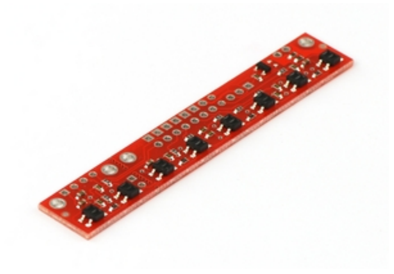
Arduino Library for the Pololu QTR Reflectance Sensors

```
 * Blink
 *
 * The basic Arduino example, modified to work for Orangutans.
 * Turns on an LED for one second, then off for one second,
 * and so on...  We use pin 1 because Orangutans have an LED
 * on PD1, which corresponds to Arduino pin 1.
 *
 * http://www.arduino.cc/en/Tutorial/Blink
 */

int ledPin = 1;             // LED co                    on Orangutans

void setup()                                          its
{
  pinMode(ledPin, OUTPUT)                          as output
}

void loop()                                 nd over again
{
  digi                                   s the LED on
                                       waits for a second
                                    // sets the LED off
                                    // waits for a second
}
```

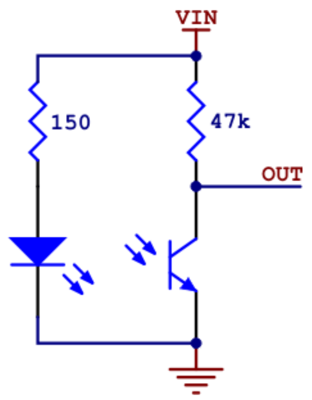
1. Introduction

QTR-1RC reflectance sensor.

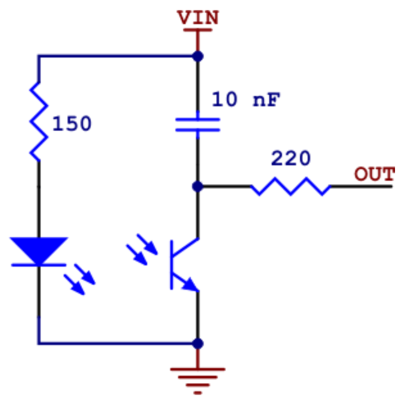

QTR-8A reflectance sensor array.

The Pololu QTR reflectance sensors carry infrared LED and phototransistor pairs that can provide analog measurements of IR reflectance, which makes them great for close-proximity edge detection and line-following applications. The modules come as compact, single-sensor units (**QTR-1A** **[http://www.pololu.com/catalog/product/958]** and **QTR-1RC** **[http://www.pololu.com/catalog/product/959]**) or as 8-sensor arrays (**QTR-8A** **[http://www.pololu.com/catalog/product/960]** and **QTR-8RC** **[http://www.pololu.com/catalog/product/961]**) that can be optionally split into a 2-sensor array and a 6-sensor array.

The modules are available in two different output formats: the QTR-xA outputs an analog voltage between 0 and Vcc that can be measured by an analog-to-digital converter (ADC), and the QTR-xRC outputs a pulse that can be measured by a digital I/O line (the duration of which provides an analog measurement of reflectance).



QTR-1A reflectance sensor schematic diagram.



QTR-1RC reflectance sensor schematic diagram.

Please see the **product pages** **[http://www.pololu.com/catalog/category/7]** for more information on how these sensors work.

This document will explain how to install Arduino libraries for the Pololu QTR reflectance sensors, and it will provide sample sketches as well as links to library documentation. The libraries will give you everything you need to interface with a QTR-8x reflectance sensor array or multiple QTR-1x reflectance sensors, including advanced features like automatic calibration and, in the case of line detection, calculation of the line's position.

2. Library Installation

Note: If you currently have an older version of our QTR reflectance sensor libraries, your first step should be to delete the **PololuQTRSensors** directory from your **arduino-0011/hardware/libraries** directory. If you don't perform this step, the newer version of the libraries might not get compiled.

Additionally, the **PololuQTRSensorsRC** object uses Timer2 for sensor pulse timing, so **PololuQTRSensorsRC might conflict with other non-Pololu libraries that use or rely upon Timer2**. During a sensor read, Timer2 is reconfigured for pulse timing, and then it is restored to its previous state once the read is complete, so you can use Timer2 any way you want while not performing an RC read. Note that the **PololuQTRSensorsAnalog** object does not use Timer2 at all, and this entire library is compatible with all other Pololu libraries, including the Orangutan Arduino libraries.

Download the following archive: **PololuArduinoLibraries_080826.zip [http://www.pololu.com/file/download/PololuArduinoLibraries-080826.zip?file_id=0J127] (80k zip)**

and extract it to your **arduino-0011/hardware/libraries** directory. Once this is complete you should see **PololuQTRSensors** as a subdirectory in the libraries directory.

You should now be able to use these libraries in your sketches by selecting **Sketch > Import Library > PololuQTRSensors** from your Arduino 0011 IDE (or simply type **#include <PololuQTRSensors.h>** at the top of your sketch). Note that you might need to restart your Arduino 0011 IDE before it sees the new libraries.

Once this is done, you can create a **PololuQTRSensorsAnalog** object for your QTR-xA sensors and a **PololuQTRSensorsRC** object for your QTR-xRC sensors:

```
// create an object for three QTR-xA sensors on analog inputs 0, 2, and 6
PololuQTRSensorsAnalog qtra((unsigned char[]) {0, 2, 6}, 3);

// create an object for four QTR-xRC sensors on digital pins 0 and 9, and on analog
// inputs 1 and 3 (which are being used as digital inputs 15 and 17 in this case)
PololuQTRSensorsRC qtrrc((unsigned char[]) {0, 9, 15, 17}, 4);
```

This library takes care of the differences between the QTR-xA and QTR-xRC sensors internally, providing you with a common interface to both sensors. The only external difference is in the constructors, as you can see in the code sample above. The first argument to the PololuQTRSensorsAnalog constructor is an array of analog input pins (0 – 7) while the first argument to the PololuQTRSensorsRC constructor is an array of digital pins (0 – 19). Note that analog inputs 0 – 5 can be used as digital pins 14 – 19.

The only other difference you might experience is in the time it takes to read the sensor values. The QTR-xRC sensors can all be read in parallel, but each requires the timing of a pulse that might take as long as 1 or 2 ms (you can specify how long the library should time this pulse before timing out and declaring the result full black). The QTR-xA sensors use the analog-to-digital converter (ADC) and hence must be read sequentially. Additionally, the analog results are produced by internally averaging a number of samples for each sensor (you can specify the number of samples to average) to decrease the effect of noise on the results.

3. PololuQTRSensors Methods & Usage Notes

PololuQTRSensor Methods

Complete documentation of this library's methods can be found in **Section 11** of the **Pololu AVR Library Command Reference** **[http://www.pololu.com/docs/0J18]**.

For QTR-xA sensors, you will want to instantiate a **PololuQTRSensorsAnalog** object, and for QTR-xRC sensors you will want to instantiate a **PololuQTRSensorsRC** object. Aside from the constructors, these two objects provide the same methods for reading sensor values (both classes are derived from the same abstract base class).

Usage Notes

Calibration

This library allows you to use the **calibrate()** method to easily calibrate your sensors for the particular conditions it will encounter. Calibrating your sensors can lead to substantially more reliable sensor readings, which in turn can help simplify your code since. As such, we recommend you build a calibration phase into your application's initialization routine. This can be as simple as a fixed duration over which you repeated call the **calibrate()** method. During this calibration phase, you will need to expose each of your reflectance sensors to the lightest and darkest readings they will encounter. For example, if you have made a line follower, you will want to slide it across the line during the calibration phase so the each sensor can get a reading of how dark the line is and how light the ground is. A sample calibration routine would be:

```
#include <PololuQTRSensors.h>

// create an object for your type of sensor (RC or Analog)
// in this example we have three sensors on analog inputs 0 - 2, a.k.a. digital pins 14 - 16
PololuQTRSensorsRC qtr((char[]) {14, 15, 16}, 3);
// PololuQTRSensorsA qtr((char[]) {0, 1, 2}, 3);

void setup()
{
  // optional: wait for some input from the user, such as  a button press

  // then start calibration phase and move the sensors over both
  // reflectance extremes they will encounter in your application:
  int i;
  for (i = 0; i < 250; i++)  // make the calibration take about 5 seconds
  {
    qtr.calibrate();
    delay(20);
  }

  // optional: signal that the calibration phase is now over and wait for further
  // input from the user, such as a button press
}
```

Reading the Sensors

This library gives you a number of different ways to read the sensors.

1. You can request raw sensor values using the **read()** method, which takes an optional argument that lets you perform the read with the IR emitters turned off (note that turning the emitters off is only supported by the QTR-8x reflectance sensor arrays).
2. You can request calibrated sensor values using the **readCalibrated()** method, which also takes an optional argument that lets you perform the read with the IR emitters turned off. Calibrated sensor values will always range from 0 to 1000, with 0 being as or more reflective (i.e. whiter) than the most reflective surface encountered during calibration, and 1000 being as or less reflective (i.e. blacker) than the least reflective surface encountered during calibration.
3. For line-detection applications, you can request the line location using the **readLine()** method, which takes as optional parameters a boolean that indicates whether the line is white on a black background or black on a white background, and a boolean that indicates whether the IR emitters should be on or off during the measurement. **readLine()** provides calibrated values for each sensor and returns an integer that tells you where it thinks the line is. If you are using *N* sensors, a returned value of 0 means it thinks the line is on or to the outside of sensor 0, and a returned value of 1000 * (*N*-1) means it thinks the line is on or to the outside of sensor *N*-1. As you slide your sensors across the line, the line position will change monotonically from 0 to 1000 * (*N*-1), or vice versa. This line-position value can be used for closed-loop PID control.

A sample routine to obtain the sensor values and perform rudimentary line following would be:

```
void loop()
{
  unsigned int sensors[3];
  // get calibrated sensor values returned in the sensors array, along with the line position
  // position will range from 0 to 2000, with 1000 corresponding to the line over the middle sensor
  int position = qtr.readLine(sensors);

  // if all three sensors see very low reflectance, take some appropriate action for this situation
  if (sensors[0] > 750 && sensors[1] > 750 && sensors[2] > 750)
  {
    // do something.  Maybe this means we're at the edge of a course or about to fall off a table,
    // in which case, we might want to stop moving, back up, and turn around.
    return;
  }

  // compute our "error" from the line position.  We will make it so that the error is zero when
  // the middle sensor is over the line, because this is our goal.  Error will range from
  // -1000 to +1000.  If we have sensor 0 on the left and sensor 2 on the right,  a reading of -1000
  // means that we see the line on the left and a reading of +1000 means we see the line on
  // the right.
  int error = position - 1000;

  int leftMotorSpeed = 100;
  int rightMotorSpeed = 100;
  if (error < -500)  // the line is on the left
    leftMotorSpeed = 0;  // turn left
  if (error > 500)  // the line is on the right
    rightMotorSpeed = 0;  // turn right

  // set motor speeds using the two motor speed variables above
}
```

PID Control

The integer value returned by **readLine()** can be easily converted into a measure of your position error for line-following applications, as was demonstrated in the previous code sample. The function used to generate this position/error value is designed to be monotonic, which means the value will almost always change in the same direction as you sweep your sensors across the line. This makes it a great quantity to use for PID control.

Explaining the nature of PID control is beyond the scope of this document, but wikipedia has a very good **article [http://en.wikipedia.org/wiki/PID_controller]** on the subject.

The following code gives a very simple example of PD control (I find the integral PID term is usually not necessary when it comes to line following). The specific nature of the constants will be determined by your particular application, but you should note that the derivative constant *Kd* is usually much bigger than the proportional constant *Kp*. This is because the derivative of the error is a much smaller quantity than the error itself, so in order to produce a meaningful correction it needs to be multiplied by a much larger constant.

```
int lastError = 0;

void loop()
{
  unsigned int sensors[3];
  // get calibrated sensor values returned in the sensors array, along with the line position
  // position will range from 0 to 2000, with 1000 corresponding to the line over the middle sensor
  int position = qtr.readLine(sensors);

  // compute our "error" from the line position.  We will make it so that the error is zero when
  // the middle sensor is over the line, because this is our goal.  Error will range from
  // -1000 to +1000.  If we have sensor 0 on the left and sensor 2 on the right,  a reading of -1000
  // means that we see the line on the left and a reading of +1000 means we see the line on
  // the right.
  int error = position - 1000;

  // set the motor speed based on proportional and derivative PID terms
  // KP is the a floating-point proportional constant (maybe start with a value around 0.1)
  // KD is the floating-point derivative constant (maybe start with a value around 5)
  // note that when doing PID, it's very important you get your signs right, or else the
  // control loop will be unstable
  int motorSpeed = KP * error + KD * (error - lastError);
  lastError = error;

  // M1 and M2 are base motor speeds.  That is to say, they are the speeds the motors should
  // spin at if you are perfectly on the line with no error.  If your motors are well matched,
  // M1 and M2 will be equal.  When you start testing your PID loop, it might help to start with
  // small values for M1 and M2.  You can then increase the speed as you fine-tune your
  // PID constants KP and KD.
  int m1Speed = M1 + motorSpeed;
  int m2Speed = M2 - motorSpeed;

  // it might help to keep the speeds positive (this is optional)
  // note that you might want to add a similiar line to keep the speeds from exceeding
  // any maximum allowed value
  if (m1Speed < 0)
    m1Speed = 0;
  if (m2Speed < 0)
    m2Speed = 0;

  // set motor speeds using the two motor speed variables above
}
```