ECEN 345/ECEN 3450-001

Mobile Robotics 1

Prof. A. Gilmore (Instructor)
Carl Endrulat (TA)

Laboratory Assignment #6

Obstacle Avoidance with Ultrasonic Sensing

Written by
Hayden Ernst
Nelson Lefebvre

Department of Electrical and Computer Engineering (ECE)
University of Nebraska Lincoln/Omaha

Due: 10/30/2022

Introduction

This lab adds an ultrasonic ranging sensor to the current implementations to incorporate an additional avoidance behavior. This was done after learning to trigger and gather distance data and exploring the limitations of the sensor.

A C function was developed to trigger and gather distance data from the ultrasonic sensor. This was then used in a sonar avoidance behavior which was added to the control structure already in place. The ultrasonic sensor has a farther range than the IR sensors. It will probably sense an obstacle first. Then, depending on how far the object is away, a difference in wheel speed will cause the CEENBoT to turn to get around it. Once the object is no longer detected, the bot will return to going straight.

This lab required roughly three to four hours of work. This was evenly distributed between partners with two hours spent in lab working with the ultrasonic sensor, setting it up, programming for it, and testing/calibrating it. Outside of lab roughly one to two hours were spent programming the avoidance function and programming it. Finally, roughly one hour was spend writing this report. The main resources used in this lab were the lab handout and ultrasonic sensor mounting instructions. These gave the information necessary to connect and use the ultrasonic sensor. Atmel Studio 7 and the CEENBoT Utility Tool were used to program the CEENBoT.

The background section will next go over some background information needed to complete this lab such as sensor and bot control methods. The procedure section will explain how each step in the lab was completed. A source code discussion section explains the program in more detail. The results section will discuss the outcomes of the experiments and answer questions found in the lab handout. The conclusion will summarize the report and offer some personal thoughts. The source code will be posted at the end of the document.

Background

This lab heavily utilized an ultrasonic distance sensor to find the distance to objects in front of the CEENBoT so an avoidance behavior could be implemented. The sensor itself has three connections, one for power (5V), ground, and one digital I/O pin. The I/O pin is used to 'trigger' the device and indicate an echo return. The sensor first waits for a triggering pulse from the controller. The sensor sends an ultrasonic ping and sets the I/O pin high to indicate to the controller that a ping has been sent. The line will remain high until the sensor detects the 'echo' from the ping. Then the signal will be set back low allowing the controller to measure the pulse-width and determine the distance from the object.

This distance can be computed with the equation $d = \frac{1}{2}(v)(t)$. The velocity of sound in air is 344.8 m/s or 34480 cm/s which is used for this purpose. This is substituted in and multiplied by 1x10^{-6} so the value of time can be in micro-seconds and the equation is then shown as $d_{cm} = 0.01724t_{\mu s}$. This returns the distance to the object.

The CEENBoT-API makes implementing this easy using the USONIC subsystem module. Here, the `USONIC_ping()` function will return the ticks elapsed from the time it took for the echo to return after a ping. This is placed into a `SWTIME` type variable and is converted to an `unsigned long int` type variable of the micro-seconds by type-casting the number of ticks and multiplying it by 10. This is then converted to the centimeters with the equation as shown above. Using this distance, the avoidance behavior could be created.

Procedure

First, the sensor was mounted to the CEENBoT. This used a mounting bracket on the front of the device and the sensor was connected to the 5V power and ground lines on the bot. The I/O line for the triggering of a ping and return of ping time was connected to PA3. This was the pin that was used for one of the light sensors inputs to the ADC. Therefore, the light sensor was moved to another one of the ADC channels. Some simple code was written to take distance readings and display them to the LCD.

Then several readings were taken to check that the sensor was accurate. These were taken at 5cm, 30cm and 60cm. The speed of sound in air is dependent on temperature, so depending on the environment, the velocity component of the distance calculation may need to be tweaked until the measurements are accurate. During the lab there was no need to do

this as the distance readings were found to be accurate. A table of these can be found in the results section.

A last test was done before adding this sensing and avoid behavior to the CEENBoT. The distance was recorded at 12in or 30.5 cm from the wall at 90-degrees. Then the angle was decreased, and the distance was recorded again. The table for this can also be found in the results section along with answering some questions.

Next, the existing skeleton code was modified to include the ultrasonic sonar avoidance. The pinging and distance calculation were added to the sensing function, and the ultrasonic avoidance behavior was created. The avoidance was created so it would be triggered under 2ft and the closer to the object, the harder the bot will turn to avoid it. This was implemented into the arbitration scheme in the second highest priority. From highest to lowest:

1. IR avoidance
2. Ultrasonic avoidance
3. Light following
4. Cruising

The program was then demonstrated with the LCD displaying the state data for debugging.

Source Code Discussion

This lab, the ultrasonic distance sensor was added. This used the USONIC subsystem module to get the distance data. As with most subsystem modules, the ultrasonic on must be opened with the `USONIC_open()` function. This used the calculations as shown in the background section and the `USONIC_ping()` function to get the distance data. This data was then used in the `US_avoid()` behavior function to avoid objects.

The `US_avoid()` behavior would trigger if the distance is under 2ft. Then, it takes 24in minus the measured distance to get the ‘error’. This multiplied by a constant kp is added to one motor speed and subtracted from the other so the turn will be away from the object. This way if the object is closer, the bot will turn harder to avoid it.

Results

Distance	Measured value
2in	1.97in
12in	11.42in
24in	24.02in

Table 1: Distance from the wall versus the values measured from the ultrasonic sensor.

As can be seen in Table 1. The sensor was fairly accurate with the given speed of sound. The largest error was only roughly half an inch allowing for a good avoidance behavior. As the data was good, the distance calculation was kept the same.

Angle to wall	Distance measured
90*	30cm
60*	29.3cm
30*	24cm
20*	NULL

Table 2: Angle versus the distance measured to the wall at 30 centimeters to the wall.

Table 2 shows the difficulties created when large angles between the object's surface and the sensor occur. As the angle increases, the distance measurements have greater and greater errors until eventually at about 20* the object cannot be detected.

The main difficulty that was overcome to get the ultrasonic sensor to work was the mounting. The mounting was important as with bad angles, the measurements become inaccurate. It was difficult to get a steady mount with the hardware which was part of the difficulty. Eventually the mounting was solidified and tightened down with the hardware. The other difficulty was in creating the avoidance as a servo behavior and using this to make the avoidance better. The servo behavior problem was solved so the closer to the object, the greater the difference in moto speeds would be allowing the bot to turn harder when the object is closer and less when it is farther away so everything could be avoided.

In the skeleton code, the sensing was done with the other sensing functions and the distance was passed into the sensor data structure. This was used in the ultrasonic avoid behavior which was triggered when the distance was under 2ft. This avoidance function would then change the motor speed values to make the bot turn around the object. As this was the second to highest priority, only the IR avoidance function was called after it. This

allows the IR avoidance to overwrite the motor speed values the ultrasonic avoidance function set to avoid using the IR sensors.

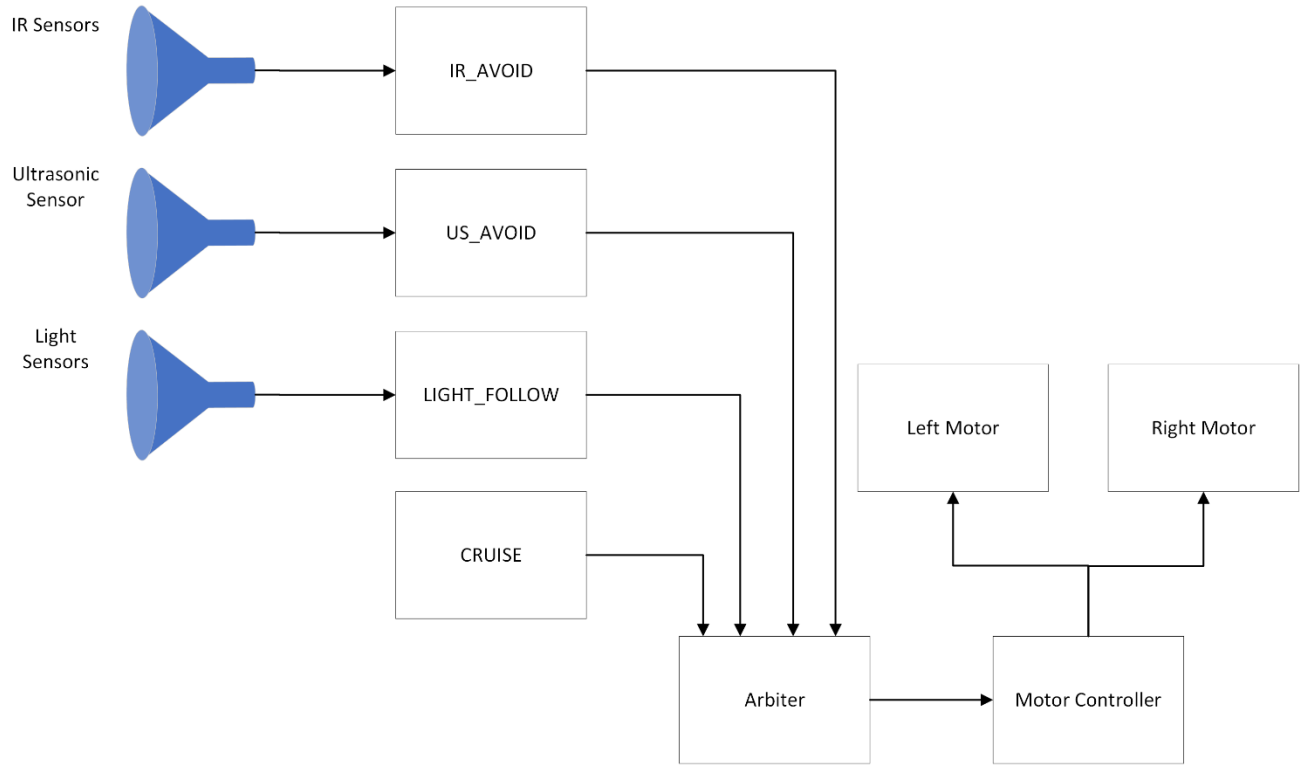

Figure 1: Behavior diagram from highest priority to lowest of the behaviors used in this lab

.

Conclusion

In conclusion, this lab provided an introduction to using an ultrasonic distance sensor to create an avoidance behavior to be used in the bbc skeleton code. This lab provided a good experience for future labs utilizing the ultrasonic sensor for more applications, and a more complex robot program will be completed. This lab worked well and gave us the information necessary for future labs with an important sensor as well.

Source Code

```
// Behavior-Based Control Skeleton code.
//
// Desc: Provides a C program structure that emulates multi-tasking
and
//       modularity for Behavior-based control with easy
scalability.
//
// Supplied for: Students of Mobile Robotics I, Fall 2013.
// University of Nebraska-Lincoln Dept. of Computer & Electronics
Engineering
// Alisa N. Gilmore, P.E., Instructor, Course Developer.  Jose
Santos, T.A.
// Version 1.3  Updated 10/11/2011.
// Version 1.4  Updated 12/2/2013
//
//      - Updated __MOTOR_ACTION() macro-function to invoke new
functions
//        added to the API: `STEPPER_set_accel2()' and
`STEPPER_run2()'.
//        In particular `STEPPER_run2()' now takes positive or
negative
//        speed values to imply the direction of each wheel.
//
// Version 1.5  Updated 2/25/2015
//
//      - Fixed an error in __MOTOR_ACTION() and __RESET_ACTION()
macros
//        where there was an extra curly brace ({) that should have
been
```

```
//        removed.
//
// Version 1.6  Updated 2/24/2016
//
//      - In the 'IR_sense()' function, we now make use of the
'TIMER_ALARM()'
//        and 'TIMER_SNOOZE()' macros that were introduced in API
version 2.x,
//        which makes usage of the timer object clear.   Before,
students
//        manipulated the 'tc' flag inside the timer object
directly, but this
//        always caused confusion, the 'TIMER_ALARM()' and
'TIMER_SNOOZE()'
//        macros achieve the same thing, but transparently.
//
//      - Also fixed __MOTOR_ACTION() macro, which previously
invoked
//        'STEPPER_run2()', but the API has been modified so that
this same
//        effect is now achieved with the function 'STEPPER_runn()',
which
//        means to run the stepper motors and allow negative values
to mean
//        reverse motion (that's what the second 'n' in 'runn'
stands for).
//        This might change again in the future, because for
consistency in the
//        API, it should have been called 'STEPPER_runn2()' since it
takes two
//        parameters and not just one.
//
```

```
//        LAB 6
//
#include "capi324v221.h"
// ---------------------- Defines:
#define DEG_90 150 /* Number of steps for a 90-degree (in place)
turn. */


// Desc: This macro-function can be used to reset a motor-action
structure
// easily. It is a helper macro-function.
#define __RESET_ACTION( motor_action ) \
do { \
( motor_action ).speed_L = 0; \
( motor_action ).speed_R = 0; \
( motor_action ).accel_L = 0; \
( motor_action ).accel_R = 0; \
( motor_action ).state = STARTUP; \
} while( 0 ) /* end __RESET_ACTION() */
// Desc: This macro-fuction translates action to motion -- it is a
helper
// macro-function.
#define __MOTOR_ACTION( motor_action ) \
do { \
STEPPER_set_accel2( ( motor_action ).accel_L, ( motor_action
).accel_R ); \
STEPPER_runn( ( motor_action ).speed_L, ( motor_action ).speed_R );
\
} while( 0 ) /* end __MOTOR_ACTION() */

// Behavior value
```

```
typedef enum ROBOT_STATE_TYPE {
STARTUP = 0, // 'Startup' state -- initial state upon RESET
EXPLORING, // 'Exploring' state -- the robot is 'roaming around'.
AVOIDING, // 'Avoiding' state -- the robot is avoiding a collision.
HOMING, // 'Homing' state -- thef robot is moving toward some
stimuli
OBSERVE,
USAVOIDL,
USAVOIDR
} ROBOT_STATE;
// Desc: Structure encapsulates a 'motor' action. It contains
parameters that
// controls the motors 'down the line' with information depicting
the
// current state of the robot. The 'state' variable is useful to
// 'print' information on the LCD based on the current 'state', for
// example.
typedef struct MOTOR_ACTION_TYPE {
ROBOT_STATE state; // Holds the current STATE of the robot.
signed short int speed_L; // SPEED for LEFT motor.
signed short int speed_R; // SPEED for RIGHT motor.
unsigned short int accel_L; // ACCELERATION for LEFT motor.
unsigned short int accel_R; // ACCELERATION for RIGHT motor.
} MOTOR_ACTION;
typedef struct SENSOR_DATA_TYPE {
BOOL left_IR; // Holds the state of the left IR.
BOOL right_IR; // Holds the state of the right IR.
float left_PR; // Holds the value of the left photoresistor
float right_PL; // Holds the value of the right photoresistor
float left_PR_CAL;// Holds the light threshold for the left
photoresistor
```

```
float right_PR_CAL;// Holds the light threshold for the right
photoresistor
float US_distacne_cm;
} SENSOR_DATA;

volatile MOTOR_ACTION action; // Holds parameters that determine the
current action that is taking place.



void sense( volatile SENSOR_DATA *pSensors, TIMER16 interval_ms );
void cruise( volatile MOTOR_ACTION *pAction );
void IR_avoid( volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors);
void light_follow( volatile MOTOR_ACTION *pAction, volatile
SENSOR_DATA *pSensors );
void light_observe( volatile MOTOR_ACTION *pAction, volatile
SENSOR_DATA *pSensors );
void act( volatile MOTOR_ACTION *pAction );
void info_display( volatile MOTOR_ACTION *pAction, volatile
SENSOR_DATA*pSensors);
BOOL compare_actions( volatile MOTOR_ACTION *a, volatile
MOTOR_ACTION *b );
void copy_motor( volatile MOTOR_ACTION *a, volatile MOTOR_ACTION *b
);
void calibrate(volatile SENSOR_DATA *pSensors);
void US_avoid(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors);
// ---------------------- Convenience Functions:
void info_display( volatile MOTOR_ACTION *pAction, volatile
SENSOR_DATA *pSensors)
{
```

```
static ROBOT_STATE previous_state = STARTUP;
if ( ( pAction->state != previous_state ) || ( pAction->state ==
STARTUP
) )
{
LCD_clear();
// Display robot behavior
switch( pAction->state )
{
case STARTUP: LCD_printf( "Starting...\n" ); break;
case EXPLORING: LCD_printf( "Exploring...\n" ); break;
case AVOIDING: LCD_printf( "Avoiding...\n" ); break;
case HOMING: LCD_printf( "Homing...\n" ); break;
case OBSERVE: LCD_printf("Observe...\n"); break;
case USAVOIDL : LCD_printf("US Avoiding L...\n"); break;
case USAVOIDR : LCD_printf("US Avoiding R...\n"); break;
default: LCD_printf( "Unknown state!\n" );
} // end switch()
// Return the state
previous_state = pAction->state;
} // end if()
}
// ------------------------------------------------------ //
BOOL compare_actions( volatile MOTOR_ACTION *a, volatile
MOTOR_ACTION *b )
{
BOOL rval = TRUE;
if ( ( a->state != b->state ) ||
( a->speed_L != b->speed_L ) ||
( a->speed_R != b->speed_R ) ||
( a->accel_L != b->accel_L ) ||
( a->accel_R != b->accel_R ) )
```

```
rval = FALSE;
// Return comparison result.
return rval;
} // end compare_actions()
void copy_motor( volatile MOTOR_ACTION *a, volatile MOTOR_ACTION *b
)
{
if(!compare_actions(a, b))// Compare to see if they are already the
same
{
a->accel_L = b->accel_L;
a->accel_R = b->accel_R;
a->speed_L = b->speed_L;
a->speed_R = b->speed_R;
a->state = b->state;
} // end if()
} // end copy_motor
// Light and ultrasonic sensors calibration
void calibrate(volatile SENSOR_DATA *pSensors)
{
// Calibrate the left light sensor
ADC_set_channel(ADC_CHAN5);
pSensors->left_PR_CAL = ADC_sample();
// Calibrate the right light sensor
ADC_set_channel(ADC_CHAN6);
pSensors->right_PR_CAL = ADC_sample();
// Set Ultrasonic heading
ATTINY_set_RC_servo(0, 1400);
} // end calibrate()

// Top-Level Behaviors:
void sense( volatile SENSOR_DATA *pSensors, TIMER16 interval_ms )
```

```
{
// Sense must know if it's already sensing.
//
// NOTE: 'BOOL' is a custom data type offered by the CEENBoT API.
//
static BOOL timer_started = FALSE;
// Sense timer is here to control the frequency of information about
sensors. The rate at which this happens must be controlled. We are
therefore required to use TIMER objects with the TIMER SERVICE. It
must be "static" because the timer object must remain even when it
is out of range, otherwise the program will crash.
static TIMEROBJ sense_timer;
unsigned long int usonic_time_us;
SWTIME usonic_time_ticks;
// If this is the FIRST time that sense() is running, we need to
start the
// sense timer. We do this ONLY ONCE!
if ( timer_started == FALSE )
{
// Start timer activate in every interval, we use then is this
variable interval_ms
TMRSRVC_new( &sense_timer, TMRFLG_NOTIFY_FLAG, TMRTCM_RESTART,
interval_ms );
// Make timer start
timer_started = TRUE;
} // end if()
// Otherwise just sense
else
{
// Ativate sensor when need
if ( TIMER_ALARM( sense_timer ) )
{
```

```
LED_toggle( LED_Green );
pSensors->left_IR = ATTINY_get_IR_state( ATTINY_IR_LEFT);
pSensors->right_IR = ATTINY_get_IR_state( ATTINY_IR_RIGHT);
// light sensor data
ADC_set_channel(ADC_CHAN5);
pSensors->left_PR = pSensors->left_PR_CAL - ADC_sample();
// right light sensors
ADC_set_channel(ADC_CHAN6);
pSensors->right_PL = pSensors->right_PR_CAL - ADC_sample();
// Ping US
usonic_time_ticks = USONIC_ping();
// Change tick to time
usonic_time_us = 10 * (( unsigned long int)
(usonic_time_ticks));
// Change time to distance
pSensors->US_distacne_cm = 0.006787 * usonic_time_us;
// Snooze the alarm, to make it trigger again.
TIMER_SNOOZE( sense_timer );
} // end if()
} // end else.
} // end sense()
// ------------------------------------------- //
void cruise( volatile MOTOR_ACTION *pAction )
{
// Nothing really changing from the other lab for the cruise
fonction, the most impotant action will be done by act
pAction->state = EXPLORING;
pAction->speed_L = 200;
pAction->speed_R = 200;
pAction->accel_L = 400;
pAction->accel_R = 400;
} // end explore()
```

```
// With the light value determine if he should home
void light_follow(volatile MOTOR_ACTION *pAction, volatile
SENSOR_DATA *pSensors)
{
// set up variables
float error;
float p = 0.0;
const float kp = 0.35;
// If light sensed > value -> homing
if ((pSensors->left_PR + pSensors->right_PL) / 2 > 30)
{
// Motors -> homing
pAction->state = HOMING;
// Difference between the two light sensors
error = (pSensors->left_PR - pSensors->right_PL);
// Multiply by constant to use it
p = error * kp;
// Change speed
pAction->speed_L = pAction->speed_L - p;
pAction->speed_R = pAction->speed_R + p;
// More information in the "Mobile Robotics I Module 8 In Class.pdf"
} // end if()
// Print values of light sensors for debugging purposes
LCD_printf_RC(1, 1, "Left: %f", pSensors->left_PR);
LCD_printf_RC(2, 1, "Right: %f", pSensors->right_PL);
} // end light_follow()
// We add that if the value are really back from before then turn
the robot
void light_observe( volatile MOTOR_ACTION *pAction, volatile
SENSOR_DATA *pSensors )
{
// If the sensor are under the value
```

```
if ((pSensors->left_PR > 800) && (pSensors->right_PL > 800))
{
// Mode : OBSERVE
pAction->state = OBSERVE;
// Make the motors stop moving
STEPPER_stop( STEPPER_BOTH, STEPPER_BRK_OFF ); // Stop motors
info_display(pAction, pSensors); // Refresh display
// Back !
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, 200, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, 200, 200, 400, STEPPER_BRK_OFF );
// Turn RIGHT 180*
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_FWD, DEG_90 * 2, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, DEG_90 * 2, 200, 400, STEPPER_BRK_OFF );
// Normal movement
pAction->speed_L = 200;
pAction->speed_R = 200;
pAction->accel_L = 400;
pAction->accel_R = 400;
}
} // end light_observe()


// If one of the IR  sensor is triggered then avoid or run away the
obstacle
void IR_avoid( volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors )
{
if ((pSensors->left_IR == TRUE || pSensors->right_IR == TRUE))
{
STEPPER_stop( STEPPER_BOTH, STEPPER_BRK_OFF ); // Stop movement
```

```
pAction->state = AVOIDING; // Mode : avoiding
info_display(pAction, pSensors);
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, 200, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, 200, 200, 400, STEPPER_BRK_OFF );
// If 2 IR sensors are triggered
if (pSensors->left_IR == TRUE && pSensors->right_IR == TRUE)
{
// Turn RIGHT 180*
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_FWD, DEG_90 * 2, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, DEG_90 * 2, 200, 400, STEPPER_BRK_OFF );
// If left IR sensors is triggered
} else if( pSensors->left_IR == TRUE )
{
// Turn RIGHT 90*
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_FWD, DEG_90, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, DEG_90, 200, 400, STEPPER_BRK_OFF );
// If right sensor is triggered
} else if (pSensors->right_IR == TRUE)
{
// Turn LEFT 90*
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, DEG_90, 200, 400, STEPPER_BRK_OFF,
STEPPER_FWD, DEG_90, 200, 400, STEPPER_BRK_OFF );
} // end if()
// Resume normal movement
pAction->speed_L = 200;
pAction->speed_R = 200;
pAction->accel_L = 400;
pAction->accel_R = 400;
```

```
} // end if()
} // end avoid()
void US_avoid(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors)
{
// variables
float error;
float p = 0.0;
const float kp = 9;
unsigned int heading = 1000;
// Avoid if the US sensor feel something at 24 inch
if (pSensors->US_distacne_cm < 24)
{
// turn away from object varying the speed based on how far away the
object is
error = pSensors->US_distacne_cm - 24;
p = error * kp;
// If sense left side turn right
if (heading > 1400)
{
pAction->state = USAVOIDL;
pAction->speed_L = pAction->speed_L - p;
pAction->speed_R = pAction->speed_R + p;
// If sense right side turn left
} else
{
pAction->state = USAVOIDR;
pAction->speed_L = pAction->speed_L + p;
pAction->speed_R = pAction->speed_R - p;
}
}
}
```

```
// ------------------------------------------ //
void act( volatile MOTOR_ACTION *pAction )
{
// act track of the past action to determine if a new action must be
execute . change only if MOTOR ACTION chg
static MOTOR_ACTION previous_action = {
STARTUP, 0, 0, 0, 0
};
if( compare_actions( pAction, &previous_action ) == FALSE )
{
__MOTOR_ACTION( *pAction );
previous_action = *pAction;
} // end if()
} // end act()
// ---------------------- CBOT Main:
void CBOT_main( void )
{
volatile SENSOR_DATA sensor_data;
LED_open();
LCD_open();
STEPPER_open();
ADC_open();
USONIC_open();
ADC_set_VREF(ADC_VREF_AVCC);
__RESET_ACTION( action );
LCD_printf( "Starting...\n" );\
calibrate(&sensor_data);
TMRSRVC_delay( TMR_SECS( 3 ) );
LCD_clear();
while( 1 )
{
sense( &sensor_data, 125 );
```

```
// Behaviors
cruise( &action );
light_follow(&action, &sensor_data);
light_observe(&action, &sensor_data);
US_avoid(&action, &sensor_data);
IR_avoid( &action, &sensor_data );
act( &action );
info_display( &action, &sensor_data );
} // end while()
} // end CBOT_main()
```