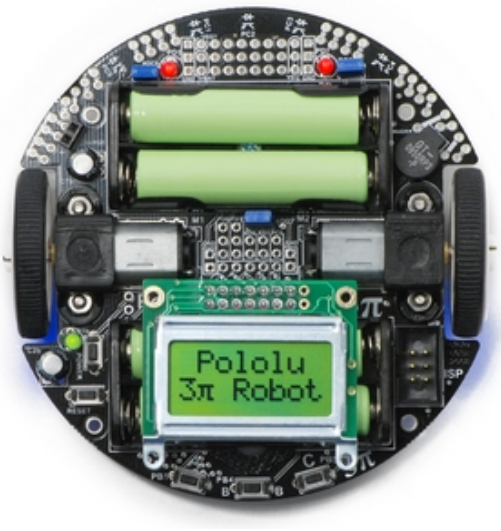
Pololu 3pi Robot Guía de usuario



1. Introducción

El 3pi de Pololu es un pequeño robot autónomo de alto rendimiento, designado para competiciones de seguimiento de línea y resolución de laberintos. Alimentado por 4 pilas AAA (no incluidas) y un único sistema de tracción para los motores trabajando a 9.25V, el 3pi es capaz de velocidades por encima de los 100cm/s mientras realiza vueltas precisas y cambios de sentido que no varían con el voltaje de las baterías. Los resultados son consistentes y están bien sintonizados con el código aún con baterías bajas. El robot está totalmente ensamblado con dos micro motores de metal para las ruedas, cinco sensores de reflexión, una pantalla LCD de 8x2 caracteres, un buzzer, tres pulsadores y más, todo ello conectado a un microcontroladores programable AVR. El 3pi mide aproximadamente 9,5 cm (3,7”) de diámetro y pesa alrededor de 83 gr. (2,9 oz.) sin baterías.
El 3pi se basa en un microcontroladores Atmel ATmega168 a 20 MHz con 16Kb de memoria flash y 1Kb de memoria de datos. El uso del ATmega168 lo hace compatible con la plataforma de desarrollo Arduino. Herramientas gratuitas de desarrollo en C y C++ están disponibles así como un extenso paquete de librerías que pueden trabajar con el hardware que lleva integrado. Están disponible simples programas que muestran como trabajan los diferentes componentes 3pi y también para mejorar otros códigos para el seguimiento de línea y laberintos.
Este documento está en construcción y se recibe en está forma para los primeros compradores.

2. Contactando con Pololu

Puedes visitar la página de 3pi para información adicional, fotos, videos, ejemplos de código y otras referencias.
Nos encantaría saber de tus proyectos y sobre tu experiencia con el 3pi robot. Puedes ponerte en contacto con nosotros directamente o por correo en nuestro foro. Cuéntanos lo que hicimos bien, lo que podría mejorar, lo que te gustaría ver en el futuro, o compartir tu código con otros usuarios 3pi.

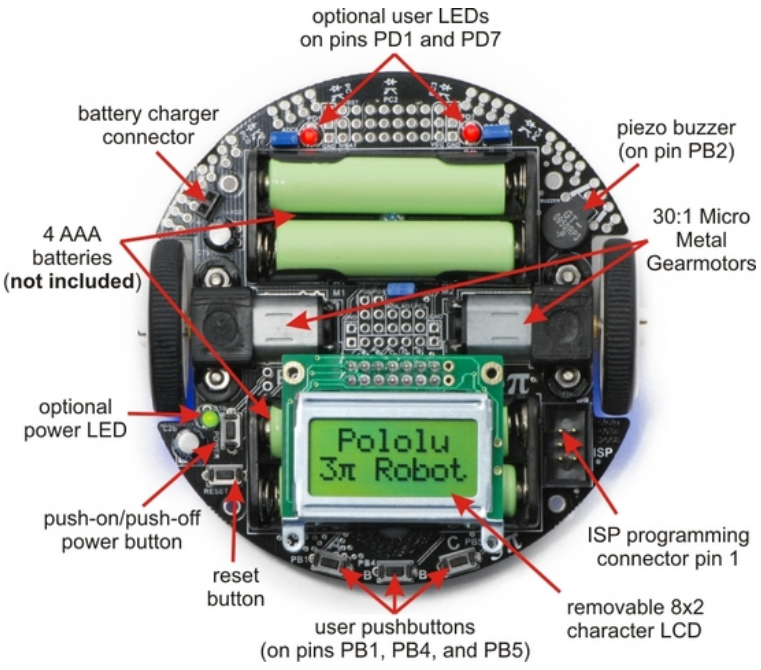
3. Advertencias en materia de seguridad y precauciones de manipulación

El robot 3pi no es para niños. Los más jóvenes deben usar este producto bajo supervisión de adultos. Mediante el uso de este producto, te compromete a no señalar a Pololu como responsable por cualquier lesión o daños relacionados con el uso incorrecto del mismo producto. Este producto no está diseñado para jugar y no debe utilizarse en aplicaciones en las que el mal funcionamiento del producto podría causar lesiones o daños. Por favor, tome nota de estas precauciones adicionales:

- El robot 3pi robot contiene plomo, no lo laves ni le eches líquidos.
- Está diseñado para trabajar en interiores y pequeñas superficies deslizantes.
- Si lo pones en contacto con superficies metálicas la PCB puede estar en contacto y producirse cortocircuitos que dañarían el 3pi.
- Dado que la PCB y sus componentes son electrónicos tome precauciones para protegerlo de ESD (descargas electrostáticas) que podrían dañar sus partes. Cuando toques el 3pi principalmente en ruedas, motores, baterías o contactos de la PCB, puedes sufrir pequeñas descargas. Cuando manejes el 3pi con otra persona, primero toca tu mano con la suya para igualar las cargas electrostáticas que podáis tener y están no se descarguen en el 3pi
- Si quitas la LCD asegúrate de que está el 3pi apagado y que la pones en la misma dirección que está, encima del extremo de la batería ya que podrías estropear la LCD o el 3pi.. Es posible poner la LCD sin iluminación o parada.

4. Empezar con tu robot 3pi

Para comenzar con tu 3pi solo debes sacarlo de la caja, ponerle pilas y encender. El 3pi se inicia con un programa demo que te muestra el funcionamiento de sus características.



Características del robot Pololu 3pi , visto desde arriba.

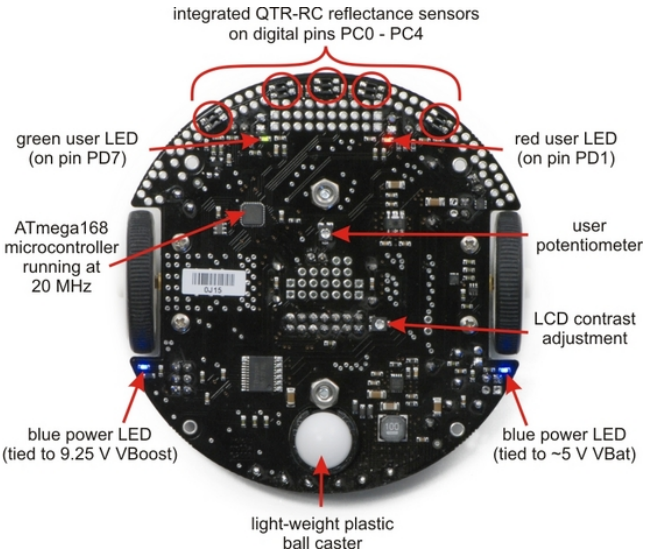
Los siguientes apartados de darán información necesaria para poder hacer que funcione tu 3pi

4.a. Que necesitas

Son necesarios para empezar:

- **4 pilas AAA.** Cualquier pila AAA puede hacerlo funcionar, pero recomendamos las de NiMH recargables, fáciles de comprar en Pololu u otro comercio. Si usa pilas recargables necesitaras un cargador de baterías. Los cargadores diseñados para conectar un pack de baterías externos pueden usarse con el puerto cargador del 3pi.

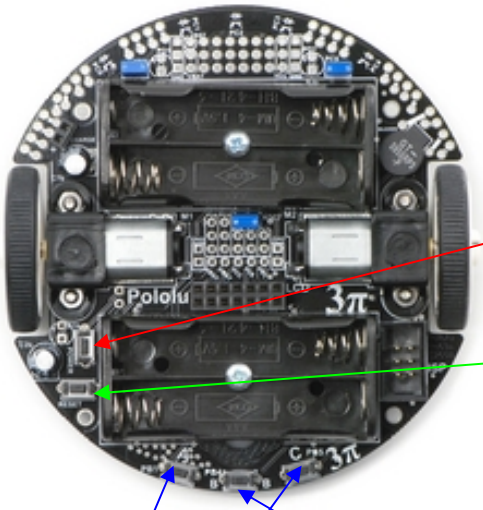
- **AVR Conector ISP programador de 6-pin.** Las características del 3pi y el microcontrolador ATmega168 requieren de un programador externo como el programador Pololu Orangután USB o la serie AVRISP de Atmel. El 3pi tiene un conector estándar de 6 pins que permite la programación directa del micro mediante un cable ISP que se conecta al dispositivo programador. (También necesitas un cable USB que conecte el programador con el PC. No está incluido. (Si, que lo tienes en el paquete combinado de 3pi+programmer).



- **Ordenador de mesa o portátil**.
  Necesitas un PC para desarrollar código y grabarlo en el 3pi. Se puede programar en Windows, Mac y Linux, pero el soporte para Mac es limitado.
- Además, **materiales para crear superficies** por donde corra el 3pi:
  - ❖ Hojas de papel largo y gruesas blancas o una pizarra blanca borrable (usadas por los constructores de casas).
  - ❖ Cinta colores brillantes para rejuntar múltiples hojas a la vez.
  - ❖ Cinta de 3/4" eléctrica para crear líneas a seguir por el robot.

4.b. Enciende el 3pi

La primera vez que uses el 3pi debes insertar 2+2 pilas AAA en cada sitio. Para ello debes quitar las LCD. Presta atención a la orientación para insertarla después. Con la LCD quitada mira las figuras marcadas a la derecha.



En cuanto estén puestas las pilas, inserta la LCD en su posición encima del paquete de baterías. Fíjate bien en los pins que deben quedar cada uno en su conector hembra correspondiente.

Después, Pulsa el botón de ENCENDER (a la izquierda al lado del porta pilas) para conectar el 3pi. Verás que dos leds azules se encienden y el 3pi empieza a ejecutar el programa de demo.

Con pulsar el botón de nuevo se apagara el 3pi o pulsa el botón de RESET situado justo mas abajo para resetear el robot mientras está funcionando.

4.c. Funcionamiento del programa demo pre-instalado

Tu 3pi viene con un programa pre-instalado de demostración y testeo de sensores, motores, leds y buzzer para ver su correcto funcionamiento.

Cuando se enciende por primera vez oirás un pitido y veras en pantalla “Pololu 3pi Robot” y luego aparece “Demo Program”, indicando que está en funcionamiento el mismo. Si oyes el beep pero no aparece nada en la LCD puedes ajustar el contraste de la LCD con el mini potenciómetro que está debajo de la placa. Seguir el programa pulsando el botón B para proceder con el menú principal.

Pulsa A o C para avanzar o retroceder a traves del menú y de nuevo B para salir.

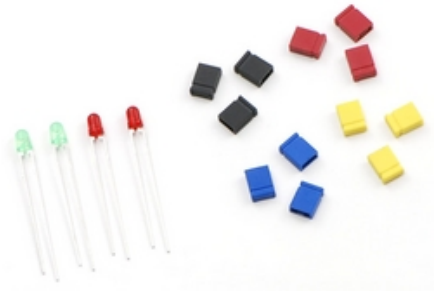
Hay siete demos accesibles desde el menú:

- Batería: Muestra el voltaje de las pilas en milivoltios, así, si marca 5000 (5.0 V) o más, es porque las baterías están a a tope. Removiendo el jumper marcado como ADC6 separa la batería del pin analógico de medida produciendo que se muestre un valor muy bajo.
- LEDs: Parpadeo de led verde y rojo que hay bajo la placa o los de usuario si los has puesto.

- Trimmer: Muestra la posición del mini potenciómetro trimmer localizado en la parte inferior de la placa marcando un numero entre 0 y 1023. Al mostrar el valor parpadean los LEDS y toca una nota musical cuya frecuencia está en función a la lectura. Puedes hacer girar los micro potenciómetros con pequeño destornillador de 2mm.
- Sensores: Muestra las lecturas actuales de los sensores IR mediante un gráfico de barras. Barras grandes indican poca reflexión (negro). Coloca un objeto reflectivo como un dedo sobre uno de los sensores y veras la lectura gráfica correspondiente. Esta demo también muestra, al pulsar C que todos los sensores están activos. En iluminación interior, cerca de bombillas de incandescencia o halógenas los sensores pueden emitir lecturas erróneas debido a la emisión de infrarrijos. Remueve el Jumper PC5 para desactivar el control de los emisores IR lo que servirá para que siempre estén activos.
- Motores: Pulsando A o C hará que funcionen cada uno de los motores en su dirección. Si pulsas ambos botones, ambos motores funcionan simultáneamente. Los motores aumentan gradualmente la velocidad; si realizas nuevos programas estudia detenidamente el funcionamiento de aceleración. Pulsa A o C para invertir la dirección del motor correspondiente (la letra del botón se vuelve minúscula si el motor funciona en sentido contrario).
- Música: Toca una melodía de J. S. Bach's Fuga en D Menor en el buzzer, mientras muestra unas notas. Es para mostrar la habilidad de 3pi como músico.
- Timer: Un simple reloj contador. Pulsa C para iniciar o parar el reloj y A para reset. El reloj puede seguir contando mientras exploras otras demos.

El código fuente del programa demo está incluido en las librerías de Pololu AVR C/C++ descritas en la sección 5. Después de descargar y desempaquetar las librerías el programa se encuentra en el directorio `examples\3pi-demo-program`.

4.d. Accesorios incluidos



Los robots 3pi se envían con dos LEDs rojos y dos verdes. Tienes tres puntos de conexión para leds opcionales: uno al lado del botón de POWER para indicar cuando el 3pi está encendido y dos puntos más controlables por el usuario en el frontal.
El uso de leds es opcional y el 3pi funciona igual sin ellos. Puedes personalizar tu 3pi con una combinación de verdes y rojos y usarlos para opciones luminosas.
Añadir LEDs es fácil, pero ten en cuenta que si tienes que desoldarlos después, los componentes que se encuentran cerca de donde hagas las soldaduras. Los LEDs tienen polaridad, fíjate, el trozo más largo corresponde al +. Antes de soldar asegúrate de la función que van a realizar y sujeta bien el led y. Recorta el exceso de patas sobrante. El 3pi también viene con cuatro juegos de tres jumpers en colores: azul, rojo, amarillo y negro. Son para personalizarlos si tienes más de un 3pi con diferentes colores.

5. Programando tu 3pi

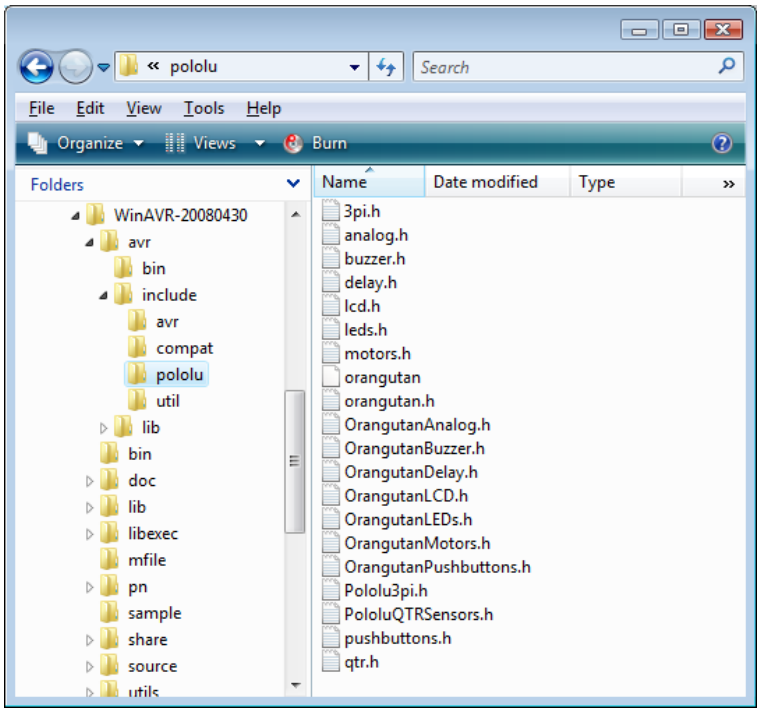
Para hacer más de lo que hace la demo necesitas programarlo, eso requiere un programador AVR ISP como el Orangután USB programer. El primer paso es ajustar el programador siguiendo las instrucciones de instalación. Si usas el Orangután USB programmer, mira la guía de usuario.
Lo siguiente es tener un software que compile y transfiera el código creado al 3pi a traves del programador.
Recomendamos estos dos paquetes de software:

- WinAVR, que es libre, un entorno de herramientas para los micros de la familia AVR, incluido un compilador GNU GCC para C/C++.
- AVR Studio, paquete de desarrollo integrado de la casa Atmel's con (IDE) que trabaja en armonía con el compilador WinAVR's. AVR Studio incluye el software AVR ISP que permite cargar tus programas creados en el robot 3pi.

Nota: Puedes también programar tu 3pi usando la interfaz Arduino IDE ay un programador externo como Orangután USB programmer. Para las instrucciones en este sistema mira la guía:
Programming Orangutans y el robot 3pi desde un entorno Arduino. El resto es del AVR Studio.
Para mas información y uso de las librerías de Pololu C/C++ con robots basados en AVR, incluyendo instrucciones de instalación en LINUX, mira Pololu AVR C/C++ Library User's Guide.

5.a. Descargar e instalar la Librería C/C++

La librería de Pololu C/C++ AVR hace más fácil para ti el uso de funciones avanzadas en tu 3pi; la librería se usa en todos los ejemplos de las siguientes secciones. Para su comprensión el código fuente de dichos ejemplos y el programa demo están incluidos en la librería, Para empezar con la instalación de la librería necesitas bajarte uno de los siguientes ficheros:



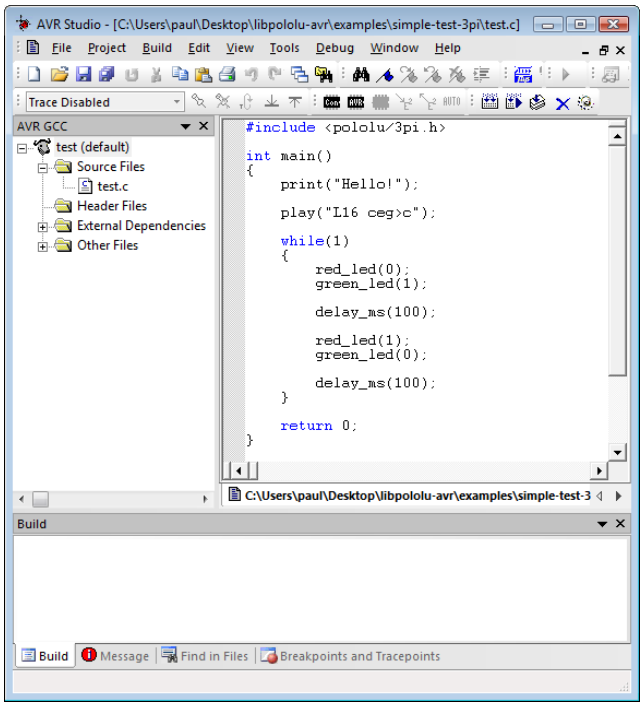
- Precompiled binary distribution (219k zip)
- Source distribution (262k zip)

Nosotros recomendamos la versión precompilada, es mas fácil de instalar para la mayoría de usuarios. Si tiene problemas al instalar la versión precompilada, o estás interesado en echar una mirada más de cerca al código fuente, descarga Source distribution y mira la sección 4 de Pololu AVR C/C++ Library User's Guide para las instrucciones de compilación.

Llibrería de ficheros de cabecera de Pololu AVR, instalados correctamente

Abre el fichero.zip y clic "Extract all" para extraer los ficheros de la librería Pololu AVR. Se creará el directorio "libpololu-avr".

Determina la localización de los ficheros de avr-gcc. En Windows, generalmente están en el directorio: `C:\WinAVR-20080610\avr`. En Linux, estarán posiblemente localizados en el directorio `/usr/avr`. Si tienes una versión antigua de la librería de Pololu AVR el primer paso será borrarla por entero incluyendo el fichero `libpololu.a` instalados anteriormente.



Luego, copia `libpololu.a` dentro del subdirectorio `lib` del directorio avr, tal como `C:\WinAVR-20080610\avr\lib`.

Finalmente, copia el subdirectorio entero de `pololu` dentro del subdirectorio `include`, tal como `C:\WinAVR-20080610\avr\include\pololu`.

Ahora estás preparado para usar Pololu AVR library con tu 3pi.

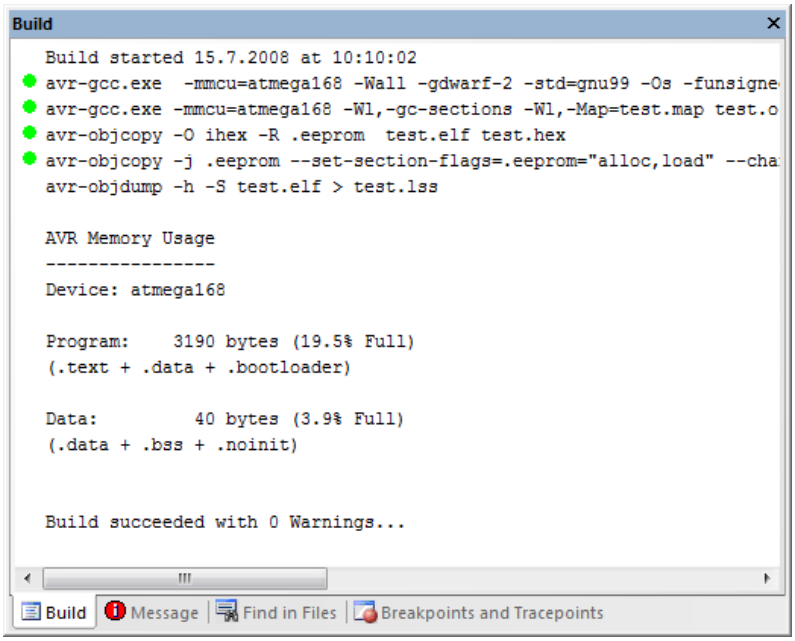
5.b. Compilando un Programa simple

Un sencillo programa demostración está disponible para tu 3pi en el directorio: `examples\simple-test-3pi`, con comandos básicos de la Pololu AVR Library.

AVR Studio con el programa simple-test-3pi

Este es su código:

```
1.  #include <pololu/3pi.h>
2.  int main()
3.  {
4.      print("Hello!");
5.      play("L16 ceg>c");
6.      while(1)
7.      {
8.          red_led(0);
9.          green_led(1);
10.         delay_ms(100);
11.         red_led(1);
12.         green_led(0);
13.         delay_ms(100);
14.     }
15.     return 0;
16. }
```



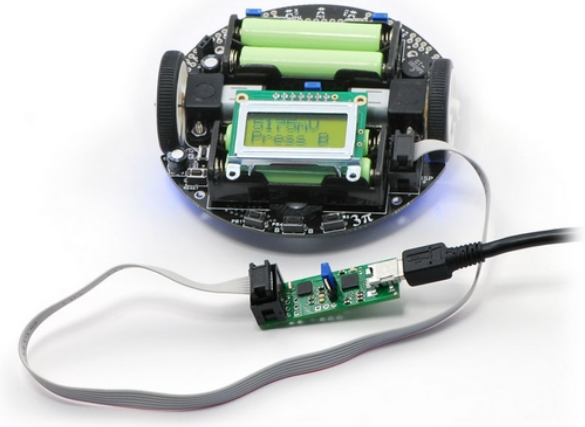
AVR Studio ventana de compilación, compilando el proyecto del ejemplo.

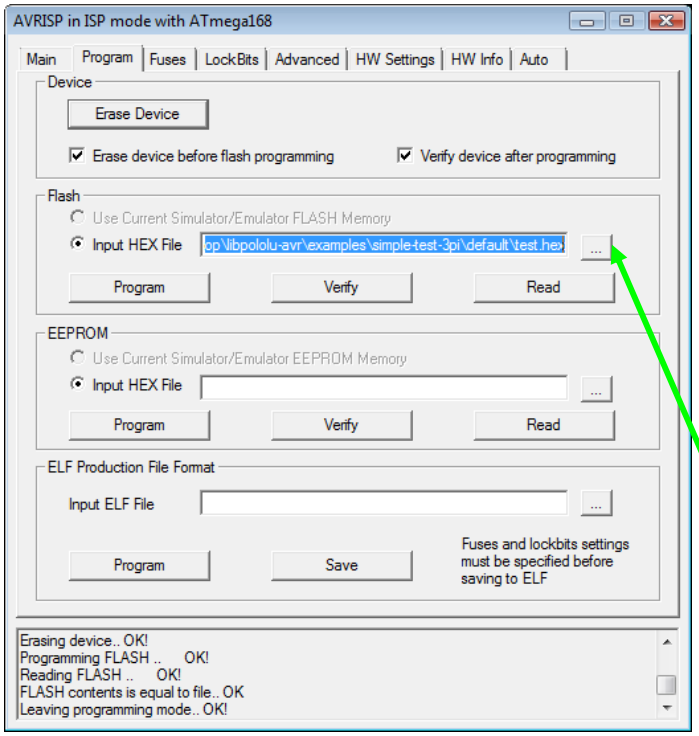
Navega por el directorio `simple-test-3pi`, doble-clic en `test.aps`, y el proyecto aparecerá en la interfaz de AVR Studio, mostrando el fichero en código C.

Para compilar, selecciona **Build > Build** o pulsa **F7**. Observa los avisos y errores (indicados con puntos amarillos y rojos) en la salida mostrada mas abajo. Si el programa se compila correctamente, el mensaje "Build se realizo con 0 avisos…" aparecerá al final de la salida, y el fichero `test.hex` se creará en el directorio: `examples\simple-test-3pi\default`.

Conecta el programador a tu PC y este al ISP port de tu 3pi, y enciende el 3pi pulsando el botón POWER.

Si estás usando el Pololu Orangután Programmer, El LED verde de estado se enciende al meter el conector USB, mientras los otros dos leds se encienden 5 segundos indicando que el programador se está iniciando.





El programador debe estar instalado correctamente antes de usarlo. Si usas Orangután USB programmer, mira las instrucciones de instalación.

Pololu 3pi robot con el Orangután USB programmer conectados por el ISP port.

Selecciona **Tools > Program AVR > Connect** para conectar el programador. Para el Orangután Programmer, las opciones por defecto "STK500 o AVRISP" y "Auto" funcionan bien, ahora clic en Connect y la ventana de programación AVRISP aparece.

Puedes usar AVRISP para leer `test.hex` dentro la memoria del 3pi. En este caso, clic "…" en la sección Flash y selecciona el fichero `test.hex` que compilaste antes.

Ten en cuenta que primero tienes que ir al directorio del `\proyecto\default\fichero.hex`!. Ahora clic en “Program” de la zona Flash, y el código se grabará dentro del Atmel del 3pi.

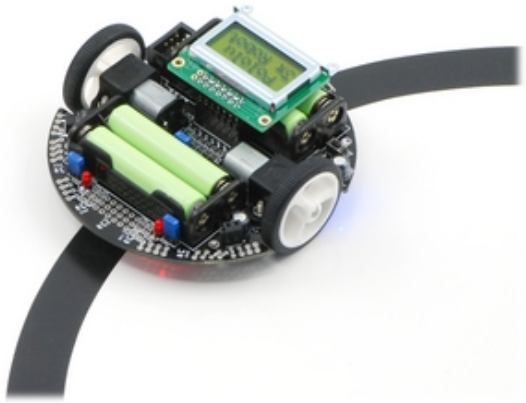
Programando el 3pi desde AVR Studio.

Si tu 3pi se ha programado debes escuchar una corta canción y ver el mensaje "Hello!" en la pantalla LCD y los LEDs parpadeando. Si oyes la melodía y ves las luces parpadeando, pero no aparece nada en la pantalla, asegúrate de que está correctamente conectada a la 3pi, o intenta ajustar el contraste con el micro potenciómetro que hay en la parte inferior de la placa.

6. Ejemplo Project #1: Siguiendo la línea

6.a. Acerca del seguimiento de línea

Ahora que has aprendido como compilar un program es tiempo de enseñar a tu robot 3pi algunas cosas más complicadas. En este ejemplo mostraremos como hacer que tu 3pi siga el camino que marca una línea negra sobre fondo blanco, coordinando los sensores y sus motores. El seguimiento de línea es una buena introducción en la programación de robots para participar en un concurso; no es díficil construir una plataforma de línea de seguimiento, fácil de entender su funcionamiento y la programación de 3pi no es complicada. Optimizar el programa para hacer que el robot 3pi se deslice sobre la línea a la velocidad más rápida posible será un reto que puede llevarte a algunos conceptos avanzados de programación.



Pololu 3pi robot sobr una línea negra de 3/4"

Una línea de seguimiento puede construirse en poco tiempo y por poco dinero, consulta el tutorial de Construcción de Líneas de seguimiento y Laberintos de línea.

6.b. Algoritmo para 3pi de seguimiento de línea

El programa de seguimiento de línea se encuentra en el directorio `examples\3pi-linefollower`.

Nota: Una versión compatible para Arduino de este programa puedes bajarla como parte de las librerías Pololu Arduino (ver Sección 5.g).

El código fuente demuestra la variedad de dispositivos de la 3pi, como sensores de línea, motores, pantalla LCD, monitor de carga de la batería y buzzer. EL programa consta de dos fases.

La primera fase consiste en la inicialización y calibración y está programada como la función `intitialize()`. Esta función es llamada una vez, al principio de la función `main()`, antes de que suceda algo y siguiendo estos pasos:

1. Llamada a *pololu_3pi_init(2000)* para ajustar el 3pi, con el timeout del sensor a 2000×0.4 us = 800 us. Este ajuste permite al sensor variar entre valores desde 0 (completamente blanco) a 2000 (completamente negro), en donde el valor de 2000 indica que el condensador tarda aproximadamente 800 us en descargarse.
2. Muestra el voltaje de la batería que nos da la función *read_battery_millivolts().* Es importante monitorizar la carga de la batería para que el robot no de sorpresas en su carrera y las pilas estén bien cargadas durante la competición. Para más información ver sección 3.
3. Calibrado de los sensores. Esto se logra mediante la activación del 3pi a derecha y a izquierda de la línea al tiempo que llamamos a la función *calibrate_line_sensors ().* El mínimo y el máximo de valores leídos durante este tiempo se almacenan en la RAM. Esto permite a la función *read_line_sensors_calibrated()* que devuelva valores ajustados en el rango de 0 a 1000 para cada sensor, aunque alguno de los sensores responda de diferente manera que los otros. La función *read_line()* usada después en el código depende de los valores calibrados. Para más información ver sección 11

4. Presentación del valor de calibrado de los sensores en barras gráficas. Sirve para mostrar el uso de la función *lcd_load_custom_character()* siempre con *print_character()* para hacer más fácil la visualización del funcionamiento correcto de los sensores de línea antes de que corra el robot. Para más información sobre los comandos de la LCD, ver Sección 5.
5. Espera a que el usuario pulse el botón. Es muy importante que tu robot empiece a rodar cuando tu quieras, o podría salir inesperadamente del cuadro o fuera de sus manos cuando estás tratando iniciar el programa. Usaremos la función *button_is_pressed()* para esperar la pulsación del botón B cuando haya mostrado el estado de la vertía y de los sensores. Para más información ver sección 8.

En la segunda fase del programa, tu 3pi realizará la lectura del sensor y ajustará la velocidad de los motores a ella. La idea general es que si el robot está fuera de linea, pueda volver, pero si está en la línea, debe tratar de seguirla. Los siguientes pasos ocurren dentro del bucle `while(1)`, que se repetirán una y otra vez hasta que lo pares o pulses el botón reset.

- Llamada a la función *read_line().* Obliga al sensor a leer y devuelve una lectura estimada entre el robot y la línea, un valor entre 0 y 4000. El valor 0 indica que la línea esta a la izquierda del sensor 0, valor de 1000 indica que la línea esta debajo del sensor 1, 2000 indica que esta debajo del sensor 2, y así sucesivamente.
- El valor devuelto por read_line() se divide en tres casos posibles:

    1. **0–1000:** el robot está lejos del lado derecho de la línea. En este caso, gira rápido a izquierda, ajustando el motor derecho a 100 y el izquierdo a 0. La máxima velocidad de los motores es de 255, luego estamos rodando el motor derecho al 40% de su velocidad.
    2. **1000–3000:** el robot está bastante centrado en la línea. En este caso, ajusta los motores a velocidad 100, para correr recto.
    3. **3000–4000:** el robot está lejos del lado izquierdo de la línea. En este caso, gira rápido a derecha ajustando los motores derecho a 0 e izquierdo a 100. Dependiendo de que motores están activados, se encienden los leds correspondientes para mejorar el aspecto. Esto puede ayudar en la depuración del código.

Para abrir el programa en el AVR Studio debes ir a `examples\3pi-linefollower` y un doble-clic en `test.aps`. Compila el programa, mételo en tu 3pi y adelante. Tienes que saber que tu robot es capaz de seguir las curvas de la línea en curso sin perderla. Sin embargo los motores se mueven a velocidades de alrededor de 100 de su máximo posible de 255, y el algoritmo debe producir una gran cantidad de cálculos para las curvas. En este punto puedes intentar mejorar el algoritmo antes de pasar a la siguiente sección. Algunas ideas para ello pondrían ser:

- Incrementar la velocidad al máximo posible.
- Añadir causas intermedias son velocidad intermedias de ajuste para hacerlo más divertido.
- Usar la memoria del robot: tiene su máxima aceleración después de haber circulado por un tramo de línea con unos pocos ciclos de código.

También puedes:

- Medir la velocidad del bucle, usando las funciones de cronómetro de la sección2 para calcular el tiempo necesario de ciclos o de parpadeos del led por cada 1000 ciclos.
- Mostrar las lecturas de los sensores en la LCD. Hay que tener en cuenta que la escritura de la LCD conlleva bastante tiempo y sobre todo si son varias veces por segundo.
- Incorpora el buzzer en programa. Puedes hacer que tu 3pi toque música mientras corre o tener información adicional con los beeps según lo que esté haciendo. Ver sección 4 para más información del uso del buzzer; para música tendrías que usar PLAY_CHECK, pero desconecta la lectura de los sensores.

El código entero del programa de seguimiento de línea es este:

```
/*
 * 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 sensores, 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
// ATmega168 has 16k of program space compared to 1k of 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.
```

```
158.            play_from_program_space(go);
159.            while(is_playing());
160.        }
161.        // This is the main function, where the code starts.  All C programs
162.        // must have a main() function defined somewhere.
163.        int main()
164.        {
165.            unsigned int sensors[5]; // an array to hold sensor values
166.            // set up the 3pi
167.            initialize();
168.            // This is the "main loop" - it will run forever.
169.            while(1)
170.            {
171.                // Get the position of the line.  Note that we *must* provide
172.                // the "sensors" argument to read_line() here, even though we
173.                // are not interested in the individual sensor readings.
174.                unsigned int position = read_line(sensors,IR_EMITTERS_ON);
175.                if(position < 1000)
176.                {
177.                    // We are far to the right of the line: turn left.
178.                    // Set the right motor to 100 and the left motor to zero,
179.                    // to do a sharp turn to the left.  Note that the maximum
180.                    // value of either motor speed is 255, so we are driving
181.                    // it at just about 40% of the max.
182.                    set_motors(0,100);
183.                    // Just for fun, indicate the direction we are turning on
184.                    // the LEDs.
185.                    left_led(1);
186.                    right_led(0);
187.                }
188.                else if(position < 3000)
189.                {
190.                    // We are somewhat close to being centered on the line:
191.                    // drive straight.
192.                    set_motors(100,100);
193.                    left_led(1);
194.                    right_led(1);
195.                }
196.                else
197.                {
198.                    // We are far to the left of the line: turn right.
199.                    set_motors(100,0);
200.                    left_led(0);
201.                    right_led(1);
202.                }
203.            }
204.            // This part of the code is never reached.  A robot should
205.            // never reach the end of its program, or unpredictable behavior
206.            // will result as random code starts getting executed.  If you
207.            // really want to stop all actions at some point, set your motors
208.            // to 0,0 and run the following command to loop forever:
209.            //
210.            while(1);
211.        }
```

6.c Seguimiento de línea avanzado con 3pi: PID Control

Un programa avanzado de seguimiento de línea para el 3pi está en el directorio `examples\3pi-linefollower-pid`.

Nota: Hay una versión compatible con el Arduino-compatible de este programa que puede bajarse como parte de Pololu Arduino Libraries (ver Sección 5.g).

La técnica usada en este ejemplo es conocida como *PID control*, dirigida a alguno de los problemas que hemos mencionado en el programa anterior y que permiten incrementar de forma notoria la velocidad de seguimiento de la línea. Muy importante, el control PID usa continuamente funciones para calcular la velocidad de los motores, la simpleza del ejemplo anterior puede reemplazarse por una respuesta más suave. PID responde a Proporcional, Integral, Derivación, estas son las tres entradas usadas en la fórmula para computar la velocidad del robot al girar a izquierda y derecha.

- El valor **proportional** es aproximadamente proporcional a la posición del robot. Esto es. Si está centrado en la línea lo expresamos con un valor exacto de 0. Si esta a la izquierda de la línea, el valor será un número *positivo* y si está a la derecha de la línea será *negativo*. Esto se computa por el resultado que devuelve read_line() simply restándole 2000.

- El valor **integral** es un histórico del funcionamiento del robot: es la suma de todos los valores del termino proporcional que recuerda desde que el robot empieza a funcionar.
- El **derivative** es el índice de cambios de los valores proporcional. Computamos en este ejemplo la diferencia entre los últimos dos valores.

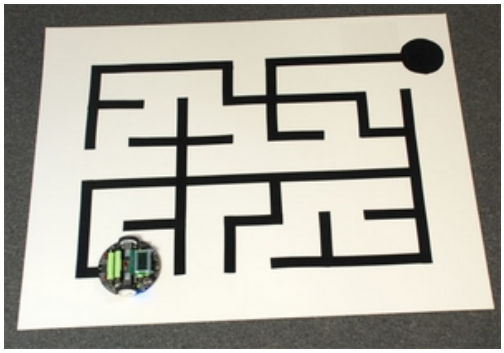
Este el trozo de código para la entrada de los valores PID:

```
1.  // Get the position of the line.  Note that we *must* provide
2.  // the "sensors" argument to read_line() here, even though we
3.  // are not interested in the individual sensor readings.
4.  unsigned int position = read_line(sensors,IR_EMITTERS_ON);
5.  // The "proportional" term should be 0 when we are on the line.
6.  int proportional = ((int)position) - 2000;
7.  // Compute the derivative (change) and integral (sum) of the
8.  // position.
9.  int derivative = proportional - last_proportional;
10. integral += proportional;
11. // Remember the last position.
12. last_proportional = proportional;
```

Hay que tener en cuenta que la variable `position` es del tipo int en la fórmula para un proporcional. Un unsigned int solo puede almacenar valores positivos, luego la expresión `position-2000`, pone fuera de prueba y puede llevar a un negative overflow. En este caso particular no afecta a los resultados pero seria una buena idea usar ajustar la fórmula para afinar en el resultado.
Cada uno de los valores entrados proviene de diferentes fuentes de información. El paso siguiente es una simple formula que combina todos los valores en una variable y que se usa para determinar las velocidades de los motores.

```
1.  // Compute the difference between the two motor power settings,
2.  // m1 - m2.  If this is a positive number the robot will turn
3.  // to the right.  If it is a negative number, the robot will
4.  // turn to the left, and the magnitude of the number determines
5.  // the sharpness of the turn.
6.  int power_difference = proportional/20 + integral/10000 + derivative*3/2;
7.  // Compute the actual motor settings.  We never set either motor
8.  // to a negative value.
9.  const int max = 60;
10. if(power_difference > max)
11.     power_difference = max;
12. if(power_difference < -max)
13.     power_difference = -max;
14. if(power_difference < 0)
15.     set_motors(max+power_difference, max);
16. else
17.     set_motors(max, max-power_difference);
```

Los valores 1/20, 1/10000, y 3/2 son parámetros ajustables que determinan la dirección del 3pi sobre la línea. En general, incrementando estos parámetros PID podemos hacer `power_difference` largas, causando reacciones más fuertes, o cuando decrecemos podemos tener reacciones débiles.
Debes reflexionar sobre los distintos valores y experimentar con tu robot para determinar qué efecto tiene cada uno de los parámetros. Este ejemplo da a los motores una velocidad máxima de 100, que es un valor inicial seguro inicial. Una vez ajustado los parámetros para que funcione bien a una velocidad de 100, intenta aumentarla. Probablemente necesitaras ajustar estos parámetros en función del recorrido para que el robot vaya lo más rápido posible.



7. Ejemplo Project #2: Resolución de laberintos

7.a. Resolución de laberinto de línea

El siguiente paso desde el seguimiento de línea es enseñar a tu 3pi a navegar entre caminos con giros recortados, callejones sin salida, e intersecciones. Crea una red

complicada de líneas entrecruzadas en negro, añade un círculo que represente el final y ya tienes un laberinto de líneas, debe ser un entorno dificil para explorar por tu 3pi. En un laberinto de líneas los robots corren tranquilamente sobre las líneas desde un inicio a un final asignado, cruzando las intersecciones y saliendo de las lineas cortadas que hay durante el circuito. Los robots tienen varias posibilidades para ejecutar el laberinto, de modo que puedan seguir el camino más rápido posible después de aprender los cruces y sobre todo los callejones sin salida.
Los laberintos que pretendemos resolver en este tutorial tienen una particularidad importante: *no contienen bucles*. Es decir, este código no está hecho para salir de un bucle en el laberinto sin tener que volver sobre sus pasos. Luego, la solución a este tipo de laberinto es mucho más fácil que resolver un laberinto con bucles, ya que una simple estrategia te permite explorar todo el laberinto. Vamos a hablar de esa estrategia en la próxima sección.
Podemos construir nuestros laberintos utilizando sólo unas líneas rectas trazadas sobre una rejilla regular, pero esto se hace principalmente para que el curso sea fácil de reproducir – la estrategia de solución del laberinto que se describe en este tutorial no requiere de estas características.
Para más información mira el tutorial Building Line Following and Line Maze Courses.

7.b. Trabajar con múltiples ficheros C en AVR Studio

El código fuente C para resolver del laberinto de líneas está en: `examples\3pi-mazesolver`.
Nota: Hay una versión compatible para Arduino en Pololu Arduino Libraries (ver Sección 5.g).
Este programa es mucho más complicado que el ejemplo anterior y esta partido en múltiples ficheros. Se usan varios ficheros para facilitar la creación de código. Por ejemplo el fichero `turn.c` contiene solo una función, usada para hacer giros en las intersecciones:

```
1.  #include <pololu/3pi.h>
2.  // Turns according to the parameter dir, which should be 'L', 'R', 'S'
3.  // (straight), or 'B' (back).
4.  void turn(char dir)
5.  {
6.      switch(dir)
7.      {
8.      case 'L':
9.          // Turn left.
10.         set_motors(-80,80);
11.         delay_ms(200);
12.         break;
13.     case 'R':
14.         // Turn right.
15.         set_motors(80,-80);
16.         delay_ms(200);
17.         break;
18.     case 'B':
19.         // Turn around.
20.         set_motors(80,-80);
21.         delay_ms(400);
22.         break;
23.     case 'S':
24.         // Don't do anything!
25.         break;
26.     }
27. }
```

La primera línea del fichero está escrita para el 3pi, contiene el fichero de cabecera que permite el acceso a las funciones de la librería Pololu AVR. Dentro de turn(), se usan funciones de librería como delay_ms() y set_motors() para mejorar los giros a izquierda, giros a derecha y salidas de curvas en U. Los “giros” rectos también son manejados por esta función, aunque estos no obligan a tomar decisiones. La velocidad del motor y los tiempos para los giros son los parámetros necesarios que necesitamos para ajustar el 3pi; a medida que se trabaja en hacer las soluciones más rápidas de los laberintos, estos serán algunos de los números necesarios para el ajuste.
Para acceder a las funciones necesitas los ficheros “header file(extensión .h)”, como `turn.h`. Este fichero de cabecera solo contiene una línea:

```
1.  void turn(char dir);
```

Esta línea declara la función turn() sin incluir una copia del código. Para acceder a la declaración cada fichero C necesita llamar a turn() añadiendo la línea siguiente:

```
1.  #include "turn.h"
```

Ten en cuenta los paréntesis usados. Esto significa que el compilador C usa este fichero de cabecera en el directorio del proyecto, en lugar de ser un sistema de ficheros de cabecera como `3pi.h`.
Recuerda siempre al crear código con funciones de poner el fichero de cabecera! Si no tienes otra solución, crea diferente copias separadas de cada fichero de código que incluya las cabeceras.
El fichero `follow-segment.c` también contiene una simple función follow_segment(), la cual lleva al 3pi recto a lo largo de una línea de segmento mientas busca una intersección o un fin de línea.
Esta es casi el mismo código que en el seguimiento de línea analizado en la sección 6, pero con más controles para las intersecciones y los extremos de línea. Aquí está la función:

```
1.  void follow_segment()
2.  {
3.      int last_proportional = 0;
4.      long integral=0;
5.      while(1)
6.      {
7.          // Normally, we will be following a line.  The code below is
8.          // similar to the 3pi-linefollower-pid example, but the maximum
9.          // speed is turned down to 60 for reliability.
10.         // Get the position of the line.
11.         unsigned int sensors[5];
12.         unsigned int position = read_line(sensors,IR_EMITTERS_ON);
13.         // The "proportional" term should be 0 when we are on the line.
14.         int proportional = ((int)position) - 2000;
15.         // Compute the derivative (change) and integral (sum) of the
16.         // position.
17.         int derivative = proportional - last_proportional;
18.         integral += proportional;
19.         // Remember the last position.
20.         last_proportional = proportional;
21.         // Compute the difference between the two motor power settings,
22.         // m1 - m2.  If this is a positive number the robot will turn
23.         // to the left.  If it is a negative number, the robot will
24.         // turn to the right, and the magnitude of the number determines
25.         // the sharpness of the turn.
26.         int power_difference = proportional/20 + integral/10000 + derivative*3/2;
27.         // Compute the actual motor settings.  We never set either motor
28.         // to a negative value.
29.         const int max = 60; // the maximum speed
30.         if(power_difference > max)
31.             power_difference = max;
32.         if(power_difference < -max)
33.             power_difference = -max;
34.         if(power_difference < 0)
35.             set_motors(max+power_difference,max);
36.         else
37.             set_motors(max,max-power_difference);
38.         // We use the inner three sensors (1, 2, and 3) for
39.         // determining whether there is a line straight ahead, and the
40.         // sensors 0 and 4 for detecting lines going to the left and
41.         // right.
42.         if(sensors[1] < 100 && sensors[2] < 100 && sensors[3] < 100)
43.         {
44.             // There is no line visible ahead, and we didn't see any
45.             // intersection.  Must be a dead end.
46.             return;
47.         }
48.         else if(sensors[0] > 200 || sensors[4] > 200)
49.         {
50.             // Found an intersection.
51.             return;
52.         }
53.     }
54. }
```

Entre el código PID y la detección de intersecciones, en la actualidad hay alrededor de seis parámetros más que se podrían ajustar. Hemos recogido aquí los valores que permiten a 3pi resolver

el laberinto con seguridad, control de velocidad, intento de aumentar la velocidad y que se ejecute con rapidez a los muchos de los problemas que tendría que manejar con un código complicado.
Poner los ficheros C y los ficheros de cabecera en tu proyecto es fácil en AVR Studio. En la columna de la izquierda de la pantalla puedes ver varias opciones para los ficheros de "Source Files" y "Header Files". Botón derecho en cada uno y tienes la opción de remover o añadir a la lista. Cuando compiles el código AVR Studio compila automáticamente todos los ficheros del proyecto.

7.c. Mano izquierda con el muro

La estrategia básica para solucionar laberintos sin bucle se llama *"left hand on the wall"*. Imagínate caminando por un laberinto de verdad – un típico laberinto rodeado de muros – y cuando pones tu mano izquierda sobre el muro varias veces. Puedes girar a izquierda siempre que sea posible y solo puedes girar a derecha en una intersección si no es una salida. Algunas veces cuando vas a para a un callejón sin salida debes girar 180º y retornar por donde has venido. Supongamos que a lo largo del trayecto no hay bucles, tu mano viajaría a lo largo de cada tramo del muro de la misma manera, al final encontrarías la entrada. Si hay una habitación en algún lugar del laberinto con un monstruo o algún tesoro, encontrarás el camino, ya que recorres cada pasillo exactamente dos veces. Usamos esta sencilla y fiable estrategia en nuestro 3pi como solución al ejemplo:

```
1.  // maze solving.  It uses the variables found_left, found_straight, and
2.  // This function decides which way to turn during the learning phase of
3.  // found_right, which indicate whether there is an exit in each of the
4.  // three directions, applying the "left hand on the wall" strategy.
5.  char select_turn(unsigned char found_left, unsigned char found_straight, unsigned char found
    _right)
6.  {
7.      // Make a decision about how to turn.  The following code
8.      // implements a left-hand-on-the-wall strategy, where we always
9.      // turn as far to the left as possible.
10.     if(found_left)
11.         return 'L';
12.     else if(found_straight)
13.         return 'S';
14.     else if(found_right)
15.         return 'R';
16.     else
17.         return 'B';
18. }
```

Los valore devueltos por select_turn() corresponden a los valores usador por turn(), siempre que estas funciones trabajen correctamente en el bucle principal.

7.d. Bucle(s) principal

La estrategia del programa se expresa en el fichero `maze-solve.c`. Muy importante, queremos hacer un seguimiento de la ruta recorrida por lo que hay que definir una matriz de almacenamiento de hasta 100 datos que serán los mismos caracteres utilizados en la función turn(). También tenemos que hacer un seguimiento de la longitud actual del camino para que sepamos que valores poner en la matriz.

```
1.  char path[100] = "";
2.  unsigned char path_length = 0; // the length of the path
```

El "main loop" se encuentra en la función maze_solve(), que es llamada después de la calibración, desde `main.c`. Esta función incluye dos bucles principales – el primero en donde "*manualmente*" se resuelve el laberinto y el segundo donde se replica la solución para mejorar el tiempo. De hecho, el segundo bucle es en realidad un bucle dentro de un bucle, ya que queremos ser capaces de reproducir la solución muchas veces. He aquí un esbozo del código:

```
1.  // This function is called once, from main.c.
2.  void maze_solve()
3.  {
4.      while(1)
5.      {
6.          // FIRST MAIN LOOP BODY
7.          // (when we find the goal, we use break; to get out of this)
```

```
    }
    // Now enter an infinite loop - we can re-run the maze as many
    // times as we want to.
    while(1)
    {
        // Beep to show that we finished the maze.
        // Wait for the user to press a button...
        int i;
        for(i=0;i<path_length;i++)
        {
            // SECOND MAIN LOOP BODY
        }
        // Follow the last segment up to the finish.
        follow_segment();
        // Now we should be at the finish!  Restart the loop.
    }
}
```

El primer bucle principal necesita para seguir un segmento del circuito, decidir cómo girar y recordar el giro en una variable. Para pasar los argumentos correctos a select_turn (), tenemos que examinar cuidadosamente la intersección a medida que la atraviesa. Tenga en cuenta que existe una excepción especial para encontrar el final del laberinto. El siguiente código funciona bastante bien, al menos a velocidad lenta que estamos utilizando:

```
// FIRST MAIN LOOP BODY
follow_segment();
// Drive straight a bit.  This helps us in case we entered the
// intersection at an angle.
// Note that we are slowing down - this prevents the robot
// from tipping forward too much.
set_motors(50,50);
delay_ms(50);
// These variables record whether the robot has seen a line to the
// left, straight ahead, and right, whil examining the current
// intersection.
unsigned char found_left=0;
unsigned char found_straight=0;
unsigned char found_right=0;
// Now read the sensors and check the intersection type.
unsigned int sensors[5];
read_line(sensors,IR_EMITTERS_ON);
// Check for left and right exits.
if(sensors[0] > 100)
    found_left = 1;
if(sensors[4] > 100)
    found_right = 1;
// Drive straight a bit more - this is enough to line up our
// wheels with the intersection.
set_motors(40,40);
delay_ms(200);
// Check for a straight exit.
read_line(sensors,IR_EMITTERS_ON);
if(sensors[1] > 200 || sensors[2] > 200 || sensors[3] > 200)
    found_straight = 1;
// Check for the ending spot.
// If all three middle sensors are on dark black, we have
// solved the maze.
if(sensors[1] > 600 && sensors[2] > 600 && sensors[3] > 600)
    break;
// Intersection identification is complete.
// If the maze has been solved, we can follow the existing
// path.  Otherwise, we need to learn the solution.
unsigned char dir = select_turn(found_left, found_straight, found_right);
// Make the turn indicated by the path.
turn(dir);
// Store the intersection in the path variable.
path[path_length] = dir;
path_length ++;
// You should check to make sure that the path_length does not
// exceed the bounds of the array.  We'll ignore that in this
// example.
// Simplify the learned path.
simplify_path();
// Display the path on the LCD.
display_path();
```

Podemos discutir la llamada a simplify_path() en la sección siguiente. Antes de eso, echemos un vistazo al segundo bucle principal, que es muy simple. Todo lo que hacemos es seguir hasta la nueva intersección y girar de acuerdo a nuestros registros. Después de hacer el último giro, el robot estará a un segmento de la meta, lo que explica la llamada final a follow_segment () en el maze_solve () anterior.

```
// SECOND MAIN LOOP BODY
follow_segment();
// Drive straight while slowing down, as before.
set_motors(50,50);
delay_ms(50);
set_motors(40,40);
delay_ms(200);
// Make a turn according to the instruction stored in
// path[i].
turn(path[i]);
```

7.e Simplificando la solución

Después de cada giro la longitud de lo recordado se incrementa en 1. Si tu laberinto, por ejemplo tiene largos zigzags sin salida las consecuencias será un ‘RLRLRLRL’ en la LCD. No hay atajo que te lleve a través de esta sección por una ruta más rápida, sólo la estrategia de la mano izquierda en la pared.

Sin embargo, cuando nos encontramos con un callejón sin salida, podemos simplificar el camino. Considera la posibilidad de la secuencia "LBL", donde "B" significa "volver" y las medidas adoptadas cuando se encuentra un callejón sin salida. Esto es lo que sucede si existe un giro a izquierda en una vía recta que conduce de inmediato a un callejón sin salida. Después de girar 90 ° a izquierda, 180º a derecha, y 90 ° de nuevo a izquierda, el efecto es que el robot se dirige en la dirección original de nuevo. La ruta puede ser simplificada con giro de 0 °: un único 'S'.

Otro ejemplo es la intersección en T con un callejón sin salida a izquierda: ‘LBS’. El giro será 90° izquierda, 180°, y 0°, para un total de 90° a derecha. La secuencia puede repetirse reemplazándola con un simple ‘R’.

En efecto, siempre que tengamos la secuencia del tipo ‘xBx’, podemos reemplazar los tres giros con un giro que sea el Angulo total, eliminando el giro U y acelerando la solución. El código será:

```
// Path simplification.  The strategy is that whenever we encounter a
// sequence xBx, we can simplify it by cutting out the dead end.  For
// example, LBL -> S, because a single S bypasses the dead end
// represented by LBL.
void simplify_path()
{
    // only simplify the path if the second-to-last turn was a 'B'
    if(path_length < 3 || path[path_length-2] != 'B')
        return;
    int total_angle = 0;
    int i;
    for(i=1;i<=3;i++)
    {
        switch(path[path_length-i])
        {
        case 'R':
            total_angle += 90;
            break;
        case 'L':
            total_angle += 270;
            break;
        case 'B':
            total_angle += 180;
            break;
        }
    }
    // Get the angle as a number between 0 and 360 degrees.
    total_angle = total_angle % 360;
    // Replace all of those turns with a single one.
    switch(total_angle)
    {
    case 0:
        path[path_length - 3] = 'S';
```

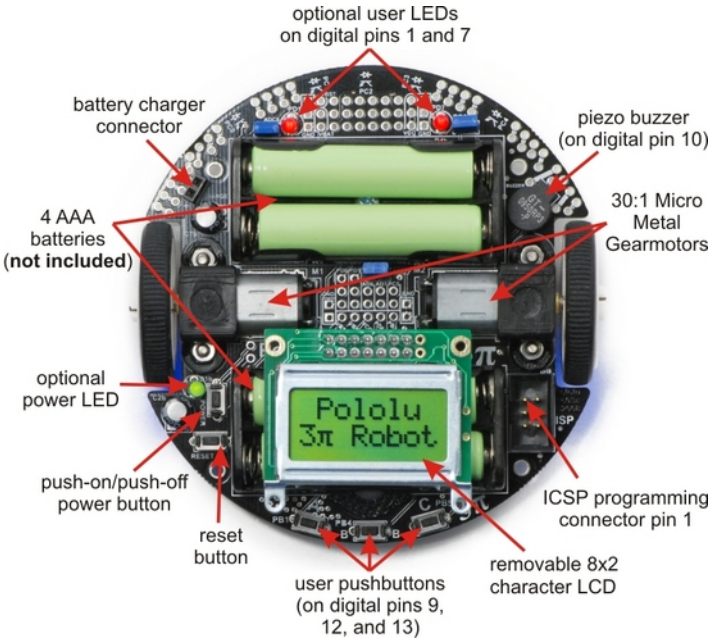
```
        break;
    case 90:
        path[path_length - 3] = 'R';
        break;
    case 180:
        path[path_length - 3] = 'B';
        break;
    case 270:
        path[path_length - 3] = 'L';
        break;
    }
    // The path is now two steps shorter.
    path_length -= 2;
}
```

Un punto interesante de este código es que hay algunas secuencias que nunca se encontrarán con un giro a izquierda del robot, como 'RBR', ya que son reemplazadas por 'S' según lo acordado. En muchos programas avanzados es posible que desee realizar un seguimiento de las incoherencias de este tipo, ya que indican que alguna vez este tipo de problema podría causar que el robot pierda el control.

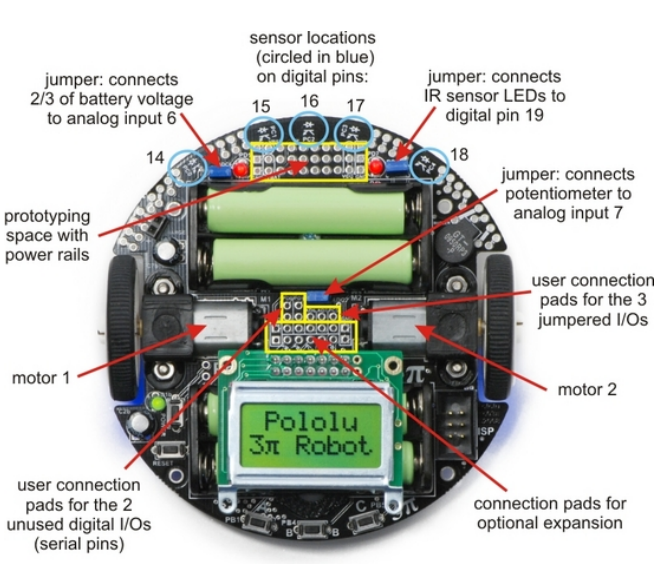
7.f. Mejorar el código de solución del laberinto

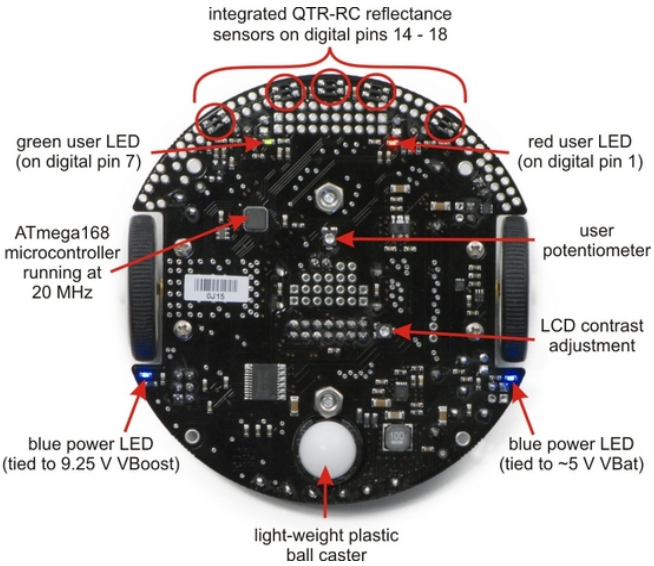
Hemos ido por las partes más importantes del código, los otros bits y trozos (como la función display_path (), la secuencia de puesta en marcha y de calibración, etc) se puede encontrar con todo lo demás en la carpeta `examples\3pi-mazesolver`. Después de tener el código trabajando y de entenderlo bien, debes tratar de mejorar tu robot para ser tan rápido como el viento. Hay muchas cosas se puede hacer para descubrir y mejorar el código:

- Incrementar la velocidad de seguimiento de la línea.
- Mejorar las variables PID.
- Incrementar la velocidad de giro.
- Identificar situaciones cuando al robot se "le va la olla" .
- Ajuste de la velocidad en base a lo que se viene, por ejemplo, conducción recta a través de una 'S' a toda velocidad.



Características del Pololu 3pi robot





8. Tablas de asignación de pins

Tabla de asignación de PINS según función

función	Arduino Pin	ATmega168 Pin
I/O digitales (x3) (quita jumper PC5 para liberar el pin 19 digital)	digital pins 0, 1, 19	PD0, PD1, PC5
Entradas analógicas (quita los jumpers, x3)	analog inputs 5 – 7	PC5, ADC6, ADC7
motor 1 (izquierdo) control (A y B)	digital pins 5 y 6	PD5 y PD6
motor 2 (derecho) control (A y B)	digital pins 3 y 11	PD3 y PB3
QTR-RC sensores de reflexión (izquierda a der, x5)	digital pins 14 – 18	PC0 – PC4
rojo (izquierda) LED de usuario	digital pin 1	PD1
verde (derecha) LED de usuario	digital pin 7	PD7
Botones de usuario (left to right, x3)	digital inputs 9, 12, y 13	PB1, PB4, y PB5
Buzzer	digital pin 10	PB2
LCD control (RS, R/W, E)	digital pins 2, 8, y 4	PD2, PB0, y PD4
LCD data (4-bit: DB4 – DB7)	digital pins 9, 12, 13, y 7	PB1, PB4, PB5, y PD7
sensor IR LED control drive low to turn IR LEDs off)	digital pin 19 (through jumper)	PC5
trimmer potenciometro de usuario	analog input 7 (con jumper)	ADC7
2/3rds de voltaje de la batería	analog input 6 (con jumper)	ADC6
ICSP lienas de programación (x3)	digital pins 11, 12, y 13	PB3, PB4, PB5
Botón de REST	reset	PC6
UART (RX y TX)	digital pins 0 y 1	PD0 y PD1
I2C/TWI	inaccesible al usuario	
SPI	inaccesible al usuario	

Tabla de asignación de PINS por pin

ATmega168 Pin	Orangutan función	Notas/Funciones alternativas
PD0	free digital I/O	USART input pin (RXD)
PD1	free digital I/O	conectado LED rojo de ususario (high turns LED on) USART output pin (TXD)
PD2	LCD control RS	interrupción 0 externa (INT0)
PD3	M2 línea control	Timer2 PWM output B (OC2B)
PD4	LCD control E	USART reloj externo input/output (XCK) Timer0 contador externo (T0)
PD5	M1 línea de control	Timer0 PWM output B (OC0B)
PD6	M1 línea de control	Timer0 PWM output A (OC0A)
PD7	LCD datos DB7	Conectado al LED ver de usuario (high turns LED on)
PB0	LCD control R/W	Timer1 input capture (ICP1) divided system clock output (CLK0)
PB1	LCD datos DB4	Boton de usuario (pulsando pulls pin low) Timer1 PWM salida A (OC1A)
PB2	Buzzer	Timer1 PWM salida B (OC1B)
PB3	M2 linea de control	Timer2 PWM salida A (OC2A) ISP linea de programación
PB4	LCD datos DB5	Boton de usuario (pulsando pulls pin low) Cuidado: también como linea de programación ISP
PB5	LCD datos DB6	Botón de usuario (pulsando pulls pin low) Cuidado: también como linea de programación ISP
PC0	QTR-RC sensor reflexión	(esta alto durante 10 us, espera entrada de linea para pasar a bajo) Sensor etiquetado como PC0 (sensor mas a izquierda)
PC1	QTR-RC sensor reflexión	(esta alto durante 10 us, espera entrada de linea para pasar a bajo) sensor etiquetado como PC1

PC2	QTR-RC sensor reflexión	(esta alto durante 10 us, espera entrada de linea para pasar a bajo) sensor etiquetado como PC2 (sensor central)
PC3	QTR-RC sensor reflexión	(esta alto durante 10 us, espera entrada de linea para pasar a bajo) sensor etiquetado como PC3
PC4	QTR-RC sensor reflexión	(esta alto durante 10 us, espera entrada de linea para pasar a bajo) sensor etiquetado como PC4 (sensor mas a derecha)
PC5	Entrada analogica y I/O digital	jumpered to sensors' IR LEDs (driving low turns off emitters) ADC input channel 5 (ADC5)
ADC6	Entrada dedicada analogica	jumpered to 2/3rds of battery voltage ADC input channel 6 (ADC6)
ADC7	Entrada dedicada analogica	jumpered to user trimmer potentiometer ADC input channel 7 (ADC7)
reset	Boton de RESET	internally pulled high; active low digital I/O disabled by default

9. Información para su expansión

9.a. Programa serie para esclavo

La librería Pololu AVR (ver sección 5.a) viene con un ejemplo de programa esclavo-serie para 3pi en `libpololu-avr\examples\3pi-serial-slave`, y el correspondiente programa maestro de comunicación serie en `libpololu-avr\examples\3pi-serial-master` Este ejemplo muestra cómo utilizar un bucle anidado en modo SERIAL_CHECK para recibir e interpretar un conjunto simple de comandos. Los comandos de control de diversas funciones del 3pi, por lo que es posible utilizar el 3pi como "base tonta" controlada por otro procesador situado en la placa de ampliación.
Es fácil añadir más comandos o adaptar la biblioteca a trabajar en una placa diferente.
La documentación completa de las funciones serie usadas se encuentra en **Section 9** de la Pololu AVR Library Command Reference.
Este programa esclavo recibes datos vía serie en el port PD0 (RX) del 3pi y transmite respuestas (si es necesario) en el port PD1 (TX), a velocidad de 115.200 baudios , nivel de protocolo serie TTL.
En este ejemplo, sin paridad, 8 bits de datos, y un stop bit (115200,N,8,1). Los comandos ejecutados están compuestos de un solo byte de comando seguido por cero o más bytes de datos. Para hacer más fácil la diferenciación entre byte de comando y byte de datos, los comandos están todos en la gama de 0x80-0xFF (1xxxxxxx), mientras que los bytes de datos se encuentran en la gama 0x00-0x7F (0xxxxxxx). Es decir, los comandos tienen el séptimo bit a 1 y los datos están a 0.
Algunos comandos resultan del envió de datos para el control del 3pi. Para los comandos en donde los enteros se envían de vuelta, el byte menos significativo es enviado primero (little endian). Si comandos o bytes de datos se detectan erróneos, el programa emite un pitido y muestra un mensaje de error en la pantalla de la LCD. Esto significa que si estás utilizando el kit de expansión sin cortes, probablemente debas eliminar los comandos relacionados con la pantalla LCD relacionados antes de cargar el programa en tu 3pi.
Los siguientes comandos son reconocidos por el programa:

Byte Comando	Comando	Byte datos	Bytes Respuesta	Descripción
0x81	signature	0	6	Envía el nombre y la versión, Ej. "3pi1.0". Este comando siempre por los motores a 0 y para el PID seguimiento de línea, si está activo, por lo que es útil como comando de inicialización.
0x86	raw sensors	0	10	Lee los cinco sensores IR y manda los valores en secuencias de 2 bytes enteros, en el rango de 0-2000
0x87	calibrated sensors	0	10	Lee los cinco sensores IR y envía los valores calibrados en el rango de 0-1000.
0xB0	trimpot	0	2	Envía la salida de voltaje del trimpot en dos bytes en el rango de 0-1023.

0xB1	battery millivolts	0	2	Envía el voltaje de la batería en mV en dos bytes.
0xB3	play music	2-101	0	Toca una melodía especificada en una cadena de comandos musicales. El primer byte es la longitud de la cadena (máx. 100) para que el programa sepa cuantos bytes debe leer. Ver comando play() en Section 4 de la Pololu AVR Library Command Reference Para saber su funcionamiento.
0xB4	calibrate	0	0	Realiza una calibración de los sensores. Debería realizarse varias veces ya que el robot se mueve en el rango del blanco y el negro.
0xB5	reset calibration	0	0	Resetear la calibración. Esto debería utilizarse cuando hay conectado un esclavo, en caso de reinicio del maestro sin restablecer el esclavo, por ejemplo, un fallo de energía.
0xB6	line position	0	2	Lee los cinco sensores IR usando valores calibrados y estima la posición de la línea negra debajo del robot. El valor a enviar será 0 cuando la línea está debajo del sensor PC0 o más a la izquierda, 1000, cuando la línea está bajo el sensor de PC1, hasta 4000, cuando está en el sensor PC4 o más a la derecha. Ver Section 12 de Pololu AVR Library Command Reference para la formula usada para posicionarse.
0xB7	clear LCD	0	0	Limpia la pantalla LCD del 3pi.
0xB8	print	2-9	0	Imprime 1-8 caracteres a la LCD. El primer byte es la longitud de la cadena de caracteres.
0xB9	LCD goto xy	2	0	Mueve el cursor de LCD a x-y carácter, línea (2 bytes).
0xBA	autocalibrate	0	1	Gira el robot a derecha e izquierda para calibrar. Se usa para posicionarlo sobre la línea. Devuelve el carácter ‘c’ calibrado.
0xBB	start PID	5	0	Ajusta los parámetros PID y comienza el seguimiento de línea. El primer byte de datos es la velocidad máxima de los motores. Los cuatro siguientes a, b, c, d representan los parámetros, la diferencia en la velocidad de los motores viene dada por la expresión (L-2000)×a/b + D×c/d, en donde L es la posición de la línea y D la derivada de ésta L. La integral term no está en este programa. Ver Section 6.c para más información acerca del seguimiento de línea PID.
0xBC	stop PID	0	0	Para el seguimiento de línea PID motores a 0.
0xC1	M1 forward	1	0	Motor M1 gira adelante a una velocidad de 0 (paro) hasta 127 (máximo avance).
0xC2	M1 backward	1	0	Motor M1 gira atrás con una velocidad entre 0 (paro) hasta 127 (máximo retroceso).
0xC5	M2 forward	1	0	Motor M2 gira adelante a una velocidad de 0 (paro) hasta 127 (máximo avance).
0xC6	M2 backward	1	0	Motor M2 gira atrás con una velocidad entre 0 (paro) hasta 127 (máximo retroceso).

Código fuente

```
#include <pololu/3pi.h>
/*
 * 3pi-serial-slave - An example serial slave program for the Pololu
 * 3pi Robot.  See the following pages for more information:
 * http://www.pololu.com/docs/0J21
 * http://www.pololu.com/docs/0J20
 * http://www.poolu.com/
 */
// PID constants
unsigned int pid_enabled = 0;
unsigned char max_speed = 255;
unsigned char p_num = 0;
unsigned char p_den = 0;
unsigned char d_num = 0;
unsigned char d_den = 0;
unsigned int last_proportional = 0;
unsigned int sensors[5];

// This routine will be called repeatedly to keep the PID algorithm running
void pid_check()
{
    if(!pid_enabled)
        return;

    // Do nothing if the denominator of any constant is zero.
    if(p_den == 0 || d_den == 0)
    {
        set_motors(0,0);
        return;
    }

    // Read the line position, with serial interrupts running in the background.
    serial_set_mode(SERIAL_AUTOMATIC);
    unsigned int position = read_line(sensors, IR_EMITTERS_ON);
    serial_set_mode(SERIAL_CHECK);

    // The "proportional" term should be 0 when we are on the line.
    int proportional = ((int)position) - 2000;

    // Compute the derivative (change) of the position.
    int derivative = proportional - last_proportional;

    // Remember the last position.
    last_proportional = proportional;

    // Compute the difference between the two motor power settings,
    // m1 - m2.  If this is a positive number the robot will turn
    // to the right.  If it is a negative number, the robot will
    // turn to the left, and the magnitude of the number determines
    // the sharpness of the turn.
    int power_difference = proportional*p_num/p_den + derivative*p_num/p_den;

    // Compute the actual motor settings.  We never set either motor
    // to a negative value.
    if(power_difference > max_speed)
        power_difference = max_speed;
    if(power_difference < -max_speed)
        power_difference = -max_speed;

    if(power_difference < 0)
        set_motors(max_speed+power_difference, max_speed);
    else
        set_motors(max_speed, max_speed-power_difference);
}

// A global ring buffer for data coming in.  This is used by the
// read_next_byte() and previous_byte() functions, below.
char buffer[100];

// A pointer to where we are reading from.
unsigned char read_index = 0;

// Waits for the next byte and returns it.  Runs play_check to keep
// the music playing and serial_check to keep receiving bytes.
// Calls pid_check() to keep following the line.
char read_next_byte()
{
```

```
78.     while(serial_get_received_bytes() == read_index)
79.     {
80.         serial_check();
81.         play_check();
82.
83.         // pid_check takes some time; only run it if we don't have more bytes to process
84.         if(serial_get_received_bytes() == read_index)
85.             pid_check();
86.
87.     }
88.     char ret = buffer[read_index];
89.     read_index ++;
90.     if(read_index >= 100)
91.         read_index = 0;
92.     return ret;
93. }
94.
95. // Backs up by one byte in the ring buffer.
96. void previous_byte()
97. {
98.     read_index --;
99.     if(read_index == 255)
100.                read_index = 99;
101.        }
102.
103.        // Returns true if and only if the byte is a command byte (>= 0x80).
104.        char is_command(char byte)
105.        {
106.            if (byte < 0)
107.                return 1;
108.            return 0;
109.        }
110.
111.        // Returns true if and only if the byte is a data byte (< 0x80).
112.        char is_data(char byte)
113.        {
114.            if (byte < 0)
115.                return 0;
116.            return 1;
117.        }
118.
119.        // If it's not a data byte, beeps, backs up one, and returns true.
120.        char check_data_byte(char byte)
121.        {
122.            if(is_data(byte))
123.                return 0;
124.
125.            play("o3c");
126.
127.            clear();
128.            print("Bad data");
129.            lcd_goto_xy(0,1);
130.            print_hex_byte(byte);
131.
132.            previous_byte();
133.            return 1;
134.        }
135.
136.        ////////////////////////////////////////////////////////////////////
137.        // COMMAND FUNCTIONS
138.        //
139.        // Each function in this section corresponds to a single serial
140.        // command.  The functions are expected to do their own argument
141.        // handling using read_next_byte() and check_data_byte().
142.
143.        // Sends the version of the slave code that is running.
144.        // This function also shuts down the motors and disables PID, so it is
145.        // useful as an initial command.
146.        void send_signature()
147.        {
148.            serial_send_blocking("3pi1.0", 6);
149.            set_motors(0,0);
150.            pid_enabled = 0;
151.        }
152.
153.        // Reads the line sensors and sends their values.  This function can
154.        // do either calibrated or uncalibrated readings.  When doing calibrated readings,
155.        // it only performs a new reading if we are not in PID mode.  Otherwise, it sends
156.        // the most recent result immediately.
157.        void send_sensor_values(char calibrated)
```

```
158.        {
159.            if(calibrated)
160.            {
161.                if(!pid_enabled)
162.                    read_line_sensors_calibrated(sensors, IR_EMITTERS_ON);
163.            }
164.            else
165.                read_line_sensors(sensors, IR_EMITTERS_ON);
166.            serial_send_blocking((char *)sensors, 10);
167.        }
168.
169.        // Sends the raw (uncalibrated) sensor values.
170.        void send_raw_sensor_values()
171.        {
172.            send_sensor_values(0);
173.        }
174.
175.        // Sends the calibated sensor values.
176.        void send_calibrated_sensor_values()
177.        {
178.            send_sensor_values(1);
179.        }
180.
181.        // Computes the position of a black line using the read_line()
182.        // function, and sends the value.
183.        // Returns the last value computed if PID is running.
184.        void send_line_position()
185.        {
186.            int message[1];
187.            unsigned int tmp_sensors[5];
188.            int line_position;
189.
190.            if(pid_enabled)
191.                line_position = last_proportional+2000;
192.            else line_position = read_line(tmp_sensors, IR_EMITTERS_ON);
193.
194.            message[0] = line_position;
195.
196.            serial_send_blocking((char *)message, 2);
197.        }
198.
199.        // Sends the trimpot value, 0-1023.
200.        void send_trimpot()
201.        {
202.            int message[1];
203.            message[0] = read_trimpot();
204.            serial_send_blocking((char *)message, 2);
205.        }
206.
207.        // Sends the batter voltage in millivolts
208.        void send_battery_millivolts()
209.        {
210.            int message[1];
211.            message[0] = read_battery_millivolts();
212.            serial_send_blocking((char *)message, 2);
213.        }
214.
215.        // Drives m1 forward.
216.        void m1_forward()
217.        {
218.            char byte = read_next_byte();
219.
220.            if(check_data_byte(byte))
221.                return;
222.
223.            set_m1_speed(byte == 127 ? 255 : byte*2);
224.        }
225.
226.        // Drives m2 forward.
227.        void m2_forward()
228.        {
229.            char byte = read_next_byte();
230.
231.            if(check_data_byte(byte))
232.                return;
233.
234.            set_m2_speed(byte == 127 ? 255 : byte*2);
235.        }
236.
237.        // Drives m1 backward.
```

```
238.        void m1_backward()
239.        {
240.            char byte = read_next_byte();
241.
242.            if(check_data_byte(byte))
243.                return;
244.
245.            set_m1_speed(byte == 127 ? -255 : -byte*2);
246.        }
247.
248.        // Drives m2 backward.
249.        void m2_backward()
250.        {
251.            char byte = read_next_byte();
252.
253.            if(check_data_byte(byte))
254.                return;
255.
256.            set_m2_speed(byte == 127 ? -255 : -byte*2);
257.        }
258.
259.        // A buffer to store the music that will play in the background.
260.        char music_buffer[100];
261.
262.        // Plays a musical sequence.
263.        void do_play()
264.        {
265.            unsigned char tune_length = read_next_byte();
266.
267.            if(check_data_byte(tune_length))
268.                return;
269.
270.            unsigned char i;
271.            for(i=0;i<tune_length;i++)
272.            {
273.                if(i > sizeof(music_buffer)) // avoid overflow
274.                    return;
275.
276.                music_buffer[i] = read_next_byte();
277.
278.                if(check_data_byte(music_buffer[i]))
279.                    return;
280.            }
281.
282.            // add the end of string character 0
283.            music_buffer[i] = 0;
284.
285.            play(music_buffer);
286.        }
287.
288.        // Clears the LCD
289.        void do_clear()
290.        {
291.            clear();
292.        }
293.
294.        // Displays data to the screen
295.        void do_print()
296.        {
297.            unsigned char string_length = read_next_byte();
298.
299.            if(check_data_byte(string_length))
300.                return;
301.
302.            unsigned char i;
303.            for(i=0;i<string_length;i++)
304.            {
305.                unsigned char character;
306.                character = read_next_byte();
307.
308.                if(check_data_byte(character))
309.                    return;
310.
311.                // Before printing to the LCD we need to go to AUTOMATIC mode.
312.                // Otherwise, we might miss characters during the lengthy LCD routines.
313.                serial_set_mode(SERIAL_AUTOMATIC);
314.                 print_character(character);
315.                serial_set_mode(SERIAL_CHECK);
316.            }
317.        }
```

```
318.
319.        // Goes to the x,y coordinates on the lcd specified by the two data bytes
320.        void do_lcd_goto_xy()
321.        {
322.            unsigned char x = read_next_byte();
323.            if(check_data_byte(x))
324.                return;
325.
326.            unsigned char y = read_next_byte();
327.            if(check_data_byte(y))
328.                return;
329.
330.            lcd_goto_xy(x,y);
331.        }
332.
333.        // Runs through an automatic calibration sequence
334.        void auto_calibrate()
335.        {
336.            time_reset();
337.            set_motors(60, -60);
338.            while(get_ms() < 250)
339.                calibrate_line_sensors(IR_EMITTERS_ON);
340.            set_motors(-60, 60);
341.            while(get_ms() < 750)
342.                calibrate_line_sensors(IR_EMITTERS_ON);
343.            set_motors(60, -60);
344.            while(get_ms() < 1000)
345.                calibrate_line_sensors(IR_EMITTERS_ON);
346.            set_motors(0, 0);
347.
348.            serial_send_blocking("c",1);
349.        }
350.
351.        // Turns on PID according to the supplied PID constants
352.        void set_pid()
353.        {
354.            unsigned char constants[5];
355.            unsigned char i;
356.            for(i=0;i<5;i++)
357.            {
358.                constants[i] = read_next_byte();
359.                if(check_data_byte(constants[i]))
360.                    return;
361.            }
362.
363.            // make the max speed 2x of the first one, so that it can reach 255
364.            max_speed = (constants[0] == 127 ? 255 : constants[0]*2);
365.
366.            // set the other parameters directly
367.            p_num = constants[1];
368.            p_den = constants[2];
369.            d_num = constants[3];
370.            d_den = constants[4];
371.
372.            // enable pid
373.            pid_enabled = 1;
374.        }
375.
376.        // Turns off PID
377.        void stop_pid()
378.        {
379.            set_motors(0,0);
380.            pid_enabled = 0;
381.        }
382.
383.        ////////////////////////////////////////////////////////////////////
384.
385.        int main()
386.        {
387.            pololu_3pi_init(2000);
388.            play_mode(PLAY_CHECK);
389.
390.            clear();
391.            print("Slave");
392.
393.            // start receiving data at 115.2 kbaud
394.            serial_set_baud_rate(115200);
395.            serial_set_mode(SERIAL_CHECK);
396.            serial_receive_ring(buffer, 100);
397.
```

```
398.            while(1)
399.            {
400.                // wait for a command
401.                char command = read_next_byte();
402.
403.                // The list of commands is below: add your own simply by
404.                // choosing a command byte and introducing another case
405.                // statement.
406.                switch(command)
407.                {
408.                case (char)0x00:
409.                    // slient error - probable master resetting
410.                    break;
411.
412.                case (char)0x81:
413.                    send_signature();
414.                    break;
415.                case (char)0x86:
416.                    send_raw_sensor_values();
417.                    break;
418.                case (char)0x87:
419.                    send_calibrated_sensor_values(1);
420.                    break;
421.                case (char)0xB0:
422.                    send_trimpot();
423.                    break;
424.                case (char)0xB1:
425.                    send_battery_millivolts();
426.                    break;
427.                case (char)0xB3:
428.                    do_play();
429.                    break;
430.                case (char)0xB4:
431.                    calibrate_line_sensors(IR_EMITTERS_ON);
432.                    send_calibrated_sensor_values(1);
433.                    break;
434.                case (char)0xB5:
435.                    line_sensors_reset_calibration();
436.                    break;
437.                case (char)0xB6:
438.                    send_line_position();
439.                    break;
440.                case (char)0xB7:
441.                    do_clear();
442.                    break;
443.                case (char)0xB8:
444.                    do_print();
445.                    break;
446.                case (char)0xB9:
447.                    do_lcd_goto_xy();
448.                    break;
449.                case (char)0xBA:
450.                    auto_calibrate();
451.                    break;
452.                case (char)0xBB:
453.                    set_pid();
454.                    break;
455.                case (char)0xBC:
456.                    stop_pid();
457.                    break;
458.
459.                case (char)0xC1:
460.                    m1_forward();
461.                    break;
462.                case (char)0xC2:
463.                    m1_backward();
464.                    break;
465.                case (char)0xC5:
466.                    m2_forward();
467.                    break;
468.                case (char)0xC6:
469.                    m2_backward();
470.                    break;
471.
472.                default:
473.                    clear();
474.                    print("Bad cmd");
475.                    lcd_goto_xy(0,1);
476.                    print_hex_byte(command);
477.
```

```
                    play("o7l16crc");
                    continue; // bad command
                }
            }
        }
```

9.b Programa serie para maestro

El programa maestro serie usado en el programa esclavo está incluido en la librería Pololu AVR Library (ver Section 5.a) en `libpololu-avr\examples\3pi-serial-master`. Está diseñado para correr con un LV-168 o en el 3pi como una demostración de lo que es posible, pero probablemente quieras adaptarlo a tu propio controlador. Para utilizar el programa, debes hacer las siguientes conexiones entre maestro y esclavo:

GND-GND	PD0-PD1	PD1-PD0

Enciende ambos maestro y esclavo. El master mostrará un mensaje “Connect” seguido de la versión del código esclavo (Ej. “3pi1.0”). El maestro da instrucciones al esclavo para mostrar “Connect” y tocar una pequeña melodía. Pulsa el botón B en el maestro lo que causará que el esclavo entre en la rutina de auto-calibration, después puedes manejar el esclavo usando los botones a y C en el maestro, mientras ves los datos de los sensores en la LCD.
Si pulsas B el esclavo entra en seguimiento de línea PID.

Código fuente

```
#include <pololu/orangutan.h>
#include <string.h>
/*
 * 3pi-serial-master - An example serial master program for the Pololu
 * 3pi Robot.  This can run on any board supported by the library;
 * it is intended as an example of how to use the master/slave
 * routines.
 *
 * http://www.pololu.com/docs/0J21
 * http://www.pololu.com/docs/0J20
 * http://www.poolu.com/
 */
// 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 6 levels of a bar graph
// plus a back arrow and a musical note character.
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+4,3); // skip level 3
    lcd_load_custom_character(levels+5,4);
    lcd_load_custom_character(levels+6,5);
    clear(); // the LCD must be cleared for the characters to take effect
}

// 10 levels of bar graph characters
const char bar_graph_characters[10] = {' ',0,0,1,2,3,3,4,5,255};
```

```
50.
51. void display_levels(unsigned int *sensors)
52. {
53.     clear();
54.     int i;
55.     for(i=0;i<5;i++) {
56.         // Initialize the array of characters that we will use for the
57.         // graph.  Using the space, an extra copy of the one-bar
58.         // character, and character 255 (a full black box), we get 10
59.         // characters in the array.
60.
61.         // The variable c will have values from 0 to 9, since
62.         // values are in the range of 0 to 1000, and 1000/101 is 9
63.         // with integer math.
64.         char c = bar_graph_characters[sensors[i]/101];
65.
66.         // Display the bar graph characters.
67.         print_character(c);
68.     }
69. }
70.
71. // set the motor speeds
72. void slave_set_motors(int speed1, int speed2)
73. {
74.     char message[4] = {0xC1, speed1, 0xC5, speed2};
75.     if(speed1 < 0)
76.     {
77.         message[0] = 0xC2; // m1 backward
78.         message[1] = -speed1;
79.     }
80.     if(speed2 < 0)
81.     {
82.         message[2] = 0xC6; // m2 backward
83.         message[3] = -speed2;
84.     }
85.     serial_send_blocking(message,4);
86. }
87.
88. // do calibration
89. void slave_calibrate()
90. {
91.     serial_send("\xB4",1);
92.     int tmp_buffer[5];
93.
94.     // read 10 characters (but we won't use them)
95.     serial_receive_blocking((char *)tmp_buffer, 10, 100);
96. }
97.
98. // reset calibration
99. void slave_reset_calibration()
100.        {
101.            serial_send_blocking("\xB5",1);
102.        }
103.
104.        // calibrate (waits for a 1-byte response to indicate completion)
105.        void slave_auto_calibrate()
106.        {
107.            int tmp_buffer[1];
108.            serial_send_blocking("\xBA",1);
109.            serial_receive_blocking((char *)tmp_buffer, 1, 10000);
110.        }
111.
112.        // sets up the pid constants on the 3pi for line following
113.        void slave_set_pid(char max_speed, char p_num, char p_den, char d_num, char d_den)
114.        {
115.            char string[6] = "\xBB";
116.            string[1] = max_speed;
117.            string[2] = p_num;
118.            string[3] = p_den;
119.            string[4] = d_num;
120.            string[5] = d_den;
121.            serial_send_blocking(string,6);
122.        }
123.
124.        // stops the pid line following
125.        void slave_stop_pid()
126.        {
127.            serial_send_blocking("\xBC", 1);
128.        }
129.
```

```
130.        // clear the slave LCD
131.        void slave_clear()
132.        {
133.            serial_send_blocking("\xB7",1);
134.        }
135.
136.        // print to the slave LCD
137.        void slave_print(char *string)
138.        {
139.            serial_send_blocking("\xB8", 1);
140.            char length = strlen(string);
141.            serial_send_blocking(&length, 1); // send the string length
142.            serial_send_blocking(string, length);
143.        }
144.
145.        // go to coordinates x,y on the slave LCD
146.        void slave_lcd_goto_xy(char x, char y)
147.        {
148.            serial_send_blocking("\xB9",1);
149.            serial_send_blocking(&x,1);
150.            serial_send_blocking(&y,1);
151.        }
152.
153.        int main()
154.        {
155.            char buffer[20];
156.
157.            // load the bar graph
158.            load_custom_characters();
159.
160.            // configure serial clock for 115.2 kbaud
161.            serial_set_baud_rate(115200);
162.
163.            // wait for the device to show up
164.            while(1)
165.            {
166.                clear();
167.                print("Master");
168.                delay_ms(100);
169.                serial_send("\x81",1);
170.
171.                if(serial_receive_blocking(buffer, 6, 50))
172.                    continue;
173.
174.                clear();
175.                print("Connect");
176.                lcd_goto_xy(0,1);
177.                buffer[6] = 0;
178.                print(buffer);
179.
180.                // clear the slave's LCD and display "Connect" and "OK" on two lines
181.                // Put OK in the center to test x-y positioning
182.                slave_clear();
183.                slave_print("Connect");
184.                slave_lcd_goto_xy(3,1);
185.                slave_print("OK");
186.
187.                // play a tune
188.                char tune[] = "\xB3 l16o6gab>c";
189.                tune[1] = sizeof(tune)-3;
190.                serial_send_blocking(tune,sizeof(tune)-1);
191.
192.                // wait
193.                wait_for_button(ALL_BUTTONS);
194.
195.                // reset calibration
196.                slave_reset_calibration();
197.
198.                time_reset();
199.
200.                slave_auto_calibrate();
201.
202.                unsigned char speed1 = 0, speed2 = 0;
203.
204.                // read sensors in a loop
205.                while(1)
206.                {
207.                    serial_send("\x87",1); // returns calibrated sensor values
208.
209.                    // read 10 characters
```

```
                    if(serial_receive_blocking(buffer, 10, 100))
                        break;
                    // get the line position
                    serial_send("\xB6", 1);

                    int line_position[1];
                    if(serial_receive_blocking((char *)line_position, 2, 100))
                        break;
                    // get the battery voltage
                                serial_send("\xB1",1);

                    // read 2 bytes
                    int battery_millivolts[1];
                    if(serial_receive_blocking((char *)battery_millivolts, 2, 100))
                        break;

                    // display readings
                    display_levels((unsigned int*)buffer);

                    lcd_goto_xy(5,0);
                    line_position[0] /= 4; // to get it into the range of 0-1000
                    if(line_position[0] == 1000)
                        line_position[0] = 999; // to keep it to a maximum of 3 characters
                    print_long(line_position[0]);
                    print("   ");
                    lcd_goto_xy(0,1);
                    print_long(battery_millivolts[0]);
                    print(" mV  ");
                    delay_ms(10);
                    // if button A is pressed, increase motor1 speed
                    if(button_is_pressed(BUTTON_A) && speed1 < 127)
                        speed1 ++;
                    else if(speed1 > 1)
                        speed1 -= 2;
                    else if(speed1 > 0)
                        speed1 = 0;

                    // if button C is pressed, control motor2
                    if(button_is_pressed(BUTTON_C) && speed2 < 127)
                        speed2 ++;
                    else if(speed2 > 1)
                        speed2 -= 2;
                    else if(speed2 > 0)
                        speed2 = 0;

                    // if button B is pressed, do PID control
                    if(button_is_pressed(BUTTON_B))
                        slave_set_pid(40, 1, 20, 3, 2);
                    else
                    {
                        slave_stop_pid();
                        slave_set_motors(speed1, speed2);
                    }
                }
            }

            while(1);
        }
```

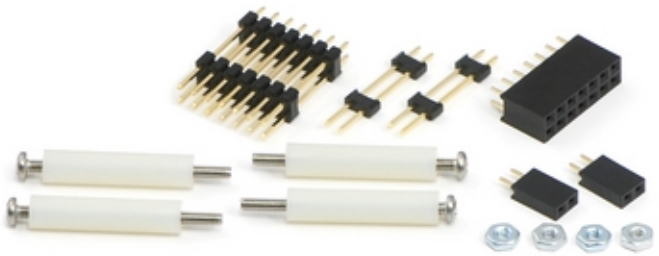
10. Enlaces relacionados

Para leer más acerca del tu robot Pololu 3pi, mira los siguientes enlaces:

- WinAVR
- AVR Studio
- Pololu AVR Library Command Reference: información detallada de cada función de la librería.
- Programar el 3pi Robot desde un entorno Arduino: una guía de programación del 3pi usando la interfaz Arduino IDE en lugar de AVR Studio.
- AVR Libc Home Page
- ATmega168 documentación
- Tutorial: AVR Programación en Mac

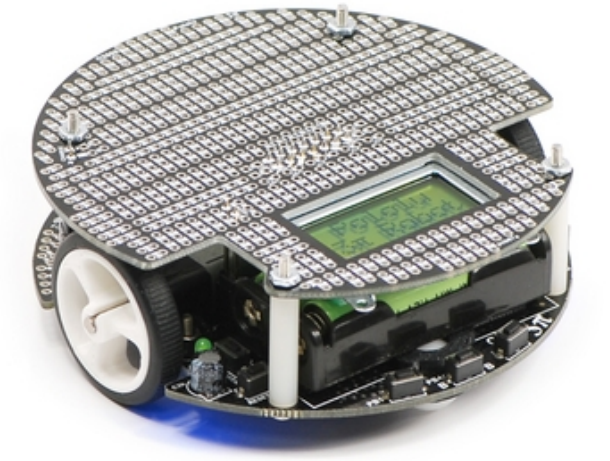
3pi kit de expansión

Este kit incluye una placa de circuito impreso (PCB) redondeada, con una parrilla de agujeros espaciados de 0,100", 1 (2 en pcb sin cortes) conector macho alargado y otro hembra de 2 × 7 pins, 2 conectores macho alargados y 2 hembras de 2 × 1 pins, 4 separadores de 7 / 8 " en plástico, 4 tornillos y sus tuercas de 1-1/4".

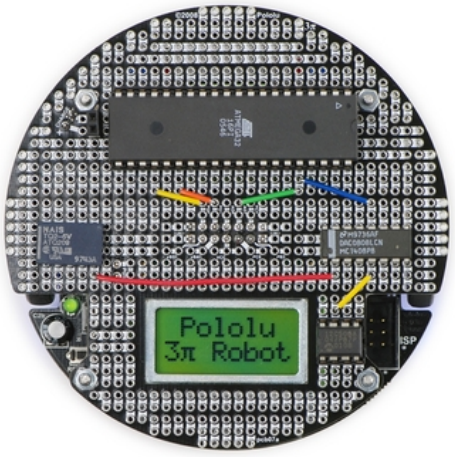


La placa de expansión coincide con la PCB de la 3 pi en color y diámetro y se monta justo por encima de las ruedas utilizando los cuatro tornillos y sus espaciadores. Una vez ensamblada la PCB se conecta a la base del 3pi lo que te permite crear tu propia interfaz electrónica (con soldaduras independientes) entre ambos circuitos. Estas conexiones te dan acceso a los pins libres del ATmega168, así como a los tres pins de tensiones: VBAT (voltaje de la batería), VCC (5 V regulados), y VBST (9,25 V regulados para los motores). Además, la expansión de PCB se conecta a la base del botón de encendido y al punto de carga de batería, lo que te permite añadir tus propios botones y conectores de carga.

El kit de ampliación del PCB que tiene cortes permite ver la pantalla de cristal líquido y el acceso al botón de encendido, botón de reinicio, y acceso al conector ISP de programación. Si necesitas más líneas de conexión o espacio extra y no vas a usar la pantalla LCD, está la versión del kit de expansión sin cortes, que aprovecha las líneas de la pantalla LCD.



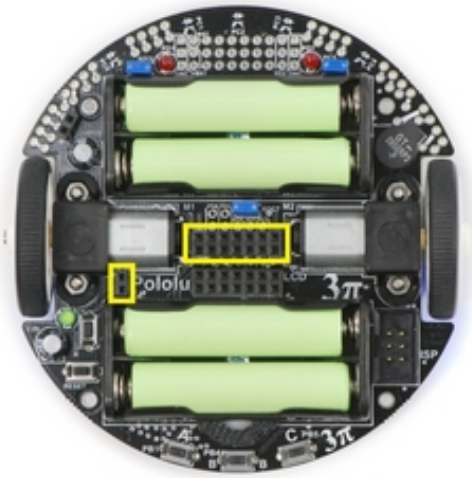
La placa de expansión está diseñada para crear un montón de prototipos con espacio suficiente para componentes. Tiene un espacio para 0,6 " para componentes de hasta 40-pin DIP (Dual in-line de paquetes) como el ATmega32 de la imagen o para numerosos componentes pequeños DIP. En el espacio del prototipo se extienden sus pistas hasta el borde de la PCB, permitiendo que puedas montar una variedad de sensores, tales como bumpers, de distancia. La serigrafía muestra cómo las pistas están conectadas, las conexiones eléctricas se encuentran en la parte inferior y puedes cortar el cobre de las pistas (con un cuchillo afilado o una pequeña herramienta de corte rotativo) si algunas de ellas interfieren en tu diseño.



Las dos líneas sin uso de E/S del microcontrolador del 3pi son las líneas de transmisión y recepción serie. Esto permite añadir un segundo microcontrolador u otras placas microcontroladas como Baby Orangutan, Basic Stamp, o Arduino Nano, a la placa de expansión. Este segundo microcontrolador se ocuparía de todos los sensores y del hardware adicional en la expansión y también del control de la base vía comandos serie. Es decir, la liberación vía serie del programa base de la 3pi y que se convertiría en una plataforma controlada que puede ser impulsada con órdenes desde otro microcontrolador.

Ensamblado

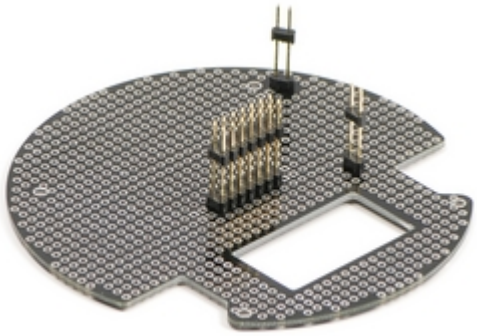
Los conectores de pins suministrados te permiten establecer todas las conexiones eléctricas necesarias entre la expansión y la base del 3pi. Recomendamos ensamblar la 3pi y el kit de expansión **antes de soldar nada**, esto asegurará que una vez hechas las soldaduras, la placa de expansión se alinea correctamente con la base. Puedes montar tu kit de expansión en el siguiente orden:



1.- Coloca el conector hembra de 2×7 y uno de los conectores hembras de 2×1 dentro de sus agujeros apropiados de la base del 3pi como vemos en la imagen (fíjate en los rectángulos amarillos).

2.- Inserta los pins del 2×7 y uno de los machos extendidos de 2×1 en los conectores hembras. Adicionalmente coloca el conector extendido macho de de 2×1 en el conector de carga de la batería. Coloca la placa de expansión encima de los pins machos y marca unos rectángulos como los de la imagen.

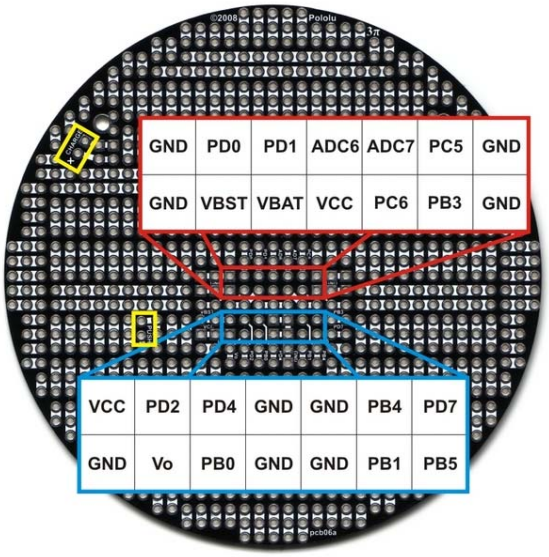
Importante: la placa de expansión se monta con la serigrafía hacia arriba.

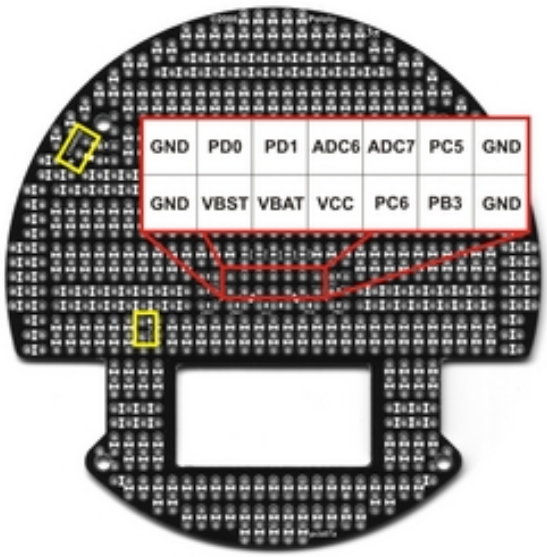


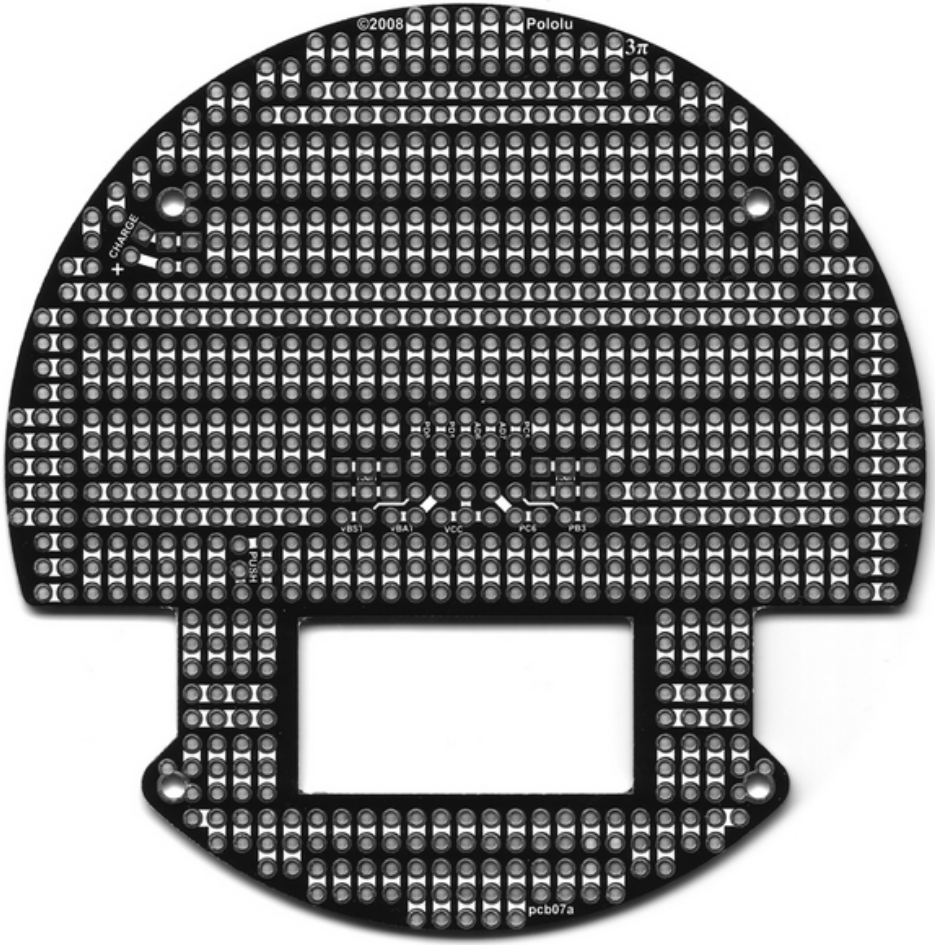
3.- Pon el separador de nylon entre la base y la expansión de PCB de manera que el montaje este en línea con el agujero en la base. Inserta un tornillo desde la parte inferior de la base a través del agujero de montaje, el espaciador y el agujero en la placa de expansión. Sosteniendo la cabeza del tornillo contra la base, gira la tuerca al otro lado, pero sin apretar del todo. Repite este proceso para los tres tornillos restantes y, a continuación, apretarlos todos a fin de que la expansión se ajuste bien con la base.

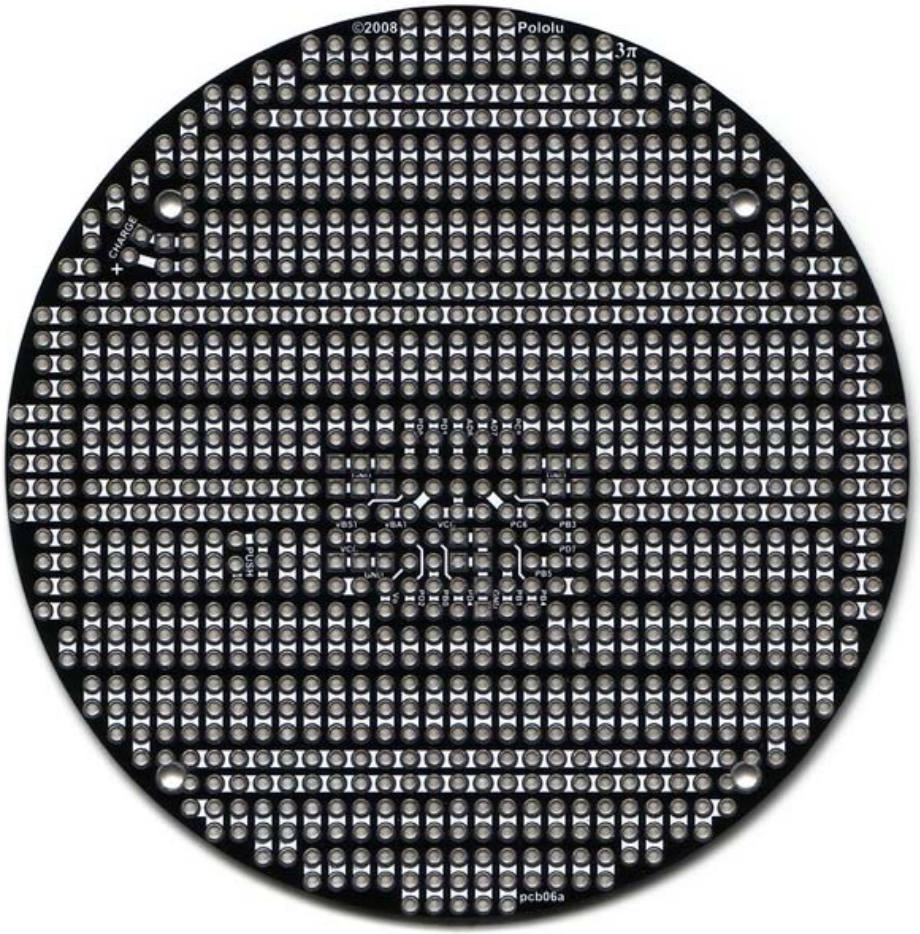
4.- Con los tornillos sujetándolo todo, ahora puedes soldar los conectores hembras a la base y los conectores macho a la PCB de expansión. Una vez que todo está soldado, puedes quitar los tornillos y sacar la placa de expansión fuera de la base; será algo parecido a lo vemos en las imágenes.

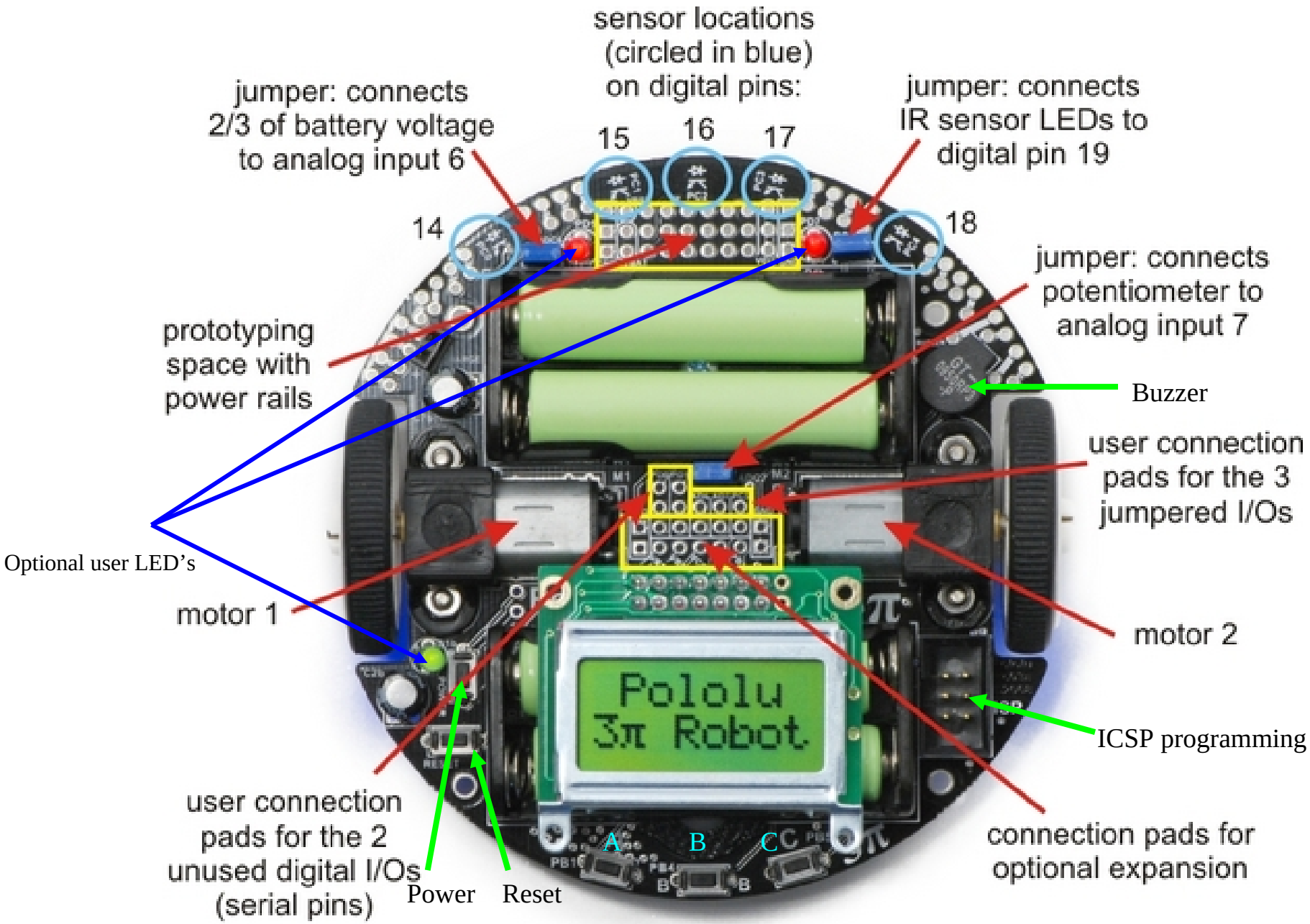
Después de ensamblar tendrás un conector hembra de 2×1 a la izquierda. Puede usar esto para crear tu propio punto de carga de batería en el puerto de expansión PCB.

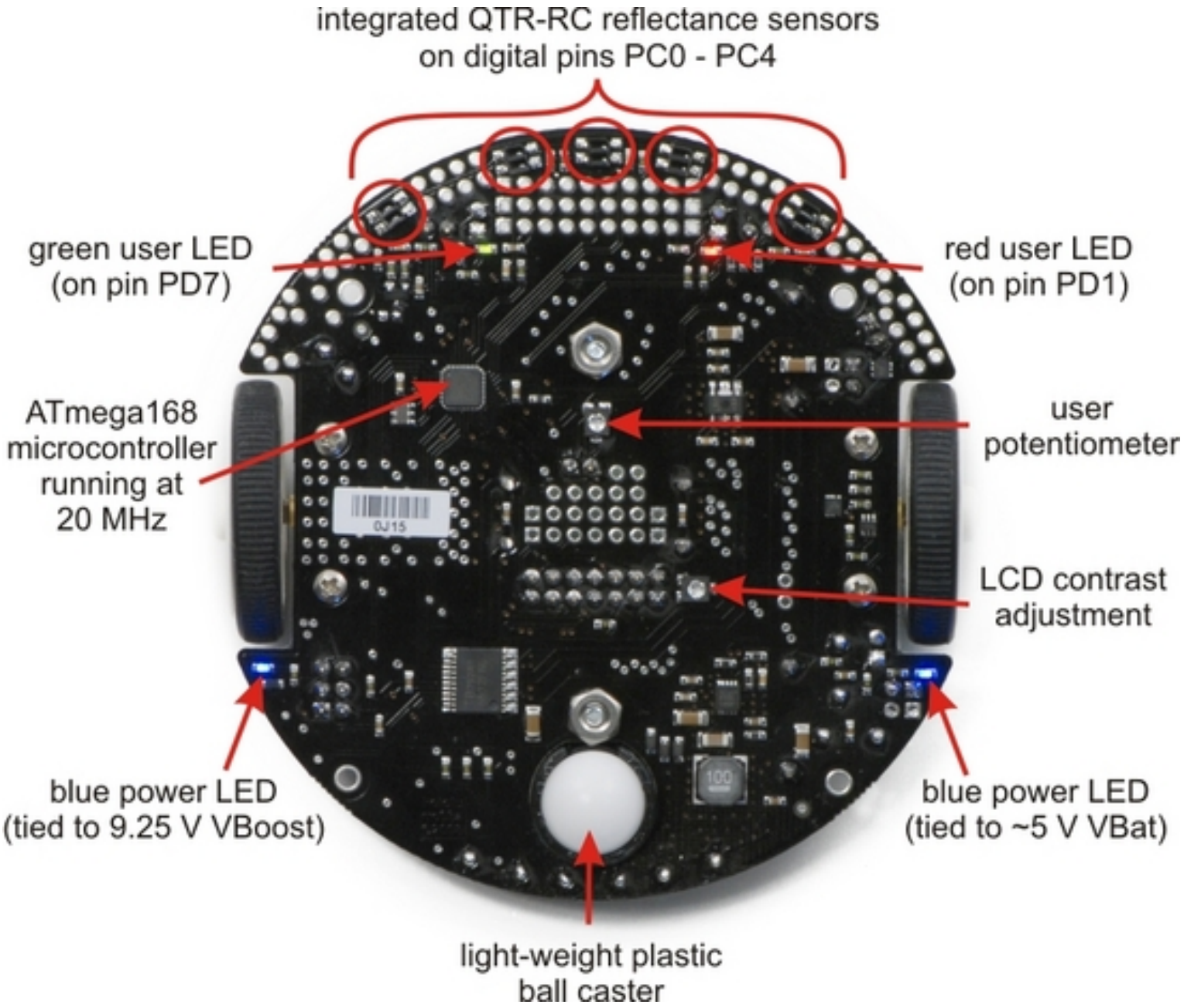





©2008 Pololu 3π
CHARGE
+
PUSH
pcb07a

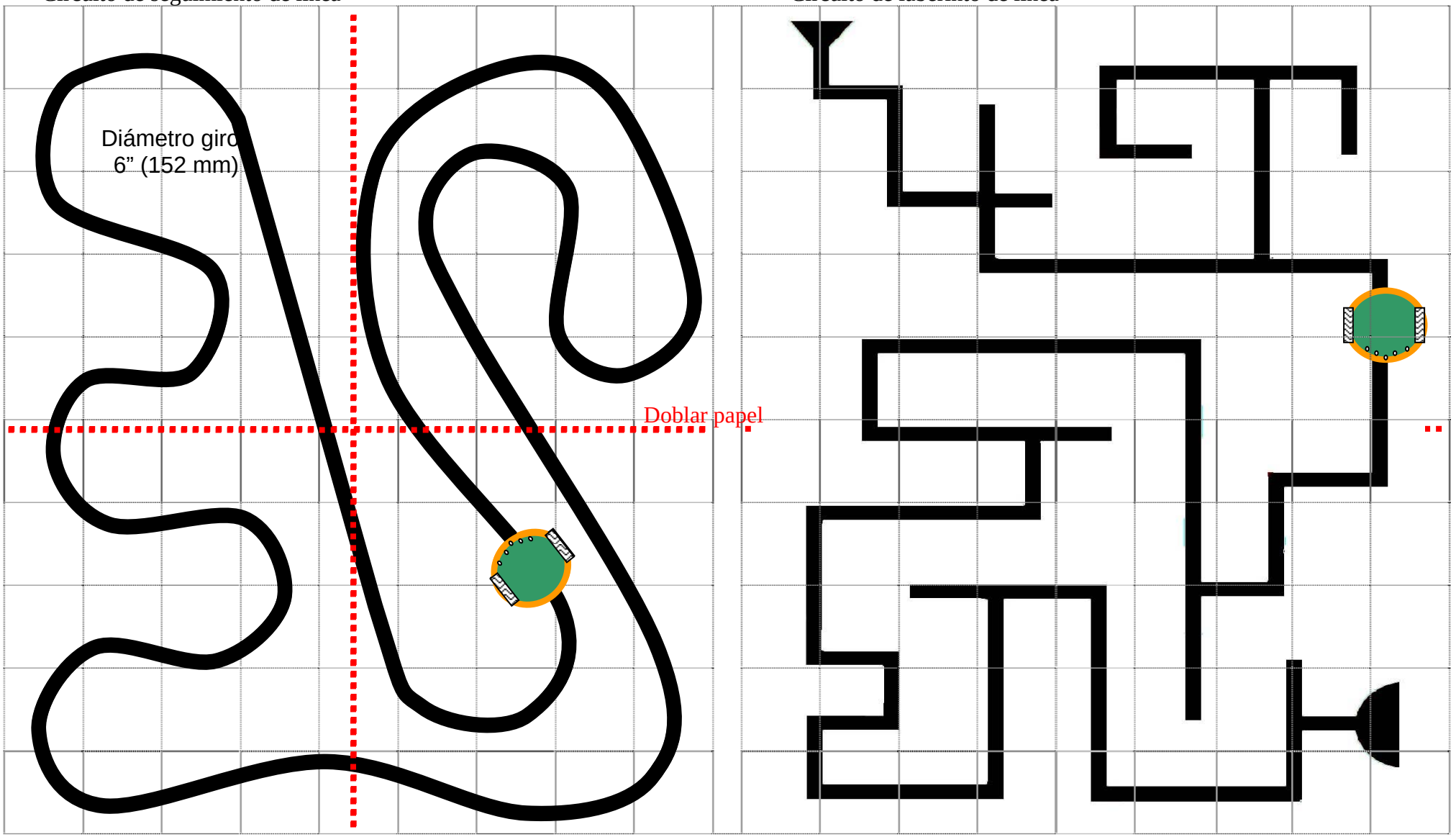

©2008 Pololu 3π
CHARGE
+
PUSH
pcb06a


sensor locations
(circled in blue)
on digital pins:
15
16
17
jumper: connects
2/3 of battery voltage
to analog input 6
jumper: connects
IR sensor LEDs to
digital pin 19
14
18
jumper: connects
potentiometer to
analog input 7
prototyping
space with
power rails
Buzzer
user connection
pads for the 3
jumpered I/Os
Optional user LED's
motor 1
motor 2
Pololu
3π Robot
ICSP programming
user connection
pads for the 2
unused digital I/Os
(serial pins)
A
B
C
Power
Reset
connection pads for
optional expansion


integrated QTR-RC reflectance sensors
on digital pins PC0 - PC4
green user LED
(on pin PD7)
red user LED
(on pin PD1)
ATmega168
microcontroller
running at
20 MHz
user
potentiometer
LCD contrast
adjustment
blue power LED
(tied to 9.25 V VBoost)
blue power LED
(tied to ~5 V VBat)
light-weight plastic
ball caster

Circuito de seguimiento de línea

Circuito de laberinto de línea



Crea tableros en papel blanco de 27” x 30“ (700x750mm) y dividelos en cuadros de 3x3” (7,6 mm)

Ancho vías
3/8” a 3/4” (10-19 mm)