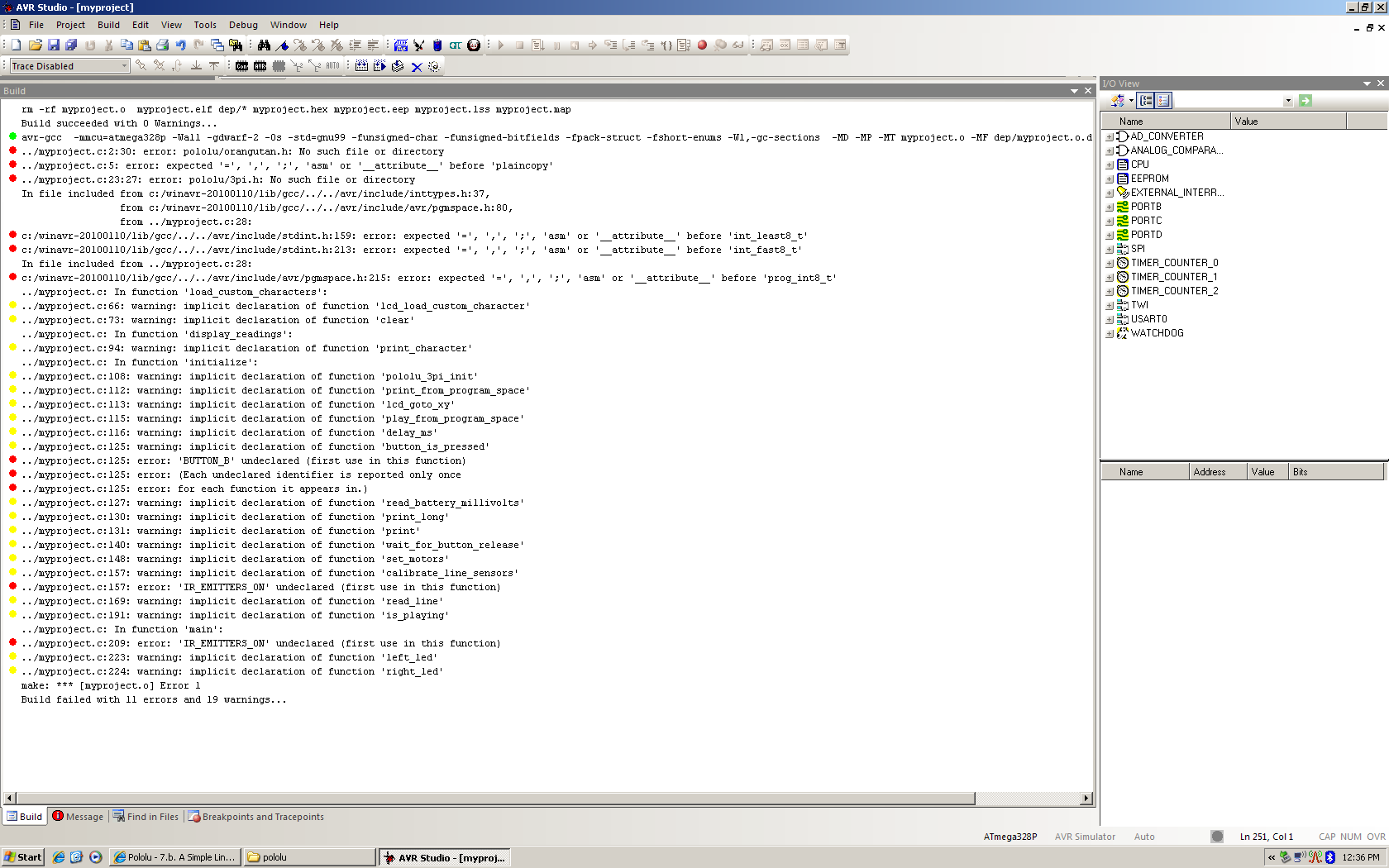
5/13/10

[Resources](http://www.pololu.com/resources) » [Pololu 3pi Robot User's Guide](http://www.pololu.com/docs/0J21) » [7. Example Project #1: Line Following](http://www.pololu.com/docs/0J21/7) »

[**7.b. A Simple Line-Following Algorithm for 3pi**](http://www.pololu.com/docs/0J21/7.b)

I’ve press F7 to Build and this is the error messages I get along with warnings. I’ve done this to numerous example programs and still get errors, can you tell me what I’m doing wrong please? Below is the same code from 7.b A simple Line following Algorithm for 3pi., I couldn’t get a good screen shot, so can you please tell me what I’m doing wrong please. Thank you.



#include <pololu/orangutan.h>

view plaincopy to clipboardprint?

/\*

\* 3pi-linefollower - demo code for the Pololu 3pi Robot

\*

\* This code will follow a black line on a white background, using a

\* very simple algorithm. It demonstrates auto-calibration and use of

\* the 3pi IR sensors, motor control, bar graphs using custom

\* characters, and music playback, making it a good starting point for

\* developing your own more competitive line follower.

\*

\* http://www.pololu.com/docs/0J21

\* http://www.pololu.com

\* http://forum.pololu.com

\*

\*/

// The 3pi include file must be at the beginning of any program that

// uses the Pololu AVR library and 3pi.

#include <pololu/3pi.h>

// This include file allows data to be stored in program space. The

// ATmegaxx8 has 16x more program space than RAM, so large

// pieces of static data should be stored in program space.

#include <avr/pgmspace.h>

// Introductory messages. The "PROGMEM" identifier causes the data to

// go into program space.

const char welcome\_line1[] PROGMEM = " Pololu";

const char welcome\_line2[] PROGMEM = "3\xf7 Robot";

const char demo\_name\_line1[] PROGMEM = "Line";

const char demo\_name\_line2[] PROGMEM = "follower";

// A couple of simple tunes, stored in program space.

const char welcome[] PROGMEM = ">g32>>c32";

const char go[] PROGMEM = "L16 cdegreg4";

// Data for generating the characters used in load\_custom\_characters

// and display\_readings. By reading levels[] starting at various

// offsets, we can generate all of the 7 extra characters needed for a

// bargraph. This is also stored in program space.

const char levels[] PROGMEM = {

0b00000,

0b00000,

0b00000,

0b00000,

0b00000,

0b00000,

0b00000,

0b11111,

0b11111,

0b11111,

0b11111,

0b11111,

0b11111,

0b11111

};

// This function loads custom characters into the LCD. Up to 8

// characters can be loaded; we use them for 7 levels of a bar graph.

void load\_custom\_characters()

{

lcd\_load\_custom\_character(levels+0,0); // no offset, e.g. one bar

lcd\_load\_custom\_character(levels+1,1); // two bars

lcd\_load\_custom\_character(levels+2,2); // etc...

lcd\_load\_custom\_character(levels+3,3);

lcd\_load\_custom\_character(levels+4,4);

lcd\_load\_custom\_character(levels+5,5);

lcd\_load\_custom\_character(levels+6,6);

clear(); // the LCD must be cleared for the characters to take effect

}

// This function displays the sensor readings using a bar graph.

void display\_readings(const unsigned int \*calibrated\_values)

{

unsigned char i;

for(i=0;i<5;i++) {

// Initialize the array of characters that we will use for the

// graph. Using the space, an extra copy of the one-bar

// character, and character 255 (a full black box), we get 10

// characters in the array.

const char display\_characters[10] = {' ',0,0,1,2,3,4,5,6,255};

// The variable c will have values from 0 to 9, since

// calibrated values are in the range of 0 to 1000, and

// 1000/101 is 9 with integer math.

char c = display\_characters[calibrated\_values[i]/101];

// Display the bar graph character.

print\_character(c);

}

}

// Initializes the 3pi, displays a welcome message, calibrates, and

// plays the initial music.

void initialize()

{

unsigned int counter; // used as a simple timer

unsigned int sensors[5]; // an array to hold sensor values

// This must be called at the beginning of 3pi code, to set up the

// sensors. We use a value of 2000 for the timeout, which

// corresponds to 2000\*0.4 us = 0.8 ms on our 20 MHz processor.

pololu\_3pi\_init(2000);

load\_custom\_characters(); // load the custom characters

// Play welcome music and display a message

print\_from\_program\_space(welcome\_line1);

lcd\_goto\_xy(0,1);

print\_from\_program\_space(welcome\_line2);

play\_from\_program\_space(welcome);

delay\_ms(1000);

clear();

print\_from\_program\_space(demo\_name\_line1);

lcd\_goto\_xy(0,1);

print\_from\_program\_space(demo\_name\_line2);

delay\_ms(1000);

// Display battery voltage and wait for button press

while(!button\_is\_pressed(BUTTON\_B))

{

int bat = read\_battery\_millivolts();

clear();

print\_long(bat);

print("mV");

lcd\_goto\_xy(0,1);

print("Press B");

delay\_ms(100);

}

// Always wait for the button to be released so that 3pi doesn't

// start moving until your hand is away from it.

wait\_for\_button\_release(BUTTON\_B);

delay\_ms(1000);

// Auto-calibration: turn right and left while calibrating the

// sensors.

for(counter=0;counter<80;counter++)

{

if(counter < 20 || counter >= 60)

set\_motors(40,-40);

else

set\_motors(-40,40);

// This function records a set of sensor readings and keeps

// track of the minimum and maximum values encountered. The

// IR\_EMITTERS\_ON argument means that the IR LEDs will be

// turned on during the reading, which is usually what you

// want.

calibrate\_line\_sensors(IR\_EMITTERS\_ON);

// Since our counter runs to 80, the total delay will be

// 80\*20 = 1600 ms.

delay\_ms(20);

}

set\_motors(0,0);

// Display calibrated values as a bar graph.

while(!button\_is\_pressed(BUTTON\_B))

{

// Read the sensor values and get the position measurement.

unsigned int position = read\_line(sensors,IR\_EMITTERS\_ON);

// Display the position measurement, which will go from 0

// (when the leftmost sensor is over the line) to 4000 (when

// the rightmost sensor is over the line) on the 3pi, along

// with a bar graph of the sensor readings. This allows you

// to make sure the robot is ready to go.

clear();

print\_long(position);

lcd\_goto\_xy(0,1);

display\_readings(sensors);

delay\_ms(100);

}

wait\_for\_button\_release(BUTTON\_B);

clear();

print("Go!");

// Play music and wait for it to finish before we start driving.

play\_from\_program\_space(go);

while(is\_playing());

}

// This is the main function, where the code starts. All C programs

// must have a main() function defined somewhere.

int main()

{

unsigned int sensors[5]; // an array to hold sensor values

// set up the 3pi

initialize();

// This is the "main loop" - it will run forever.

while(1)

{

// Get the position of the line. Note that we \*must\* provide

// the "sensors" argument to read\_line() here, even though we

// are not interested in the individual sensor readings.

unsigned int position = read\_line(sensors,IR\_EMITTERS\_ON);

if(position < 1000)

{

// We are far to the right of the line: turn left.

// Set the right motor to 100 and the left motor to zero,

// to do a sharp turn to the left. Note that the maximum

// value of either motor speed is 255, so we are driving

// it at just about 40% of the max.

set\_motors(0,100);

// Just for fun, indicate the direction we are turning on

// the LEDs.

left\_led(1);

right\_led(0);

}

else if(position < 3000)

{

// We are somewhat close to being centered on the line:

// drive straight.

set\_motors(100,100);

left\_led(1);

right\_led(1);

}

else

{

// We are far to the left of the line: turn right.

set\_motors(100,0);

left\_led(0);

right\_led(1);

}

}

// This part of the code is never reached. A robot should

// never reach the end of its program, or unpredictable behavior

// will result as random code starts getting executed. If you

// really want to stop all actions at some point, set your motors

// to 0,0 and run the following command to loop forever:

//

// while(1);

}