DamarBot: The Pothole Filling Robot
Embedded System Lab Project

Garvit Juniwal 08005008
Ravi Bhoraskar 08005002
Kunal Shah 08005005
Namit Katariya 08005007
Group 7

April 9, 2011

Contents

1 Code Documentation

We use an overhead camera to detect and manoeuvre the robot on the appropriate path. The pothole detection and filling is done locally on board the firebird robot. Hence, our code consists of:

1. *c files* to be run on the firebird, consisting on the code to perform appropriate motion on receiving commands from the base station, to detect potholes, and to fill potholes.
2. *MATLAB files* to run on the computer that is attached to the overhead camera, consisting of image processing code to detect the current position of the robot, and to determine the appropriate order to give it, so that it moves on the desired path.
3. Communication is done using *XBee*, and code must be written for both the base station and the robot to handle this communication.

In the subsequent sections, we shall mention the files that we have created, and briefly describe the functionalities of the various functions in these files.

1.1 C Code

1.1.1 xbee.c

This file contains the code to communicate using XBee on the robot. It only contains the function `uart0_init`, which initializes the appropriate registers to enable the UART interrupts, required to process XBee commands.

1.1.2 servo.c

This file contains the functions required to use the servo motor, that is attached to the hopper mechanism. When a pothole is detected, the servo motro repeatedly opens and closes the flap, in order to let the filling material drop into the hole and fill it.

- `timer1_init` and `servo1_pin_config`: These functions initialize the appropriate registers in order to enable timer1, and to enable the servo motor.
- `servo_1`: This function takes an angle (in degrees) as parameter, and sets the servo motor to that angle.

1.1.3 motion.c

This file contains code to navigate the motion of the robot. It also has code to set the velocity, using PWM.

- `motion_pin_config`, `motion_port_init` and `timer5_init`: These functions initialize the appropriate registers to enable PWM mode, and the ports which make the motion of the robot possible.
- `velocity`: Function to set the velocity of the left and right side motors of the firebird.
- `motion_set`: Function to set the appropriate bits of the appropriate register in order to move the robot in the appropriate direction as per the parameter provided.
- `forward, back, left, right, soft_left, soft_right, stop`: Functions to move the robot in the appropriate direction.

1.1.4 lcd.c

We display 2 things on the LCD screen: The road depth estimate, and the current depth estimate. The functions to use the LCD panel have been taken from the *e-yantra* website, hence we are not mentioning the details of the functions present inside this file.

1.1.5 buzzer.c

We use the buzzer to indicate that a pothole has been detected. The buzzer is on while the pothole is being filled, and turns off when the filling is done, and the robot resumes its motion. The functions to use the buzzer have been taken from the *e-yantra* website, hence we are not mentioning the details of the functions present inside this file.

1.1.6 adc.c

A *sharp sensor* is mounted at the top, near the front of the robot. It is used to detect potholes in the road, by sampling the depth of the road at periodic intervals. The ADC is used to read and process the readings of the sharp sensor. We took most of the code in this file from the e-yantra website, and will describe only the salient functions that we have used in our project.

- `adc_pin_config, display_port_init` and `adc_init`: These function sets the appropriate ports and registers in order to enable the ADC conversion, and to use the sharp sensors properly.
- `ADC_Conversion`: Given a certain ADC channel as a parameter, it converts the Analog value on that channel to a digital value and returns it. This function is used to read the value from the Sharp Sensor.
- `Sharp_GP2D120_estimation`: This function takes as input the reading that we get from a certain Sharp Sensor, and returns an *int* corresponding to the distance (in mm) which that reading represents.

1.1.7 main.c

This is the main file of our project. It contains the functions written by us in order to perform pothole detection and filling. The functions here call functions from all the helper files, and hence integrate the code spread over several files. The firebird is essentially an actuation device, which performs its motion on receiving orders from the central server. At a very abstract level, the firebird performs Algorithm 1.

Algorithm 1 Firebird Algorithm for DamarBot

```
1: while true do
2:   Await command from base station
3:   Move as commanded
4:   if Pothole is detected then
5:     while Pothole is not filled do
6:       Drop filling material in pothole
7:     end while
8:   end if
9: end while
```

The functions which we have used are as follows:

- `init_devices`: This function initializes all the ports and registers for the robot to start its operation. It calls the initialization functions of all the helper files in order to accompalish this.

- `SIGNAL`: This is the signal handler for the UART. It is called when a command is received from the base station.
- `execute_last_command`: This function executes the motion order received from the base station. It calls the appropriate functions to move in the appropriate direction.
- `take_initial_estimate`: This function is called only once, at the beginning of the run of the program. It takes a large number of readings from the sharp sensor and averages over them. It takes this value to be the *estimated road depth.* A pothole is detected with reference to this depth.
- `sample_readings`: This function implements the pothole detection algorithm, which is described here. At fixed amounts of time, the bot stops and takes the "average" reading of the depth of the road. The averaging procedure is as follows: Take the sharp sensor reading at fixed intervals of time and add them. (Note: The bot is at the same place while these multiple readings are being taken) Divide it by the number of samples taken to get the average reading at that time. The algorithm has at its disposal the estimate of the current depth of the road in the variable "road_depth_estimate" (given by the function take_initial_estimate()). If the current average reading is greater than the current estimate plus a certain THRESHOLD (set by us), then we increment a counter. Here this counter is the "consecutive_falls_in_avg". The name of the counter provides an insight into the pothole detection algorithm. We infer that a hole has been detected when this counter reaches a THRESHOLD_COUNT i.e when the average depth consecutively decreases for a THRESHOLD_COUNT number of times. If the average is less than depth_estimate+THRESHOLD, we reset the counter.
- `fill_hole`: This is the function that is called when a pothole is detected, and we are required to fill it with the filling material. This function repeatedly calls the function to change the position of the servo motor, thus opening and closing the flap of the hopper mechanism. We have adjusted the parameters such that only a small quantity of the filling material falls every time. This function continues to drop filling material, and check the road depth. Once the estimated depth is equal to the road depth estimate, we conclude that the road has been filled, and this function returns. The firebird proceeds with its motion once this is done.

1.2 Matlab Code

The Matlab code is relatively smaller in size as compared to the C-code, and hence it is in the form of a single *.m* file. The file consists of linear imperative code, and we did not need to define or call functions. The salient features of the code, and its important parts are described below:

1.2.1 Initialization

First, we close all pending connections to the hardware. Then we set up the XBee serial port, on which we shall write the commands to send to the firebird, and set up the overhead camera, which is used to detect and navigate the robot. Matlab sees the serial port as a file on which it can write, and the camera video as an object, on which we can perform various functions like taking a snapshot, previewing etc.

1.2.2 Callibration for Image Processing

Our image processing algorithm works on the assumption that the robot is fitted with two coloured patches: one each on its front and rear end. This section of the code callibrates the Matlab code so that it detects these patches correctly, so that it can take the correct action. We are required to do this for every run of the code, as even the slightest change in the lighting conditions cause the RGB values of the patches to deviate, and thus make the code incapable of detecting them.

To detect the coloured patches, we take a snapshot of the video, and then open an interface for the user to enter the lower and upper bounds on the Red, Blue and Green values for the front and the back patch respectively. We expect the user to enter these values, using the still image and Colour Detector - a free software available for Windows, that tells the RGB values for a pixel on the screen.

1.2.3 Set up the Grid

The part of the arena that is visible on the camera is logically divided into a PXQ grid, where P and Q are taken as inputs from the user. The path which the robot is supposed to follow is specified in terms of a $NEXT\_GRID$, which specifies which is the next cell of the grid that the robot needs to move to, given that it is in a particular cell. Once the grid and the colour thresholds are set up, we are all set to begin running our algorithm.

1.2.4 Locating the Robot

The sections of the code described henceforth run within a loop. The first thing to do within the loop is to locate the position of the robot, and the two coloured patches corresponding to the front and the back end respectively. This is done in the following steps:

1. `For loop`:For each patch, create a black and white image, which is white where the colour corresponds to the range of the coloured patch, and black everywhere else.

2. `bwareaopen`: In each of the black and white images, eliminate all the white regions, whose contiguous area is less than a certain threshold. Hopefully, only one area will remain, and it will correspond to the patch.

3. `BoundingBox`: For the area corresponding to the patch, find the *Bounding Box*. This is the smallest rectangle that entirely encloses the area of the patch.

4. `center_x` and `center_y`: Find the centroids of each of the Bounding Boxes. The center of the line segment joining these line segments is taken to be the center of the robot, and the vector joining these two is taken to be the direction in which the robot is pointing.

1.2.5 Moving the Robot

Once the location and orientation of the robot is determined, all that is left to do is to find out which direction the robot needs to move in, and give it the appropriate instruction. The different kinds of states which a robot can be in are summarized in Table 1.

State	Description	Action
0	On Correct Path, and within 10 degrees of expected direction	Move Forward
1	On Correct Path, but deviated between 10 and 30 deg towards right	Soft Turn Left
2	On Correct Path, and deviated more than 30 deg towards right	Hard Turn Left
3	On Correct Path, but deviated between 10 and 30 deg towards left	Soft Turn Right
4	On Correct Path, and deviated more than 30 deg towards left	Hard Turn Right
5	Deviated from Correct Path, but going towards correct path	Move Forward
6	Deviated from Correct Path, and not going towards correct path	Hard Turn Right

Table 1: The possible states of the Firebird

Once the correct state of the robot is determined, the command is written onto the XBee serial port using `fwrite` command, and the next iteration of the loop continues.