```
1  /** led.h
2
3    @file Header providing abstraction functions for the LEDs.
4    @author Matt Kokshoorn
5
6  **/
7
8  #ifndef _LED_H_
9  #define _LED_H_
10
11 #include <avr/io.h>
12
13 #define YELLOW 0
14 #define RED 6
15 #define GREEN 7
16
17 /**
18   Initilises the pins to drive the LEDs as outputs.
19 **/
20 void led_init(void);
21
22 /**
23   Toggles the LED.
24   @param (int) Logic output position on PORTB.
25 **/
26 void led_toggle(
27   int led
28   );
29
30 /**
31   Turns the LED on.
32   @param (int) Logic output position on PORTB.
33 **/
34 void led_on(
35   int led
36   );
37
38 /**
39   Turns the LED off.
40   @param (int) Logic output position on PORTB.
41 **/
42 void led_off(
43   int led
44   );
45
46 /**
47   Tests all leds by entering an infinite loop toggling all LEDs sequentially.
48 **/
49 void led_test(void);
50
51 #endif
```

```
1  /** led.c
2
3    @file Abstracted functions to initilise, toggle and turn on/off LED's.
4    Also has a test led function to cycle through LED's to ensure functionality.
5    @author Matt Kokshoorn
6
7  **/
8
9  #include "led.h"
10
11 /**
12   Initilises the pins to drive the LEDs as outputs.
13 **/
14 void led_init(void)
15 {
16   DDRB|=(1<<YELLOW)|(1<<RED)|(1<<GREEN);
17   led_off(RED); led_off(GREEN); led_off(YELLOW);
18 }
19
20 /**
21   Toggles the LED.
22   @param (int) Logic output position on PORTB.
23 **/
24 void led_toggle(
25   int led
26   )
27 {
28   PORTB=PORTB^(1<<led);
29 }
30
31 /**
32   Turns the LED on.
33   @param (int) Logic output position on PORTB.
34 **/
35 void led_on(
36   int led
37   )
38 {
39   PORTB=PORTB&(~(1<<led));
40
41 }
42
43 /**
44   Turns the LED off.
45   @param (int) Logic output position on PORTB.
46 **/
47 void led_off(
48   int led
49   )
50 {
51   PORTB=PORTB|(1<<led);
52 }
53
54 /**
55   Tests all leds by entering an infinite loop toggling all LEDs sequentially.
56 **/
57 void led_test(void)
```

```
58 {
59   volatile long i;
60   volatile long j=0;
61   while(1) {
62
63     if(j==1) led_on(YELLOW);
64     if(j==2) led_on(GREEN);
65     if(j==3) led_on(RED);
66     if(j==4) led_off(YELLOW);
67     if(j==5) led_off(GREEN);
68     if(j==6) led_off(RED);
69     if(j==7) j=0;
70     j++;
71     for (i = 0; i < 1000; i++) continue; //delay loop
72   }
73
74 }
```

```
1  /** button.h
2
3    @file Header providing abstraction functions for the button.
4    @author Matt Kokshoorn
5
6  **/
7
8  #ifndef _BUTTON_H_
9  #define _BUTTON_H_
10
11 #include <avr/io.h>
12
13 #define BUTTON_BIT 0
14
15 /**
16   Function initilises pin for with button as an input.
17 **/
18 void button_init(void);
19
20 /**
21   Abstracted fucntion to check the state of the button.
22   @return  (char)  The button state 1 or 0.
23 **/
24 char button_pressed(void);
25
26 /**
27   Provides button debounce support.
28 **/
29 void button_debounce(void);
30
31 #endif
32
```

```
1   /** button.c
2
3     @file Abstrated fucntions to initilise, check the state of the button.
4     @author Matt Kokshoorn and Nick Bingham
5
6   **/
7
8   #include "button.h"
9
10  /**
11    Function initilises pin for with button as an input.
12  **/
13  void button_init(void)
14  {
15    DDRB=DDRB&(~(1<<BUTTON_BIT));
16  }
17
18
19  /**
20    Abstracted fucntion to check the state of the button.
21    @return  (char)  The button state 1 or 0.
22  **/
23  char button_pressed(void)
24  {
25    return !((PINB&(1<<BUTTON_BIT))>0);
26  }
27
28  /**
29    Provides button debounce support.
30  **/
31  void button_debounce(void)
32  {
33    uint8_t button_on_count = 255;
34    while (button_on_count) {
35      if (button_pressed()) {
36        button_on_count = 255;
37      }
38      else {
39        button_on_count--;
40      }
41    }
42  }
43
```

```
1  /** adc.h
2
3    @file Header providing abstraction functions for the ADC.
4    @author Matt Kokshoorn
5
6  **/
7
8  #ifndef _ADC_H_
9  #define _ADC_H_
10
11 #include <avr/io.h>
12 #include <stdint.h>
13 #include <avr/interrupt.h>
14 #include "adc.h"
15
16 #define BIT(x) (1 << (x))
17
18 #define ADC0 0
19 #define ADC1 1
20 #define ADC2 2
21 #define MUX_ADC0 (ADMUX & ~(BIT(MUX3) | BIT(MUX2) | BIT(MUX1) | BIT(MUX0) ) )
22 #define MUX_ADC1 ((ADMUX & ~(BIT(MUX3) | BIT(MUX2) | BIT(MUX1) | BIT(MUX0) ) ) |
   BIT(MUX0))
23 #define MUX_ADC2 ((ADMUX & ~(BIT(MUX3) | BIT(MUX2) | BIT(MUX1) | BIT(MUX0) ) ) |
   BIT(MUX1))
24
25 /**
26   Initilises the ADC for use and begins the corresponding ISR.
27 **/
28 void adc_init(void);
29
30 #endif
```

```
1  /** adc.c
2
3    @file Initilises the Analog to Digital Convertor.
4    @author Matt Kokshoorn
5
6  **/
7
8  #include "adc.h"
9
10 /**
11   Initilises the ADC for use and begins the corresponding ISR.
12 **/
13 void adc_init(void)
14 {
15   /* prescaler 8 so sample rate is 125 kHz */
16   ADCSRA |= BIT(ADPS1) | BIT(ADPS0);
17   /* Set reference as AVcc*/
18   ADMUX |= BIT(REFS0);
19   /* Turn ADC into 8 bit*/
20   ADMUX |= BIT(ADLAR);
21   /* ADC0 to be ADC input */
22   ADMUX = MUX_ADC0;
23   /* Set ADC to Free-Running mode */
24   ADCSRA |= BIT(ADFR);
25   /* Enable the ADC */
26   ADCSRA |= BIT(ADEN);
27   // Enable ADC Interrupt
28   ADCSRA |= BIT(ADIE);
29   // Enable Global Interrupts
30   sei();
31   /* Start the ADC measurements */
32   ADCSRA |= BIT(ADSC);
33 }
34
35
36
37
```

```
1  /** pwm.h
2
3    @file Header providing basic drivers for the PWM.
4    @author Matt Kokshoorn
5
6  **/
7
8
9  #ifndef _PWM_H_
10 #define _PWM_H_
11
12 #include <avr/io.h>
13
14 #define BIT(x) (1 << (x))
15 #define DUTY_CYCLE OCR2
16
17 /**
18   Initilise the PWM output to PB3.
19 **/
20 void pwm_init (void);
21
22 #endif
```

```
1  /** pwm.c
2
3    @file Provides basic drivers for the PWM.
4    @author Matt Kokshoorn
5
6  **/
7
8  #include "pwm.h"
9
10 /**
11   Initilise the PWM output to PB3.
12 **/
13 void pwm_init (void)
14 {
15   /* PB3 as output */
16   DDRB = BIT(DDB3);
17
18   /* Assign PWM to PB3 using Timer2 */
19   TCCR2 = BIT(WGM20) | BIT(WGM21)  /* Fast PWM mode */
20           | BIT(COM21)             /* Clear OC2 on Compare Match */
21           | BIT(CS21);             /* 1/8 prescale = 488 Hz */
22
23   OCR2 = 0x80;
24 }
```

```
1  /** motor.h
2
3    @file Header providing drivers and abstraction functions for the motor.
4    @author Matt Kokshoorn and Nick Bingham
5
6  **/
7
8  #include <avr/io.h>
9  #include "delay.h"
10
11 #define LEFT 0
12 #define RIGHT 1
13
14 #define FWD 0
15 #define BK 1
16
17 #define ON 1
18 #define OFF 0
19
20 #define RIGHT_FWD 0x04
21 #define LEFT_BK 0x01
22 #define RIGHT_BK 0x08
23 #define LEFT_FWD 0x02
24
25 /**
26   Initilises motor logic outputs.
27 **/
28 void motor_init(void);
29
30 /**
31   Function that changes the state of one of the motor outputs.
32   @param  (int)  The side of the motor to drive, either LEFT or RIGHT.
33   @param  (int)  The direction of the motor to drive, either FWD or BK.
34   @param  (int)  The value of the motor state, either OFF or ON.
35 **/
36 void motor_logic(
37   int side,
38   int dir,
39   int val
40   );
41
42 /**
43   Abstracted function to stop the motors.
44 **/
45 void motor_stop(void);
46
47 /**
48   Abstracted function to drive the motors in reverse.
49 **/
50 void motor_backward(void);
51
52 /**
53   Abstracted function to drive the motors forward.
54 **/
55 void motor_forward(void);
56
57 /**
```

```
58   Abstracted function to drive the motors in an arcing right turn.
59 **/
60 void motor_right(void);
61
62 /**
63   Abstracted function to drive the motors in an arcing left turn.
64 **/
65 void motor_left(void);
66
67 /**
68   Abstracted function to drive the motors in a pivoting right turn.
69 **/
70 void motor_pivot_right(void);
71
72 /**
73   Abstracted function to drive the motors in a pivoting left turn.
74 **/
75 void motor_pivot_left(void);
76
77 /**
78   Abstracted function to provide multiple turning components in one turn.
79   This function leaves active the last one of the modes used, in the order:
80     Stop > Forward > Turn > Pivot.
81   @param  (unsigned int)  Minimum time desired to turn.
82   @param  (unsigned int)  Time desired to pivot.
83   @param  (unsigned int)  Time desired to move forward.
84   @param  (unsigned int)  Time desired to stop.
85   @param  (unsigned int)  Direction of movement, either LEFT or RIGHT.
86 **/
87 void motor_turn(unsigned int turn_delay,
88   unsigned int pivot_delay,
89   unsigned int forward_delay,
90   unsigned int stop_delay,
91   unsigned int direction
92   );
93
94
```

```
1  /** motor.c
2
3    @file Provides basic drivers and abstracted functions for the motor.
4    @author Matt Kokshoorn and Nick Bingham
5
6  **/
7
8  #include "motor.h"
9
10 /**
11   Initilises motor logic outputs.
12 **/
13 void motor_init(void)
14 {
15       DDRD|=LEFT_FWD|RIGHT_FWD|LEFT_BK|RIGHT_BK;
16 }
17
18 /**
19   Function that changes the state of one of the motor outputs.
20   @param  (int)  The side of the motor to drive, either LEFT or RIGHT.
21   @param  (int)  The direction of the motor to drive, either FWD or BK.
22   @param  (int)  The value of the motor state, either OFF or ON.
23 **/
24 void motor_logic(
25   int side,
26   int dir,
27   int val
28   )
29 {
30   if(dir==FWD){
31     if (val==ON){
32       if (side==LEFT){
33         PORTD|=(1<<1);
34       }
35       else if(side==RIGHT){
36         PORTD=PORTD|RIGHT_FWD;
37       }
38     }
39     else if(val==OFF){
40       if (side==LEFT){
41         PORTD&=~(LEFT_FWD);
42       }
43       else if(side==RIGHT){
44         PORTD&=~(RIGHT_FWD);
45       }
46     }
47   }
48   else if(dir==BK){
49     if (val==ON){
50       if (side==LEFT){
51         PORTD|=LEFT_BK;
52       }
53       else if(side==RIGHT){
54         PORTD|=RIGHT_BK;
55       }
56     }
57     else if(val==OFF){
```

```
58        if (side==LEFT){
59          PORTD&=~(LEFT_BK);
60        }
61        else if(side==RIGHT){
62          PORTD&=~(RIGHT_BK);
63        }
64      }
65    }
66  }
67
68  /**
69    Abstracted function to stop the motors.
70  **/
71  void motor_stop(void)
72  {
73    motor_logic(LEFT,FWD,OFF);
74    motor_logic(RIGHT,FWD,OFF);
75    motor_logic(LEFT,BK,OFF);
76    motor_logic(RIGHT,BK,OFF);
77  }
78
79  /**
80    Abstracted function to drive the motors in reverse.
81  **/
82  void motor_backward(void)
83  {
84    motor_logic(LEFT,FWD,ON);
85    motor_logic(RIGHT,FWD,ON);
86    motor_logic(LEFT,BK,OFF);
87    motor_logic(RIGHT,BK,OFF);
88  }
89
90  /**
91    Abstracted function to drive the motors forward.
92  **/
93  void motor_forward(void)
94  {
95    motor_logic(LEFT,FWD,OFF);
96    motor_logic(RIGHT,FWD,OFF);
97    motor_logic(LEFT,BK,ON);
98    motor_logic(RIGHT,BK,ON);
99  }
100
101 /**
102   Abstracted function to drive the motors in an arcing right turn.
103 **/
104 void motor_right(void)
105 {
106   motor_logic(LEFT,FWD,OFF);
107   motor_logic(RIGHT,FWD,OFF);
108   motor_logic(LEFT,BK,ON);
109   motor_logic(RIGHT,BK,OFF);
110 }
111
112 /**
113   Abstracted function to drive the motors in an arcing left turn.
114 **/
```

```
115 void motor_left(void)
116 {
117   motor_logic(LEFT,FWD,OFF);
118   motor_logic(RIGHT,FWD,OFF);
119   motor_logic(LEFT,BK,OFF);
120   motor_logic(RIGHT,BK,ON);
121 }
122
123 /**
124   Abstracted function to drive the motors in a pivoting right turn.
125 **/
126 void motor_pivot_right(void)
127 {
128   motor_logic(LEFT,FWD,OFF);
129   motor_logic(RIGHT,FWD,ON);
130   motor_logic(LEFT,BK,ON);
131   motor_logic(RIGHT,BK,OFF);
132 }
133
134 /**
135   Abstracted function to drive the motors in a pivoting left turn.
136 **/
137 void motor_pivot_left(void)
138 {
139   motor_logic(LEFT,FWD,ON);
140   motor_logic(RIGHT,FWD,OFF);
141   motor_logic(LEFT,BK,OFF);
142   motor_logic(RIGHT,BK,ON);
143 }
144
145 /**
146   Abstracted function to provide multiple turning components in one turn.
147   This function leaves active the last one of the modes used, in the order:
148     Stop > Forward > Turn > Pivot.
149   @param  (unsigned int)  Minimum time desired to turn.
150   @param  (unsigned int)  Time desired to pivot.
151   @param  (unsigned int)  Time desired to move forward.
152   @param  (unsigned int)  Time desired to stop.
153   @param  (unsigned int)  Direction of movement, either LEFT or RIGHT.
154 **/
155 void motor_turn(unsigned int turn_delay,
156   unsigned int pivot_delay,
157   unsigned int forward_delay,
158   unsigned int stop_delay,
159   unsigned int direction
160   )
161 {
162   if (direction == LEFT){
163     motor_pivot_left();
164     delay_flat(pivot_delay);
165     if (turn_delay > 0) {
166       motor_left();
167       delay_flat(turn_delay);
168     }
169   }
170   else {
171     motor_pivot_right();
```

```
172     delay_flat(pivot_delay);
173     if (turn_delay > 0) {
174       motor_right();
175       delay_flat(turn_delay);
176     }
177   }
178   if (forward_delay > 0) {
179     motor_forward();
180     delay_flat(forward_delay);
181   }
182   if (stop_delay > 0) {
183     motor_stop();
184     delay_flat(stop_delay);
185   }
186 }
187
188
189
```

```
1  /** delay.h
2
3    @file Header providing abstracted delay functions.
4    @author Nick Bingham
5
6  **/
7
8  #ifndef _DELAY_H_
9  #define _DELAY_H_
10
11 /**
12   Delays the processor for a short period from 0 to 255 cycles.
13   @param  (unsigned long int)  Delay period in the range 0-255.
14 **/
15 void delay_flat(
16   volatile unsigned long int value
17   );
18
19 #endif
20
21
```

```
1  /** delay.c
2
3    @file Provides abstracted delay functions.
4    @author Nick Bingham
5
6  **/
7
8  #include "delay.h"
9
10 /**
11   Delays the processor for a short period from 0 to 255 cycles.
12   @param  (unsigned long int)  Delay period in the range 0-255.
13 **/
14 void delay_flat(
15   volatile unsigned long int value
16   )
17 {
18   while (value > 0) {
19     value--;
20   }
21 }
22
23
```

```
1  /** sensor.h
2
3    @file Header providing abstraction functions for the sensor.
4    @author Nick Bingham
5
6  **/
7
8  #ifndef _SENSOR_H_
9  #define _SENSOR_H_
10
11 #include "delay.h"
12 #include "button.h"
13 #include "led.h"
14 #include "motor.h"
15
16 #define CALIBRATE_COUNT 255
17 #define CALIBRATE_DELAY 100
18 #define DEBOUNCE_DELAY 120
19
20 #define GREY_LOWER grey_val-8
21 #define GREY_UPPER grey_val+8
22 #define GREY_MIN 12
23 #define ANTIGREY_MAX 1
24 #define ANTIGREY_PENALTY 3
25
26 #define BLACK 1
27 #define WHITE 0
28
29 /**
30   Determines the average value of the sensor over a short time interval.
31   @param  (unsigned int *)  Address of the sensor ISR value.
32   @return  (unsigned int)  Average sensor reading, in the range 0-255.
33 **/
34 unsigned int sensor_calibrate(
35   unsigned int *sensor_reading
36   );
37
38 /**
39   Determines the average value of the sensor over a short time interval.
40   For use in a continuous loop waiting for the button conditions to be
41   satisfied during the various stages of sensor calibration.
42   @param  (unsigned int *)  Address of the sensor ISR value.
43   @param  (unsigned int *)  Address at which the calibrated value is stored.
44   @param  (unsigned int *)  State of calibration, either 1 (ready) or 0.
45   @return  (unsigned int)  Average sensor reading, in the range 0-255.
46 **/
47 unsigned int sensor_init(
48   unsigned int *sensor_middle,
49   unsigned int *value,
50   unsigned int *ready
51   );
52
53 /**
54   Checks whether the sensor is currently over grey. If it is, this function
55   will increment counters and stop after several cycles, displaying the looped
56   led flashing function.
57   @param  (unsigned int)  The calibrated grey value.
```

```
58   @param  (unsigned int *)  Address holding the sensor ISR value.
59   @param  (unsigned int *)  Address storing the number of times that a grey
60   value has been reached recently.
61   @param  (unsigned int *)  Address storing the number of times that a non-grey
62   value has been reached recently.
63 **/
64 void sensor_grey_check(
65   unsigned int grey_val,
66   unsigned int *sensor_middle,
67   unsigned int *grey_count,
68   unsigned int *grey_anticount
69   );
70
71 /**
72   Checks whether the sensor has changed since it was last checked. If it is, the
73   value of the global 'middle' will be adjusted to accommodate this.
74   @param  (unsigned int)  The calibrated grey value.
75   @param  (unsigned int *)  Address holding the sensor ISR value.
76   @param  (unsigned int *)  Address holding the current state of the sensor,
77   either BLACK (1) or WHITE (0).
78 **/
79 void sensor_change_detect(
80   unsigned int grey_val,
81   unsigned int *sensor_middle,
82   unsigned int *middle
83   );
84
85 #endif
86
87
```

```
1  /** sensor.c
2
3    @file Provides abstracted functions for the sensor.
4    @author Nick Bingham
5
6  **/
7
8  #include "sensor.h"
9
10 unsigned int prev_mid = 1;
11 unsigned int curr_mid = 1;
12
13 /**
14   Determines the average value of the sensor over a short time interval.
15   @param  (unsigned int *)  Address of the sensor ISR value.
16   @return  (unsigned int)  Average sensor reading, in the range 0-255.
17 **/
18 unsigned int sensor_calibrate(
19   unsigned int *sensor_reading
20   )
21 {
22   unsigned long int calibrate_total = 0;
23   unsigned int count = 0;
24
25
26   while (count < CALIBRATE_COUNT) {
27     calibrate_total += *sensor_reading;
28     delay_flat(CALIBRATE_DELAY);
29     delay_flat(CALIBRATE_DELAY);
30     count++;
31   }
32   return (unsigned int)calibrate_total/CALIBRATE_COUNT;
33 }
34
35 /**
36   Determines the average value of the sensor over a short time interval.
37   For use in a continuous loop waiting for the button conditions to be
38   satisfied during the various stages of sensor calibration.
39   @param  (unsigned int *)  Address of the sensor ISR value.
40   @param  (unsigned int *)  Address at which the calibrated value is stored.
41   @param  (unsigned int *)  Address holding the state of calibration, either 1
42   (ready) or 0.
43   @return  (unsigned int)  Average sensor reading, in the range 0-255.
44 **/
45 unsigned int sensor_init(
46   unsigned int *sensor_middle,
47   unsigned int *value,
48   unsigned int *ready
49   )
50 {
51   unsigned int complete = 0;
52   if(button_pressed() && !*ready){
53     led_on(RED);
54     *ready = 1;
55     *value = sensor_calibrate(sensor_middle);
56     button_debounce();
57   }
```

```
58    else if (button_pressed()){
59      led_on(RED);
60      complete = 1;
61      button_debounce();
62      led_init();
63    }
64    delay_flat(DEBOUNCE_DELAY);
65    led_off(RED);
66    return complete;
67  }
68
69  /**
70    Checks whether the sensor is currently over grey. If it is, this function
71    will increment counters and stop after several cycles, displaying the looped
72    led flashing function.
73    @param  (unsigned int)  The calibrated grey value.
74    @param  (unsigned int *)  Address holding the sensor ISR value.
75    @param  (unsigned int *)  Address storing the number of times that a grey
76    value has been reached recently.
77    @param  (unsigned int *)  Address storing the number of times that a non-grey
78    value has been reached recently.
79  **/
80  void sensor_grey_check(
81    unsigned int grey_val,
82    unsigned int *sensor_middle,
83    unsigned int *grey_count,
84    unsigned int *grey_anticount
85    )
86  {
87    if (*sensor_middle > GREY_LOWER && *sensor_middle < GREY_UPPER) {
88      *grey_count += 1;
89      if (*grey_count > GREY_MIN) {
90        motor_stop();
91        led_test();
92      }
93    }
94    else {
95      *grey_anticount += 1;
96      if (*grey_anticount > ANTIGREY_MAX) {
97        if (*grey_count > (ANTIGREY_PENALTY+1)) {
98          *grey_count -= ANTIGREY_PENALTY;
99        }
100       else {
101         *grey_count = 0;
102       }
103       *grey_anticount = 0;
104     }
105   }
106 }
107
108 /**
109   Checks whether the sensor has changed since it was last checked. If it is,
      the
110   value of the global 'middle' will be adjusted to accommodate this.
111   @param  (unsigned int)  The calibrated grey value.
112   @param  (unsigned int *)  Address holding the sensor ISR value.
113   @param  (unsigned int *)  Address holding the current state of the sensor,
```

```
114   either BLACK (1) or WHITE (0).
115 **/
116 void sensor_change_detect(
117   unsigned int grey_val,
118   unsigned int *sensor_middle,
119   unsigned int *middle
120   )
121 {
122   prev_mid = curr_mid;
123   if (*sensor_middle > grey_val) {
124     curr_mid = BLACK;
125   }
126   else {
127     curr_mid = WHITE;
128   }
129   if (curr_mid == prev_mid){
130     *middle = curr_mid;
131   }
132 }
133
134
135
```

```
1  /** control.h
2
3    @file Header providing control functions for the robot.
4    @author Nick Bingham
5
6  **/
7
8  #ifndef _CONTROL_H_
9  #define _CONTROL_H_
10
11 #include "motor.h"
12 #include "sensor.h"
13
14 #define OUTSIDE_MAX 250
15 #define INSIDE_MAX 145
16 #define WOBBLE_LIMIT 5
17
18 #define OUTSIDE_TURN 2
19 #define OUTSIDE_PIVOT 0
20 #define OUTSIDE_FWD 15
21 #define OUTSIDE_STOP 1
22
23 #define INSIDE_TURN 0
24 #define INSIDE_PIVOT 30
25 #define INSIDE_FWD 0
26 #define INSIDE_STOP 0
27
28 /**
29   Function to move in an arcing forward direction at a predetermined rate. The
30   radius of the arc will depend on the current value (BLACK or WHITE) of the
31   sensor and also which side of the line is being followed.
32   @param  (unsigned int)  The side currently being followed, either LEFT or
33   RIGHT.
34   @param  (unsigned int)  The number of turns made on the current side of the
35   line up to that point.
36   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
37   @param  (unsigned int *)  The number of successive left turns.
38   @param  (unsigned int *)  The number of successive right turns.
39   @param  (unsigned int *)  The number of wobbles since a left or right turn.
40 **/
41 void control_forward(
42   unsigned int current_side,
43   unsigned int turn_count,
44   unsigned int middle,
45   unsigned int *turn_left_count,
46   unsigned int *turn_right_count,
47   unsigned int *wobble_count
48   );
49
50 /**
51   Function to stabilise the robot after crossing over a black/white
52   intersection. This compensates for the length of the previous turn.
53   @param  (unsigned int)  The side currently being followed, either LEFT or
54   RIGHT.
55   @param  (unsigned int)  Address holding the number of turns made on the
56   current side of the line up to that point.
57   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
```

```
58   @param  (unsigned int *)  Address holding the number of wobbles since a left
59   or right turn.
60 **/
61 void control_stabilise(
62   unsigned int current_side,
63   unsigned int *turn_count,
64   unsigned int middle,
65   unsigned int *wobble_count
66   );
67
68 /**
69   This function comes into play when the robot is stuck inside a loop. It causes
70   the robot to rotate and move to the other side of the line before continuing.
71   @param  (unsigned int)  The side currently being followed, either LEFT or
72   RIGHT.
73   @param  (unsigned int *)  Address holding the number of turns made on the
74   current side of the line up to that point.
75   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
76   @param  (unsigned int *)  Address recording which side to next travel to.
77 **/
78 void control_switch(
79   unsigned int current_side,
80   unsigned int *turn_count,
81   unsigned int middle,
82   unsigned int *move_direction
83   );
84
85 #endif
86
87
```

```
1  /** control.c
2
3    @file Provides control functions for the robot.
4    @author Nick Bingham
5
6  **/
7
8  #include "control.h"
9
10 unsigned int move_first = 1;
11
12 /**
13   Function to move in an arcing forward direction at a predetermined rate. The
14   radius of the arc will depend on the current value (BLACK or WHITE) of the
15   sensor and also which side of the line is being followed.
16   @param  (unsigned int)  The side currently being followed, either LEFT or
17   RIGHT.
18   @param  (unsigned int)  The number of turns made on the current side of the
19   line up to that point.
20   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
21   @param  (unsigned int *)  Address holding the number of consecutive left
     turns.
22   @param  (unsigned int *)  Address holding the number of consecutive right
     turns.
23   @param  (unsigned int *)  Address holding the number of wobbles since a left
24   or right turn.
25 **/
26 void control_forward(
27   unsigned int current_side,
28   unsigned int turn_count,
29   unsigned int middle,
30   unsigned int *turn_left_count,
31   unsigned int *turn_right_count,
32   unsigned int *wobble_count
33   )
34 {
35   unsigned int outside = current_side;
36   unsigned int inside = !current_side;
37   if (middle == WHITE){
38     motor_turn(OUTSIDE_TURN, OUTSIDE_PIVOT, OUTSIDE_FWD, OUTSIDE_STOP,
       outside);
39     if (turn_count == OUTSIDE_MAX && *wobble_count > WOBBLE_LIMIT) {
40       if (current_side == LEFT) {
41         *turn_right_count++;
42         *turn_left_count=0;
43         *wobble_count = 0;
44       }
45       else {
46         *turn_left_count++;
47         *turn_right_count = 0;
48         *wobble_count = 0;
49       }
50     }
51   }
52   else{
53     motor_turn(INSIDE_TURN, INSIDE_PIVOT, INSIDE_FWD, INSIDE_STOP, inside);
54     if (turn_count == INSIDE_MAX && *wobble_count > WOBBLE_LIMIT) {
```

```
55        if (current_side == RIGHT) {
56          *turn_right_count++;
57          *turn_left_count=0;
58          *wobble_count = 0;
59        }
60        else {
61          *turn_left_count++;
62          *turn_right_count = 0;
63          *wobble_count = 0;
64        }
65      }
66    }
67  }
68
69  /**
70    Function to stabilise the robot after crossing over a black/white
71    intersection. This compensates for the length of the previous turn.
72    @param  (unsigned int)  The side currently being followed, either LEFT or
73    RIGHT.
74    @param  (unsigned int *)  Address holding the number of turns made on the
75    current side of the line up to that point.
76    @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
77    @param  (unsigned int *)  Address holding the number of wobbles since a left
78    or right turn.
79  **/
80  void control_stabilise(
81    unsigned int current_side,
82    unsigned int *turn_count,
83    unsigned int middle,
84    unsigned int *wobble_count
85    )
86  {
87    unsigned int outside = current_side;
88    unsigned int inside = !current_side;
89    if (*turn_count > 100) {
90      *turn_count = 100;
91    }
92    *turn_count = (*turn_count/2);
93    while (*turn_count > 0) {
94      if (middle == WHITE){
95        motor_turn(0, (OUTSIDE_TURN+OUTSIDE_PIVOT), 0, 0, outside);
96      }
97      else{
98        motor_turn(0, (INSIDE_TURN+INSIDE_PIVOT), 0, 0, inside);
99      }
100     *turn_count -= 1;
101   }
102   *wobble_count++;
103   *turn_count = 0;
104 }
105
106 /**
107   This function comes into play when the robot is stuck inside a loop. It
      causes
108   the robot to rotate and move to the other side of the line before continuing.
109   @param  (unsigned int)  The side currently being followed, either LEFT or
110   RIGHT.
```

```
111   @param  (unsigned int *)  Address holding the number of turns made on the
112   current side of the line up to that point.
113   @param  (unsigned int)  The current sensor reading, either BLACK or WHITE.
114   @param  (unsigned int *)  Address recording which side to next travel to.
115 **/
116 void control_switch(
117   unsigned int current_side,
118   unsigned int *turn_count,
119   unsigned int middle,
120   unsigned int *move_direction
121   )
122 {
123   unsigned int move_continue = 0;
124   unsigned int outside = current_side;
125   if (*turn_count > 140) {
126     *turn_count = 140;
127   }
128   *turn_count = (*turn_count/2);
129   while (*turn_count > 0 && move_first) {
130     motor_turn(0, (INSIDE_TURN+INSIDE_PIVOT), 0, 0, outside);
131     *turn_count -= 1;
132   }
133   move_first = 0;
134   if (middle == BLACK) {
135     motor_turn(0, 0, 1, 0, outside);
136   }
137   else {
138     move_continue = 1;
139   }
140   if (move_continue) {
141     move_first = 1;
142     *move_direction = !*move_direction;
143   }
144 }
145
146
```

```
1  /** main.c
2
3    @File Main function for the ENEL300 Assignment 2 2012.
4    @Author Matt Kokshoorn and Nick Bingham
5
6  **/
7
8  #include <avr/io.h>
9  #include <avr/interrupt.h>
10 #include "led.h"
11 #include "adc.h"
12 #include "pwm.h"
13 #include "motor.h"
14 #include "button.h"
15 #include "sensor.h"
16 #include "delay.h"
17 #include "control.h"
18
19 unsigned int choice = ADC0;
20 unsigned int sensor_left = 0;
21 unsigned int sensor_right = 0;
22 unsigned int sensor_middle = 0;
23
24 unsigned int grey_val = 0;
25 unsigned int grey_ready = 0;
26 unsigned int executing = 0;
27
28 unsigned int grey_count = 0;
29 unsigned int grey_anticount = 0;
30
31 unsigned int middle = 1;
32 unsigned int old_mid = 1;
33
34 unsigned int turn_count = 0;
35 unsigned int turn_left_count = 0;
36 unsigned int turn_right_count = 0;
37 unsigned int wobble_count = 0;
38 unsigned int move_direction = RIGHT;
39
40 int main (void){
41
42   adc_init();
43   pwm_init();
44   motor_init();
45   led_init();
46   button_init();
47   DUTY_CYCLE=250;  // 0-255
48
49   while(!executing){
50     executing = sensor_init(&sensor_middle, &grey_val, &grey_ready);
51   }
52
53   while(executing){
54     // Update sensor information.
55     sensor_grey_check(grey_val, &sensor_middle, &grey_count, &grey_anticount);
56     sensor_change_detect(grey_val, &sensor_middle, &middle);
57
```

```
58              // Execute one cycle of the algorithm.
59              if (turn_left_count > 4) {
60                // Respond to sensor information following right edge.
61                if (move_direction == RIGHT) {
62                  control_switch(LEFT, &turn_count, middle, &move_direction);
63        }
64                else if (middle != old_mid){
65          old_mid = middle;
66          control_stabilise(RIGHT, &turn_count, middle, &wobble_count);
67        }
68        else {
69          control_forward(RIGHT, turn_count, middle, &turn_left_count,
            &turn_right_count, &wobble_count);
70        }
71      }
72      else {
73        // Respond to sensor information following left edge.
74        if (move_direction == LEFT) {
75          control_switch(RIGHT, &turn_count, middle, &move_direction);
76        }
77        else if (middle != old_mid){
78          old_mid = middle;
79          control_stabilise(LEFT, &turn_count, middle, &wobble_count);
80        }
81        else {
82          control_forward(LEFT, turn_count, middle, &turn_left_count,
            &turn_right_count, &wobble_count);
83        }
84      }
85      turn_count++;
86
87      // Limit values.
88      if (turn_count > 250) {
89        turn_count = 250;
90      }
91      if (wobble_count > 10) {
92        wobble_count = 10;
93      }
94
95      // Binary counter for LEDs.
96      if (turn_left_count & (1<<0)) {
97        led_on(RED);
98      }
99      else {
100       led_off(RED);
101     }
102     if (turn_left_count & (1<<1)) {
103       led_on(GREEN);
104     }
105     else {
106       led_off(GREEN);
107     }
108     if (turn_left_count & (1<<2)) {
109       led_on(YELLOW);
110     }
111     else {
112       led_off(YELLOW);
```

```
113     }
114   }
115 }
116
117 ISR(ADC_vect){
118       // 0 ->2; 2->1; 1->0
119     if (choice == ADC0){
120     sensor_left = ADCH;
121     /* start measuring adc1 */
122     ADMUX = MUX_ADC1;
123     choice = ADC1;
124     }
125     else if (choice == ADC1){
126       sensor_right = ADCH;
127       /* start measuring adc2 */
128     ADMUX = MUX_ADC2;
129     choice = ADC2;
130     }
131     else if (choice == ADC2){
132        sensor_middle = ADCH;
133        /* start measuring adc0 */
134        ADMUX = MUX_ADC0;
135        choice = ADC0;
136     }
137 }
138
139
140
```