Lil'Bot™

user manual

Table of Contents

Getting started

If you have purchased a kit, please assemble it according to the assembly instructions, available at Lil'Bot's website: **http://www.lil-bot.com/downloads/**. You must then go through the calibration procedure.

If you purchased an assembled Lil'Bot, it is already calibrated and should do a little demo for you right out of the box. Here is how to get it started:

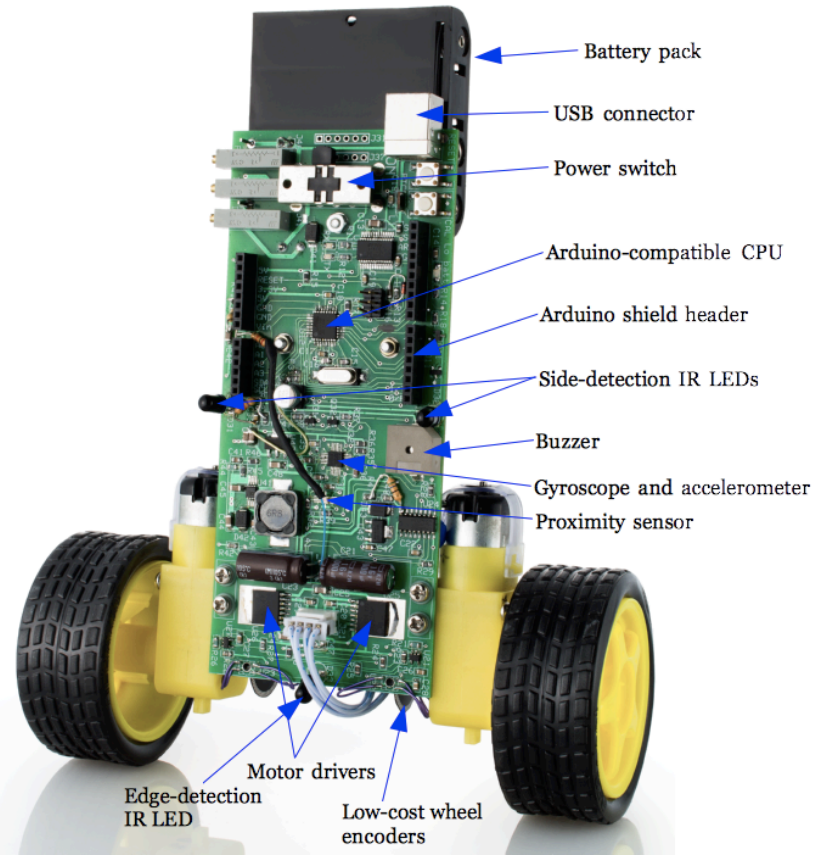
1. With the robot lying down, turn the power switch on.
2. Leave the robot undisturbed for a few seconds. Lil'Bot will beep once to indicate it has performed its internal housekeeping.
3. Lift Lil'Bot in the upright position and see it come to life. It will proceed to fine-tune its own balance point and its proximity sensors. Let it do its own thing away from obstacles for a few seconds. After a few seconds of self-calibration, it will run whatever Arduino user commands were last programmed.

Calibration

Lil'Bot needs to be calibrated every time the balance point changes significantly, for instance if you add or remove shields or accessories. You may also redo the calibration if you have any doubt about the balancing. Please read the entire sequence before starting calibration. Once the calibration procedure has begun, it must be completed in a timely manner — you only have a few seconds. It is very important to perform this step as precisely as possible, since that will affect the robot's balance and precise rotations.

1. Start with Lil'Bot powered down and lying on a stable, horizontal surface.
2. Power the robot up while holding the CAL button down. Hold the CAL button down until *two low beeps* are heard. Those indicate calibration mode.
3. Lift the robot up and rotate it along its vertical axis (along the main board's length), and put it down in the same resting position gently. The rotation may be in either direction. This must be completed with the robot back at rest before *three low beeps* are heard. Repeat the procedure if you ran out of time. This calibrates the gyroscope for precise robot rotations.
4. Bring Lil'Bot vertically onto its wheels and feel its balance point between your fingers. You must be able to hold it in balance very gently, feeling no force forward or backward between your fingers. Holding this balance point must be achieved before *two high beeps* are heard.
5. Hold the balance as gently and precisely as possible until *three high beeps* are heard. If the robot was not balanced properly the entire time between the two high beeps and the three high beeps, repeat the entire procedure.
6. Let Lil'Bot go. It may go a few inches forward and back while it is learning to balance itself. At the end of this last phase, it will play four musical tones, indicating that calibration has been completed and the calibration parameters have been stored into EEPROM (permanent memory). It will then proceed with whatever it is holding in its program memory.

Getting to know your robot



Battery pack and power supply

Lil'Bot requires seven AA batteries. The preferred non-rechargeable type is alkaline. For rechargeable batteries, nickel metal hydride (NiMH) batteries are ideally suited. The motors are powered by a nominal 10.5V battery voltage. As the batteries discharge, software compensates the lower voltage by increasing the PWM duty cycle in reverse proportion with the battery voltage. The battery voltage can be as low as 6V for Lil'Bot to operate. When the battery voltage reaches below 6.123V (875mV per cell), the LED blinks as a low-battery indicator.

The Arduino circuitry is powered by 5V, which is either regulated from the battery voltage when the power switch is on, or taken from the USB when Lil'Bot is connected to the computer. The proximity sensor and the IMU (gyroscope and accelerometer chip) are powered at 3.3V, regulated from the 5V supply.

USB connector and power

The USB port allows programming Lil'Bot from an Arduino environment. It also powers Lil'Bot, so that when the power switch is turned off, the following circuits can operate from the USB-provided 5V source:

- The Arduino portion of Lil'Bot, including the LED
- The buzzer
- The wheel encoders
- The proximity sensor
- The IMU

The motor drivers require the full battery voltage to operate.

For experimental purposes, an unpopulated DC-to-DC converter circuit based on Analog Devices' ADP1613 switcher controller is provided on Lil'Bot's printed circuit board.

Arduino hardware

Lil'Bot is build around the Arduino Uno architecture and is software and hardware compatible. Please refer to the hardware reference section for implementation details.

Just like the Arduino Uno, Lil'Bot accepts shields of the same form factor. Its emoShield is one such shield. Other shields can be added, provided no resource conflict exists.

Proximity sensor

The proximity sensor is based on Silicon Labs' Si1143. It is an extremely versatile I²C device. Lil'Bot uses three LEDs to detect front, right and left obstacles, and edges. Obstacle detection is accomplished by measuring infrared light emitted by an LED and reflected into the Si1143. The right and left LEDs have very broad radiant angles. Thus when the same amount of light is reflected from both LEDs, a front obstacle is inferred.

Current software does not implement edge detection.

Buzzer

The buzzer allows Lil'Bot to emit chirps patterned after astromech droid sounds. Its bandwidth is from a few hundred hertz to about 3kHz.

Gyroscope and accelerometer

The IMU is Invensense's six-axis MPU-6050 (three-axis gyroscope and three-axis accelerometer). This I²C device is programmed to take one set of measurements every ten milliseconds, which is the period of the balancing cycle. The software takes approximately two milliseconds to compute updated motor commands, and leaves the remaining time until the next IMU measurement for user programs.

Lil'Bot's right and left turns are controlled by one axis of the gyroscope.

Wheel encoders

The wheel encoders help determine Lil'Bot's backward and forward motion. A slot switch reads dark and transparent patterns from the encoder wheel. At each transition from transparent to opaque and back, a rising or falling logical signal is transmitted to the Arduino as an interrupt. The software infers the direction of motion from the polarity of the motor control signals, and the extent of motion from the number of transitions.

emoShield

The emoShield is based on yet another I²C device, the CAT9555 i/o expander. Sixteen different logic outputs are required to form all the different robot expressions Lil'Bot is capable of. Since the Arduino Uno does not provide enough digital outputs, this device is programmed from the I²C bus to control sixteen outputs, which power the emoShield's branches of LEDs.

Programming in the Arduino environment

Installation

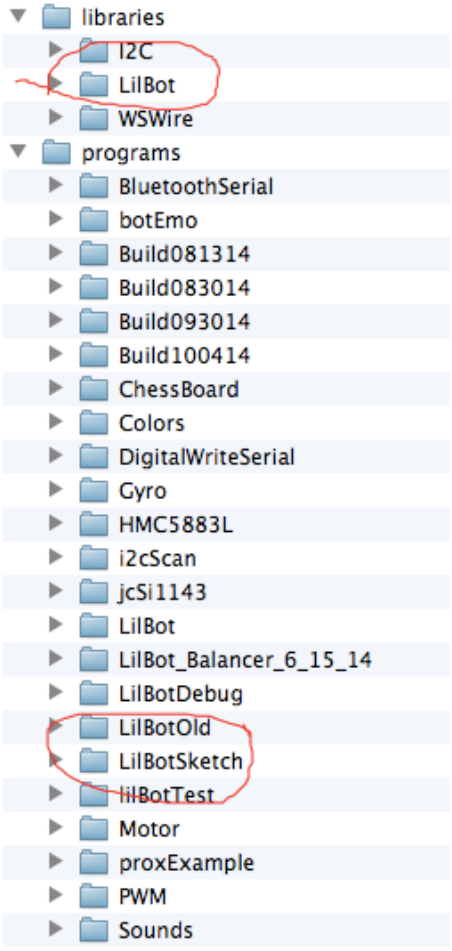
In order to program Lil'Bot, please install the latest Arduino development environment by following the instructions provided. Here is a link to the Arduino download area:

http://arduino.cc/en/Main/Software

Then download the latest Lil'Bot software by visiting the Lil'Bot download area:

http://www.lil-bot.com/downloads/

Copy the **LilBot** library folder to your Arduino library area, and copy the **LilBotSketch** folder to your Arduino program area. When you are done, your Arduino folders should look something like this:



Other than the **LilBot** folder in **libraries** and the **LilBotSketch** folder in **programs**, the exact contents of the Arduino installation depends on your system. Only those two folders are relevant. Now start the Arduino environment and open the **LilBotSketch**. Verify that the program compiles without error.

Connect Lil'Bot to the USB. In the Arduino application, go to Tools → Serial Port and select the USB port to which Lil'Bot is connected. Go to Tools → Board and select Arduino Uno, which is compatible with Lil'Bot. Then upload the program, and power Lil'Bot according to the procedure described in Getting Started.

If you want to dig much deeper into the robot code, you may want to look at the contents of the library files provided. This is definitely an advanced code exploration.

Minimum robot program

Now you are ready to make your robot do nothing other than balancing itself. The **LilBotSketch** contains template code, which may be tailored to do just that. Enter the following program:

```
#include "Wire.h"
#include "EEPROM.h"
#include <LilBot.h>

LilBot Bot;

void setup(void) {
  Bot.begin();
}

void loop(void) {
  Bot.balance();
}
```

The first few **#include** lines tell the Arduino compiler to load up some libraries of code that is necessary for robot operation. The LilBot library contains the balancing code, sensor operation, sound and emoShield routines, etc., in short, all that LilBot needs to operate without the user having to figure it all out from scratch.

The next line reads "**LilBot Bot;**" it is called instantiation. In the LilBot library, there is a C++ class named **LilBot**, which says how a good Lil'Bot should behave. Now we are telling the compiler that there exists such a robot of the **LilBot** class, which we are going to name **Bot** for short. All subsequent mention of out robot will refer to its name, **Bot**.

An Arduino program contains two main functions, **setup()** and **loop()**. The first function runs once after the power has been turned on or after the reset button has been pushed. In this case, it is used to initialize all the hardware, balancing parameters, etc., before anything useful can happen. After **setup()** has run once, the **loop()** function is executed indefinitely. At a minimum, we want our robot to stay upright and keep balancing, which is what the **Bot.balance()** function is for.

Making Lil'Bot do a square figure

Making Lil'Bot go in a square figure is almost as simple. Enter the same code as above, but replace the **loop()** function as follows:

```
void loop(void) {
  Bot.go(5000);
  Bot.rotate(90.0);
}
```

Since the **loop()** function is repeated indefinitely, it is not necessary to describe each side of the square. Just tell the robot to go straight for a number of odometry units, and then make a quarter turn left. Repeating this pattern will result in the robot going in a square figure. Now upload the program in the same manner as before.

Programming in Lil'Blocks

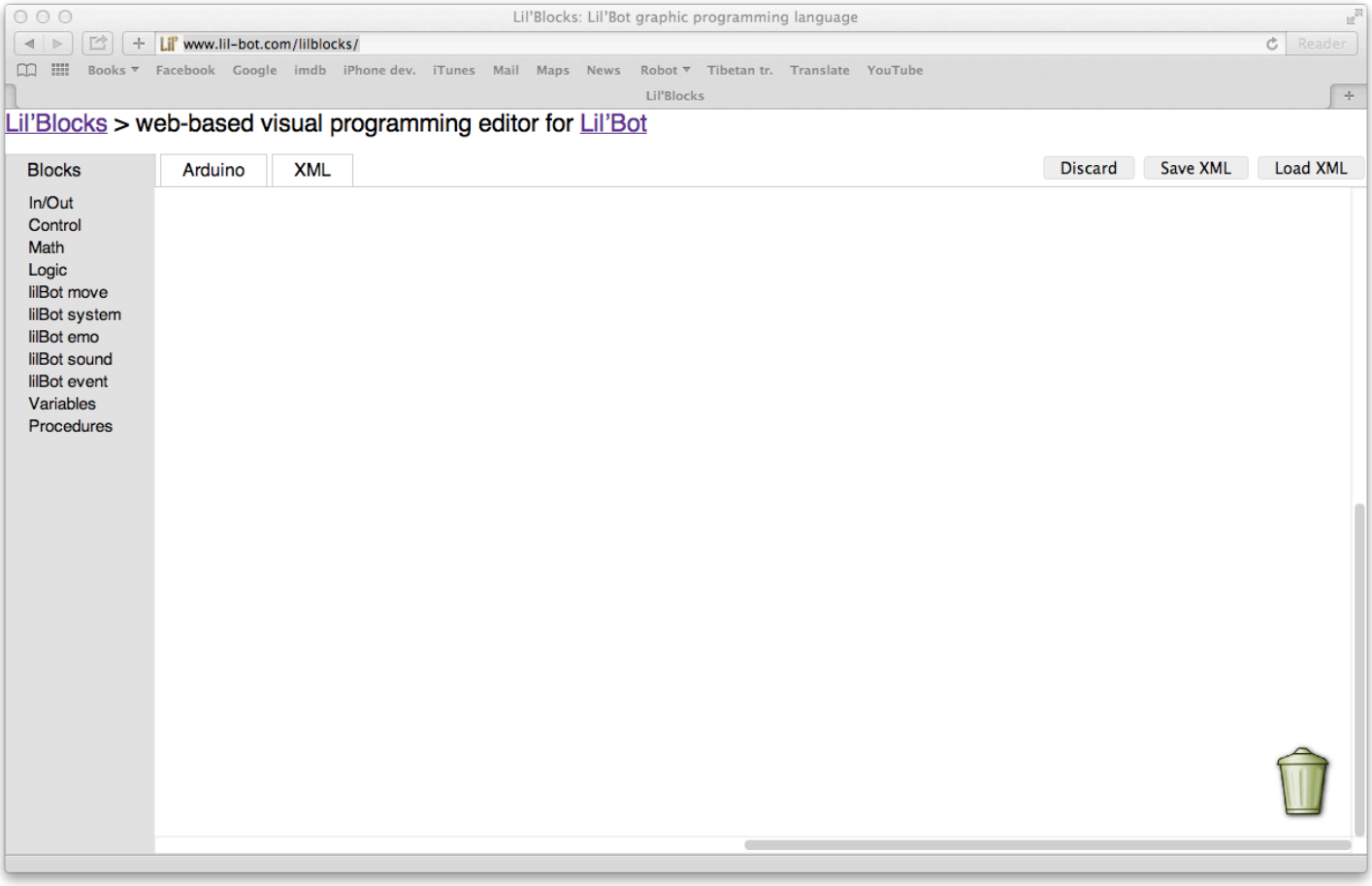
Getting started, and a minimum robot program

Lil'Blocks is a port of Fred Lin's BlocklyDuino, a port of Blockly that creates Arduino code. Blockly is a creation of Neil Fraser at Google. Blockly is the Hour of Code's language of choice.

In order to use Lil'Blocks, you must have the Arduino installation working, as described in the section above, including the Lil'Bot library. You may then use Lil'Blocks by visiting the Lil'Blocks web page. No additional installation is necessary. Please open the link below and bookmark it for easy access from your browser. Your browser must have JavaScript enabled.

http://www.lil-bot.com/lilblocks/

The Lil'Blocks page looks like this:



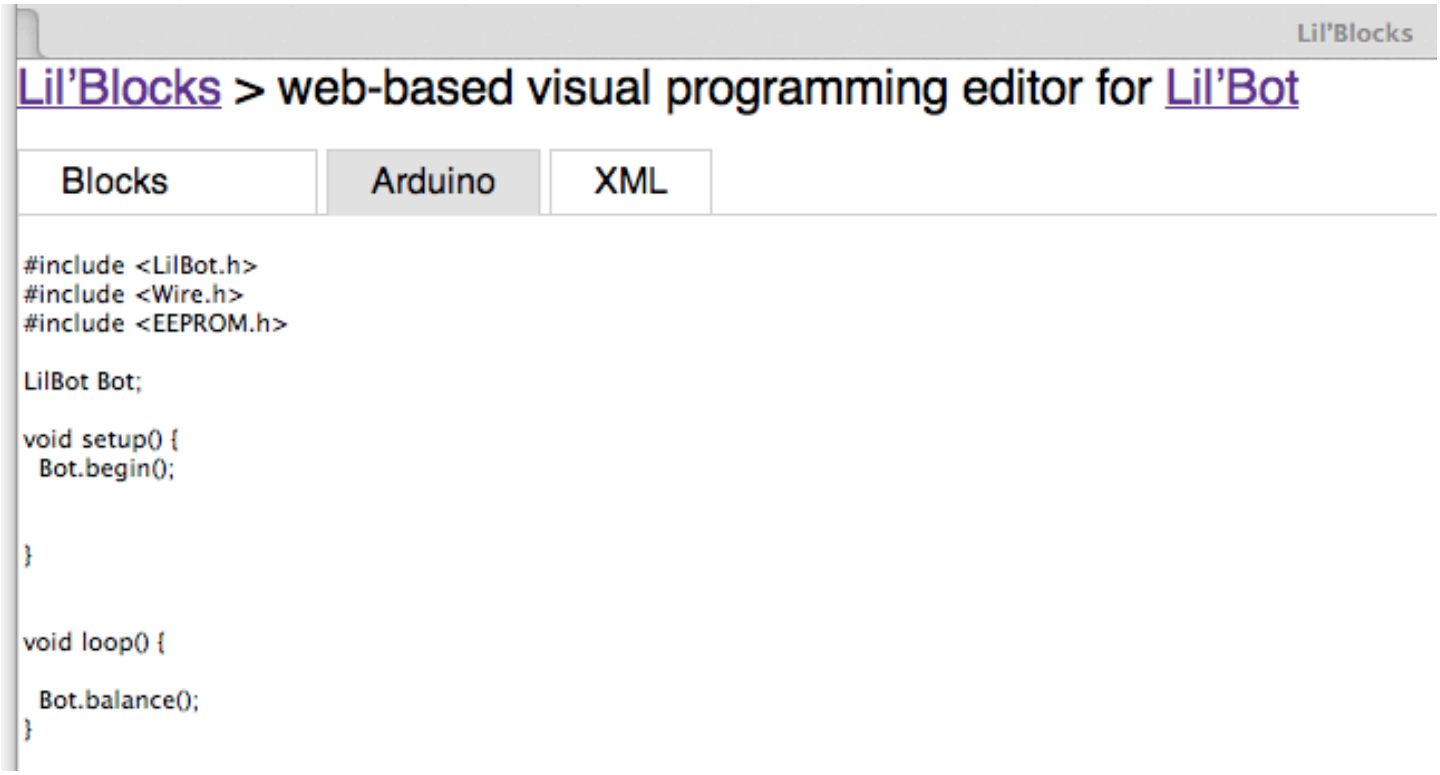
The Blocks tab, which is selected when you first open the page, contains a menu of blocks for you to build programs. Please take time to familiarize yourself with the various kinds of blocks. Further chapters in this manual go through the blocks systematically.

The XML tab contains an XML representation of the blocks. It is the format used to save your work. In order to save your block program, click Save XML. A new browser tab will open with the XML code for your blocks. You may save it to your computer, using the browser Save As command (usually in the File menu), choosing Page Source for the file format, and choosing a file name and location on your computer.

To load up previously saved blocks, click Load XML and select the file you saved previously.

Clicking on Discard will delete all the blocks in the work area.

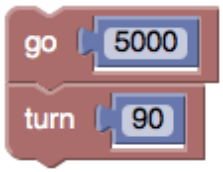
The Arduino tab contains the Arduino code that results from your Lil'Blocks program. When there is no block at all in the work area, there already is some minimal code. It looks like this:



Using the cursor, copy all the code and paste it into a blank Arduino project. Upload the program onto your Lil'Bot using the steps described in the previous section. Again, this is minimal code that will just let the robot stay balanced. As you add blocks to your program, code will be added in the **`loop()`** function.

Making Lil'Bot do a square figure

Assemble the following blocks:



Click on the Arduino tab, and be amazed by Lil'Blocks awesome power. Some code was created by translating your blocks into a complete Arduino program ready to compile and upload. Just as in the Arduino example above, only one side of a square needs to be described. Lil'Bot will repeat the pattern continuously.

If you are interested in finding out how the Arduino code works, please refer to the previous section.

For more information on BlocklyDuino:

`http://www.gasolin.idv.tw/public/blockly/demos/blocklyduino/index.html`

You may want to visit Blockly's page:

`https://blockly-games.appspot.com`

The Hour of Code is a great and lively introduction to coding using Blockly and other languages:

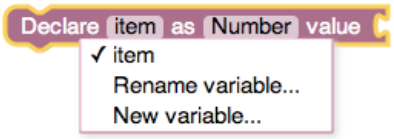
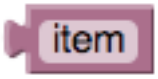
`http://code.org`

Programming reference

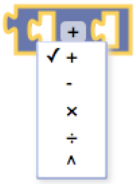
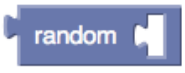
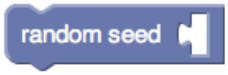
Blockly original blocks

Blockly provides a rich set of blocks, which can be mapped to many programming languages. In this case, Lil'Blocks and BlocklyDuino are mapped to Arduino C/C++. Lil'Blocks also supports the Lil'Bot hardware. The general blocks that are part of Lil'Blocks include provisions for variables, controls, loops, math and Boolean logic. Those are summarized in this section.

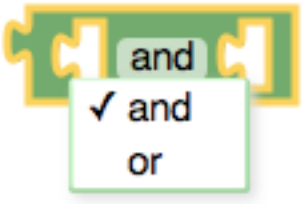
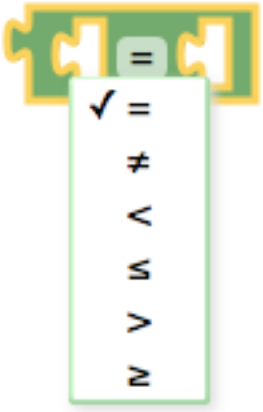
Variables

Block	Action
	Declare a variable. Only "Number" (integer) is supported.
	Set a previously-declared variable to a value
	Use a variable value

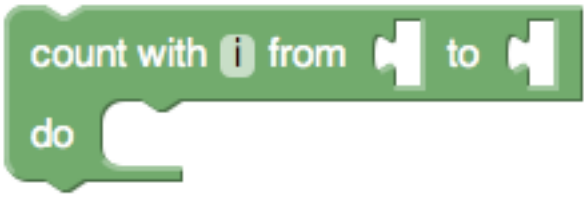
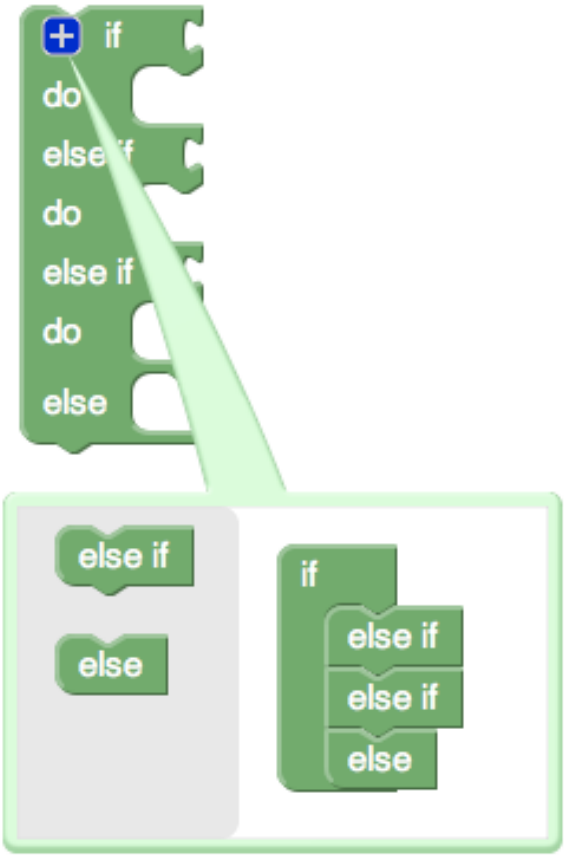
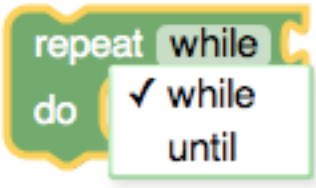
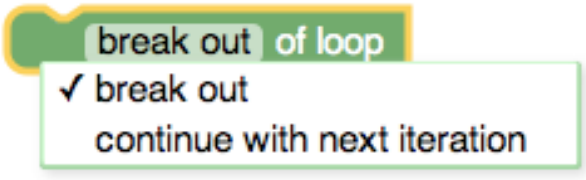
Math

Block	Action
	Immediate integer value
	Perform an operation. Addition, subtraction, multiplication, division, and raising to a power are supported.
	Produce a random number from zero to the number specified (not included).
	Seed the random-number generator.

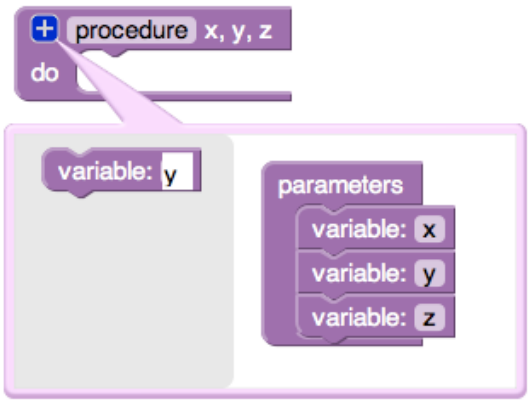
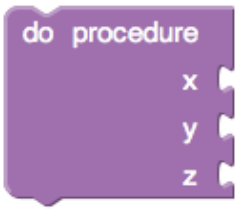
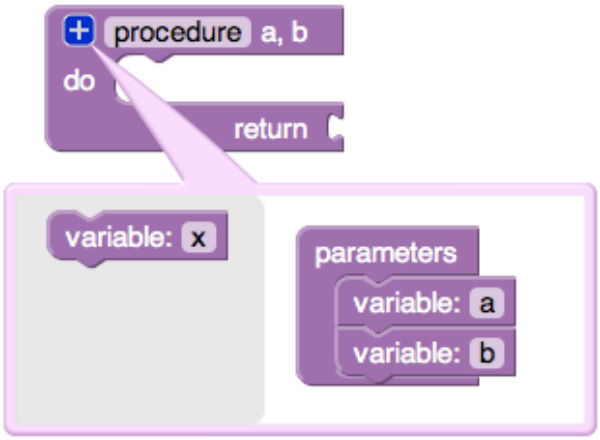
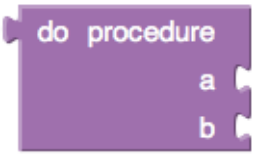
Logic

Block	Action
	Perform a logic operation
	Perform a comparison
	Negate a logic statement
	Always false

Control

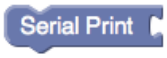
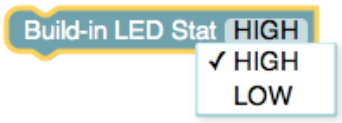
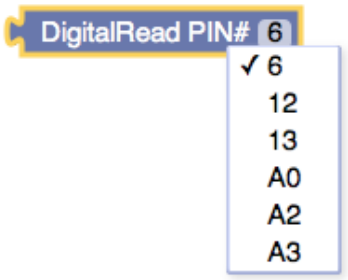
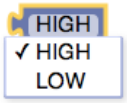
Block	Action
	Loop with index counter
	Build if / else if / else statement block
	Iterate a block of code while a condition is true or false
	Equivalent to C **`break`** or **`continue`** statement

Procedures

Block	Action
	Create a procedure block, which does not return a value, and declare its parameters
	Call a procedure block that does not return a value
	Create a procedure block, which returns a value, and declare its parameters
	Return a value from a procedure if a condition is met
	Call a procedure block that returns a value

Arduino input/output blocks

Lil'Blocks includes most of BlocklyDuino's original Arduino blocks. This includes blocks that read and write to and from digital and analog Arduino pins, and a block that prints to the console. The delay block (which maps the Arduino **`delay()`** function) was replaced with a wait block (which maps to **`Bot.wait()`**). Please refer to the wait block's description for more explanation as to why it was replaced.

Block	Action
	Print to the console. The output is available on a terminal connected to the Lil'Bot USB, e.g. the Arduino console.
	Turn the LED on (HIGH) or off (LOW). The led is periodically updated by the Lil'Bot software to indicate battery status.
	Read the state of a digital input pin. Only the pins not used by Lil'Bot are available.
	Set an output pin high or low
	Obtain the numerical equivalent of the HIGH or LOW state
	Measure the voltage at an analog input pin. Only the analog input pins not used by Lil'Bot are available.
	Set a PWM output pin to a value. Digital pin 6 is the only PWM-capable pin that isn't used by Lil'Bot.

Robot movement

Go

Lil'Block	Arduino	Action
	`Bot.go(long rawUnits);`	Go straight for the specified number of odometry units

Lil'Blocks example	Arduino example	Remarks
	`Bot.go(5000);`	The odometry is provided by the wheel encoders. It is not calibrated.

Turn

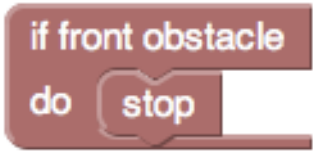
Lil'Block	Arduino	Action
	`Bot.rotate(double angle);`	Turn left (positive) or right (negative) a specified number of degrees

Lil'Blocks example	Arduino example
	`Bot.rotate(-90.0);`

Stop

Lil'Block	Arduino	Action
	`Bot.stop(void);`	Stop when an obstacle has been detected

Lil'Blocks example	Arduino example	Remark

	`void frontObstacle(void) {`
	`  Bot.stop();`
	`}`
		The robot stops by itself after completing motion commands. This command is required only for exception handling.

Robot sounds

Sound

Lil'Block	Arduino	Action
	`Bot.sound(int soundLine);`	Make one astromech droid sound sequence based on a line number from 0 to 48

Lil'Blocks example	Arduino example	Remarks
	`Bot.sound(random(0, 48));`	Each sound line is stored in a table as four parameters: duration, start tone, end tone, pause duration. See library source code for details.

Say

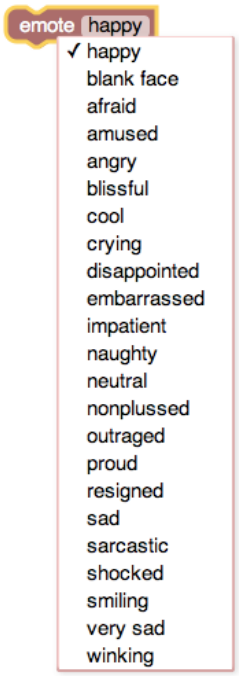
Lil'Block	Arduino	Action
	`Bot.say(char *phrase);`	Translate ASCII text into astromech droid

Lil'Blocks example	Arduino example	Remark
	`Bot.say("Hello, world!");`	See the library source code for the sound mapping algorithm

Robot emotions

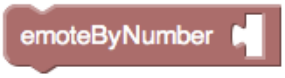
Emote

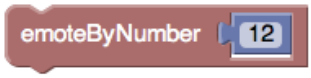
Lil'Block	Arduino	Action

	`Bot.emote(short int emotion);`	Express a robot emotion on the emoShield. These codes are available:
		`Bot.happy`
		`Bot.blankFace`
		`Bot.afraid`
		`Bot.amused`
		`Bot.angry`
		`Bot.blissful`
		`Bot.cool`
		`Bot.crying`
		`Bot.disappointed`
		`Bot.embarrassed`
		`Bot.impatient`
		`Bot.naughty`
		`Bot.neutral`
		`Bot.nonplussed`
		`Bot.outraged`
		`Bot.proud`
		`Bot.resigned`
		`Bot.sad`
		`Bot.sarcastic`
		`Bot.shocked`
		`Bot.smiling`
		`Bot.verySad`
		`Bot.winking`

Lil'Blocks example	Arduino example
	`Bot.emote(Bot.happy);`

Emote by number

Lil'Block	Arduino	Action
	`Bot.emoteByNumber(int index);`	Expresses a robot emotion on the emoShield based on an index value from 0 to 22

Lil'Blocks example	Arduino example
	`Bot.emoteByNumber(random(0, 22));`

Robot events

Obstacle detection causes the user program to be interrupted by an optional event handler ("callback") function. In Arduino, the user must declare and define the handler functions explicitly. This is done first by declaring the function, e.g. `frontObstacleHandler = frontObstacle;`, in `setup()`, and then defining code for the function declared, in this case `frontObstacle()`. Handler functions take no parameter and return no value.

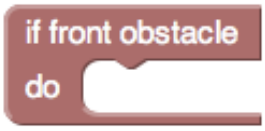
In Lil'Blocks, this is done automatically when the desired handler block is placed. The user only needs to place the blocks that will handle the exception, e.g. turning back when a front obstacle has been detected.

Event handlers are not recursive; during event handling, obstacle detection is disabled.

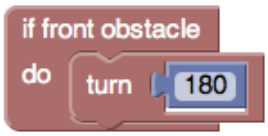
The Lil'Bot hardware provides for edge detection (e.g. coming to the edge of a tabletop), but the software is not implemented. Please refer to the library source code in `LilBot.cpp` for a possible implementation.

Front obstacle

Lil'Block	Arduino	Action

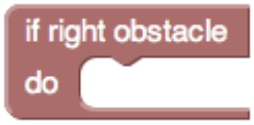
	`frontObstacleHandler = frontObstacle;`
	`void frontObstacle(void) {`
	`}`
		Defines the handler function in case a front obstacle has been detected

Lil'Blocks example	Arduino example	Remark

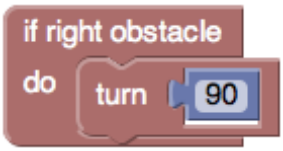
	`frontObstacleHandler = frontObstacle;`
	`void frontObstacle(void) {`
	`  Bot.turn(180.0);`
	`}`
		Define the first line in `setup()`

Right obstacle

Lil'Block	Arduino	Action

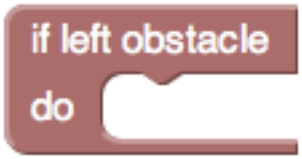
	`rightObstacleHandler = rightObstacle;`
	`void rightObstacle(void) {`
	`}`
		Defines the handler function in case a right obstacle has been detected

Lil'Blocks example	Arduino example	Remark

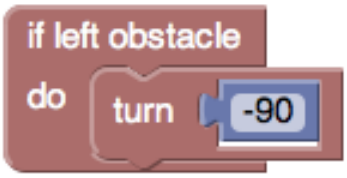
	`rightObstacleHandler = rightObstacle;`
	`void rightObstacle(void) {`
	`  Bot.turn(90.0);`
	`}`
		Define the first line in `setup()`

Left obstacle

Lil'Block	Arduino	Action

	`leftObstacleHandler = leftObstacle;`
	`void leftObstacle(void) {`
	`}`
		Defines the handler function in case a left obstacle has been detected

Lil'Blocks example	Arduino example	Remark

	`leftObstacleHandler = leftObstacle;`
	`void leftObstacle(void) {`
	`  Bot.turn(-90.0);`
	`}`
		Define the first line in **`setup()`**

System blocks

Wait

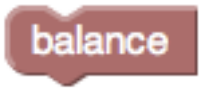
Lil'Bot cannot use the Arduino **`delay()`** function, because that would halt the balancing. **`Bot.wait()`** pauses the user program, but periodically invokes the balancing code so as to continue balancing while waiting.

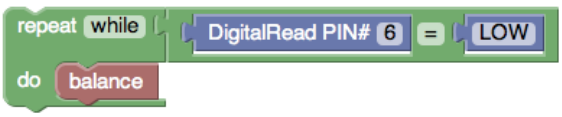
Lil'Block	Arduino	Action
	**`Bot.wait(int milliseconds);`**	Pause the user program for the specified duration in milliseconds, but continue invoking balancing code

Lil'Blocks example	Arduino example
	**`Bot.wait(1000);`**

Balance

This function is required only in unusual situations. If there is any chance the processor might be caught up in tight loops, it might lose track of the balancing. In such a case, this function should be called periodically (more often than every ten milliseconds) for the software to catch up with the balancing.

Lil'Block	Arduino	Action
	**`Bot.balance(void);`**	Calls the balancing code. Once every 10 milliseconds, this call computes robot balancing for about 2 milliseconds before returning.

Lil'Blocks example	Arduino example	Remarks
	**`while (digitalRead(6) == LOW) {`** **`  Bot.balance();`** **`}`**	Wait for Arduino digital input 6 to go high. This is a potentially infinite loop. The user must remember to call the balancing code.

Hardware reference

Arduino resource summary

Lil'Bot is hardware- and software-compatible with the Arduino Uno. The table below describes the utilization of Arduino Uno resources for Lil'Bot operation:

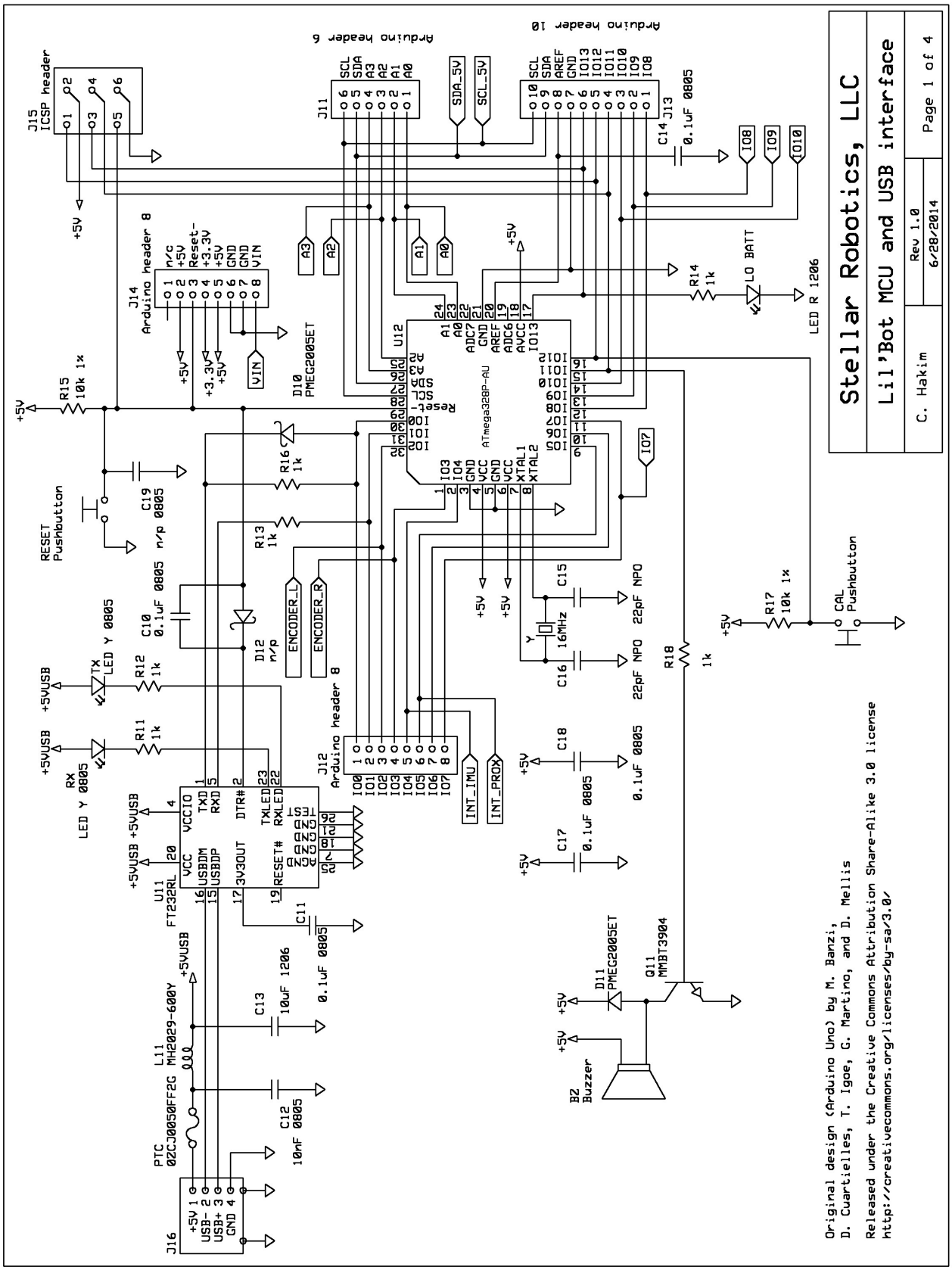
Arduino pin	Where used	Remarks
Digital 0	Serial receiver, connected to the USB interface	
Digital 1	Serial transmitter, connected to the USB interface	
Digital 2	Left wheel encoder interrupt input	
Digital 3	Right wheel encoder interrupt input	
Digital 4	Accelerometer and gyroscope interrupt line (polled)	
Digital 5	Proximity sensor interrupt line (not used)	To reuse this pin, disable the Si1143 INT output
Digital 6	Not used	May be reused
Digital 7	Motor-driver direction output	
Digital 8	Motor-driver direction output	
Digital 9	Motor-driver PWM output	
Digital 10	Motor-driver PWM output	
Digital 11	Buzzer output	
Digital 12	Calibration pushbutton input	May be reused, but do not push the CAL button
Digital 13	LED output	May be reused, but is rewritten periodically
Analog 0	Not used	May be reused as analog input or digital i/o
Analog 1	Battery voltage measurement input	
Analog 2	Not used	May be reused as analog input or digital i/o
Analog 3	Not used	May be reused as analog input or digital i/o
Analog 4	I^2C SDA bus line	May be used to attach other I^2C peripherals
Analog 5	I^2C SCL bus line	May be used to attach other I^2C peripherals

Schematics and other relevant documents are at Lil'Bot's website: **http://www.lil-bot.com/downloads/**

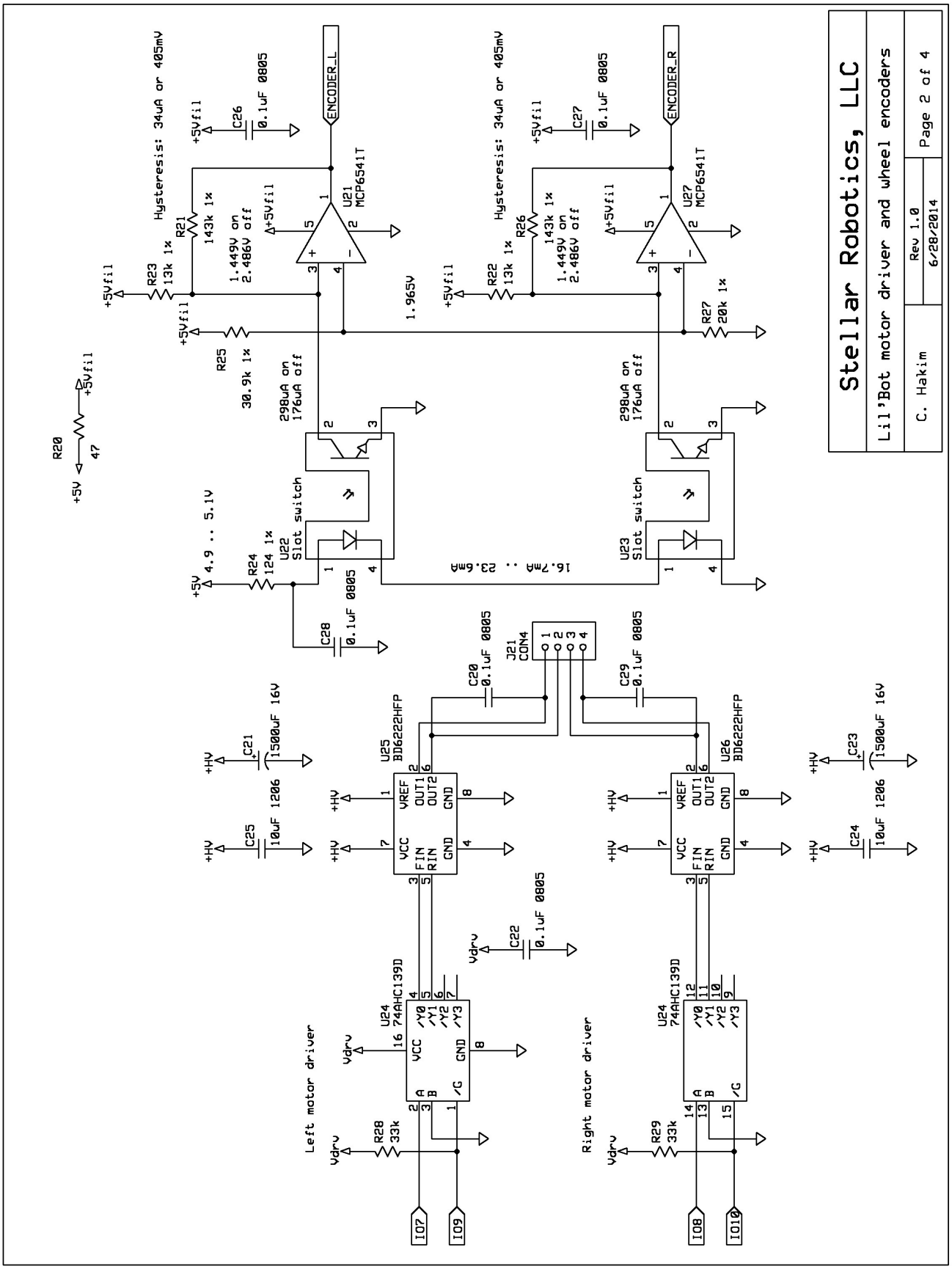
Much useful information is also available from Arduino: **http://arduino.cc/en/Main/ArduinoBoardUno/**

Lil'Bot schematics

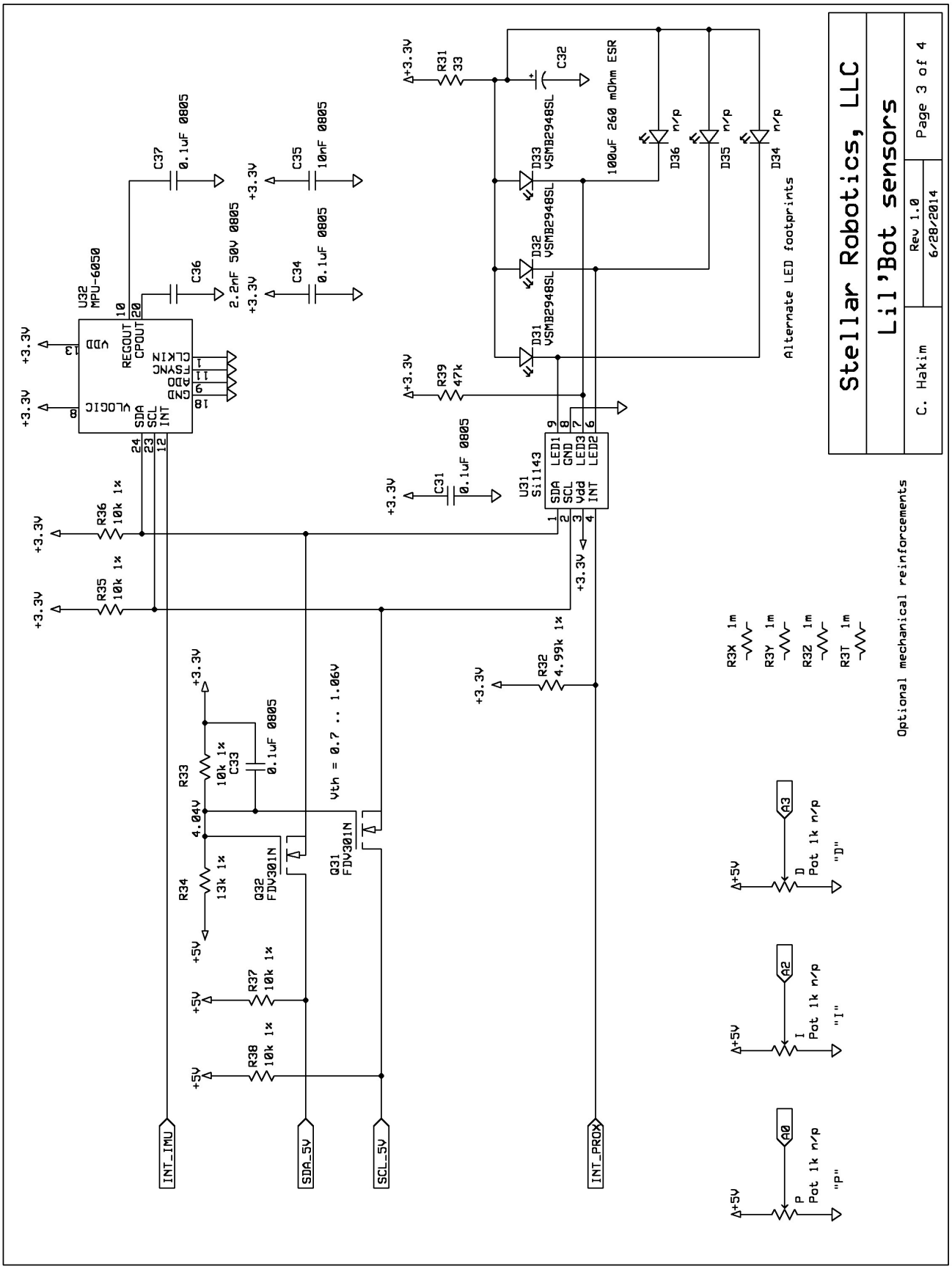
Arduino



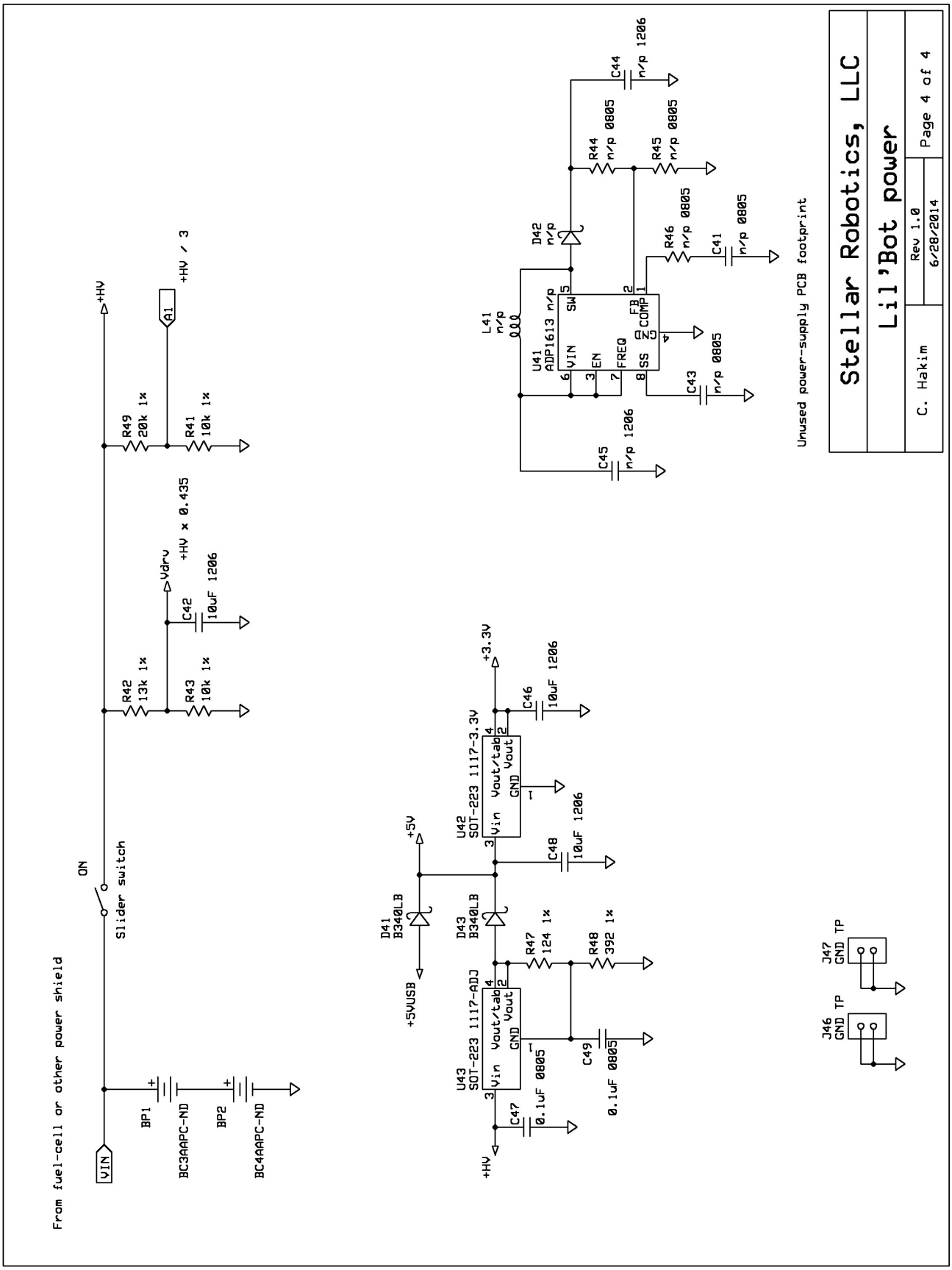
Motor drivers and wheel encoders



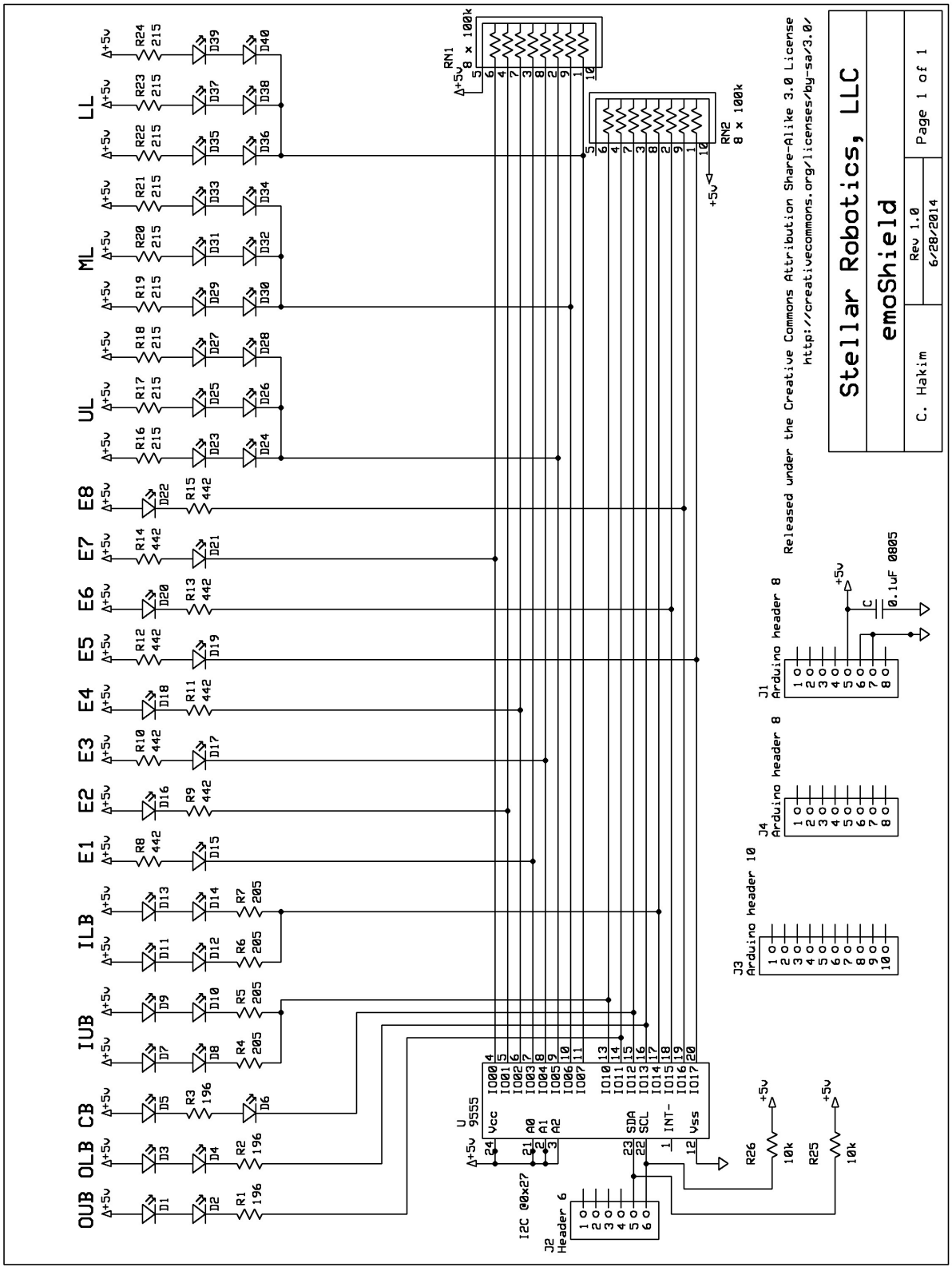
Sensors


Stellar Robotics, LLC
Lil'Bot sensors
C. Hakim
Rev 1.0
6/28/2014
Page 3 of 4
U32 MPU-6050
U31 Si1143
D31 VSMB2948SL
D32 VSMB2948SL
D33 VSMB2948SL
D34 n/p
D35 n/p
D36 n/p
100uF 260 mOhm ESR
Alternate LED footprints
Optional mechanical reinforcements
Vth = 0.7 .. 1.06V
4.04V
INT_IMU
SDA_5V
SCL_5V
INT_PROX
Pot 1k n/p
"P"
"I"
"D"

Power supply



emoShield schematic



For more information

Please visit our website: **http://www.lil-bot.com/**