventdata, handles)
% hObject    handle to start_Tack (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end
    fprintf(s,'$Z');


function MIN_RUDDERS_Callback(hObject, eventdata, handles)
% hObject    handle to MIN_RUDDERS (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of MIN_RUDDERS as text
%        str2double(get(hObject,'String')) returns contents of MIN_RUDDERS as a
    double


% --- Executes during object creation, after setting all properties.
function MIN_RUDDERS_CreateFcn(hObject, eventdata, handles)
% hObject    handle to MIN_RUDDERS (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function MAX_RUDDERS_Callback(hObject, eventdata, handles)
% hObject    handle to MAX_RUDDERS (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of MAX_RUDDERS as text
%        str2double(get(hObject,'String')) returns contents of MAX_RUDDERS as a
    double


% --- Executes during object creation, after setting all properties.
function MAX_RUDDERS_CreateFcn(hObject, eventdata, handles)
```

```
% hObject    handle to MAX_RUDDERS (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in setRudders.
function setRudders_Callback(hObject, eventdata, handles)
% hObject    handle to setRudders (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end
    MIN_RUDDERS = str2num(get(handles.MIN_RUDDERS,'String'));
    MAX_RUDDERS = str2num(get(handles.MAX_RUDDERS,'String'));
    fprintf(s,sprintf('$F,%d,%d',round(MIN_RUDDERS),round(MAX_RUDDERS)));


function SAIL_GRAD_STEP_Callback(hObject, eventdata, handles)
% hObject    handle to SAIL_GRAD_STEP (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of SAIL_GRAD_STEP as text
%        str2double(get(hObject,'String')) returns contents of SAIL_GRAD_STEP as a
    double


% --- Executes during object creation, after setting all properties.
function SAIL_GRAD_STEP_CreateFcn(hObject, eventdata, handles)
% hObject    handle to SAIL_GRAD_STEP (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
```

```
1425 if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
         defaultUicontrolBackgroundColor'))
         set(hObject,'BackgroundColor','white');
1427 end

1429

1431 function HEAD_GRAD_STEP_Callback(hObject, eventdata, handles)
     % hObject    handle to HEAD_GRAD_STEP (see GCBO)
1433 % eventdata  reserved - to be defined in a future version of MATLAB
     % handles    structure with handles and user data (see GUIDATA)
1435
     % Hints: get(hObject,'String') returns contents of HEAD_GRAD_STEP as text
1437 %        str2double(get(hObject,'String')) returns contents of HEAD_GRAD_STEP as a
         double

1439
     % --- Executes during object creation, after setting all properties.
1441 function HEAD_GRAD_STEP_CreateFcn(hObject, eventdata, handles)
     % hObject    handle to HEAD_GRAD_STEP (see GCBO)
1443 % eventdata  reserved - to be defined in a future version of MATLAB
     % handles    empty - handles not created until after all CreateFcns called
1445
     % Hint: edit controls usually have a white background on Windows.
1447 %       See ISPC and COMPUTER.
     if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
         defaultUicontrolBackgroundColor'))
1449     set(hObject,'BackgroundColor','white');
     end
1451


1453
     function sailDeadband_Callback(hObject, eventdata, handles)
1455 % hObject    handle to sailDeadband (see GCBO)
     % eventdata  reserved - to be defined in a future version of MATLAB
1457 % handles    structure with handles and user data (see GUIDATA)

1459 % Hints: get(hObject,'String') returns contents of sailDeadband as text
     %        str2double(get(hObject,'String')) returns contents of sailDeadband as a
         double
1461

1463 % --- Executes during object creation, after setting all properties.
     function sailDeadband_CreateFcn(hObject, eventdata, handles)
1465 % hObject    handle to sailDeadband (see GCBO)
     % eventdata  reserved - to be defined in a future version of MATLAB
1467 % handles    empty - handles not created until after all CreateFcns called

1469 % Hint: edit controls usually have a white background on Windows.
```

```
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'
    defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in setGrads.
function setGrads_Callback(hObject, eventdata, handles)
% hObject    handle to setGrads (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    hMainGui = getappdata(0,'hMainGui');
    s = getappdata(hMainGui,'xBee');
    stopVal = getappdata(hMainGui,'STOP');
    stoppedVal = getappdata(hMainGui,'STOPPING');
    if( stopVal ~= 0 || stoppedVal ~= 0)
        return; % do nothing
    end
    SAIL_GRAD_STEP = str2num(get(handles.SAIL_GRAD_STEP,'String'));
    sailDeadband = str2num(get(handles.sailDeadband,'String'));
    HEAD_GRAD_STEP = str2num(get(handles.HEAD_GRAD_STEP,'String'));
    fprintf(s,sprintf('$D,%f,%f,%f',SAIL_GRAD_STEP,sailDeadband,HEAD_GRAD_STEP));
    fprintf('$D,%f,%f,%f\n',SAIL_GRAD_STEP,sailDeadband,HEAD_GRAD_STEP);

;
% this function parses the GPS as we sent it to the boat. this is an
% abbreviated amount of data compared to the Garming GPS string that is
% recieved by the board
function [out, GPS_Location] = parseGPS_custom(gpsString, GPS_Location)
    if( ~strcmp(gpsString(1:4),'$GPS')),
       out = -1;
       return
    end
    breakInd = strfind(gpsString,',');

    if( length( breakInd ) < 5 )
       disp('Incomplete GPS message recieved');
       out = -1;
       return;
    end

    %time stamp
    GPS_Location.timestamp = str2num(gpsString(breakInd(1)+1:breakInd(2)-2)); %
        extra -1 on the end is because it has a + at the end of the string, unknown
         reason (memcpy in C issue)

%     %link quality
%     if( strcmp(gpsString(breakInd(2)+1:breakInd(3)-1),'V')),
```

```
%         out = -1;
%         return;
%     end

    %latitude minutes
    lat_min = str2num(gpsString(breakInd(2)+1:breakInd(3)-1));
    GPS_Location.lat = lat_min;
%     direction = gpsString(breakInd(4)+1:breakInd(5)-1);
%     GPS_Location.latD = direction;

    %longitude
    lon_min = str2num(gpsString(breakInd(3)+1:breakInd(4)-1)); %#ok<*ST2NM>
    GPS_Location.long = lon_min;

    %SOG
    sog = gpsString(breakInd(4)+1:breakInd(5)-1);
    if( isempty(sog)),
        GPS_Location.sog = 0;
    else
        GPS_Location.sog = str2num(sog);
    end

    %TOG
    %tog = gpsString(breakInd(5)+1:breakInd(6)-1);
    tog = gpsString(breakInd(5)+1:end);
    if( isempty(tog)),
       GPS_Location.tog = 0;
    else
        GPS_Location.tog = str2num(tog);
    end

    out = 0; % successful addition

end

;
% this method returns the boat state information that we have stored on the
% boat. this is parsed and stored accordingly
function [out, boat_State] = parseBoatState(stateString, boat_State)
if( ~strcmp(stateString(1:3),'$ST')),
    out = -1;
    return
end
breakInd = strfind(stateString,',');

if( length(breakInd) < 7),
   disp('Recieved partial state information line');
   out = -1;
   return;
```

```
end

%rudder angle
boat_State.rAngle = str2num(stateString(breakInd(1)+1:breakInd(2)-1));
%sail angle
boat_State.sAngle = str2num(stateString(breakInd(2)+1:breakInd(3)-1));
boat_State.desiredHeading = str2num(stateString(breakInd(3)+1:breakInd(4)-1));
boat_State.windDir = str2num(stateString(breakInd(4)+1:breakInd(5)-1));
boat_State.windVel = str2num(stateString(breakInd(5)+1:breakInd(6)-1));
boat_State.trueDir = str2num(stateString(breakInd(6)+1:breakInd(7)-1));
boat_State.trueVel = str2num(stateString(breakInd(7)+1:end));


out = 0; % successful parsing

;
```

```
function out = xBeeInit(comport)
    % check to see if there are any connections to this com port
    old = instrfind('port',comport);
    if( ~isempty(old))
        fclose(old);
        delete(old);
    end

    out = serial(comport);
    out.Terminator = 'LF';
    %out.Terminator = 'CR';

    %try to open the port
    try
        fopen(out);
    catch
        warndlg(sprintf('Cannot open %s! Try Again!',comport))
        throw(exception);
    end


    % check to see if there are bytes available, if there are clear them
    if ( out.BytesAvailable > 0)

        fscanf(out,'%s',out.BytesAvailable);
    end

;
```

D Microcontroller Code

```
typedef struct {
        int rAngle;             // rudder servo angle
    int sAngle;             // sail servo angle
    int gpsState;           // GPS sending state
    int boatState;          // Boat state values
    int controlState;       // control state of the sailboat
    float Kri;                          // rudder integral gain param
    float Krp;              // rudder proportional gain param
   // int Ksi;                          // sail integral gain param
    float desiredHeading; // desired heading angle
    float windDir;          // Current realtive wind direction   (degrees)
    float windVel;          // Current relative wind velocity     (m/s)
    float trueDir;          // True wind direction
    float trueVel;          // True wind velocity
    int initialize_Wind;    // flag to initialze the wind
    int windSet;            // flag for xBee ack
    int Abort;              // abort flag
    int Goal;                           // Waypoint goal

    float NO_SAIL;          //How far from the wind direction we must
                            //sail to avoid being stuck in irons (degrees)
    float TACK_LENGTH;      //Maximum length between tacks (km)
    float ANG_OFF_WIND;     //How far off the wind we want to sail between
        tacks (degrees)
    int TACK;               // current boat tack
    int tackFlag;


    //maximization info
    float SAIL_GRAD_STEP;//       .1              // stepsize of gradient
        search
    float sailDeadband; //                .05    // zero determination
    float HEAD_GRAD_STEP;//   .1

    //min/max rudder info
    int MIN_RUDDER;// -45
    int MAX_RUDDER;// 45
```

```
} sailState;
// This structure will be used to hold the state of the sailboat as well as any
// control and debug information
typedef struct {
 float velocity; //wind velocity
 long old_anem_pos_st;//old anemometer position
 } velOutput;
//Dynamic C is stupid and doesn't like global variables and pointers

#ifndef PI
#define PI  3.141592654
#endif

//Force functions to be compiled to extended memory.  Helps when the
// program gets large
#memmap xmem


/** DEBUGGING DEFINES     */
// comment out to shut off different debug/settings

//#define _debug_
//#define _debugXbee_    // prints out all xbee writes
//#define _debugWind_    // prints out the true wind speed and directoin calc
#define _debugHeadingControl_    // prints out heading information
//#define _debugWindReadings_  // prints out wind velcity and
                               //direction measurements
//#define _debugStrategy_  // prints strategy selected
//#define _printState_
//#define _wayCheck_
//#define _bearingPrint_
//#define _sailPrint_
#define _statusChange_
//#define _GPS_Debug_
//#define _xBeeDebug_


```

```
#define _send_xBee_ //turns on xbee communication out

/** END DEBUGGING DEFINTES


/*** SET BUFFER SIZES */
#define CINBUFSIZE 511
#define COUTBUFSIZE 511
#define EINBUFSIZE 511
#define EOUTBUFSIZE 511
#define FINBUFSIZE 31
#define FOUTBUFSIZE 31
/*** END BUFFER SIZES */


/*** DEFINE GPS DEFAULTS */

#define GPS_POSITION_LENGTH 28
#define MAX_SENTENCE 200 // max sentence length for GPS as specified
//Define return states
#define GPS_VALID 0
#define GPS_PARSING_ERROR -1
#define GPS_INVALID_MESSAGE 1
//#define GPS_VALID_POSITION 2
//#define GPS_VALID_SATELLITE 3
/*** END GPS DEFINES */

/*** DEFINE GPS SERIAL PORT TO USE */
#define BAUD_GPS 38400
#define serGPSOpen(Baud)                serCopen(Baud)
#define serGPSRead(data,length,tout) serCread(data,length,tout)
#define serGPSputc(data)                serCputc(data)
#define serGPSrdFlush()                                    serCrdFlush()
#define serGPSwrFlush()                                    serCwrFlush()
#define serGPSrdUsed()                                     serCrdUsed()
#define serGPSMode(value)                          serMode(value)
#define serGPSgetc()
   serCgetc();
#define GPS_MODE 0
```

```
/*** DEFINE WAYPOINT ITEMS */
#define numWayPoints                                                    2
#define latD                            43
#define longD
            72
#define SOG
                        0
#define TOG
                        0
#define latN
            'N'
#define longE
            'E'
#define timeWay                                                         080000


/*** DEFINE XBEE SPECFICIC SERIAL PORT TO USE */
#define BAUD_XBEE 9600
#define serXbeeOpen(Baud)                               serEopen(Baud)
#define serXbeeRead(data,length, tout) serEread(data,length,tout)
#define serXbeeputc(data)                                  serEputc(data)
#define serXbeerdFlush()                                      serErdFlush()
#define serXbeewrFlush()                                      serEwrFlush()
#define serXbeerdUsed()                                 serErdUsed()
#define serXbeeMode(value)                 serMode(value)
#define serXbeegetc()                                                   serEgetc()
#define XBEE_MODE 0

/** DEFINE MAXIMIZTION PARAMETERS */
// moved these to the statestruct --- can be set by radio
//#define SAIL_GRAD_STEP        .1                // stepsize of gradient
    search
//#define sailDeadband          .05    // zero determination
//#define HEAD_GRAD_STEP  .1
#define aveNum                  10  // vmg measurements before
   averaging
/** END MAXIMIZATION PARAMETERS */


/** SET SERVO ADDRESS VALUES */
```

```
#define serServoOpen(Baud)
    serFopen(Baud)
#define serServordFlush()                                   serFrdFlush()
#define serServowrFlush()                                   serFwrFlush()
#define serServoMode(value)             serMode(value)

#define RUDDER 6
#define SAIL 7
/*** END SERVO VALUES */          \

/*** SERVO SETTINGS */
// Min and max sail angles
#define MIN_SAIL -40
#define MAX_SAIL 75

// min an max rudder angles
// moved to state struct to be set by radio
//#define MIN_RUDDER -45
//#define MAX_RUDDER 45
/*** END SERVO SETTTINGS */


/*** BEGIN CONTROL MODES */
#define USER_CONTROL_MODE       0
#define GPS_CONTROL_MODE        1
#define ESCAPE_CONTROL_MODE     2
#define HEADING_CONTROL_MODE    3
#define MAX_VELOCITY_MODE       4
#define MAX_HEADING_MODE        5
/*** END CONTROL MODES */

/*** BEGIN BOAT STATES */
#define BROADCAST_STATE 0
#define NO_BROADCAST_STATE 1
/* END BOAT STATES */

/*** GPS STATES */
#define GPS_BROADCAST_STATE 0
#define NO_GPS_BROADCAST_STATE 1
/* END GPS STATES
```

```
185
   /*** START TELEMETRY SETTINGS */
187 #define TELEMETRY_INTERVAL 600 // time to wait between
                                   //each send of GPS, state
189 /* END TELEMETRY SETTINGS */

191 /*** START WIND INFORMATION */
   #define WIND_HISTORY 20
193 /* END WIND INFORMATION */

195 /*** START WAYPOINT INFO */
   #define WAYPOINT_RADIUS 0.5
197 /* END WAYPOINT INFO */

199 #use GPS4.LIB
   #use BL26XX.LIB
201
   /*ENCODER DEFINITIONS*/
203 #define DATA     0x00ff
   #define RD       8
205 #define WR       9
   #define CS1      10
207 #define CS2      11
   #define YX       12
209 #define CD       13
   #define BP_RESET                0X01            // reset byte pointer
211 #define EFLAG_RESET             0X86            // reset E bit of flag
       register
   #define CNT                                     0x01  // access control
        register
213 #define DAT                                     0x00  // access data
       register
   #define BP_RESETB               0X81            // reset byte pointer (
       x and y)
215 #define CLOCK_DATA              2                       // FCK
       frequency divider
   #define CLOCK_SETUP             0X98            // transfer PR0 to PSC
       (x and y)
217 #define INPUT_SETUP             0XC1            // enable inputs A and
       B (x and y)
```

```
#define QUAD_X1                 0XA8    // quadrature multiplier to 1 (x and y)
#define QUAD_X2                 0XB0    // quadrature multiplier to 2 (x and y)
#define QUAD_X4                 0XB8    // quadrature multiplier to 4 (x and y)
#define CNTR_RESET              0X02            // reset counter
#define CNTR_RESETB             0X82            // reset counter (x and y)
#define TRSFRPR_CNTR            0X08    // transfer preset register to counter
#define TRSFRCNTR_OL                    0X90    // transfer CNTR to OL (x and y)
#define XYIDR_SETUP                     0XE1    // x and y index cntrl register
#define XYIOR_SETUP                     0XDB    // x and y i/o cntrl register
#define HI_Z_STATE          0xFF
/*END ENCODER DEFINITIONS*/

/*START STRATEGY DEFINITIONS*/
#define TACKING 1
#define DOWNWIND 0
/*END STRATEGY DEFINITIONS*/

/*START SAILING PARAMATERS*/
#define HEADING_ERROR 10 //accpetable heading error, degrees
#define TACK_SPEED 0.5 //minimum tacking speed, knots
/*END SAILING PARAMATERS*/



// servo serial function prototypes
void writeServo(char cmd, int serv, char data);
void initServo();
void setServoSpeed(int serv, int spd);
int setServoPosition(int serv, int ang, sailState *state);

// generic write to servo function
void writeServo(char cmd, int serv, char data)
```

```
{
251    char buf[5];
       serFputc(0x80); // start com with servo contorller
253    serFputc(0x01); // servo type id
       serFputc(cmd);          // cmd
255
       serFputc(sprintf(buf,"%d",serv));       // servo number
257    serFputc(data);         // put in data
}
259
// initialize servo, open each servo pwm line
261 void initServo()
{
263    int i;
   serFwrFlush();
265    serFrdFlush();
       for (i =0 ; i<8; i++)
267   {writeServo(0, i, 0x2F);     // setup servo one
   }
269   setServoSpeed(RUDDER, 0);
   setServoSpeed(SAIL, 0);
271 }

273 // set the servos to instant response mode
void setServoSpeed(int serv, int spd)
275 {  writeServo(0x01, serv, spd);  }

277 /* set servo position
 takes: servo number (0-7)
279                 ang   -  the angle to go to (-90 to 90, with 0
                       straight ahead)
 gives: retu  - a status bit indicating
281                                        -1 angle set too low
              1  angle set too high
283           0  angle set correctly
*/
285 int setServoPosition(int serv, int ang, sailState *state)
{
287    int temp, retu;
   int minVal, maxVal; // maximum values
```

```
289     char buf;

291     serFputc((char)0x80); // start byte
        serFputc((char)0x01); // device id
293     serFputc((char)0x02); // set position command (7 bit)
        serFputc((char)serv);         // set servo
295
        // determine what min / max val to use
297     if( serv == SAIL ){
              minVal = MIN_SAIL;
299        maxVal = MAX_SAIL;
        }
301     else if( serv == RUDDER ){
              minVal = (int)state->MIN_RUDDER;
303        maxVal = (int)state->MAX_RUDDER;
        }
305     else{
              minVal = (int)state->MIN_RUDDER;
307        maxVal = (int)state->MAX_RUDDER;
           // printf("hi");
309          // should never get here
  #ifdef _debug_
311                   printf("Error, not listed servo commanded\n");

313 #endif
        }
315
              if (ang < minVal)
317                   {
              //temp = (int) 31.75;    // set at -45
319           temp = (int) (63.5*minVal/90+63.5);
              serFputc((char)temp);
321        retu = -1;          // angle set too low
                   }
323           else if (ang > maxVal)
                   {
325        //temp = (int) 31.75;      // set at -45
           temp = (int)(63.5*maxVal/90+63.5);
327        serFputc((char)temp);
           retu = 1; // angle set too high
```

```
329                 }
          else
331                 {
        temp = (int) (63.5*ang/90+63.5);        //
333                 //printf("index = %d\n",temp);
              serFputc((char)temp);
335                 //printf("serial put value: %c\n",(char)temp);
        retu = 0; // all is well
337                 }
    return retu;
339 }


341
 long old_anem_pos;
343 /*** Begin Header EncRead */
int EncRead(int channel, int reg);
345 /*** End Header */
/* START FUNCTION DESCRIPTION
    ******************************************
347 EncRead

349 SYNTAX:         int EncRead(int channel, int reg);
KEYWORDS:       encoder,read
351 DESCRIPTION:    Returns data from the encoder board, depending on input
PARAMETER 1:    channel - an int from 0 to 3 indicating which encoder is
     to be read
353 PARAMATER 2:    reg - an int which is 1 or 0 indicating whether control
    or data is desired
RETURN VALUE:   the requested data, as an int
355
END DESCRIPTION
    ******************************************************/
357 int EncRead(int channel, int reg)
{
359 // channel is an int from 0 to 3 indicating which encoder
// reg is an int which is 1 or 0 indicating whether control or data is
    desired
361     int EncData;
    int i, delayvar;
363     EncData = 0;
```

```
365     digOutConfig(0xff00); // set data lines as inputs and everything
           else as outputs

367     // select which chip
        if (channel <= 1)
369           digOut(CS1,0);
              else
371           digOut(CS2,0);

373      // Select control or data register
         digOut(CD, reg);
375
        // select which channel, X or Y
377     if ((channel == 0) | (channel == 3) )
              digOut(YX,0);
379    else
           digOut(YX,1);
381       // assert Read low
        digOut(RD,0);
383
        EncData = digInBank(0);       // read the data from the data lines
385
    //      for(i = 0; i < 5; i++)
387 //          { delayvar = i; }

389 //deassert read reads the data. Deassert delay to allow rise then
        deselect
        digOut(RD,1);
391     digOut(CS1,1);
        digOut(CS2,1);
393     return EncData;
    }
395 /*** Begin Header EncWrite */
    void EncWrite(int channel, int data, int reg);
397 /*** End Header */
    /* START FUNCTION DESCRIPTION
        ****************************************
399 EncWrite
```

```
401 SYNTAX:         void EncWrite(int channel, int data, int reg);
    KEYWORDS:        encoder,write
403 DESCRIPTION:    puts data onto registers in the encoder board
    PARAMETER 1:    channel - an int from 0 to 3 indicating which encoder is
        to be written to
405 PARAMATER 2:     data - the data to be put on the register
    PARAMATER 2:    reg - an int which is 1 or 0 indicating whether control
       or data register is desired
407 RETURN VALUE: none

409 END DESCRIPTION
       **************************************************/
    void EncWrite(int channel, int data, int reg)
411 {
       int i, delayvar;
413
       digOutConfig(0xffff);// set all digI/O lines as outputs
415
      // select which chip - channel 0 & 1 are chip 1 and channel 2 & 3 are
          chip 2
417    if (channel <= 1)
          digOut(CS1,0);
419    else
            digOut(CS2,0);
421    // select which channel, X or Y  X = 0 and 2, Y = 1 and 3
       digOut(CD,reg);
423
       if ((channel == 0) | (channel == 3) )
425          digOut(YX,0);
      else
427       digOut(YX,1);
           digOutBank((char)0,(char)data);
429
        // assert write
431    digOut(WR,0);

433    // deassert write
       digOut(WR,1);
435
       // deselect chip
```

```
437     digOut(CS1,1);
        digOut(CS2,1);
439     //Set all outputs to 1 so that open collector transistor is off
        digOutBank((char)0,(char)HI_Z_STATE);
441     digOutConfig(0xff00);
    }
443 /*** Begin Header EncInit */
    int encInit(void);
445 /*** End Header */
    /* START FUNCTION DESCRIPTION
        ******************************************
447 encInit

449 SYNTAX:         int encInit(void)
    KEYWORDS:       encoder,init
451 DESCRIPTION:    Sets up the encoder board
    RETURN VALUE:   a failute check, zero means no fault
453
    END DESCRIPTION
        ********************************************************/
455 int encInit(void)
    {
457     int fail;
        int i,j;
459     fail = 0;
            digOutConfig(0xff00);
461     digOut(CS1,1);
        digOut(CS2,1);
463     digOut(RD,1);
        digOut(WR,1);
465
        for (i = 0; i<4; i++)
467     {
    //              EncWrite(i, XYIOR_SETUP,CNT);    // Setup I/O control
        register
469 //              EncWrite(i, XYIDR_SETUP,CNT);            // Disable
        Index
            EncWrite(i,EFLAG_RESET,CNT);
471         EncWrite(i,BP_RESETB,CNT);
            //EncWrite(i,CNTR_RESETB,CNT);
```

```
473           EncWrite(i,CLOCK_DATA,DAT);
              EncWrite(i,CLOCK_SETUP,CNT);
475           EncWrite(i,INPUT_SETUP,CNT);
              EncWrite(i,QUAD_X4,CNT);
477
          EncWrite(i,BP_RESETB,CNT);
479       EncWrite(i,0x12,DAT);
          EncWrite(i,0x34,DAT);
481       EncWrite(i,0x56,DAT);

483       EncWrite(i,TRSFRPR_CNTR,CNT);
          EncWrite(i,TRSFRCNTR_OL,CNT);
485
          EncWrite(i,BP_RESETB,CNT);
487       #ifdef _debug_
                    printf("written = %d, read = %d\n",0x12,EncRead(i,DAT))
                        ;
489       printf("written = %d, read = %d\n",0x34,EncRead(i,DAT));
          printf("written = %d, read = %d\n",0x56,EncRead(i,DAT));
491       #endif
        // Reset the counter now so that starting position is 0
493         EncWrite(i,0x5A0,DAT);
            EncWrite(i,CNTR_RESET,CNT);
495         EncWrite(i,CNTR_RESETB,CNT);

497     }
      return fail;
499 }


501


503 /*** Begin Header getWindPos */
    float getWindPos(long offset);
505 /*** End Header */
     /* START FUNCTION DESCRIPTION
        ****************************************
507 getWindPos

509 SYNTAX:          float getWindPos(long offset);
    KEYWORDS:        encoder,read,vane
```

```
511 DESCRIPTION:    Returns the wind direction using the wind vane position
    PARAMETER 1:    offset - the count treated as straight ahead (using the
        boat
513 Coordinate system)
    RETURN VALUE:   the wind direction, in degrees
515
    END DESCRIPTION
        **************************************************/
517 float getWindPos(long offset){
      int asb, bsb, csb;
519   long position;
      float floatPos  ;
521
       EncWrite(3, TRSFRCNTR_OL, CNT);
523          EncWrite(3,BP_RESETB,CNT);
       asb = EncRead(3,DAT);
525    bsb = EncRead(3,DAT);
       csb = EncRead(3,DAT);
527   //printf(" position: %d, %d, %d, \n",csb, bsb, asb);
             position  = (long)asb;                    // least significant
                byte
529          position += (long)(bsb << 8);
             position += (long)(csb <<16);
531    position=position-offset;
             position= (position % 1440);
533          floatPos=(position * 360.0)/1440.0;

535    if (floatPos < 0 ){ return 360 + floatPos;}

537    return floatPos;
    }
539 /*** Begin Header getWindVel */
    velOutput getWindVel(float dt, long old_anem_pos);
541 /*** End Header */
    /* START FUNCTION DESCRIPTION
        ****************************************
543 getWindVel

545 SYNTAX:          float getWindVel(float dt);
    KEYWORDS:       encoder,read,anemometer
```

```
DESCRIPTION:    Returns the wind velocity using the anemometer position
PARAMETER 1:    dt - the timestep, used in the derivitive calculation
RETURN VALUE:   the wind velocity

END DESCRIPTION
    ************************************************/
velOutput getWindVel(float dt, long old_anem_pos_in){
  int asb, bsb, csb;
  float vel;
  long position;
  velOutput output;
  EncWrite(2, TRSFRCNTR_OL, CNT);
              EncWrite(2,BP_RESETB,CNT);
              asb = EncRead(2,DAT);
              bsb = EncRead(2,DAT);
              csb = EncRead(2,DAT);

//              printf("%d, %d, %d, \n",csb, bsb, asb);
              position  = (long)asb;                        // least
                 significant byte
                                  position += (long)(bsb << 8);
                                  position += (long)(csb <<16);
              //printf("%d, %d \n",position,old);
             vel= ((float)position-(float)old_anem_pos_in)/(dt);
             vel=(0.001*vel-0.0369)*1.944;
              if (vel<0){ vel=-vel;}
              output.velocity=vel;
              output.old_anem_pos_st=position;
              //printf("velocity: %f,dir: %f\n",output.velocity,output.
                 old_anem_pos_st);
              return output;


}
/*** Begin Header getCount */
long getCount(int channel);
/*** End Header */
/* START FUNCTION DESCRIPTION
    ****************************************
getCount
```

```
583
  SYNTAX:          long getCount(int channel);
585 KEYWORDS:       encoder,read
  DESCRIPTION:     Returns the current encoder position in counts
587 PARAMATER 1: channel - an int from 0 to 3 indicating which encoder is
      to be written to
  RETURN VALUE:  The current encoder count
589
  END DESCRIPTION
      **********************************************************/
591 long getCount(int channel){
   int asb, bsb, csb;
593  long position;
      EncWrite(channel, TRSFRCNTR_OL, CNT);
595           EncWrite(channel,BP_RESETB,CNT);
      asb = EncRead(channel,DAT);
597   bsb = EncRead(channel,DAT);
      csb = EncRead(channel,DAT);
599   // printf("%d, %d, %d, \n",csb, bsb, asb);
           position  = (long)asb;                    // least significant
              byte
601        position += (long)(bsb << 8);
           position += (long)(csb <<16);
603   return position;
  }
605
  /*** Begin Header  printGPSPosition */
607 void printGPSPosition(GPSPosition* pos);
  /*** End Header */
609
  /* START FUNCTION DESCRIPTION
      ****************************************
611 printGPSPosition

613 SYNTAX:          void printGPSPosition(GPSPosition* pos);
  KEYWORDS:        gps
615 DESCRIPTION:     Prints out the information contained in the GPSPosition
                                        structure.
617 PARAMETER 1:     pos - a GPSPosition structure that is to be printed out
  RETURN VALUE:    none
```

```
619
   END DESCRIPTION
      **********************************************************/
621 void printGPSPosition(GPSPosition* pos)
   {
623         printf("----------------------\n");
            printf("\nLat degrees: %d\n",pos->lat_degrees);
625      printf("Lat minutes: %f\n",pos->lat_minutes);
         printf("Lat Direction: %c\n",pos->lat_direction);
627         printf("Lon degrees: %d\n",pos->lon_degrees);
            printf("Lon minutes: %f\n",pos->lon_minutes);
629      printf("Lon Direction: %c\n",pos->lon_direction);
         printf("speed over ground: %f\n",pos->sog);
631      printf("heading (deg): %f\n",pos->tog);
         printf("----------------------\n");
633 }


635
   void printSailState(sailState *state)
637 {
            printf("\n---------------\n");
639         printf("Rudder Angle: %d\n", state->rAngle );
      printf("Sail Angle: %d\n",state->sAngle);
641      printf("GPS state: %d\n",state->gpsState);
         printf("---------------\n");
643
   }
645


647 /*** Begin Header writeGPSsTRING */
   void writeGPSString(char* str);
649 /* End Header */

651 /* START FUNCTION DESCRIPTION
      ****************************************
   writeGPSString
653
   SYNTAX:          void writeGPSString(char* str);
655 KEYWORDS:        gps, write
   DESCRIPTION:     Takes the input string, str and sends it out to the gps
```

```
    in
                the correct format
PARAMETER 1:    str - string to be transmitted to the GPS
RETURN VALUE:   none

END DESCRIPTION
    **********************************************************/
 void writeGPSString(char* str)
{
        int len;
        int n;

        len = strlen(str);
        //printf("string length: %d",len);
        //printf("string %s\n",str);
     for(n=0;n<len;n++) {
       serGPSputc(str[n]);
        }
    serGPSputc('*');
    serGPSputc(13);
    serGPSputc(10);
}


/*** Begin Header initGPS */
void initGPS();
/* End Header */

/* START FUNCTION DESCRIPTION
    ******************************************
initGPS

SYNTAX:         void initGPS()
KEYWORDS:       gps, initialize
DESCRIPTION:    Initializes the GPS to the desired settings
RETURN VALUE:   none

END DESCRIPTION
    **********************************************************/
void initGPS()
```

```
{
   serGPSrdFlush();
   serGPSwrFlush();
        writeGPSString("$PGRMO,GPRMC,2"); //Disable all strings
        writeGPSString("$PGRMO,GPRMC,1");  //enable GPRMC string
   writeGPSString("$PGRMC1,1,1,2,,,,,2,W,N,,,,1"); //enable NMEA 0183
      2.3
   writeGPSString("$PGRMI,,,,,,,R");
   writeGPSString("$PGRMO,GPGSA,1"); //get sat info
        printf("using garmin GPS\n");

}

/*** Begin Header writeXBeeString */
void writeXBeeString(char *str);
/*** End Header */


/* START FUNCTION DESCRIPTION
   ******************************************
writeXBeeString

SYNTAX:         void writeXBeeString(char *str)
KEYWORDS:       xbee, write
DESCRIPTION:    Writes the given string to the xbee
PARAMETER 1:    str -- string to be written to the Xbee
RETURN VALUE:   None
END DESCRIPTION
   ********************************************************/
void writeXBeeString(char *str)
{
        int len, i;

   len = strlen(str);
   for(i=0;i<len;i++){
      serXbeeputc(str[i]);
   }
   serXbeeputc((char)10); // add the LF character at the end (message
      term flag)
}
```

```
/*** Begin Header setCostates */
void setCostates(sailState *state);
/*** End Header */

/* Initialize Costate data */
   CoData  setTack, averaging,WaypointCheck, maxHeading, maxVelocity,
      headingControl, UserControl, generate_offset, decide_Strategy,
      check_speed, tacking, downwind; // costate names


/* START FUNCTION DESCRIPTION
   ****************************************
setCostates

SYNTAX:        void setCostates(sailState *state)
KEYWORDS:      setCostates, initialize
DESCRIPTION:   This will set the costates to the correct state (pause/
   resume)
               based on the current control state
PARAMETER 1:   state - sailState struct to update with information

RETURN VALUE: None
END DESCRIPTION
   ****************************************************/
void setCostates(sailState *state)
{
        switch (state->controlState ){
                // only control methods remaining on
                case (USER_CONTROL_MODE):
                CoPause(&maxHeading);                // disable heading
                   maximizer
                CoPause(&maxVelocity);               // disable velocity
                   maximizer
                CoPause(&headingControl);            // disable heading
                   controller
                CoPause(&setTack);
                CoPause(&averaging);
                CoPause(&WaypointCheck);             // no waypoint info
                CoResume(&UserControl);              // enable user
```

```
            control
761
            break;
763         case( GPS_CONTROL_MODE ):
            CoResume(&maxHeading);              // enable heading
                maximizer
765         CoResume(&maxVelocity);             // enable velocity
                maximizer
            CoResume(&headingControl);          // enable heading
                controller
767         CoResume(&averaging);
            CoResume(&setTack);
769         CoResume(&WaypointCheck);
            CoPause(&UserControl);             // disable user control
771         break;
            case( HEADING_CONTROL_MODE):
773         CoResume(&headingControl);          // enable heading
                controller
            CoPause(&maxVelocity);              // enable velocity
                maximizer
775         CoPause(&maxHeading);               // disable heading
                maximizer
            CoPause(&averaging);
777         CoPause(&WaypointCheck);
            CoPause(&UserControl);
779         CoResume(&setTack);  //want to be able to tack
            break;
781         case( MAX_HEADING_MODE ):
            CoResume(&maxHeading);              // enable heading
                maximizer
783         CoPause(&maxVelocity);             // disable velocity
                maximizer
            CoResume(&headingControl);          // enable heading
                controller
785         CoResume(&averaging);
            CoResume(&setTack);
787         CoResume(&WaypointCheck);
            CoResume(&UserControl);            // enable (debug) user
                control
789         break;
```

```
            case( MAX_VELOCITY_MODE):
791             CoPause(&maxHeading);               // enable heading
                    maximizer
                CoResume(&maxVelocity);              // disable velocity
                    maximizer
793             CoResume(&headingControl);           // enable heading
                    controller
                CoResume(&averaging);
795             CoResume(&setTack);
                CoResume(&WaypointCheck);
797             CoResume(&UserControl);             // enable (debug) user
                     control
                break;
799         default:
                CoPause(&maxHeading);                // disable heading
                    maximizer
801             CoPause(&maxVelocity);               // disable velocity
                    maximizer
                CoPause(&headingControl);            // disable heading
                    controller
803             CoPause(&setTack);
                CoPause(&averaging);
805             CoPause(&WaypointCheck);             // no waypoint info
                CoResume(&UserControl);              // enable user
                    control
807             break;
            // IMU case could be added here
809     }

811 }// END setCostates


813
    /*** BEGIN GLOBAL */
815 GPSPosition GPS_Way[numWayPoints];
    /** END */
817
    /*** Begin Header update_Sail_State */
819 int update_Sail_State(sailState *newstate, char *sentence);
    /*** End Header */
821
```

```
/* START FUNCTION DESCRIPTION
   ****************************************
update_Sail_State

SYNTAX:         int update_Sail_State(sailState *newstate, char *sentence
   )
KEYWORDS:       sailState
DESCRIPTION:    Updates the new sail state with the given sentence which
   is
                                   assumed to come from our xBee
                                      communication
PARAMETER 1:    newstate - sailState struct to update with information
PARAMETER 2:    sentence - recieved radio communication string to parse
   out
RETURN VALUE: 0 if the message was parsed correctly
                                   -1 invalid message
                1 right syntax but unable to parse (should never happen)
END DESCRIPTION
   ****************************************************/
int update_Sail_State(sailState *newstate, char *sentence)
{
        auto int i, valid;
    auto int angle;
    auto int item;
    float heading;

    //auto char temp[8];
    valid = 0;
    // at a minimum the sentence should have verb, which is 2 characters
    if( strlen(sentence) < 2) {
        return -1;
      }
    // switch on the types

    switch( sentence[1] )
    {
    case 'A':
                        // abort
        newstate->Abort = 1;
#ifdef _statusChange_
```

```
855             printf("Abort loop\n");
#endif
857         break;
        case 'R':
                                        // rudder action
859     //if( strncmp(sentence, "R",1) == 0){
            sentence = strchr(sentence,',');
861         sentence++;
            angle = atoi(sentence);
863         if( angle > newstate->MAX_RUDDER){
                    angle = newstate->MAX_RUDDER;
865         }
            else if( angle < newstate->MIN_RUDDER){
867                 angle = newstate->MIN_RUDDER;
            }
869
            newstate->rAngle = angle;
871 #ifdef _statusChange_
            printf("rAngle: %d\n",newstate->rAngle);
873 #endif
            break;
875     //}
        // sail angle setting
877     //else if( sentence[0] == 'S'){
        //else if( strncmp(sentence, "S",1) == 0){
879      case 'S':
                                        // sail          action
            sentence = strchr(sentence,',');
881         sentence++;
            //printf("sentence: %s\n",sentence);
883         angle = atoi(sentence);
            if( angle > MAX_SAIL ){
885                 angle = MAX_SAIL;
            }
887         else if( angle < MIN_SAIL){
                    angle = MIN_SAIL;
889         }
            newstate->sAngle = angle;
891 #ifdef _statusChange_
            printf("sAngle: %d\n",newstate->sAngle);
```

```
893 #endif
        break;
895     //}
    case 'G':
                                    // new GPS state
897     //else if( sentence[0] == 'G' ){
    //else if( strncmp(sentence, "G",1) == 0){
899         sentence = strchr(sentence,',');
        sentence++;
901         newstate->gpsState = atoi(sentence);
  #ifdef _statusChange_
903         printf("GPS state: %d\n",newstate->gpsState);
  #endif
905         break;
    case 'E':                                       // edit waypoint
907          // $E,1,7111,2874
       // set waypoint 1
909       // lat minutes at 71.11
       // long minutes at 28.74
911       //printf("got: %s\n",sentence);
         sentence = strchr(sentence,',');
913       sentence++;
       item = atoi(sentence);
915       //printf("item: %d\n",item);
       if (item > numWayPoints){
917       #ifdef _statusChange_
         printf("Not enough Waypoints");
919       #endif
          break;
921          }
         sentence = strchr(sentence,',');
923       sentence++;
       GPS_Way[item].lat_minutes = atof(sentence);
925       //printf("set lat: %f\n",GPS_Way[item].lat_minutes);
         sentence = strchr(sentence,',');
927       sentence++;
       GPS_Way[item].lon_minutes = atof(sentence);
929 #ifdef _statusChange_
                printf("New Waypoint set");
931 #endif
```

```
        break;
933     case 'W':
                                          // identify waypoint
          sentence = strchr(sentence,',');
935       sentence++;
          newstate->Goal = atoi(sentence);                    // set the
             waypoint goal
937       break;

939     case 'C':
                                    // control state action
      //reset the control state
941       sentence = strchr(sentence,',');
          sentence++;
943       newstate->controlState = atoi(sentence);

945       setCostates(newstate);  // update costates

947 #ifdef _statusChange_
          printf("Control State: %d\n",newstate->controlState);
949 #endif
          break;
951
      case 'O':                                                          // wind
          initialize
953       newstate->initialize_Wind = 1;      // set flag
  #ifdef _statusChange_
955       printf("Initialize Wind set \n");
  #endif
957       break;
      case 'H':
                                          // set heading
959         sentence = strchr(sentence,',');
          sentence++;
961       heading=atof(sentence);
          if (heading>360.0) {heading=360.0;}
963       else if (heading<0.0) {heading=0.0;}
          newstate->desiredHeading = heading;
965 #ifdef _statusChange_
          printf("Desired heading set \n");
```

```
967 #endif
                    break;
969     case 'K':                                       // gain param adjust
        //reset the rudder gain
971           sentence = strchr(sentence,',');
          sentence++;
973       newstate->Kri = atof(sentence);

975       sentence = strchr(sentence,',');
          sentence++;
977       newstate->Krp = atof(sentence);

979 #ifdef _statusChange_
          printf("KRI: %f, KRP %f\n",newstate->Kri, newstate->Krp);
981 #endif
                    break;
983     //}

985     case 'T':
          sentence = strchr(sentence,',');
987       sentence++;
          newstate->TACK_LENGTH = atof(sentence);
989       sentence = strchr(sentence,',');
          sentence++;
991       newstate->NO_SAIL = atof(sentence);
          sentence = strchr(sentence,',');
993       sentence++;
          newstate->ANG_OFF_WIND = atof(sentence);
995
          break;
997      case 'Z':
         newstate->tackFlag=1;
999      #ifdef _statusChange_
          printf("tack flag set\n");
1001 #endif
         break;
1003
     case 'D':
1005       sentence = strchr(sentence,',');
          sentence++;
```

```
1007        newstate->SAIL_GRAD_STEP = atof(sentence);
            sentence = strchr(sentence,',');
1009        sentence++;
            newstate->sailDeadband = atof(sentence);
1011        sentence = strchr(sentence,',');
            sentence++;
1013        newstate->HEAD_GRAD_STEP = atof(sentence);
               #ifdef _statusChange_
1015        printf("sail grad step: %f, sail deadband: %f, heading gradient
               step: %f\n", newstate->SAIL_GRAD_STEP, newstate->sailDeadband,
                newstate->HEAD_GRAD_STEP);
#endif
1017        break;

1019    case 'F':
             sentence = strchr(sentence,',');
1021        sentence++;
            newstate->MIN_RUDDER = atoi(sentence);
1023        sentence = strchr(sentence,',');
            sentence++;
1025        newstate->MAX_RUDDER = atoi(sentence);
               #ifdef _statusChange_
1027        printf("min rudder set: %d, max rudder set: %d \n",newstate->
               MIN_RUDDER,newstate->MAX_RUDDER);
#endif
            break;

1031    default:
             valid = 1;
1033        break;
        }
1035

1037    return valid;
}
1039
int parseAddPoints(char *sentence){
1041                        int i, points;
            // $E,1,7111,2874
1043                    // set waypoint 1
```

```
                    // lat minutes at 71.11
1045                // long minutes at 28.74
                            sentence = strchr(sentence,',');
1047                sentence++;
                    points = atoi(sentence);
1049                if (points > numWayPoints){
                            printf("Not enough Waypoints");
1051                return -1;
                                    }
1053            else{
                    for(i = 0; i < points; i++)
1055                {
                    sentence = strchr(sentence,',');
1057                        sentence++;
                            GPS_Way[i].lat_minutes = atof(sentence);
1059                        //printf("set lat: %f\n",GPS_Way[i].lat_minutes
                                );
                                    sentence = strchr(sentence,',');
1061                        sentence++;
                            GPS_Way[i].lon_minutes = atof(sentence);
1063                return 0;
                    }
1065                }
    }
1067

1069 void main()
    {
1071  // variable initialization
      int i,n,g, temp,points;
1073  auto int GPSConnectionStatus; // holds a status of GPS connection
      int sail_Flag;
1075  auto char c;
      int sendMsg;
1077  auto int nIn;
      auto char cOut,ch;
1079  GPSPosition GPS_Location;  // current and destination
      auto char sentence[MAX_SENTENCE];
1081  auto char gpsString[MAX_SENTENCE];
      auto char xBeeString[MAX_SENTENCE];
```

```
1083    auto char xBee_write_buffer[MAX_SENTENCE];
        auto float tempFloat; // temporary float used in true wind
           calculation
1085    sailState state;
        velOutput output;
1087

1089    // maximizing desired heading
        auto int newGPS_FLAG;                                   // denotes gps
           data came in valid
1091    auto int AVE_DONE;                                      // averaging
           done flag
        auto float wayBear, wayDist;                    // bearing, distance to
            waypoint
1093    auto int p;                                             // averaging
           indicies for vmg
        auto float sum_vmg, ave_vmg, pvmg;    // previous vmg
1095    auto float slope;                       // improvement on vmg

1097    // maximizing sail position
        auto float sailSlope;
1099    auto float sum_vel, ave_vel, velLast;         // previous velocity
        auto int q;                     // averaging index for velocity
1101
        //finding true wind direction and position
1103
        auto float histWindD[WIND_HISTORY]; //direction
1105    auto float histWindS[WIND_HISTORY]; //speed
        auto int rollingPointer; //pointer to location of current last wind
           point
1107    auto float windDSum;
        auto float windSSum;
1109
        // heading control variables
1111    auto float ehead;
        auto int uheadm1;
1113    auto int uhead;


1115
        auto int rud_stat;              // status return of setting rudder
```

```
    servo
auto int sail_stat;          // status return of setting the sail
    servo


auto int usedXbeeIn; // number of characters in the Xbee in buffer
auto int usedGPSIn; // number of characters in the GPS in buffer
auto int reachedGoal; // flag to indicate that we have reached our
    goal
auto float currentToGoal; // distance to our current waypoint
//encoder stuff
int N2;
float dt;
float T0;
long offset;
long old_anem_pos;

//Sailing startegy stuff
int tack_possible;
int strategy;
int loop;
GPSPosition tackOrigin;
float bearing;
int goal_reached;



GPS_Location.lat_degrees =  latD;
GPS_Location.lon_degrees = longD;
GPS_Location.lat_direction = latN;
GPS_Location.lon_direction = longE;
GPS_Location.lat_minutes = 42.66947;
GPS_Location.lon_minutes = 17.76415;
//occom waypoint
GPS_Way[0].lat_degrees = latD;
GPS_Way[0].lon_degrees = longD;
GPS_Way[0].lat_direction = latN;
GPS_Way[0].lon_direction = longE;
GPS_Way[0].lat_minutes = 42.72882;
GPS_Way[0].lon_minutes = 17.17637;
```

```
1155    GPS_Way[1].lat_degrees = latD;
        GPS_Way[1].lon_degrees = longD;
1157    GPS_Way[1].lat_direction = latN;
        GPS_Way[1].lon_direction = longE;
1159    GPS_Way[1].lat_minutes = 42.73626;
        GPS_Way[1].lon_minutes = 17.1867;
1161    reachedGoal = 0; // set the goal flag to 0


1163

        /*
1165    // initialize unchanging struct values of waypoints
        for(i = 0; i<numWayPoints; i++)
1167    {
           GPS_Way[i].lat_degrees = latD;
1169       GPS_Way[i].lon_degrees = longD;
           GPS_Way[i].lat_direction = latN;
1171       GPS_Way[i].lon_direction = longE;
           GPS_Way[i].sog = SOG;
1173       GPS_Way[i].tog = TOG;
        } // end wayload for
1175

        // initialize thayer waypoint  g
1177    GPS_Way[0].lat_minutes = 740732;
        GPS_Way[0].lon_minutes = 294146;
1179    */

1181    // initialize program state
        state.rAngle = 0;                               // degrees
1183         state.sAngle = 0;                                  // degrees
        state.gpsState = GPS_BROADCAST_STATE;          // send raw gps data on
            xbee
1185    state.boatState = BROADCAST_STATE;                                  // set
           boat status on xbee
        state.controlState = USER_CONTROL_MODE;       // initially in user
           control
1187    state.initialize_Wind = 0;                     // initially not
           initializing
        state.windSet = 0;                             // flag for xBee ack
           initialy 0
```

```
1189    state.Goal = 0;
                        // initially look at waypoint 0 - THAYER
        state.Abort = 0;                                    // unset abort flag
1191    GPSConnectionStatus = GPS_INVALID_MESSAGE;  // initially no GPS
           contact
        state.controlState = USER_CONTROL_MODE;      // initially in user
           Control
1193    state.desiredHeading=0;                             // Placeholder
        state.windDir=0;
1195    state.windVel=0;
        state.tackFlag=0;
1197    state.SAIL_GRAD_STEP = 0.1; // gradient step amount for sail control
        state.sailDeadband = 0.05; //
1199    state.HEAD_GRAD_STEP = 0.1; // gradient step amount for heading w/
           rudders
        state.MIN_RUDDER = -45;
1201    state.MAX_RUDDER = 45;
        sail_Flag = 0;
1203    newGPS_FLAG = 0;
        AVE_DONE = 0;
1205    windDSum = 0; //initial sum for wind direction is 0
        windSSum = 0; //initial sum for wind speed is 0
1207    rollingPointer = 0; //initial pointer of location is 0
        // zero out history
1209    for(i =0; i< WIND_HISTORY; i++){
              histWindD[i] = 0;
1211       histWindS[i] = 0;

1213    }

1215    state.Kri = 0;
        state.Krp = -.6;
1217


1219    state.NO_SAIL=30.0;
        state.TACK_LENGTH=0.005;
1221    state.ANG_OFF_WIND=45.0;
        //state.trueDir = 0;
1223    //state.trueVel = 0;
```

```
1225    state.trueDir = 120;
        state.trueVel = 4;
1227
        brdInit(); /// initialize the board
1229
        // setup Xbee
1231    serXbeeOpen(BAUD_XBEE);
        serXbeerdFlush();
1233    serXbeewrFlush();
        serXbeeMode(XBEE_MODE);
1235
        // setup GPS
1237    serGPSOpen(BAUD_GPS);
        serGPSrdFlush();
1239    serGPSwrFlush();
        serGPSMode(GPS_MODE);
1241
        //initialize serial data check flags
1243    usedXbeeIn = -1;
        usedGPSIn = -1;
1245    memset(&xBeeString,0x00, sizeof(xBeeString));
        // initialize the GPS
1247    initGPS();

1249    // initialize the servos
        serServoOpen(19200);
1251        serServoMode(0);
        initServo();
1253    uhead = 0;
        uheadm1 = 0;
1255
        //BEGIN Enconder initialization
1257    N2 = 70;
        dt = ((float)N2)/1024;
1259        offset=0;
        encInit();
1261    offset=getCount(3);
        old_anem_pos=getCount(2);
1263    // END Encoder
```

```
1265      //Sailing startegy stuff
      tack_possible=0;
1267    strategy = TACKING;
      loop=1;
1269    bearing=0;
      goal_reached=0;
1271
    tackOrigin.lat_degrees = 0;
1273  tackOrigin.lon_degrees = 0;
    tackOrigin.lat_direction = latN ;
1275  tackOrigin.lon_direction = longE;
    tackOrigin.sog = 0;
1277  tackOrigin.tog = 0;

1279  GPS_Location.lat_degrees =  latD;
      GPS_Location.lon_degrees = longD;
1281    GPS_Location.lat_direction = latN;
      GPS_Location.lon_direction = longE;
1283    GPS_Location.lat_minutes = 42.66947;
      GPS_Location.lon_minutes = 17.76415;
1285    GPS_Location.sog = 0;
      GPS_Location.tog = 0;
1287    //occom waypoint
      GPS_Way[0].lat_degrees = latD;
1289    GPS_Way[0].lon_degrees = longD;
      GPS_Way[0].lat_direction = latN;
1291    GPS_Way[0].lon_direction = longE;
      GPS_Way[0].lat_minutes = 42.72882;
1293    GPS_Way[0].lon_minutes = 17.17637;

1295    GPS_Way[1].lat_degrees = latD;
      GPS_Way[1].lon_degrees = longD;
1297    GPS_Way[1].lat_direction = latN;
      GPS_Way[1].lon_direction = longE;
1299    GPS_Way[1].lat_minutes = 42.73626;
      GPS_Way[1].lon_minutes = 17.1867;
1301
      currentToGoal = gps_ground_distance(&GPS_Way[state.Goal] ,&
          GPS_Location);
1303    printGPSPosition(&GPS_Location);
```

```
// OUTERMOST WHILE LOOP. THIS WILL ONLY LOOP FORWARD FOR A RESET
while(1){

    /*** START LOOP. WILL WAIT FOR GO COMMAND */
    while(1){
        costate initcheck always_on
       {
         waitfor( serXbeerdUsed() > 0);
          DelayMs(20);
          serXbeeRead(xBeeString,2,5);
          //xBeeString[3] = '\0';
           // recieved a 1. This means start the main control loop

#ifdef _debug_

          printf("Xbee Values: %s\n",xBeeString);
#endif
          if(strncmp(xBeeString,"1",1) == 0){

                                  break;
          } // ENDIF
          // loading waypoints
       //else if(&xBeeString[0] == '$' && &xBeeString[1]== 'F' )
          else if( strncmp(xBeeString,"$F", 2) == 0)
          {
                          waitfor(DelayMs(100));
                  // wait a long time for points to come in
                          usedXbeeIn = serXbeerdUsed();
                          serXbeeRead(xBeeString,usedXbeeIn, 0);
                                  //usedXBeeIn++;
                                  xBeeString[usedXbeeIn] = '\0'; // add
                                     terminating string
                  parseAddPoints(xBeeString);
          }
          // initialize wind encoder
          else if(strncmp(xBeeString,"$O",2) == 0)
          {

                  offset = getCount(3);    //generates offset
                  state.initialize_Wind = 0;
```

```
1343            #ifdef _printState_
                        printf("offset= %ld",offset);
1345            #endif
                    writeXBeeString("$S,1");     // send acknowledgements
1347 #ifdef _debug_
                    printf("Wind initialization complete");
1349 #endif
          }
1351          else{

1353            serXbeerdFlush();
          }
1355
          }// end costate
1357          }// end while(1)


1359


1361    usedXbeeIn = -1;
        usedGPSIn = -1;
1363      serGPSrdFlush();
        serGPSwrFlush();
1365 #ifdef _debug_
        printf("Begining Control while loop\n");
1367 #endif

1369    while(1)
        {
1371                    T0 = TICK_TIMER;
       costate stateUpdate init_on
1373       {
          #ifdef _debug_
1375          printf("update state\n");
                    #endif
1377        // update costates
           setCostates(&state);
1379       }
          costate checkSerial always_on
1381       {
            #ifdef _debug_
```

```
1383            printf("check serial\n");
                #endif
1385           usedXbeeIn = serXbeerdUsed();
                usedGPSIn  = serGPSrdUsed();
1387 #ifdef _debug_
            if( usedXbeeIn > 0 || usedGPSIn > 0 )
1389            {
                    printf("usedXbee: %d, usedGPS: %d \n",usedXbeeIn,
                       usedGPSIn);
1391            }

1393 #endif
        } // end costate
1395

1397 #ifdef _send_xBee_
        costate xBeeWr always_on
1399        {
             #ifdef _debug_
1401          printf("xBeeWr \n");
                #endif
1403          // wait for a set amount of time between transmissions
             waitfor(DelayMs(TELEMETRY_INTERVAL));
1405          #ifdef _debug_
             printf("in xbeewr\n");
1407            #endif
             // GPS state 1 means send GPS data
1409          if(state.gpsState == GPS_BROADCAST_STATE &&
                GPSConnectionStatus == GPS_VALID){
                    // string
1411            // $GPS, timestamp, lat minutes, long minutes, sog, tog
                    sprintf(xBee_write_buffer,"$GPS,%s,%f,%f,%f,%f",
1413                                    GPS_Location.timestamp,
                                        GPS_Location.lat_minutes,
1415                                    GPS_Location.lon_minutes,
                                        GPS_Location.sog,
1417                                    GPS_Location.tog);
                    writeXBeeString(xBee_write_buffer);
1419 #ifdef _debugXbee_
                    printf("wrote: %s\n",xBee_write_buffer);
```

```
1421 #endif
            }
1423            // send error message instead of the gps coordinates
            else if(state.gpsState == GPS_BROADCAST_STATE){
1425                sprintf(xBee_write_buffer,"$E,%d",GPSConnectionStatus);
               writeXBeeString(xBee_write_buffer);
1427 #ifdef _debugXbee_
                printf("wrote: %s\n",xBee_write_buffer);
1429 #endif

1431            }// ENDIF GPS error if
            //waitfor(DelayMs(10));
1433            waitfor(DelayMs(TELEMETRY_INTERVAL));
            if( state.boatState == BROADCAST_STATE){
1435                sprintf(xBee_write_buffer,"$ST,%d,%d,%f,%f,%f,%f,%f",
                      state.rAngle,
                                                        state.sAngle,
1437                                                    state.
                                                          desiredHeading,
                                                        state.windDir,
1439                                                    state.windVel,
                                                        state.trueDir,
1441                                                    state.trueVel);
               writeXBeeString(xBee_write_buffer);
1443 #ifdef _debugXbee_
                printf("wrote: %s\n",xBee_write_buffer);
1445 #endif
            }
1447
            if( state.windSet == 1 ){
1449                writeXBeeString("$S,1");
               state.windSet = 0;
1451            CoPause(&generate_offset);     // pause the costate
   #ifdef _debugXbee_
1453                printf("wrote: $S,1\n");
   #endif
1455            }

1457            // reached waypoint
            if( reachedGoal == 1){
```

```
1459                sprintf(xBee_write_buffer,"$B,%d",state.Goal-1);
                    writeXBeeString(xBee_write_buffer);
1461                reachedGoal = 0;
                }
1463
        }// END costate xBeeWr
1465 #endif
        costate xBeeRead always_on
1467    {
            #ifdef _debug_
1469        printf("xbeeRead\n");
                #endif
1471    waitfor(usedXbeeIn > 0);// wait for there to be a message in the
            buffer
        //while(usedXbeeIn<=0) {
1473    //      #ifdef _debug_
        //    printf("usedXbeeIn = %d\n",usedXbeeIn);
1475                //#endif
        //yield;
1477    //}

1479        #ifdef _debug_
            printf("in xbeerd \n");
1481            #endif
            // give the message a chance to finish sending
1483        waitfor(DelayMs(20));
            // read in the entire string
1485        usedXbeeIn = serXbeerdUsed();
            serXbeeRead(xBeeString,usedXbeeIn, 0);
1487        //usedXBeeIn++;
            xBeeString[usedXbeeIn] = '\0'; // add terminating string
1489
#ifdef _xBeeDebug_
1491        printf("Xbee String read in: %s\n",xBeeString);
#endif
1493        // update the state
          update_Sail_State(&state, xBeeString);
1495        if (state.Abort == 1)
            {
1497                        state.Abort = 0; // reset flag
```

```
#ifdef _statusChange_
1499                printf("Abort executed\n");
#endif
1501                break; //exit the loop
          }
1503
          else if (state.initialize_Wind==1)
1505          {
#ifdef _statusChange_
1507                printf("initialize wind\n");
#endif
1509           CoResume(&generate_offset); //start the initialization costate
          }// ENDIF check if wind is to be initialized
1511       }// END costate xBeeRead

1513       costate GPSRead always_on
          {
1515           #ifdef _debug_
           printf("GPS read\n");
1517                #endif
           waitfor(usedGPSIn > 0); // wait for msg in buffer
1519        //     while(usedGPSIn<=0) {
           //    #ifdef _debug_
1521        //  printf("usefdGPSIn = %d\n",usedXbeeIn);
                //#endif
1523      // yield;
         //}
1525         #ifdef _debug_
           printf("in gps read \n");
1527                #endif
           c = 0;
1529           while(1)
           {
1531                 c = serGPSgetc();
               if( c == -1)
1533             {
                  serGPSrdFlush();
1535              abort;
               } // END if
1537            else if ( c == '$')
```

```
                {
1539                    waitfor(DelayMs(5));  // wait for message to come
                    break;
1541                } //END else if
            }// end while(1)
1543
            // now that some time has passed read in the gps string
1545            getgps(gpsString);
#ifdef _GPS_Debug_
1547                            printf("gps: %s\n",gpsString);
#endif
1549                            //update our gpsLocation
            GPSConnectionStatus = gps_get_position(&GPS_Location,gpsString)
                ;
1551            //printf("gps timestamp: %s\n",GPS_Location.timestamp);

1553            if( GPSConnectionStatus == GPS_VALID)
                {
1555 #ifdef _printState_
                    printGPSPosition(&GPS_Location);
1557 #endif
                //printf("flag high\n");
1559                newGPS_FLAG = 1;             // set flag denoting new GPS
                    came in valid
            }// END if
1561


1563        } // END costate GPSRead

1565        costate get_wind always_on
            {
1567         #ifdef _debug_
               printf("get wind\n");
1569                #endif
            //T0=TICK_TIMER;
1571        state.windDir=getWindPos(offset);

1573        output=getWindVel(dt,old_anem_pos);
            old_anem_pos=output.old_anem_pos_st;
1575                    state.windVel=output.velocity;
```

```
#ifdef _debugWindReadings_
1577        printf("Measured WindVel: %f, WindDir: %f\n",state.windVel,state.
               windDir);
#endif
1579


1581      }  // END get_wind costate

1583        costate generate_offset always_on
            {
1585            #ifdef _debug_
               printf("gen offset\n");
1587                  #endif
               waitfor(state.initialize_Wind==1);
1589            offset = getCount(3);    //generates offset
               state.initialize_Wind = 0;
1591            state.windSet = 1; // set flag for xBee ack
#ifdef _debug_
               printf("Wind initialization complete");
#endif
1595
            } // END generate_offset costate
1597
            costate averaging always_on
1599        {
               #ifdef _debug_
1601            printf("averaging\n");
                     #endif
               for (q=0; q<aveNum; q++)
               {
1605            waitfor(newGPS_FLAG == 1);
               //printf("q = %d",q);
1607            sum_vel += GPS_Location.sog;    // sum aveNum speeds then
                  average
               wayBear = gps_bearing2(&GPS_Way[state.Goal], &GPS_Location);
                     // bearing to waypoint
1609            sum_vmg += GPS_Location.sog*cos(rad(wayBear));  // vmg toward
                  waypoint
               newGPS_FLAG = 0;
1611            yield;
```

```
            }
1613        AVE_DONE = 1;
        } // end averaging costate
1615

1617    costate maxHeading always_on
        {
1619     #ifdef _debug_
            printf("maxHeading\n");
1621                #endif
         waitfor(AVE_DONE == 1);
1623                ave_vmg = sum_vmg/aveNum;
                    slope = (ave_vmg - pvmg)*1024/N2;     // improvement on
                        vmg over loop time
1625        // take a step in appropriate heading change
            //state.desiredHeading = (1 + HEAD_GRAD_STEP*slope)*state.
               desiredHeading;
1627        state.desiredHeading = state.desiredHeading + state.
               HEAD_GRAD_STEP*slope;
            pvmg = ave_vmg;
1629 #ifdef _bearingPrint_
            printf("bearing: %f, vmg: %f, pvmg: %f, slope: %f, desired
               Heading: %f\n",wayBear,ave_vmg,pvmg,slope,state.
               desiredHeading);
1631 #endif
            AVE_DONE = 0;
1633
        }// END costate maxHeading
1635


1637

         costate maxVelocity always_on
1639    {
            waitfor(AVE_DONE == 1);
1641        ave_vel = sum_vel/aveNum;
                        sailSlope = ave_vel - velLast;
1643      if (sailSlope > -state.sailDeadband && sailSlope < state.
             sailDeadband)
              {}  // if within band of zero, do nothing
1645      else
```

```
                {
1647            state.sAngle = (int) (state.sAngle + sailSlope*state.
                   SAIL_GRAD_STEP);
                sail_stat = setServoPosition(SAIL,state.sAngle,&state);
1649            }
          velLast = ave_vel;
1651                        AVE_DONE = 0;
#ifdef _sailPrint_
1653              printf("ave_vel: %f, velLast %f, sailSlope %f\n",
                     ave_vel,velLast,sailSlope);
#endif
1655      }// END costate maxVelocity

1657      costate headingControl always_on
          {
1659          #ifdef _debug_
            printf("heading control\n");
1661              #endif
              ehead = state.desiredHeading - GPS_Location.tog;
1663        //ehead=state.desiredHeading-state.windDir;
          if(ehead<-180.0)
1665        {
          ehead=(360+ehead);
1667        }
          else if (ehead>180.0)  {
1669        ehead=-(360-ehead);
          }
1671         // u/e = k1/(z-1)
           // uk = ukm1*ki - ek*kp
1673       uhead = (int)(state.Kri*uheadm1 - ehead*state.Krp);
           uheadm1 = uhead;
1675       state.rAngle = uhead; // keep track of rudder angle
           rud_stat = setServoPosition(RUDDER, uhead, &state);
1677 #ifdef _debugHeadingControl_
          //printf("des: %f , tog %f, ehead: %f, uheadm1: %d, rud_stat:
             %d\n",state.desiredHeading,state.windDir,ehead,uhead,
             rud_stat);
1679        printf("des: %f , tog %f, ehead: %f, KRP: %f ,uhead: %d,
             rud_stat: %d\n",state.desiredHeading,GPS_Location.tog,ehead
             ,state.Krp,uhead,rud_stat);
```

```
#endif

        //printf("headingcontrol\n");

    }// END costate headingControl

    costate WaypointCheck always_on
    {
       #ifdef _debug_
        printf("waypoint check\n");
              #endif
               waitfor(DelayMs(1000)); // only run once a second
        //printf("waypointCheck\n");
        if( currentToGoal < WAYPOINT_RADIUS ){
               reachedGoal = 1; // set notification flag
               state.Goal ++;   // iterate to next waypoint
           //check to see if we have reached the end
           if( state.Goal > numWayPoints){
               state.Goal = 0; //return to begining
           }
        }
        // we have not reached our goal find our new distance
        else
        {
               currentToGoal = gps_ground_distance(&GPS_Way[state.Goal
                  ] ,&GPS_Location);
        }

    }// END WaypointCheck

    costate UserControl always_on
    {
        #ifdef _debug_
        printf("userControl\n");
              #endif
      //       waitfor(DelayMs(100));
     rud_stat = setServoPosition(RUDDER,state.rAngle, &state);
     sail_stat = setServoPosition(SAIL,state.sAngle, &state);
#ifdef _printState_
```

```
1719        printSailState(&state);
            //printf("user status \tRUDDER: %d\tSAIL: %d\n",rud_stat,
               sail_stat);
1721 #endif

1723        }// END UserControl

1725    /*
            costate check_speed always_on
1727        {
                #ifdef _debug_
1729            printf("check_speed\n");
                      #endif
1731            waitfor(newGPS_FLAG==1 && GPSConnectionStatus == GPS_VALID);
                  //Wait for a new message, then re-evaluate

1733            //If the boat is moving faster than TACK_SPEED, set the
                  tack_possible flag to true
                if (GPS_Location.sog>TACK_SPEED){
1735                    tack_possible=1;
                }
1737            else
                {
1739                     tack_possible=0;
                }
1741
            } // end check_speed costate
            /*-------------------------------------------------------------------*/


1745
           costate setTack always_on
1747       {
            #ifdef _debug_
1749            printf("set tack\n");
                      #endif
1751       #ifdef _debugWind_
            printf("end tack, waiting for tack flag \n",bearing);
1753        #endif
             waitfor(state.tackFlag==1);
1755         state.tackFlag=0;
```

```
                    bearing=gps_bearing2(&GPS_Location,&GPS_Way[state.
                       Goal]);      // set the bearing to waypoint
1757            #ifdef _debugWind_
          printf("bearing is %f\n",bearing);
1759          #endif
              if (bearing>state.trueDir || bearing<(state.trueDir-180))
1761          {//head right of wind
                                    state.desiredHeading=state.trueDir+
                                       state.ANG_OFF_WIND;
1763
                              }
1765                          else
                              { //head left of wind
1767                                  state.desiredHeading=state.trueDir-
                                       state.ANG_OFF_WIND;
                              }
1769          if (state.desiredHeading>360)
              {
1771              state.desiredHeading=state.desiredHeading-360;
              }
1773        #ifdef _debugWind_
           printf("begin tack to %f\n",state.desiredHeading);
1775        #endif
           waitfor(ehead<HEADING_ERROR);
1777



1779



1781        }
           costate windCalc always_on
1783       {
               #ifdef _debug_
1785            printf("wind calc\n");
                     #endif
1787          //state.trueDir=0;
              //state.trueVel=0;
1789          //printGPSPosition(&GPS_Location);
              windDSum -= histWindD[rollingPointer];
1791          //calculate true wind
```

```
1793
            windSSum -=  histWindS[rollingPointer];
1795        //calculate true wind speed
            //tempFloat = pow(state.windVel,2.0);
1797
            //windSSum += sqrt(pow(state.windVel,2.0)+ pow(GPS_Location.sog
                ,2.0) + 2.0*state.windVel*GPS_Location.sog);
1799        histWindS[rollingPointer] = sqrt(pow(state.windVel,2.0)+ pow(
                GPS_Location.sog,2.0) + 2.0*state.windVel*GPS_Location.sog*
                cos(rad(state.windDir)));//insert true wind calc

1801        windSSum += histWindS[rollingPointer];
            state.trueVel = windSSum/WIND_HISTORY;
1803

1805        tempFloat = (state.windVel*cos(rad(state.windDir))-GPS_Location
                .sog)/(state.trueVel);
            if( tempFloat > 1 || tempFloat < 1){
1807 #ifdef _debugWind_
                                    printf("Bad wind calculation\n");
1809            printGPSPosition(&GPS_Location);
    #endif
1811                                histWindD[rollingPointer] = state.
                                        windDir;
            windDSum += histWindD[rollingPointer]; // just to not have
                everything blow up
1813        }
            else
1815        {
                        histWindD[rollingPointer] = GPS_Location.tog +
                            deg(acos(tempFloat));
1817                    windDSum += histWindD[rollingPointer];
            }
1819        state.trueDir = windDSum/WIND_HISTORY;

1821
    #ifdef _debugWind_
1823        printf("trueVel: %f, trueDir: %f\n",state.trueDir,state.trueVel
                );
    #endif
```

```
1825                    rollingPointer++;
              if( rollingPointer == WIND_HISTORY ){
1827                    rollingPointer = 0;
            }
1829
        }
1831

1833          //set loop length
#ifdef _debug_
1835        if( (TICK_TIMER - T0) > N2)
                    {
1837                    printf("overrun sampling time\n");
            }
1839 #endif

1841                    while((TICK_TIMER-T0)<=N2){}
    }// END costate while(1)
1843    }// END far outside while(1) loop
    return;
1845 }// END main loop
```