Team EE80T : The Crystal Ship

Lab Report

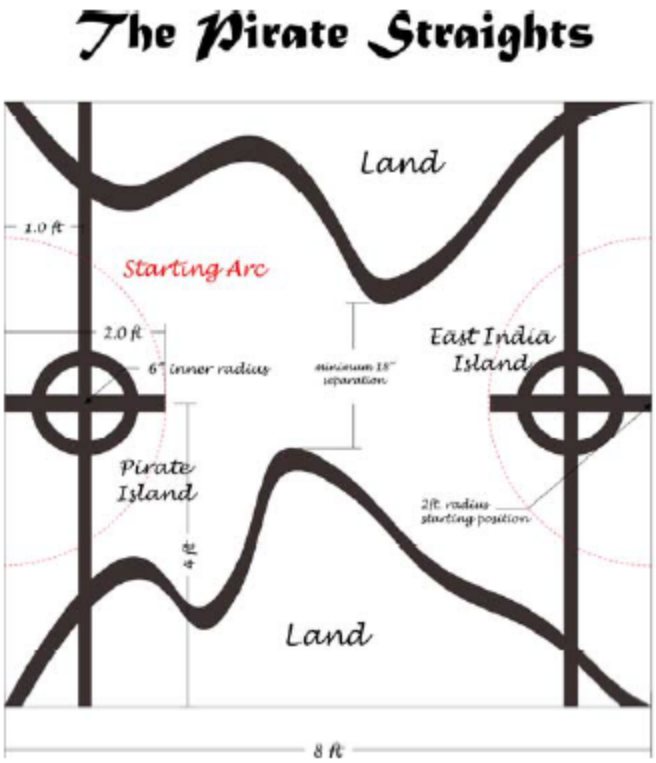
CMPE 118/L

March 19, 2012

Sina Kahnemouyi
Randy Chen
Hart Wanetick

Introduction:

This project involves three simple tasks: get from one island to the other and back, shoot two ping-pong balls at the enemy bot, and resolve collisions. The setting is an 8ft x 8ft platform with the two 6” radius islands and coastlines marked out by black Pro-Tec PVC tape. See image below for a more detailed version:



Other than that there are some other criteria. First, the solution must be contained in an 11” x 11” x 11” cube upon initiation. Second, it must have an open (not blocked by anything) beacon mount exactly 11” off the ground. Third, it must be designed under a budget of $150. The solution should be designed to start at one of the dotted arcs on the map above in a random orientation. The bot cannot enter into any of the regions marked as “Land.” The coastlines are going to be randomly generated, so they will not look like the one shown in the map above. There is a guaranteed minimum of an 18” separation between the coastlines.

List Of Materials:

The following is a list of materials that we used to design our solution:

Part No.	Part Name	Quantity
1	20” x 32” Sheet of Acrylic ⅛” thick	1
2	CMPE118 UNO32 Board	1
3	CMPE118 Dual H-Bridge Driver	2
4	CMPE118 ULN2003A Board	1
5	CII/A1464 - 12 VDC Pull-Type Solenoid w/ Compression Spring	1
6	12VDC, 65 RPM Gearhead Motor	5
7	JD 550 18V JIN DING Motor from a Power Drill	1
8	TCRT5000 Sensor	6
9	Generic Roller Blade Wheel	4
10	Multipurpose PC Board with 417 Holes (preferably with power rails) http://www.radioshack.com/product/index.jsp?productId=2102845	1
11	A Pack of Std Value Resistors and Capacitors	1
12	LM324 Op Amp	1
13	LM339 Comparator	1
14	TIP122 Darlington Transistor	1
15	LTE-301 Light Sensor	1
16	Highly Snap-Action Switch (VT16061C2)	2
17	LP2950 3.3V Regulator	1
18	L7805CV 5.0V Regulator	1
19	4” Lazy Susan Bearing	1
20	Three-Position Switch	1
21	⅛” Heat-Shrink Tubing	31

CMPE118 Uno32 I/O Board Connections Table:

Protection Pin #	Designated use	Notes
PORTV - 3 (ADC)	Tape Sensor 1 (tape[0])	
PORTV - 4 (ADC)	Tape Sensor 2 (tape[1])	
PORTV - 5 (ADC)	Tape Sensor 3 (tape[2])	
PORTV - 6 (ADC)	Tape Sensor 4 (tape[3])	
PORTV - 7 (ADC)	Tape Sensor 5 (tape[4])	
PORTV - 8 (ADC)	Tape Sensor 6 (tape[5])	
PORTW - 3		
PORTW - 4 (ADC)	Beacon Sensor	
PORTW - 5		
PORTW - 6		
PORTW - 7		
PORTW - 8		
PORTX - 3	Tape Sensors LED side	
PORTX - 4	Tape Sensors LED side	
PORTX - 5	Tape Sensors LED side	
PORTX - 6	Tape Sensors LED side	
PORTX - 7	Tape Sensors LED side	
PORTX - 8	Tape Sensors LED side	
PORTX - 9		
PORTX - 10		
PORTX - 11		
PORTX - 12		
PORTY - 3	Right Side Wheels Direction	Forward is a logic high - Reverse is a logic low
PORTY - 4 (PWM)	Right Side Wheels Enable	
PORTY - 5	Solenoid	Logic High Pulls - Logic Low Releases
PORTY - 6		
PORTY - 7		
PORTY - 8		
PORTY - 9	Turret Direction	Counter-Clockwise is a logic high - Clockwise is a logic low
PORTY - 10 (PWM)	Turret Enable	
PORTY - 11	Left Side Wheels Direction	
PORTY - 12 (PWM)	Left Side Wheels Enable	

PORTZ - 3		
PORTZ - 4		
PORTZ - 5	Launcher Wheel Direction	Set to high for the desired direction
PORTZ - 6 (PWM)	Launcher Wheel Enable	Set at 65% PWM
PORTZ - 7		
PORTZ - 8		
PORTZ - 9		
PORTZ - 10		
PORTZ - 11	Bumper Sensors	Nose bumper and both side bumpers were hooked up to the same input
PORTZ - 12		

Design Methods:

The general design to be implemented was a four-wheel driven box with another rotating turret box on top of it, with a protruding tube to be used as a gun-barrel for ping-pong balls. This design was made using five identical DC motors with gear-heads; this was to provide torque for both driving the bot and rotating the turret. We also used a larger DC motor that was faster, but with less torque, to launch the balls, as well as a pulling solenoid with a compression spring to shoot balls into the launching wheel driven by this motor. By making to-scale drawings of these actuators in the program SolidWorks 2011, we were able to construct a 3D scale-model of our robot, in which the actuators were mounted.

All parts designed in the model assembly, excluding the actuators, launching tube, wheels and swivel bearing, we made a universal thickness set with a variable. This gave us the flexibility of using different prototyping materials which could all be cut out of the same sheet in a laser cutter. The ultimate robot was constructed this way using a 0.114” clear acrylic sheet. This material had many advantages including remarkable visibility and the ability to weld pieces together using acrylic cement, which created very strong joints. To assembly a body for the robot, we constructed rectangular boxes for both the base and turret using six faces with locking tab and slot

edges. In order to fit within the specified dimensions of an 11” cube, we designed the turret box to be 6.85” H x 5” W x 6.5” L and the base box to be 3” H x 8” W x 7” L, with length meaning from the front to the back of the robot. The bottom plate of the base we made with a triangular extension that stuck out to a point. This gave the base box a triangular “cow-catcher” to which two triangular nose-pieces were attached that gave the nose a sloping, pyramid shape for deflecting enemy robots.

The sides of the base box were given three screw-holes at each of the corners for a motor to be mounted inside the box, with screws from the outside, and a larger hole for the motor shaft to stick perpendicularly out of the box. Also the top plate of the base box had screw-holes cut into it for attaching the “lazy susan” swivel bearing, as well as a larger hole in the center of the plate for passing wires through, with small holes on either side of it for attaching washers and a gear for the turret to rotate around. The back face for the base had a larger square hole cut into it for accessing the microcontroller as well as two screw-holes for mounting an H-bridge board on the inside. The bottom plate of the base had six small, rectangular holes for mounting the TCRT5000 tape detectors, with three in a triangular configuration near the tip of the nose and three in an in-line configuration further back. The bottom plate on the base was also given a hole near the nose for bumper-sensor wires to pass through. The driving wheels we used were 2.72” in diameter rubber rollerblade wheels, with a 0.72” inner diameter. To mount these to the shafts of the driving motors, we designed a disk part that fit inside the inner circle of the wheels, with a hole in the middle, that the motor shafts locked into, on the outside of the base box. To ensure that the wheels would stay on, we used two of these disks on each motor, both of which fit inside the wheel.

The turret box was given a hole in the bottom large enough for the base gear to pass through, or 1.66” in in diameter. The turret mechanism was designed to have a gear which rotated the turret itself around a larger, hollow gear connected to the base. The purpose for this was to allow the turret to rotate at least 360 degrees without the wires to the base getting in the way. It also slowed down the turning speed of the turret and increased the torque, given the 42:12 gear ratio. The motor which enabled this turning was mounted, shaft-down, to the left side wall and bottom of the turret. This was achieved using one piece tabbed into the left wall, which itself was mounted on two stands which were tabbed into the bottom of the turret. The turret motor was then

fastened to this mount piece with screws. Attached to the shaft of this motor was the 12-tooth small gear which rested on the bottom. The 42-tooth gear on the top of the base box was elevated using three washers with the same sized inner hole for the wires to pass through. The turret motor was powered by an H-bridge module, which was fastened to the left side wall of the turret. The control signals sent to this module came from the micro-controller in the base box.

The pipe used for the gun-barrel had a 2” outer diameter and a 1.75” inner diameter for the 1.5” diameter ping-pong balls to fit in. The solenoid was encased in the tube using two disks which fit the inner diameter of the tube and the outer diameter of the solenoid. One of the disks needed to be cut in half to be acrylic-welded around the solenoid and lock it in place. Also, two slots were cut into the pipe, just ahead of where the solenoid was mounted, one for balls to drop into, and one for the launcher wheel to be lowered into in order to make contact with and launch the balls. This gun-barrel device was then mounted in the turret box using a 2” circular cut in the front face of the box, up against the right side of the box. After fitting through this cut, the back of the barrel then rested on the bottom of the turret. The slot in the pipe for the balls to drop in was on the inside of the box and the slot for the launcher wheel was on the outside of the box since the launcher wheel would be located outside. A slot was also cut above and connected to the circular cut in the face of the turret for the wheel to stick through inside the turret as well as the pipe. Sticking out from the plunger of the solenoid, in order to block the ball-loading slot in the pipe until the solenoid was retracted, was attached a plunger disk. This plunger facilitated one-at-a-time loading of the balls in the turret, as well as knocking them into the launcher wheel when the solenoid was released. The power wires for the solenoid were fed down into the base box and powered by a TIP 122 darlington transistor, from which the control came from the microcontroller.

The motor for the launching wheel was mounted to the front face of the turret using three hemispherical pieces that were tabbed into it: one in the front of the motor, with the shaft going through it, and the other two with the motor itself going through them. The launcher wheel was mounted to this motor using a disk piece which fit inside the rim of the wheel and around the gear piece on the shaft. The motor mounts were positioned so that the wheel would stick 0.28” into the pipe to make contact with the balls while spinning. This motor was also controlled by the turret H-bridge module.

This motor and wheel also served to shake up the balls in the turret and prevent jams at the pipe loading slot, since the wheel itself stuck into the turret. The balls in the turret were contained in the space above the pipe using two angled sheets One of these was angled from the top back of the turret box and the other from the top left. Both of these plates converged at the loading slot in the launcher barrel. Finally the top plate of the turret was mounted to contain the balls from the top. This piece had three holes drilled into it for mounting the IR beacon.

In order for our launching with the turret to be able to aim, we needed an IR sensor, to detect the other beacon, in line with our gun-barrel. To do this we constructed a small box, above the pipe, mounted to the front and right faces. The phototransistor from this circuit was extended with wires and fit into this box through a slot facing it out from the face. This box was lined with black tape to give it directional detection and shield the back and sides of the sensor and aim it in the same direction as the barrel. The sensor circuit was placed on the bottom plate of the turret, below the gun-barrel. The signal and power wires for this circuit went to the base box and were connected to the microcontroller and power distribution board respectively.

Coding:

The program, MPLAB ® X, and the associated library file for the UNO32 board (p32xxxx.h) give us the ability to write code for the robot directly in C programming language. After creating an MPLAB ® X project for pic32 board (Instructions available on CMPE118s webpage) other library files such as pwm.h, ports.h, and … were used for coding the robot. In addition to that, a few more library files and modules (motors.c, TAPES.c, and turret.c) had to be written for better handling the actuators and sensors used in the bot.

State machines:

State machines were needed in different parts of the code as a method to program the main behavior of the robot, and even to sample the sensors.

The state machines for sampling the tape sensors and the beacon consisted of simple states just to cause a non-blocking delay between taking each sample. However, the

state machines for robot's navigation and the turret were more complex to code and required hierarchical event-driven finite state machines.

Hierarchical event-driven finite state machines

Like other simple finite state machines, Hierarchical event-driven state machines are usually written in the following format:

```
switch(states)
{
    case state1:
        state1();
        if (event= state1Done)
        states = state2;
        break;
    case state2:
        state2();
        if (event=...)
        states = state2;
    break;
}
```

However, in Hierarchical state machines usually each state that consists of a lower level state machine, calls another function every time it loops back in to itself. In this example, state1 calls the state1(); function which can be a separate state machine, here used as a second level state machine. The higher level state machine will then stay in the same state, until the event flag is changed in a lower level state machine.

```
void state1(void)
{
```

```
    switch(state1States)
    {
        case state1State1:
            …
            …
            state1States = state1State2;
        break;
        case state1State2:
            event = state1Done;
        break;
    }
}
```

This method is used in this project to better organize big state machines like the navigation state machine, however other state machines were kept simple.

Tape Sensors:

An individual module was created for including the functions needed for tape sensors. TAPES.c includes three functions:

GetTape(void):

This function samples the readings from the photo transistor part of the tape sensors, once when the LED is off, and once when the LED is on. The samples from when LED is off represent the background noise which would then be subtracted from the “LED on” samples.

A simple state machine was used to implement this. In the first state all LEDs are turned on and a timer is initiated. Then the second state checks if the timer is expired. If it was, it takes the illuminated sample and turns the LEDs off waiting in the next state for the LEDs to stay off for 3ms before it takes background[] samples.

This process is then repeated during the whole program if this function is called in a while loop.
Since there are 6 tape sensors used, an array of size 6 was created for each sample type(Illuminated[6], background[6]) These arrays were not private to the function so their values could later be used in other functions to determine the final condition of the tape sensors.

TapeRefresh():
This function is a supplement to GetTape function. It gathers the samples of illuminated and background. Subtracts them and generates a final measurement. Then it compares the values of high and Low thresholds with the measurement obtained to determine the final state of the tape sensors. If it determined that the reading was on tape, it assigns “1” to the related element of the tapeDetection array and 0 otherwise.

TapeChecker ():
After the previous two functions another sampling mechanism was needed for the program to distinguish a small stain with real tape, and in general to avoid false positives. To do so, a state machine was designed to take the final value of tapeDetection every 50ms and compare the two values. If both values were the same it changes the finalTapeSample to the value from either the first or second sample. finalTapeSample is the final value that was then used in main to determine the final reading of each tape sensor.

motors.c:
This module was used to include the basic and high level functions which take care of motor behaviors.

void Init_Driving_Motor():
This function initiates all the PWMs for the driving motors as well as the ping pong launching motor and the turret motor. It also sets the digital ports used for the direction of the motors to outputs.

void Init_Launcher_Wheel():
This function sets the duty cycle(speed) of the launching wheel, and it is used in turret.c. the reason this is a separate function is that we didn't want the launcher wheel to start rotating from the beginning, and we wanted to call this function only when the turret was initiated.

Right_Wheels_Drive (unsigned int wheelSpeed, unsigned int wheelDirection):
This function and two other functions for rotating the left wheels and the turret basically have the same code, except that their PWM pins are different.
They all receive a speed and a direction as parameters, and rotate the motors correspondingly.

Some more functions such as driveStraight, tankTurnRight, and… were then included for more convenience and better coding. These functions call the other more basic functions(described above) to do their job.

Navigation State Machine:
The basic idea of our approach was to first find the beacon, then drive veering right for a certain amount of time for which the tape sensors ignore all input values. After the timer expires, depending on which tape goes on first-- tape 4 or tape 2. If tape sensor 4 goes on first, then the bot will know it is following the left coast and therefore needs to tank turn right once it find the corner. Similarly, if tape 2 is detected after the timer expires, then the bot knows to follow the right coast. Of course there is the off chance that the timer is too long and the bot has already run aground once it expires, so we had to fine tune this timer value. Once the bot finds the center, it will do a 45 degree turn and repeat the process all over again. Set a timer, veer right, look for either tape 2 or 4 and follow again. The bot should then be carving out a circuit in which it can run laps around take the roughly the same path every time. Below is a list of all the state machines involved in this complicated process with a detailed description on what

they do and the thought process behind it.

TopLevelStateMachine():

This state machine contains all the lower level state machines including the state machines for following left and right tape as well as the ones for finding center tape from the left and right sides. Breaking it down one state at a time, the first one is finding the beacon. This involves tank turning in place until the beacon is successfully sampled. When the beacon is found, a timer is initialized and the bot starts veering right immediately. This continues until either coast is found, then the state transitions to finding corner. After corner found, it goes to finding center, and after that it tank turns 45 degrees and starts all over from the beginning. The following are the second level state machines the top level one uses to navigate. To follow tape, we have three tape sensors in the middle of the bottom plate of the driving portion of our bot (tape sensors 4, 3, 2 in that order from left to right). Each are spaced at an inch and a quarter apart. When tape 3 is on exclusively, the bot will drive straight, and when either 4 or 2 go on with or without 3, the bot will correct itself by turning slightly in accordance to which tape sensor is on at the time. The state machine detects corners when the front triangular tape sensor triad is all on. There are a couple of cases in which a center might be found. For corner cases, we have tape sensors 5 1 6, 3 5 1, 2 5 1, and 4 5 1 (see appendix Top Level State Machine). For center cases, we just check tape sensors 5 3 6 and 5 6 1.

FindingCenterLeftTapeSM();

This state machine begins only when the event variable is equal to corner found. It then knows to tank right and will continue until the nose tape sensor (sensor 6) hits on tape. It will then run through the follow tape state machine until event becomes center found. It then drives forward for one and a quarter second to make sure half of the bot is in the island circle. Then it tank turns 45 degrees right and re-initializes itself to the initialize state and exits.

FindingCenterRightTapeSM();

This is the same state machine as the previous one, however two things are changed. One, the bot will turn left initially instead of right and two, after finding the center, the bot will tank turn right again.

FollowLeftTapeSM();
This state machine will start as soon as one of the tape sensor 4 hits on tape. Then it will follow the tape in the manner described in TopLevelStateMachine(); until the corner is detected. Then the state machine re-initializes to the first state and exits.

FollowRightTapeSM();
This state machine does the same thing as the previous one above. The only difference is the condition to when it starts. This state machine will only run when tape sensor 2 hits on tape.

turret.c:

The idea of the turret was to make a firing mechanism which would give the robot the ability to launch ping pong balls at the opponent, all the time regardless of which navigation state the robot is in, and which direction it is headed to. The turret should have rotated 360 degrees in each direction and look for the other robot using the beacon detector attached to it. Whenever it found the other robot, it would stop the rotation of the turret and start shooting ping pong balls for as long as the other robot was still in range.
The state machine for implementing the turret was not complicated; however, a hierarchical state machine could optimize the coding. Therefore we used to state machines to implement this: one just for rotating the turret, and the other just for handling the shooting. These two state machines work together for the best results.

Turret State Machine:

This state machine consists of simple states for rotating the turret in different directions for the amount of time it takes to make 180 or 360 turns. We want the turret to make turns no more than 360 so the wires would not get tangled. At each state of turning the shooting state machine is called all the time. Event variables were used to

determine whether the shooting state machine should be activated, and the turret should stop spinning.

turretEvents:

NoEvent:

whenever NoEvent is set for the events variable the turret should continue rotating in each direction and no ball should be shot, until the beacon is found. After the timer for rotation in one direction is up, the event will be cleared to NoEvent for the other state of rotation.

launchShooting:

The turret state machine sets the states variable to launchShooting whenever the beacon is found, and won't continue to turn again until the event is changed to enemydown by the shooting state machine.

enemyDown:

This is the event that the other robot has gone out of range after one or few times of shooting at it. when the turret state machine receives this event, it starts to rotate again until either the enemy is at range again or until the timer is up and it should rotate in the other direction.

Shooting State Machine:

The shooting state machine will stay in the Idle state until the events variable is changed to launchshooting. Right after this, it stops the timer of the turret's rotation, and starts its own timer for pulling the solenoid. Then in the next state it will keep the solenoid pulled until the timer is up, as soon as the timer was expired it stops the turret from spinning (about500ms), then it releases the solenoid for about 500ms. If the enemy was still in range, it doesn't change the event and goes back to the pulling the solenoid state to shoot again. The state machine repeats this process until the other robot is not in range anymore(beacon reading == 0), then it sets the events variable to enemydown, resumes the turret timer, and goes back to the idle state.

Sampling the beacon-detector:

The first time that we tested the turret without a sampling state machine, we noticed that the beacon detector read a lot of false positives; therefore, we designed a

state machine to take 5 samples from the beacon detector each after about 40ms of non-blocking delay. Then all the samples were compared, if they all confirmed that the beacon was in range, then a final value was issued. 1 for beacon in range, and 0 for beacon out of range.

Conclusion:

This project started out as a list of minimum specifications and resulted in a multi-functional autonomous robot. It was a true exercise in mechatronic design in that it required mechanical, electronic and computer language development. For our design, we decided on a separate firing mechanism from our navigational one. This involved a mechanical design that allowed this as well as proper state-machine hierarchy and integration. The result was a robot that did meet the minimum specifications as well as demonstrating a multitasking capability to shoot and navigate at the same time.