ECEN 345/ECEN 3450-001

Mobile Robotics 1

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

Laboratory Assignment #9

Robotic Vision with the Pixy (CMUcam5)

Written by
Hayden Ernst
Nelson Lefebvre

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

Due: 12/4/2022

Introduction

The purpose of this lab was to learn about the Pixy (CMUcam5) camera and utilize it with a proportional control program to allow the CEENBoT to follow and stay close to a colored object. This would be done by first learning about and understanding how to use the CMUcam for color tracking. A feedback-controlled behavior could be created utilizing the color-tracking data gained from the camera. This can be implemented into Proportional, Derivative, and Integral control methods and then fine tuned to gain the best performance.

This lab required roughly 5 hours of work. This was evenly distributed between partners. Roughly one hour was spent working with the CMUcam, working to get it to function correctly with the CEENBoT. 30 more minutes were spent understanding the information with another hour was spent planning the feedback controller. Another hour was spent in the lab creating a preliminary program. The final hour and 30 minutes were spent outside of lab finishing the feedback control and tuning it. The main resources used in this lab were the lab handout as well as the lectures and notes. The CMUcam manual was another important resource. These resources gave a guide for how to mount and make use of the CMUcam as well as how to program with a feedback controller. 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 camera 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 utilizes feedback control. This has been experimented with in previous labs, so information regarding this can be found in those lab reports. The background section will cover the CMUcam information. The CMUcam uses UART communication to send information between it and the CEENBoT. The UART RX of the camera is connected to TXD1 of the CEENBoT on pin 15 of J3. The UART TX of the camera is connected to RXD1

of the CEENBoT on pin 14 of J3. Then the power pin of the camera can be connected to +5V on the CEENBoT and the ground connected to a ground pin. The camera then needs to be mounted over the CEENBoT. According to the guide, this should be 5 to 7 in above the ground with a 45-degree angle to the ground for the best tracking detection.

The Pixy camera uses the Pixymon app to configure the camera. The Pixy camera can be connected to the computer via a USB cable. Then the configuration parameters can be set. The 'Data out port' can be set up here. For the purposes of working with the CEENBoT it should be set to correspond to the UART port. The UART baud rate and I2C address can also be set up here. Again, for the purposes of this lab, the I2C address isn't important, and the UART baud rate should be set to 115200 to interface correctly with the CEENBoT.

There may also be a Blocks setting option. Here, the max number of color signatures as well as the max number of blocks per signature can be configured. The max number of signatures can be set to whatever, but it should be relatively low. The max number of blocks per signature should be set to one so the program doesn't become confused if multiple returns are found.

The Pixy camera needs to be trained on what colors to track. This can be done in the app by selecting action and a signature to set. Once this is selected, the frame will be frozen. Then the user can click and drag to select a region of pixels corresponding to the colors to track. This can be done for up to seven colors which will then be tracked.

Using the CEENBoT api, the data from the camera can be easily accessed and used. The submodule is opened with the `PIXY_open()` function. This will return true if the submodule opens successfully and the camera responds. The `PIXY_register_callback()` function is used to set any function to be called as well as variables to be set when the camera sends new data. This requires a function name as a parameter as well as a pointer to a `PIXY_DATA` struct. Either one can be set to `NULL` so nothing will be done when the data is received. The camera tracking is started with the `PIXY_track_start()` function. Once this is done, the `PIXY_has_data()` function can be used to check if the camera has sent new data. It will return true if so. The `PIXY_process_finishes()` should be called to let the camera know it is ok to write new data. Note, these two functions should only be used in polling mode, not when a callback

function is set. The `PIXY_DATA` struct type holds several different pieces of data, including the checksum of the transmission (which is not necessary for this lab program), signature number (signum), as well as two nested structs. The first nested struct is the signature's position (pos) in x and y. The width and height of the images the camera takes are 320 by 240. The next struct is the object's size (size) and includes a width and height.

With an understanding of how to set up the camera as well as how to interact with it in the program and access the data received from it, the lab could be completed.

Procedure

First, the camera was mounted onto the CEENBoT over the main board using the mounting hardware provided. It was then connected to the CEENBoT utilizing the instructions as explained roughly in the background. Once this was done, the Pixymon utility was downloaded and used to configure the camera to work with the CEENBoT. It was also to set a green and orange color signature. Some experimentation was done here with the different color signatures and the information received by the program to gain a good understanding of how it worked. Once the setup was complete the test example program from the Pixy guide was uploaded and used to show the correct object tracking as well as how this may be used to create the object tracking program.

Next, the BBC-based program provided was modified to work with the camera to track and follow objects in order to keep them in the center of the camera frame. Some testing and implementation was done with each level until a working program was created.

Source Code Discussion

As explained in the background section, The Pixy submodule is opened with the `PIXY_open()` function. This will return true if the submodule opens successfully and the camera responds. The `PIXY_register_callback()` function is used to set any function to be called as well as variables to be set when the camera sends new data. This requires a function name as a parameter as well as a pointer to a `PIXY_DATA` struct. Either one can be set to `NULL` so nothing will be done when the data is received. The camera tracking is started with the `PIXY_track_start()` function. Once this is done, the `PIXY_has_data()` function can be used to check if the camera has sent new data. It will

return true if so. The `PIXY_process_finishes()` should be called to let the camera know it is ok to write new data. Note, these two functions should only be used in polling mode, not when a callback function is set. The `PIXY_DATA` struct type holds several different pieces of data, including the checksum of the transmission (which is not necessary for this lab program), signature number (signum), as well as two nested structs. The first nested struct is the signature's position (pos) in x and y. The width and height of the images the camera takes are 320 by 240. The next struct is the object's size (size) and includes a width and height.

The program built for this lab utilizes the position data of the object to track where it is located and the error from the center of the screen to make the adjustments necessary to move to the correct position. The two variables declared below are the error in x and y. They are generated by taking the difference in the center and the actual coordinates and as the comments state, above the center and to the right is positive.

```
int Abs_X = pSensors -> pixy_data.pos.x - CENTER_X;  // Right of
center is positive
int Abs_Y = -(pSensors -> pixy_data.pos.y - CENTER_Y);  // Below
center is negative
```

Below, the speed sets the forward and backwards movement. Here the proportional constant (kpy) is set to two. This is only used with the values of error in y. Then the base speed is set to zero so there won't be any forwards or reverse movement unless the object is not centered.

```
// Speed
int kpy = 2;
int base_speed = 0;  // Bot is not moving by default
//int base_speed = 100;
```

The turn values below generate any turn in the x direction. The proportional constant (kpx) is set to one and the base turn is set to zero. Below these values, a dead zone is setup around the center by 15, so if the error is inside of this, there won't be any action taken as it is close enough to the center.

```
// Turn values
int kpx = 1;
int turn = 0; // Bot goes straight by default

// Dead zone logic
int zoneX = 15;
int zoneY = 15;
```

Below, the turn and base speed values are set by the x and y and their respective proportional constants. If the object is to the right and up, both will be set positive; the opposite is true if the object is to the left and down. Further below that, the motor speeds are set. If both are positive, the base speed will be forwards with the left motor set faster than the right. If both are negative, the base speed will be backwards with the right motor set at a slower backwards speed. This should theoretically allow for the object tracking.

```
if ( abs(Abs_X) > zoneX ) {
// CEENBoT needs to turn left or right
turn = kpx * Abs_X;
}
if ( abs(Abs_Y) > zoneY ) {
// CEENBoT needs to move forward or back
base_speed = kpy * Abs_Y;
}

pAction->speed_L = base_speed + turn;
pAction->speed_R = base_speed - turn;
```

Results

- **Discuss how you implemented the object-following behavior: discuss development iterations you went to realize this task.**

  The error in x was set to a turn variable while the error in y was set to a base speed variable. This way if y was negative, a reverse direction should be set and if it was positive a forward direction would be set. x would work as intended for right or left movement. First, the y movement was tested with the bot able to move forwards to the object and stop when it was in the center established, the right and left following was established.

- **Discuss the feedback control mechanism YOU used in achieving the object-following task (i.e., PID components), and the performance improvements achieved by adding each.**

  The proportional controller was the only one utilized for this lab as it seemed to work well and an integral and derivative control element seemed to be difficult to implement correctly.

- **Discuss the overall performance of your Camera_follow() behavior, and the hurdles you found necessary to overcome. (Do NOT state you did not have any hurdles - everyone has them, big or small).**

  The performance was good for the forwards following as well as any right or left movement, however, it struggled in reversing. This wasn't initially tested as we didn't realize the requirement for it. Then when we did try it, we could not make it reverse even though the code suggested it would. We hypothesized that this could be due to the camera not having a negative y coordinate as the object got closer, instead it may just become bigger which could be an issue with trying to simulate depth of field.

- **Discuss the chosen object you chose for color detection and what attributes it had that you felt would give you improved color tracking (size, texture, color, etc).**
  We chose a green cylinder as it was one of the available options when testing in lab. The camera didn't seem to have vary many difficulties acquiring the color for tracking. However, reflecting on it, the size may have contributed to the reversing problem. The cylindrical shape also caused some shading differences on the object if a strong light source was placed perpendicular to the line it and the camera were on. This didn't seem to cause many difficulties in testing or demonstration as the areas used had uniform lighting. However, this may have caused issues in different environments, and it should be a consideration if multiple environments are expected.

- **Discuss whether your chosen detection object was relatively easy for the Pixy to track, or had challenges or difficulties doing so. Did you have to change objects? Colors? What worked? Why? Why didn't it worked?**
  As stated above, it initially appeared that the object worked well with the tracking. The difference in brightness caused by a single brighter light source was avoided due to the lab testing environment. The size of the object wasn't considered until after the demonstration had been done, however it could have contributed to the reversing issue mentioned previously.

- **Discuss whether you experimented with the mounting angle of the Pixy and made any attempt to see if detection of your chosen object was detectable at larger distances.**
  The guides specifically stated that a roughly 45 degree angle should be the best for object tracking, so not much work was done regarding the angle of the camera.

- **Discuss whether, during experimentation (in Part I), object tracking was consistent regardless of changes in light, or location in the lab, or whether tracking was easier under certain light conditions (and locations), versus others.**
  Object tracking under the ideal conditions of the lab made it easy, however, if the objects were taken out of lab and used in different environments, lighting could have a large effect on the ability of the camera to track the object. This was not observed with the robot, but rather on the Pixymon utility.

- **In the Conclusion section of your report: Provide any suggestions that you feel would help to make this particular lab exercise better.**

Conclusion

In conclusion, this lab provided experience in feedback control with multiple dimensions as well as computer vision in the form of the Pixy (CMUcam5) camera. This lab allowed for work with object tracking with different colors, shapes, lighting, and different environments. Overall, this gave a good experience on object tracking as well as feedback control in multiple directions. A large difficulty faced in the lab was in regard to the movement forward and reverse as well as side to side. Perhaps one suggestion to make this lab exercise better is a little more guidance on how to make this work better. Other than this there weren't that many complaints on the lab as a whole. But it was our fault for missing the reverse section and not doing as much testing as we should have. As a whole, this will be good experience for future applications where feedback control can be used. As feedback control is a useful tool in any sort of robotics it may be encountered many times and the experience with different implementations of it is quite valuable.

Source Code

```
#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() */

// Desc: This macro-function is used to set the action, in a more
natural
//       manner (as if it was a function).
```

```
// ---------------------- Type Declarations:

// Desc: The following custom enumerated type can be used to specify
the
//       current state of the robot.  This parameter can be expanded
upon
//       as complexity grows without intefering with the 'act()'
function.
//         It is a new type which can take the values of 0, 1, or 2
using
//         the SYMBOLIC representations of STARTUP, EXPLORING, etc.
typedef enum ROBOT_STATE_TYPE {

STARTUP = 0,    // 'Startup' state -- initial state upon RESET.
CRUISING,      // 'Exploring' state -- the robot is 'roaming
around'.
AVOIDING,        // 'Avoiding' state -- the robot is avoiding a
collision.
CAMERAFOLLOW,       // 'Following' state -- the robot is following a
Pixy object

} 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;

// Desc: Structure encapsulates 'sensed' data.  Right now that only
consists
//       of the state of the left & right IR sensors when queried.
You can
//       expand this structure and add additional custom fields as
needed.
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.
PIXY_DATA pixy_data;    // Holds relevant PIXY tracking data.

// *** Add your -own- parameters here.

} SENSOR_DATA;

// ---------------------- Globals:
volatile MOTOR_ACTION action;      // This variable holds parameters
that determine
// the current action that is taking place.
// Here, a structure named "action" of type
```

```
// MOTOR_ACTION is declared.

// ---------------------- Prototypes:
void IR_sense( volatile SENSOR_DATA *pSensors, TIMER16 interval_ms
);
void cruise( volatile MOTOR_ACTION *pAction );
void pixy_process( volatile MOTOR_ACTION *pAction,
volatile SENSOR_DATA *pSensors );

void IR_avoid( volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors );
void act( volatile MOTOR_ACTION *pAction );
void info_display( volatile MOTOR_ACTION *pAction );
BOOL compare_actions( volatile MOTOR_ACTION *a, volatile
MOTOR_ACTION *b );

// ---------------------- Convenience Functions:
void info_display( volatile MOTOR_ACTION *pAction )
{

// NOTE:  We keep track of the 'previous' state to prevent the LCD
//        display from being needlessly written, if there's  nothing
//        new to display.  Otherwise, the screen will 'flicker' from
//        too many writes.
static ROBOT_STATE previous_state = STARTUP;

if ( ( pAction->state != previous_state ) || ( pAction->state ==
STARTUP ) )
{

LCD_clear();
```

```
//  Display information based on the current 'ROBOT STATE'.
switch( pAction->state )
{

case STARTUP:
LCD_printf( "Starting...\n" );
break;

case CRUISING:
LCD_printf( "Exploring...\n" );
break;

case AVOIDING:
LCD_printf( "Avoiding...\n" );
break;

case CAMERAFOLLOW:
LCD_printf( "CAMERAFOLLOW...\n" );
break;

default:
LCD_printf( "Unknown state!\n" );

} // end switch()

// Note the new state in effect.
previous_state = pAction->state;

} // end if()

} // end info_display()
```

```
BOOL compare_actions( volatile MOTOR_ACTION *a, volatile
MOTOR_ACTION *b )
{

// NOTE:  The 'sole' purpose of this function is to
//        compare the 'elements' of MOTOR_ACTION structures
//        'a' and 'b' and see if 'any' differ.

// Assume these actions are equal.
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()

// ---------------------- Top-Level Behaviorals:
void IR_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;

// The 'sense' timer is used to control how often gathering sensor
// data takes place.  The pace at which this happens needs to be
// controlled.  So we're forced to use TIMER OBJECTS along with the
// TIMER SERVICE.  It must be 'static' because the timer object must
remain
// 'alive' even when it is out of scope -- otherwise the program
will crash.
static TIMEROBJ sense_timer;

// 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 the 'sense timer' to tick on every 'interval_ms'.
//
// NOTE:  You can adjust the delay value to suit your needs.
//
TMRSRVC_new( &sense_timer, TMRFLG_NOTIFY_FLAG, TMRTCM_RESTART,
interval_ms );

// Mark that the timer has already been started.
timer_started = TRUE;

} // end if()

// Otherwise, just do the usual thing and just 'sense'.
else
{
```

```
// Only read the sensors when it is time to do so (e.g., every
// 125ms).  Otherwise, do nothing.
if ( TIMER_ALARM( sense_timer ) )
{

// NOTE: Just as a 'debugging' feature, let's also toggle the green
LED
//       to know that this is working for sure.  The LED will only
//       toggle when 'it's time'.
LED_toggle( LED_Green );


// Read the left and right sensors, and store this
// data in the 'SENSOR_DATA' structure.
pSensors->left_IR  = ATTINY_get_IR_state( ATTINY_IR_LEFT  );
pSensors->right_IR = ATTINY_get_IR_state( ATTINY_IR_RIGHT );

// NOTE: You can add more stuff to 'sense' here.

// Snooze the alarm so it can trigger again.
TIMER_SNOOZE( sense_timer );

} // end if()

} // end else.

} // end sense()
// ------------------------------------------- //
void cruise( volatile MOTOR_ACTION *pAction )
{
```

```
// Nothing to do, but set the parameters to explore.  'act()' will
do
// the rest down the line.
pAction->state = CRUISING;
pAction->speed_L = 100;
pAction->speed_R = 100;
pAction->accel_L = 250;
pAction->accel_R = 250;

// That's it -- let 'act()' do the rest.

} // end explore()
// ------------------------------------------- //
void pixy_process( volatile MOTOR_ACTION *pAction,
volatile SENSOR_DATA *pSensors )
{

// Only process this behavior -if- there is NEW data from the pixy.
if( PIXY_has_data() )
{
pAction->state = CAMERAFOLLOW;

int CENTER_X = 160;
int CENTER_Y = 120;
int Abs_X = pSensors -> pixy_data.pos.x - CENTER_X;  // Right of
center is positive
int Abs_Y = -(pSensors -> pixy_data.pos.y - CENTER_Y);  // Below
center is negative



// Speed
```

```
int kpy = 2;
int base_speed = 0;  // Bot is not moving by default
//int base_speed = 100;

// Turn values
int kpx = 1;
int turn = 0; // Bot goes straight by default

// Dead zone logic
int zoneX = 15;
int zoneY = 15;



if ( abs(Abs_X) > zoneX ) {
// CEENBoT needs to turn left or right
turn = kpx * Abs_X;
}
if ( abs(Abs_Y) > zoneY ) {
// CEENBoT needs to move forward or back
base_speed = kpy * Abs_Y;
}

pAction->speed_L = base_speed + turn;
pAction->speed_R = base_speed - turn;

PIXY_process_finished();

} // end if()

} // end pixy_process()
// ------------------------------------------ //
```

```
void IR_avoid( volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors )
{

// NOTE: Here we have NO CHOICE, but to do this 'ballistically'.
//       **NOTHING** else can happen while we're 'avoiding'.

if( pSensors->right_IR == TRUE && pSensors->left_IR == TRUE)
{
pAction->state = AVOIDING;
LCD_clear();
LCD_printf( "AVOIDING...\n");

STEPPER_stop(STEPPER_BOTH, STEPPER_BRK_OFF);

// Back up...
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, 250, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, 250, 200, 400, STEPPER_BRK_OFF );

// ... and turn LEFT ~90-deg
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, DEG_90, 200, 400, STEPPER_BRK_OFF,
STEPPER_FWD, DEG_90, 200, 400, STEPPER_BRK_OFF);

// ... and set the motor action structure with variables to move
forward.
pAction->speed_L = 200;
pAction->speed_R = 200;
pAction->accel_L = 400;
pAction->accel_R = 400;
}
```

```
// If the LEFT sensor tripped...
else if( pSensors->left_IR == TRUE )
{
pAction->state = AVOIDING;
LCD_clear();
LCD_printf( "AVOIDING...\n");

STEPPER_stop(STEPPER_BOTH, STEPPER_BRK_OFF);

// Back up...
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, 250, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, 250, 200, 400, STEPPER_BRK_OFF );

// ... and turn LEFT ~90-deg
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, DEG_90, 200, 400, STEPPER_BRK_OFF,
STEPPER_FWD, DEG_90, 200, 400, STEPPER_BRK_OFF);

// ... and set the motor action structure with variables to move
forward.
pAction->speed_L = 200;
pAction->speed_R = 200;
pAction->accel_L = 400;
pAction->accel_R = 400;

}
else if( pSensors->right_IR == TRUE)
{
pAction->state = AVOIDING;
LCD_clear();
LCD_printf( "AVOIDING...\n");
```

```
STEPPER_stop(STEPPER_BOTH, STEPPER_BRK_OFF);

// Back up...
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, 250, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, 250, 200, 400, STEPPER_BRK_OFF );

// ... and turn LEFT ~90-deg
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, DEG_90, 200, 400, STEPPER_BRK_OFF,
STEPPER_FWD, DEG_90, 200, 400, STEPPER_BRK_OFF);

// ... and set the motor action structure with variables to move
forward.
pAction->speed_L = 200;
pAction->speed_R = 200;
pAction->accel_L = 400;
pAction->accel_R = 400;
}
} // end avoid()
// ------------------------------------------- //
void act( volatile MOTOR_ACTION *pAction )
{

// 'act()' always keeps track of the PREVIOUS action to determine
// if a new action must be executed, and to execute such action ONLY
// if any parameters in the 'MOTOR_ACTION' structure have changed.
// This is necessary to prevent motor 'jitter'.
static MOTOR_ACTION previous_action = {

STARTUP, 0, 0, 0, 0
```

```
};

if( compare_actions( pAction, &previous_action ) == FALSE )
{

// Perform the action.  Just call the 'free-running' version
// of stepper move function and feed these same parameters.
__MOTOR_ACTION( *pAction );

// Save the previous action.
previous_action = *pAction;

} // end if()

} // end act()
// ---------------------- CBOT Main:
void CBOT_main( void )
{

volatile SENSOR_DATA sensor_data;

// ** Open the needed modules.
LED_open();     // Open the LED subsystem module.
LCD_open();     // Open the LCD subsystem module.
STEPPER_open(); // Open the STEPPER subsystem module.

// Initialize the Pixy subsystem.
if ( PIXY_open() == SUBSYS_OPEN )
{

// Register the pixy structure, but NO callback.  The 'type casting'
```

```
// is to eliminate the warning from 'sensor_data' being declared
// as 'volatile' above.
PIXY_register_callback( NULL, (PIXY_DATA *) &sensor_data.pixy_data
);

// Start tracking.
PIXY_track_start();

} // end if()
else
{

// If the PIXY doesn't open, we can't continue.  This is a FATAL
error.
LCD_clear();
LCD_printf( "FATAL: Pixy failed!\n" );

// Get stuck here forever.
while( 1 );

} // end else.

// Reset the current motor action.
__RESET_ACTION( action );


ATTINY_get_sensors();
LCD_clear();

while( 1 )
{
```

```
cruise( &action );

pixy_process( &action, &sensor_data );

act( &action );

info_display( &action );

DELAY_ms(20);

} // end while()

} // end CBOT_main()
```