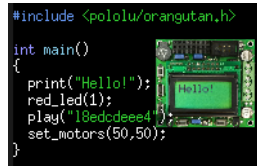


Pololu AVR C/C++ Library User's Guide



1. Introduction	2
2. Prerequisites	3
3. Downloading and extracting the files	4
4. Compiling the Pololu AVR Library (Source distribution only)	5
5. Installation of the Pololu AVR Library	6
6. Example programs	7
6.a. Example program – AVR Studio	7
6.b. Example program – Linux	9
6.c. Orangutan Analog Input Functions	10
6.d. Orangutan High-Level Buzzer Control Functions	12
6.e. Orangutan LCD Control Functions	17
6.f. Orangutan LED Control Functions	20
6.g. Orangutan Motor Control Functions	21
6.h. Orangutan Pushbutton Interface Functions	23
6.i. Pololu QTR Sensor Functions	24
6.j. Orangutan Serial Port Communication Functions	26
7. Using the Pololu AVR Library for your own projects	28
8. Additional resources	30

1. Introduction

This document is a guide to using the Pololu AVR C/C++ library, including installation instructions, tutorials, and example programs. The Pololu AVR Library makes it easy for you to get started with the following Pololu products:



Pololu 3pi robot: a mega168-based robot controller. The 3pi robot essentially contains an LV-168 and a 5-sensor version of the QTR-8RC, both of which are in the list below.



Pololu Orangutan SV-168: a full-featured, mega168-based robot controller that includes an LCD display. The SV-168 runs on an input voltage of 6-13.5V, giving you a wide range of robot power supply options, and can supply up to 3 A on its regulated 5 V bus.



Pololu Orangutan LV-168: a full-featured, mega168-based robot controller that includes an LCD display. The LV-168 runs on an input voltage of 2-5V, allowing two or three batteries to power a robot.



Pololu Baby Orangutan B-48: a compact, complete robot controller based on the mega48. The B-48 packs a voltage regulator, processor, and a two-channel motor-driver into a 24-pin DIP format.



Pololu Baby Orangutan B-168: a mega168 version of the above. The mega168 is a more powerful processor, with more memory for your programs.



Pololu QTR-1A and QTR-8A reflectance sensors (analog): an analog sensor containing IR/phototransistor pairs that allows a robot to detect the difference between shades of color. The QTR sensors can be used for following lines on the floor, for obstacle or drop-off (stairway) detection, and for various other applications.



Pololu QTR-1RC and QTR-8RC reflectance sensors (RC): a version of the above that is read using digital inputs; this is compatible with the Parallax QTI sensors.

Note that the library is designed for Atmel's mega168 and mega48-based boards like the Orangutans: to use it with the QTR sensors, your controller must be either an Orangutan or another board built with one of these processors.

This document covers the C/C++ version of the library, but it may also be used with **Arduino** [<http://www.arduino.cc>]: a popular, beginner-friendly programming environment for the mega168, using simplified C++ code. See our **guide to using Arduino with Orangutan controllers** [<http://www.pololu.com/docs/0J17>] for more information.

For detailed information about all of the functions available in the library, see the **command reference** [<http://www.pololu.com/docs/0J18>].

2. Prerequisites

The free `avr-gcc` compiler, `avr-libc`, and other associated tools must be installed before the Pololu AVR library. For Windows users, these tools are made available as the **WinAVR distribution** [<http://winavr.sourceforge.net/>]. Linux users should install all AVR-related packages available in their distribution's package manager. In particular, under Ubuntu you will need to install the following packages:

- `avr-libc`
- `gcc-avr`
- `avra`
- `binutils-avr`
- `avrdude` (for use with the Pololu Orangutan Programmer)

For Windows users, we also recommend the **AVR Studio development environment** [<http://www.atmel.com/avrstudio/>], which may be downloaded free of charge from Atmel.

To program the Pololu Orangutan or 3pi, we recommend the **Pololu Orangutan USB Programmer** [<http://www.pololu.com/catalog/product/740>], but any AVR ISP programmer will work. If you will be using a Pololu programmer, follow the **installation instructions** [<http://www.pololu.com/docs/0J6>] to install it on your computer, before continuing with these instructions.

3. Downloading and extracting the files

To begin the installation process for the Pololu AVR C/C++ Library, you will need to download one of the following zip files:

- **Precompiled binary distribution** [http://www.pololu.com/file/download/libpololu-avr-081209.zip?file_id=0J144] (254k zip)
- **Source distribution** [http://www.pololu.com/file/download/libpololu-avr-081209.src.zip?file_id=0J145] (295k zip)

We recommend the precompiled version, which will be easier to install for most people. If you have trouble installing the precompiled version, or you are interested in taking a closer look at the source code, please download the source distribution.

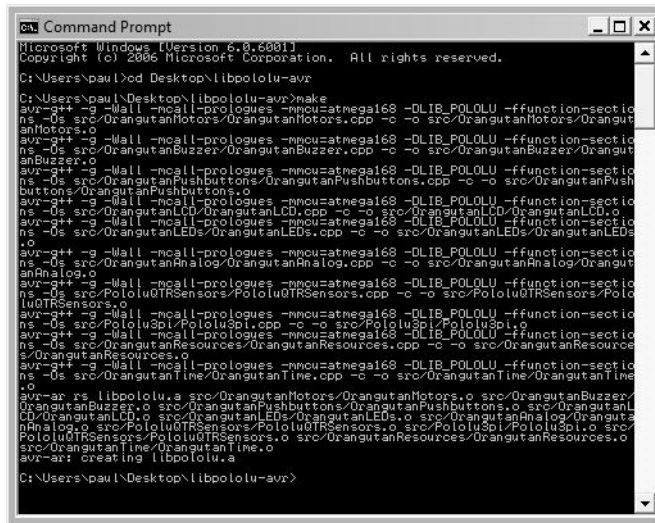
Next, if you are using Windows: open the .zip file and click “Extract all” to extract the Pololu AVR Library files.

If you are using Linux, run the command `unzip libpololu-avr-xxx.zip`, where `xxx` is replaced by the version of the library that you have downloaded.

A directory called “libpololu-avr” will be created. If you chose the source distribution, proceed to **Section 4** for instructions on compiling the library. Otherwise, you may skip to **Section 5** for installation instructions.

4. Compiling the Pololu AVR Library (Source distribution only)

Unpack the entire archive and open a command prompt within the libpololu-avr directory. If `avr-gcc` is correctly installed on your system, you will be able to type “make” to compile the entire library. Pay attention to any errors that occur during the build process. If you see errors, it is likely that `avr-gcc` was installed improperly or in a way that is incompatible with the Makefile.



```

Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\pau\Desktop\libpololu-avr>
C:\Users\pau\Desktop\libpololu-avr>make
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/OrangutanMotors/OrangutanMotors.cpp -o src/OrangutanMotors/OrangutanMotors.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/OrangutanBuzzer/OrangutanBuzzer.cpp -o src/OrangutanBuzzer/OrangutanBuzzer.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/OrangutanPushbuttons/OrangutanPushbuttons.cpp -o src/OrangutanPushbuttons/OrangutanPushbuttons.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/OrangutanLCD/OrangutanLCD.cpp -c -o src/OrangutanLCD/OrangutanLCD.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/OrangutanLEDs/OrangutanLEDs.cpp -o src/OrangutanLEDs/OrangutanLEDs.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/OrangutanAnalog/OrangutanAnalog.cpp -o src/OrangutanAnalog/OrangutanAnalog.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/PololuIOTransensors/PololuIOTransensors.cpp -o src/PololuIOTransensors/PololuIOTransensors.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/PololuSPI/PololuSPI.cpp -c -o src/PololuSPI/PololuSPI.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/OrangutanResources/OrangutanResources.cpp -c -o src/OrangutanResources/OrangutanResources.o
avr-g++ -g -Wall -ncall-prologues -mmcu=atmega168 -DLIB_POLOLU -ffunction-sections -Os src/OrangutanTime/OrangutanTime.cpp -c -o src/OrangutanTime/OrangutanTime.o
avr-ar rs libpololu.a src/OrangutanMotors/OrangutanMotors.o src/OrangutanBuzzer/OrangutanBuzzer.o src/OrangutanPushbuttons/OrangutanPushbuttons.o src/OrangutanLCD/OrangutanLCD.o src/OrangutanLEDs/OrangutanLEDs.o src/OrangutanAnalog/OrangutanAnalog.o src/PololuIOTransensors/PololuIOTransensors.o src/PololuSPI/PololuSPI.o src/OrangutanResources/OrangutanResources.o src/OrangutanTime/OrangutanTime.o
avr-ar: creating libpololu.a
C:\Users\pau\Desktop\libpololu-avr>

```

Compiling the Pololu AVR Library from the command prompt in Windows.

5. Installation of the Pololu AVR Library

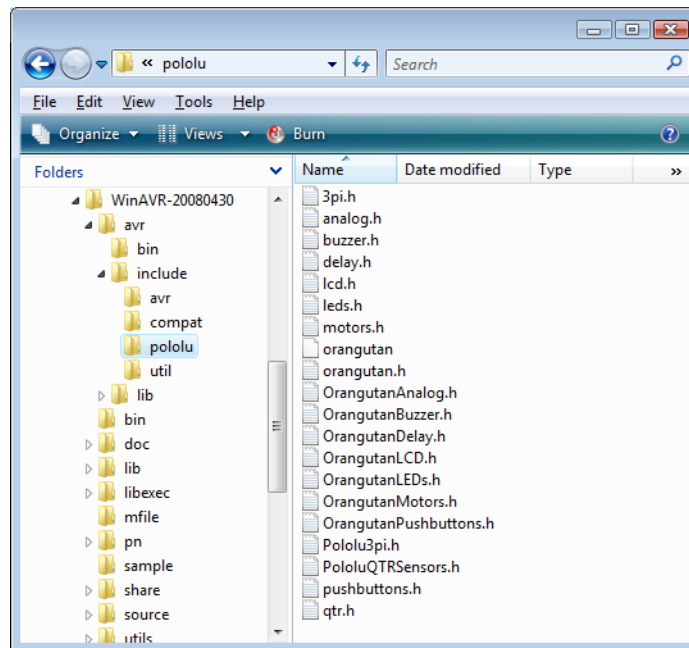
Determine the location of your `avr-gcc` files. In Windows, they will usually be in a folder such as: `C:\WinAVR-20080610\avr`. In Linux, the `avr-gcc` files are probably located in `/usr/avr`.

If you are using Linux and the `avr-gcc` files are in `/usr/avr`, you can install the library simply by typing `sudo make install` from within the `libpololu-avr` directory.

If you currently have an older version of the Pololu AVR Library, your first step should be to delete all of the old include files and the `libpololu.a` file that you installed previously.

Next, copy `libpololu.a` into the `lib` subdirectory of your `avr` directory (e.g. `C:\WinAVR-20080610\avr\lib`). Note that there is also a `lib` subdirectory directly below the main WinAVR directory; it will not work to put `libpololu.a` here.

Finally, copy the entire `pololu` subfolder into the `include` subfolder. The Pololu include files should now be located in `avr\include\pololu`.



The Pololu AVR Library header files, installed correctly.

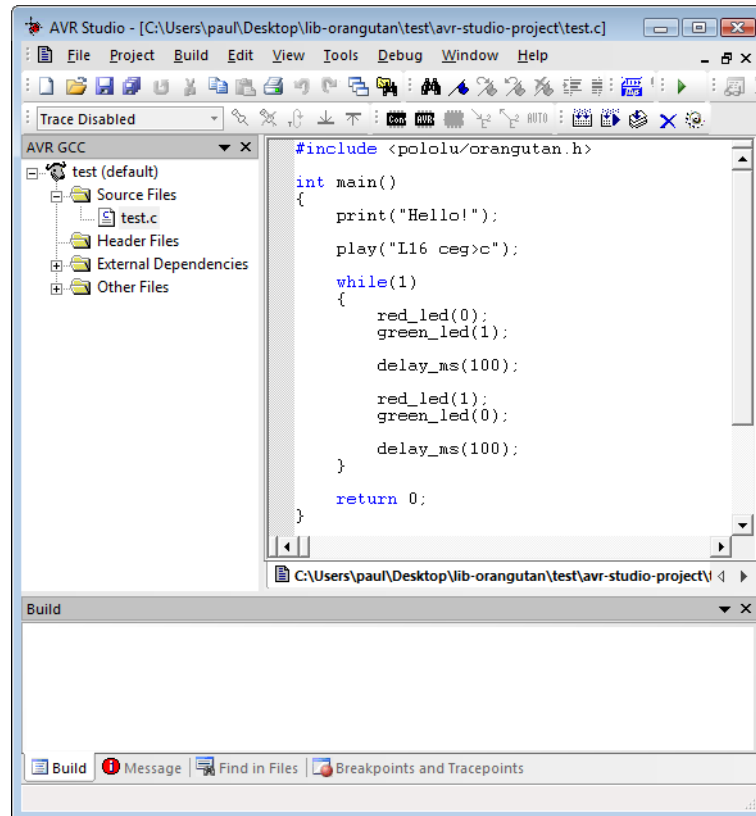
You are now ready to use the Pololu AVR library. The next section provides example programs that are already set up to use the library. For information on using the Pololu AVR library in your own programs (e.g. configuring AVR Studio projects to use the library), please see **Section 7**.

6. Example programs

6.a. Example program - AVR Studio

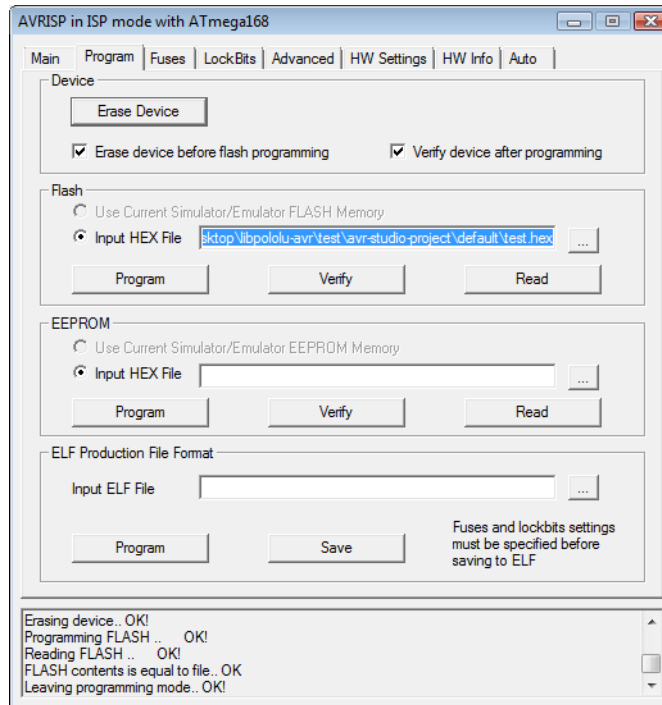
A very simple demo program for the Orangutan or 3pi is available in the folder `examples\simple-test`.

Double-click on the file `test.aps`, and the project should open automatically in AVR Studio, showing a C file that uses a few basic commands from the Pololu AVR Library:



AVR Studio showing the sample program.

To compile this program, select **Build > Build** or press **F7**. Look for warnings and errors (indicated by yellow and red dots) in the output displayed below. If the program compiles successfully, the message “Build succeeded with 0 Warnings...” will appear at the end of the output, and a file `test.hex` will have been created in the `test\avr-studio-project\default` folder.



Programming the Orangutan from AVR Studio.

If your controller was successfully programmed and you are using an Orangutan SV-168, Orangutan LV-168, or 3pi robot, you should hear a short tune, see the message “Hello!” on the LCD (if one is present and the contrast is set correctly), and the LEDs on the board should blink. If you are using a Baby Orangutan B, you will just see the red user LED blink.

In case you are having trouble performing the compilation, precompiled hex files for this example and all of the other examples included with the C/C++ library are available in `examples\hex-files`. You can load these hex files onto your controller using AVR Studio as described above.

6.b. Example program - Linux

A simple demo program is supplied in the directory `examples/simple-test/`.

Change to this directory and inspect the Makefile. Depending on your system, you may need to update the paths to the `avr-gcc` binaries and the device for your Orangutan Programmer. Then, you should be able to compile the example with `make`, which should generate the following output as it compiles the source code:

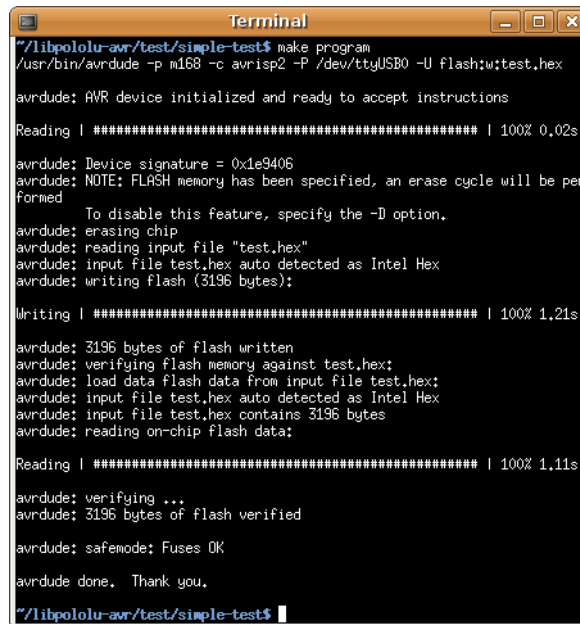
```

Terminal
~/libpololu-avr/test/simple-test$ make
/usr/bin/avr-gcc -g -Wall -mcall-prologues -mmcu=atmega168 -Os -c -o test.o test.c
/usr/bin/avr-gcc -g -Wall -mcall-prologues -mmcu=atmega168 -Os test.o -ll_gcc-sections -lpololu -o test.obj
/usr/bin/avr-objcopy -R .eeprom -O ihex test.obj test.hex
rm test.obj
~/libpololu-avr/test/simple-test$

```

Compiling the test program under Linux.

If make completed successfully, connect your Orangutan Programmer to your computer and your Orangutan board or 3pi robot, and turn on the target's power. The green status LED close to the USB connector should be on, while the other two LEDs should be off, indicating that the programmer is ready. Type `make program` to load the program onto the Orangutan or 3pi.



```
Terminal
~/libpololu-avr/test/simple-test$ make program
/usr/bin/avrdude -p m168 -c avrisp2 -P /dev/ttyUSB0 -U flash:w:test.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ##### | 100% 0.02s

avrdude: Device signature = 0x1e9405
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be per
formed
      To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "test.hex"
avrdude: input file test.hex auto detected as Intel Hex
avrdude: writing flash (3196 bytes):

Writing | ##### | 100% 1.21s

avrdude: 3196 bytes of flash written
avrdude: verifying flash memory against test.hex;
avrdude: load data flash data from input file test.hex;
avrdude: input file test.hex auto detected as Intel Hex
avrdude: input file test.hex contains 3196 bytes
avrdude: reading on-chip flash data:

Reading | ##### | 100% 1.11s

avrdude: verifying ...
avrdude: 3196 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done. Thank you.

~/libpololu-avr/test/simple-test$
```

Programming the Orangutan with avrdude under Linux.

If your controller was successfully programmed and you are using an Orangutan SV-168, Orangutan LV-168, or 3pi, you should hear a short tune, see the message “Hello!” on the LCD (if one is present and the contrast is set correctly), and the LEDs on the board should blink. If you are using a Baby Orangutan B, you will just see the red user LED blink.

6.c. Orangutan Analog Input Functions

Overview

This section of the library provides a set of methods that can be used to read analog voltage inputs, as well as functions specifically designed to read the value of the trimmer potentiometer (on the **3pi Robot** [<http://www.pololu.com/catalog/product/975>], **Orangutan SV-168** [<http://www.pololu.com/catalog/product/1225>], **Orangutan LV-168** [<http://www.pololu.com/catalog/product/775>], and **Baby Orangutan B** [<http://www.pololu.com/catalog/product/1216>]), the value of the temperature sensor in tenths of a degree F or C (on the Orangutan LV-168 only), and the battery voltage (3pi Robot only).

C++ users: See **Section 5.a of Programming Orangutans and the 3pi Robot from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] for examples of this class in the Arduino environment, which is almost identical to C++.

Complete documentation of the functions can be found in **Section 3 of the Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>].

Usage Examples

This library comes with two examples in `libpololu-avr/examples`. The Orangutan Motors examples also make limited use of this section.

1. analog1

Demonstrates how you can use the methods in this library to read the analog voltage of the trimmer potentiometer in the background while the rest of your code executes. If the ADC is free, the program starts a conversion on the **TRIMPOT** analog input (channel 7), and then it proceeds to execute the rest of the code in `loop()` while the ADC hardware works. Polling of the `analog_is_converting()` method allows the program to determine when the conversion is complete and to update its notion of the trimpot value accordingly. Feedback is given via the red user LED, whose brightness is made to scale with the trimpot position.

```
#include <pololu/orangutan.h>

/*
 * analog1:
 *
 * This example uses the OrangutanAnalog functions to read the voltage
 * output of the trimpot in the background while the rest of the main
 * loop executes. The LED is flashed so that its brightness appears
 * proportional to the trimpot position.
 *
 * http://www.pololu.com/docs/0J20/6.c
 * http://www.pololu.com
 * http://forum.pololu.com
 */

unsigned int sum;
unsigned int avg;
unsigned char samples;

int main()
{
  set_analog_mode(MODE_8_BIT); // 8-bit analog-to-digital conversions
  sum = 0;
  samples = 0;
  avg = 0;
  start_analog_conversion(TRIMPOT); // start initial conversion

  while(1)
  {
    if (!analog_is_converting()) // if conversion is done...
    {
      sum += analog_conversion_result(); // get result
      start_analog_conversion(TRIMPOT); // and start next conversion
      if (++samples == 20)
      {
        avg = sum / 20; // compute 20-sample average of ADC result
        samples = 0;
        sum = 0;
      }

      // when avg == 0, the red LED is almost totally off
      // when avg == 255, the red LED is almost totally on
      // brightness should scale approximately linearly in between
      red_led(0); // red LED off
      delay_us(256 - avg);
      red_led(1); // red LED on
      delay_us(avg+1);
    }

    return 0;
  }
}
```

2. analog2

Intended for use on the Orangutan LV-168. Note that it will run on the 3pi robot and Orangutan SV-168, but the displayed temperature will be incorrect as the analog input connected to the temperature sensor on the Orangutan LV-168 is connected to 2/3rds of the battery voltage on the 3pi and to 1/3rd of the battery voltage on the Orangutan SV-168. It displays on the LCD the trimmer potentiometer output in millivolts and the temperature sensor output in degrees Fahrenheit. If you hold a finger on the underside of the Orangutan LV-168's PCB near the center of the board, you should see the temperature reading slowly start to rise. Be careful not to zap the board with electrostatic discharge if you try this!

```

#include <pololu/orangutan.h>

/*
 * analog2:
 *
 * This sketch uses the OrangutanAnalog library to read the voltage output
 * of the trimpot (in millivolts) and to read the Orangutan LV-168's
 * temperature sensor in degrees Farenheit. These values are printed to
 * the LCD 10 times per second. This example is intended for use with the
 * Orangutan LV-168, though all but the temperature-measuring portion
 * will work on the 3pi robot (on the 3pi, analog input 6 connects to 2/3rds
 * of the battery voltage rather than a temperature sensor) and the
 * Orangutan SV-168 (on the SV-168, analog input 6 connects to 1/3rd of
 * the battery voltage).
 *
 * You should see the trimpot voltage change as you turn it, and you can
 * get the temperature reading to slowly increase by holding a finger on the
 * underside of the Orangutan LV-168's PCB near the center of the board.
 * Be careful not to zap the board with electrostatic discharge if you
 * try this!
 *
 * http://www.pololu.com/docs/0J20/6.c
 * http://www.pololu.com
 * http://forum.pololu.com
 */

int main()
{
  set_analog_mode(MODE_10_BIT); // 10-bit analog-to-digital conversions

  while(1) // run over and over again
  {
    lcd_goto_xy(0,0); // LCD cursor to home position (upper-left)
    print_long(to_millivolts(read_trimpot())); // trimpot output in mV
    print(" mV "); // added spaces are to overwrite left over chars

    lcd_goto_xy(0, 1); // LCD cursor to start of the second line

    unsigned int temp = read_temperature_f(); // get temp in tenths of a degree F
    print_long(temp/10); // get the whole number of degrees
    print_character('.'); // print the decimal point
    print_long(temp - (temp/10)*10); // print the tenths digit
    print_character(223); // print a degree symbol
    print("F "); // added spaces are to overwrite left over chars

    delay_ms(100); // wait for 100 ms (otherwise LCD flickers too much)
  }

  return 0;
}

```

6.d. Orangutan High-Level Buzzer Control Functions

Overview

These functions allow you to easily control the buzzer on the **3pi Robot** [<http://www.pololu.com/catalog/product/975>], **Orangutan SV-168** [<http://www.pololu.com/catalog/product/1225>], and **Orangutan LV-168** [<http://www.pololu.com/catalog/product/775>]. You have the option of playing either a note or a frequency for a specified duration at a specified volume, or you can use the **play()** method to play an entire melody in the background. Buzzer control is achieved using one of the Timer1 PWM outputs, and duration timing is performed using a Timer1 overflow interrupt, so **these functions will conflict with any other code that relies on or reconfigures Timer1**.

This library is incompatible with some older releases of WinAVR. If you experience any problems when using this library, make sure that your copy of the compiler is up-to-date. We know that it works with WinAVR 20080610.

The benefit to this approach is that you can play notes on the buzzer while leaving the CPU mostly free to execute the rest of your code. This means you can have a melody playing in the background while your Orangutan does its main task. You can poll the `isPlaying()` method to determine when the buzzer is finished playing.

C++ users: See **Section 5.b of Programming Orangutans and the 3pi Robot from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] for examples of this class in the Arduino environment, which is almost identical to C++.

Complete documentation of the functions can be found in **Section 4 of the Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>].

Usage Examples

This library comes with three examples in `libpololu-avr\examples`.

1. buzzer1

Demonstrates one way to use this library's `play_note()` method to play a simple melody stored in RAM. It should immediately start playing the melody, and you can use the top user pushbutton to stop and replay the melody. The example is structured so that you can add your own code to the main loop and the melody will still play normally in the background, assuming your code executes quickly enough to avoid inserting delays between the notes. You can use this same technique to play melodies that have been stored in EEPROM (the mega168 has enough room in EEPROM to store 170 notes).

```
#include <pololu/orangutan.h>
/*
 * buzzer1:
 *
 * This example uses the OrangutanBuzzer library to play a series of notes
 * on the Orangutan's/3pi's buzzer. It also uses the OrangutanLCD library
 * to display the notes its playing, and it uses the OrangutanPushbuttons
 * library to allow the user to stop/reset the melody with the top
 * pushbutton.
 *
 * http://www.pololu.com/docs/0J20/6.d
 * http://www.pololu.com
 * http://forum.pololu.com
 */

#define MELODY_LENGTH 95

// These arrays take up a total of 285 bytes of RAM (out of a 1k limit)
unsigned char note[MELODY_LENGTH] =
{
  E(5), SILENT_NOTE, E(5), SILENT_NOTE, E(5), SILENT_NOTE, C(5), E(5),
  G(5), SILENT_NOTE, G(4), SILENT_NOTE,

  C(5), G(4), SILENT_NOTE, E(4), A(4), B(4), B_FLAT(4), A(4), G(4),
  E(5), G(5), A(5), F(5), G(5), SILENT_NOTE, E(5), C(5), D(5), B(4),

  C(5), G(4), SILENT_NOTE, E(4), A(4), B(4), B_FLAT(4), A(4), G(4),
  E(5), G(5), A(5), F(5), G(5), SILENT_NOTE, E(5), C(5), D(5), B(4),

  SILENT_NOTE, G(5), F_SHARP(5), F(5), D_SHARP(5), E(5), SILENT_NOTE,
  G_SHARP(4), A(4), C(5), SILENT_NOTE, A(4), C(5), D(5),

  SILENT_NOTE, G(5), F_SHARP(5), F(5), D_SHARP(5), E(5), SILENT_NOTE,
  C(6), SILENT_NOTE, C(6), SILENT_NOTE, C(6),

  SILENT_NOTE, G(5), F_SHARP(5), F(5), D_SHARP(5), E(5), SILENT_NOTE,
  G_SHARP(4), A(4), C(5), SILENT_NOTE, A(4), C(5), D(5),

  SILENT_NOTE, E_FLAT(5), SILENT_NOTE, D(5), C(5)
};

unsigned int duration[MELODY_LENGTH] =
{
  100, 25, 125, 125, 125, 125, 125, 250, 250, 250, 250, 250,
```

```

375, 125, 250, 375, 250, 250, 125, 250, 167, 167, 167, 250, 125, 125,
125, 250, 125, 125, 375,

375, 125, 250, 375, 250, 250, 125, 250, 167, 167, 167, 250, 125, 125,
125, 250, 125, 125, 375,

250, 125, 125, 125, 250, 125, 125, 125, 125, 125, 125, 125, 125,

250, 125, 125, 125, 250, 125, 125, 200, 50, 100, 25, 500,

250, 125, 125, 125, 250, 125, 125, 125, 125, 125, 125, 125, 125,

250, 250, 125, 375, 500
};

unsigned char currentIdx;

int main()                // run once, when the sketch starts
{
  currentIdx = 0;
  print("Music!");

  while(1)                // run over and over again
  {
    // if we haven't finished playing the song and
    // the buzzer is ready for the next note, play the next note
    if (currentIdx < MELODY_LENGTH && !is_playing())
    {
      // play note at max volume
      play_note(note[currentIdx], duration[currentIdx], 15);

      // optional LCD feedback (for fun)
      lcd_goto_xy(0, 1);                // go to start of the second LCD line
      if(note[currentIdx] != 255) // display blank for rests
        print_long(note[currentIdx]); // print integer value of the current note
      print(" ");                        // overwrite any left over characters
      currentIdx++;
    }

    // Insert some other useful code here...
    // the melody will play normally while the rest of your code executes
    // as long as it executes quickly enough to keep from inserting delays
    // between the notes.

    // For example, let the top user pushbutton function as a stop/reset melody button
    if (button_is_pressed(TOP_BUTTON))
    {
      stop_playing(); // silence the buzzer
      if (currentIdx < MELODY_LENGTH)
        currentIdx = MELODY_LENGTH; // terminate the melody
      else
        currentIdx = 0; // restart the melody
      wait_for_button_release(TOP_BUTTON); // wait here for the button to be released
    }
  }

  return 0;
}

```

2. buzzer2

Demonstrates how you can use this library's **play()** function to start a melody playing. Once started, the melody will play all the way to the end with no further action required from your code, and the rest of your program will execute as normal while the melody plays in the background. The **play()** function is driven entirely by the Timer1 overflow interrupt. The top user pushbutton will play a fugue by Bach from program memory, the middle user pushbutton will quietly play the C major scale up and back down from RAM, and the bottom user pushbutton will stop any melody that is currently playing or play a single note if the buzzer is currently inactive.

```

#include <pololu/orangutan.h>

/*
 * buzzer2:

```

```

*
* This example uses the OrangutanBuzzer functions to play a series of notes
* on the Orangutan's/3pi's buzzer. It uses the OrangutanPushbuttons
* library to allow the user select which melody plays.
*
* This example demonstrates the use of the play() method,
* which plays the specified melody entirely in the background, requiring
* no further action from the user once the method is called. The CPU
* is then free to execute other code while the melody plays.
*
* http://www.pololu.com/docs/0J20/6.d
* http://www.pololu.com
* http://forum.pololu.com
*/

#include <avr/pgmspace.h> // this lets us refer to data in program space (i.e. flash)
// store this fugue in program space using the PROGMEM macro.
// Later we will play it directly from program space, bypassing the need to load it
// all into RAM first.
const char fugue[] PROGMEM =
"! O5 L16 agafaea dac+adaea fa<aa<bac#a dac#adaea f"
"O6 dcd<b-d<ad<g d<f+d<gd<ad<b- d<dd<ed<f+d<g d<f+d<gd<ad"
"L8 MS <b-d<b-d MLe-<ge-<g MS<ac<a ML d<fd<f O5 MS b-gb-g"
"ML >c#e>c#e MS afaf ML gc#gc# MS fdfd ML e<b-e<b-"
"O6 L16ragafaea dac#adaea fa<aa<bac#a dac#adaea faeadaca"
"<b-acadg<b-g egdgcg<b-g <ag<b-gcf<af d<fc<b-f<af"
"<gf<af<b-e<ge c#e<b-e<ae<ge <fe<ge<ad<fd"
"O5 e>ee>ef>df>d b->c#b->c#a>df>d e>ee>ef>df>d"
"e>d>c#>db>d>c#b >c#agaegfe f O6 dc#dfdc#<b c#4";

void loop() // run over and over again
{
  // wait here for one of the three buttons to be pushed
  unsigned char button = wait_for_button(ALL_BUTTONS);
  clear();

  if (button == TOP_BUTTON)
  {
    play_from_program_space(fugue);

    print("Fugue!");
    lcd_goto_xy(0, 1);
    print("flash ->");
  }
  if (button == MIDDLE_BUTTON)
  {
    play("! V8 cdefgab>cbagfedc");
    print("C Major");
    lcd_goto_xy(0, 1);
    print("RAM ->");
  }
  if (button == BOTTOM_BUTTON)
  {
    if (is_playing())
    {
      stop_playing();
      print("stopped");
    }
    else
    {
      play_note(A(5), 200, 15);
      print("note A5");
    }
  }
}

int main() // run once, when the program starts
{
  print("Press a");
  lcd_goto_xy(0, 1);
  print("button..");

  while(1)
    loop();

  return 0;
}

```

3. buzzer3

Demonstrates the use of this library's `playMode()` and `playCheck()` methods. In this example, automatic play mode is used to allow the melody to keep playing while it blinks the red user LED. Then the mode is switched to play-check mode during a phase where we are trying to accurately measure time. There are three `#define` macros that allow you to run this example in different ways and observe the result. Please see the comments at the top of the sketch for more detailed information.

```
#include <pololu/orangutan.h>

/*
 * buzzer3:
 *
 * This example uses the OrangutanBuzzer functions to play a series of notes
 * on the Orangutan's/3pi's buzzer. It uses the OrangutanPushbuttons
 * functions to allow the user select which melody plays.
 *
 * This example demonstrates the use of the play_mode()
 * and play_check() methods, which allow you to select
 * whether the melody sequence initiated by play() is
 * played automatically in the background by the Timer1 interrupt, or if
 * the play is driven by the play_check() method in your main loop.
 *
 * Automatic play mode should be used if your code has a lot of delays
 * and is not time critical. In this example, automatic mode is used
 * to allow the melody to keep playing while we blink the red user LED.
 *
 * Play-check mode should be used during parts of your code that are
 * time critical. In automatic mode, the Timer1 interrupt is very slow
 * when it loads the next note, and this can delay the execution of your.
 * Using play-check mode allows you to control when the next note is
 * loaded so that it doesn't occur in the middle of some time-sensitive
 * measurement. In our example we use play-check mode to keep the melody
 * going while performing timing measurements using Timer2. After the
 * measurements, the maximum time measured is displayed on the LCD.
 *
 * Immediately below are three #define statements that allow you to alter
 * the way this program runs. You should have one of the three lines
 * uncommented while commenting out the other two:
 *
 * If only WORKING_CORRECTLY is uncommented, the program should run in its
 * ideal state, using automatic play mode during the LED-blinking phase
 * and using play-check mode during the timing phase. The maximum recorded
 * time should be 20, as expected.
 *
 * If only ALWAYS_AUTOMATIC is uncommented, the program will use automatic
 * play mode during both the LED-blinking phase and the timing phase. Here
 * you will see the effect this has on the time measurements (instead of 20,
 * you should see a maximum reading of around 27 or 28).
 *
 * If only ALWAYS_CHECK is uncommented, the program will be in play-check
 * mode during both the LED-blinking phase and the timing phase. Here you
 * will see the effect that the LED-blinking delays have on play-check
 * mode (the sequence will be very choppy while the LED is blinking, but
 * sound normal during the timing phase). The maximum timing reading should
 * be 20, as expected.
 *
 * http://www.pololu.com/docs/0J20/6.d
 * http://www.pololu.com
 * http://forum.pololu.com
 */

// *** UNCOMMENT ONE OF THE FOLLOWING PRECOMPILER DIRECTIVES ***
// (the remaining two should be commented out)
#define WORKING_CORRECTLY // this is the right way to use playMode()
// #define ALWAYS_AUTOMATIC // playMode() is always PLAY_AUTOMATIC (timing is inaccurate)
// #define ALWAYS_CHECK // playMode() is always PLAY_CHECK (delays interrupt the sequence)

#include <avr/pgmspace.h>
const char rhapsody[] PROGMEM = "O6 T40 L16 d#<b<f#<d#<f#<bd#f#"
  "T80 c#<b-<f#<c#<f#<b-c#8"
  "T180 d#b<f#d#f#>bd#f#c#b-<f#c#f#>b-c#8 c>c#<c#>c#<b>c#<c#>c#<c#>c#<b>c#<c#>c#"
  "c>c#<c#>c#<b->c#<c#>c#<c#>c#<c#>c#<b->c#<c#>c#"
  "c>c#<c#>c#f>c#<c#>c#c>c#<c#>c#f>c#<c#>c#"
  "c>c#<c#>c#f#>c#<c#>c#c>c#<c#>c#f#>c#<c#>c#<c#>c#d#bb-bd#bf#d#c#b-ab-c#b-f#d#";
```



```

int main()
{
    TCCR2A = 0;          // configure timer2 to run at 78 kHz
    TCCR2B = 0x06;      // and overflow when TCNT2 = 256 (~3 ms)
    play_from_program_space(rhapsody);

    while(1)
    {
        // allow the sequence to keep playing automatically through the following delays
#ifdef ALWAYS_CHECK
        play_mode(PLAY_AUTOMATIC);
#else
        play_mode(PLAY_CHECK);
#endif
        lcd_goto_xy(0, 0);
        print("blink!");
        int i;
        for (i = 0; i < 8; i++)
        {
#ifdef ALWAYS_CHECK
            play_check();
#endif
            red_led(1);
            delay_ms(500);
            red_led(0);
            delay_ms(500);
        }

        lcd_goto_xy(0, 0);
        print("timing");
        lcd_goto_xy(0, 1);
        print(" "); // clear bottom LCD line
        // turn off automatic playing so that our time-critical code won't be interrupted by
        // the buzzer's long timer1 interrupt. Otherwise, this interrupt could throw off our
        // timing measurements. Instead, we will now use playCheck() to keep the sequence
        // playing in a way that won't throw off our measurements.
#ifdef ALWAYS_AUTOMATIC
        play_mode(PLAY_CHECK);
#endif
        unsigned char maxTime = 0;
        for (i = 0; i < 8000; i++)
        {
            TCNT2 = 0;
            while (TCNT2 < 20) // time for ~250 us
            ;
            if (TCNT2 > maxTime)
                maxTime = TCNT2; // if the elapsed time is greater than the previous max, save it
#ifdef ALWAYS_AUTOMATIC
            play_check(); // check if it's time to play the next note and play it if so
#endif
        }
        lcd_goto_xy(0, 1);
        print("max=");
        print_long((unsigned int)maxTime);
        print(" "); // overwrite any left over characters
    }

    return 0;
}

```

6.e. Orangutan LCD Control Functions

Overview

This section of the library gives you the ability to control the 8×2 character LCD on the **3pi Robot** [<http://www.pololu.com/catalog/product/975>], **Orangutan SV-168** [<http://www.pololu.com/catalog/product/1225>], and **Orangutan LV-168** [<http://www.pololu.com/catalog/product/775>]. It implements the standard 4-bit HD44780 protocol, and it uses the busy-wait-flag feature to avoid the unnecessarily long delays present in other 4-bit LCD control libraries. This comprehensive library is meant to offer as much LCD control as possible, so it most likely gives you more methods than you need. Make sure to use the linker option `-Wl,-gc-sections` when compiling your code, so that these extra functions will not be included in your hex file. See **Section 7** for more information.

This library is designed to gracefully handle alternate use of the four LCD data lines. It will change their data direction registers and output states only when needed for an LCD command, after which it will immediately restore the registers to their previous states. This allows the LCD data lines to additionally function as pushbutton inputs and an LED driver.

C++ users: See **Section 5.c of Programming Orangutans and the 3pi Robot from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] for examples of this class in the Arduino environment, which is almost identical to C++.

Complete documentation of this library's methods can be found in **Section 5 of the Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>].

Usage Examples

This library comes with two examples in `libpololu-avr\examples`.

1. lcd1

Demonstrates shifting the contents of the display by moving the word “Hello” around the two lines of the LCD.

```
#include <pololu/orangutan.h>

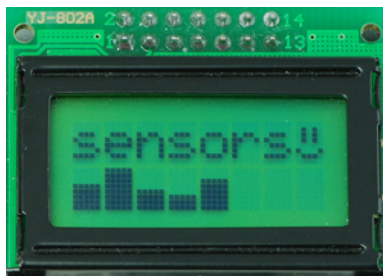
/*
 * lcd1:
 *
 * This example uses the OrangutanLCD library to display things on the LCD.
 *
 * http://www.pololu.com/docs/0J20/6.e
 * http://www.pololu.com
 * http://forum.pololu.com
 */

int main()                // run over and over again
{
  while(1)
  {
    print("Hello");       // display "Hello" at (0, 0), a.k.a. upper-left
    delay_ms(200);
    lcd_scroll(LCD_RIGHT, 3, 200); // shift the display right every 200ms three times
    clear();              // clear the LCD
    lcd_goto_xy(3, 1);    // go to the fourth character of the second LCD line
    print("Hello");       // display "Hello" at (3, 1), a.k.a. lower-right
    delay_ms(200);
    lcd_scroll(LCD_LEFT, 3, 200); // shift the display left every 200ms three times
    clear();              // clear the LCD
  }

  return 0;
}
```

1. lcd2

Demonstrates creating and displaying custom characters on the LCD. The following picture shows an example of custom characters, using them to display a bar graph of sensor readings and a smiley face:



```

#include <pololu/orangutan.h>

// get random functions
#include <stdlib.h>

/*
 * lcd2:
 *
 * This example uses the OrangutanLCD functions to display custom
 * characters on the LCD. Simply push a any user pushbutton to
 * display a new, randomly chosen, custom mood character.
 *
 * http://www.pololu.com/docs/0J20/6.e
 * http://www.pololu.com
 * http://forum.pololu.com
 */

// define some custom "mood" characters
#include <avr/pgmspace.h> // this lets us refer to data in program space (i.e. flash)
const char happy[] PROGMEM = {
    0b00000, // the five bits that make up the top row of the 5x8 character
    0b01010,
    0b01010,
    0b01010,
    0b00000,
    0b10001,
    0b01110,
    0b00000
};

const char sad[] PROGMEM = {
    0b00000,
    0b01010,
    0b01010,
    0b01010,
    0b00000,
    0b01110,
    0b10001,
    0b00000
};

const char indifferent[] PROGMEM = {
    0b00000,
    0b01010,
    0b01010,
    0b01010,
    0b00000,
    0b00000,
    0b01110,
    0b00000
};

const char surprised[] PROGMEM = {
    0b00000,
    0b01010,
    0b01010,
    0b00000,
    0b01110,
    0b10001,
    0b10001,
    0b01110
};

const char mocking[] PROGMEM = {
    0b00000,
    0b01010,
    0b01010,
    0b01010,
    0b00000,
    0b11111,
    0b00101,
    0b00010
};

char prevMood = 5;

int main() // run once, when the sketch starts

```

```

{
  lcd_load_custom_character(happy, 0);
  lcd_load_custom_character(sad, 1);
  lcd_load_custom_character(indifferent, 2);
  lcd_load_custom_character(surprised, 3);
  lcd_load_custom_character(mocking, 4);
  clear(); // this must be called before we can use the custom characters
  print("mood: ?");

  // initialize the random number generator based on how long they hold the button the first time
  wait_for_button_press(ALL_BUTTONS);
  long seed = 0;
  while(button_is_pressed(ALL_BUTTONS))
    seed++;
  srand(seed);

  while(1)
  {
    lcd_goto_xy(6, 0); // move cursor to the correct position

    char mood;
    do
    {
      mood = random()%5;
    } while (mood == prevMood); // ensure we get a new mood that differs from the previous
    prevMood = mood;

    print_character(mood); // print a random mood character
    wait_for_button(ALL_BUTTONS); // wait for any button to be pressed
  }

  return 0;
}

```

6.f. Orangutan LED Control Functions

Overview

These functions allow you to easily control the LED(s) on the **3pi Robot** [<http://www.pololu.com/catalog/product/975>], **Orangutan SV-168** [<http://www.pololu.com/catalog/product/1225>], **Orangutan LV-168** [<http://www.pololu.com/catalog/product/775>], and **Baby Orangutan B** [<http://www.pololu.com/catalog/product/1216>]. The 3pi Robot has two user LEDs on the bottom of the board: a red LED on the left and a green LED on the right. Additional LEDs included with the 3pi may be soldered in on the top side (in parallel with the surface-mount LEDs on the underside) for easier viewing. The Orangutan SV-168 and Orangutan LV-168 have two user LEDs: a red LED on the bottom left and a green LED on the top right. The Baby Orangutan B has a single, red user LED.

C++ users: See **Section 5.d of Programming Orangutans and the 3pi Robot from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] for examples of this class in the Arduino environment, which is almost identical to C++.

Complete documentation of these functions can be found in **Section 9 of the Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>].

This library comes with an example program in `libpololu-avr\examples\led1`.

1. led1

A simple example that blinks LEDs.

```

#include <pololu/orangutan.h>

/*
 * led1:
 *
 * This program uses the OrangutanLEDs functions to control the red and green
 * LEDs on the 3pi, Orangutan SV-168, and Orangutan LV-168.
 * It will also work to control the red LED on the Baby Orangutan B

```

```

* (which lacks a second, green LED).
*
* http://www.pololu.com/docs/0J20/6.f
* http://www.pololu.com
* http://forum.pololu.com
*/

int main()
{
  while(1)
  {
    red_led(1);           // red LED on
    delay_ms(1000);      // waits for a second
    red_led(0);          // red LED off
    delay_ms(1000);      // waits for a second
    green_led(1);        // green LED on (will not work on the Baby Orangutan)
    delay_ms(500);       // waits for 0.5 seconds
    green_led(0);        // green LED off (will not work on the Baby Orangutan)
    delay_ms(500);       // waits for 0.5 seconds
  }

  return 0;
}

```

6.g. Orangutan Motor Control Functions

Overview

This set of functions gives you the ability to control the motor drivers on the **3pi Robot** [<http://www.pololu.com/catalog/product/975>], **Orangutan SV-168** [<http://www.pololu.com/catalog/product/1225>], **Orangutan LV-168** [<http://www.pololu.com/catalog/product/775>], and **Baby Orangutan B** [<http://www.pololu.com/catalog/product/1216>]. It accomplishes this by using the four hardware PWM outputs from timers Timer0 and Timer2, so **this library will conflict with any other libraries that rely on or reconfigure Timer0 or Timer2.**

C++ users: See **Section 5.e of Programming Orangutans and the 3pi Robot from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] for examples of this class in the Arduino environment, which is almost identical to C++.

Complete documentation of these functions can be found in **Section 7 of the Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>].

Usage Examples

This library comes with two examples in `libpololu-avr\examples`.

1. motors1

Demonstrates controlling the motors using the trimmer potentiometer and uses the red LED for feedback.

```

#include <pololu/orangutan.h>

/*
* motors1:
*
* This example uses the OrangutanMotors functions to drive
* motors in response to the position of user trimmer potentiometer
* and blinks the red user LED at a rate determined by the trimmer
* potentiometer position. It uses the OrangutanAnalog library to measure
* the trimpot position, and it uses the OrangutanLEDs library to provide
* limited feedback with the red and green user LEDs.
*
* http://www.pololu.com/docs/0J20/6.g
* http://www.pololu.com
* http://forum.pololu.com
*/

unsigned long prevMillis = 0;

int main()

```

```

{
  while(1)
  {
    // note that the following line could also be accomplished with:
    // int pot = analogRead(7);
    int pot = read_trimpot(); // determine the trimpot position
    int motorSpeed = pot/2-256; // turn pot reading into number between -256 and 255
    if(motorSpeed == -256)
      motorSpeed = -255; // 256 is out of range
    set_motors(motorSpeed, motorSpeed);

    int ledDelay = motorSpeed;
    if(ledDelay < 0)
      ledDelay = -ledDelay; // make the delay a non-negative number
    ledDelay = 256-ledDelay; // the delay should be short when the speed is high

    red_led(1); // turn red LED on
    delay_ms(ledDelay);

    red_led(0); // turn red LED off
    delay_ms(ledDelay);
  }

  return 0;
}

```

2. motors2

Demonstrates controlling the motors using the trimmer potentiometer.

```

#include <pololu/orangutan.h>

/*
 * motors2:
 *
 * This example uses the OrangutanMotors and OrangutanLCD functions to
 * drive motors in response to the position of user trimmer potentiometer
 * and to display the potentiometer position and desired motor speed
 * on the LCD. It uses the OrangutanAnalog functions to measure the
 * trimpot position, and it uses the OrangutanLEDs functions to provide
 * limited feedback with the red and green user LEDs.
 *
 * http://www.pololu.com/docs/0J20/6.g
 * http://www.pololu.com
 * http://forum.pololu.com
 */

int main() // run over and over again
{
  while(1)
  {
    // note that the following line could also be accomplished with:
    // int pot = analogRead(7);
    int pot = read_trimpot(); // determine the trimpot position

    // avoid clearing the LCD to reduce flicker
    lcd_goto_xy(0, 0);
    print("pôt=");
    print_long(pot); // print the trim pot position (0 - 1023)
    print(" "); // overwrite any left over digits

    int motorSpeed = (512 - pot) / 2;
    lcd_goto_xy(0, 1);
    print("spd=");
    print_long(motorSpeed); // print the resulting motor speed (-255 - 255)
    print(" ");
    set_motors(motorSpeed, motorSpeed); // set speeds of motors 1 and 2

    // all LEDs off
    red_led(0);
    green_led(0);
    // turn green LED on when motors are spinning forward
    if (motorSpeed > 0)
      green_led(1);
    // turn red LED on when motors are spinning in reverse
    if (motorSpeed < 0)

```

```

    red_led(1);
    delay_ms(100);
}
return 0;
}

```

6.h. Orangutan Pushbutton Interface Functions

Overview

This library allows you to easily interface with the three user pushbuttons on the **3pi Robot** [<http://www.pololu.com/catalog/product/975>], **Orangutan SV-168** [<http://www.pololu.com/catalog/product/1225>], and **Orangutan LV-168** [<http://www.pololu.com/catalog/product/775>] by either polling for the state of specific buttons or by waiting for press/release events on specifiable buttons. The `wait_for_button_____()` methods in this library automatically take care of button debouncing.

C++ users: See **Section 5.f of Programming Orangutans from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] for examples of this class in the Arduino environment, which is almost identical to C++.

Complete documentation of these functions can be found in **Section 8 of the Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>].

Usage Examples

This library comes with two examples in `libpololu-avr/examples`.

1. pushbuttons1

Demonstrates interfacing with the user pushbuttons. It will wait for you to push either the top button or the bottom button, at which point it will display on the LCD which button was pressed. It will also detect when that button is subsequently released and display that to the LCD.

```

#include <pololu/orangutan.h>

/*
 * pushbuttons1:
 *
 * This example uses the OrangutanPushbuttons library to detect user
 * input from the pushbuttons, and it uses the OrangutanLCD library to
 * display feedback on the LCD.
 *
 * http://www.pololu.com/docs/0J20/6.h
 * http://www.pololu.com
 * http://forum.pololu.com
 */

int main()
{
    while(1)
    {
        clear();
        print("Waiting");

        // wait for either the top or bottom buttons to be pressed
        // store the value of the pressed button in the variable 'button'
        unsigned char button = wait_for_button_press(TOP_BUTTON | BOTTOM_BUTTON);
        clear();
        if (button == TOP_BUTTON) // display the button that was pressed
            print("top down");
        else
            print("bot down");
        wait_for_button_release(button); // wait for that button to be released
        clear();
        print("released"); // display that the button was released
        delay_ms(1000);
    }
}

```

```
    return 0;
}
```

6.i. Pololu QTR Sensor Functions

Overview

This set of functions provides access to the QTR family of reflectance sensors, which come as 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>]). To initialize the set of sensors that you are using, choose either the **qtr_rc_init()** or **qtr_analog_init()** function, and specify the set of pins connected to the sensors that you will be using. The initialization may only be called once within the C environment, while C++ allows the sensors to be used in more complicated ways.

These functions are used by the 3pi support described in the **3pi Robot User's Guide** [<http://www.pololu.com/docs/0J21>]. We do not recommend using these functions directly on the 3pi unless you are adding additional sensors.

C++ users: See **Section 3 of Programming Orangutans and the 3pi Robot from the Arduino Environment** [<http://www.pololu.com/docs/0J19>] for examples of this class in the Arduino environment, which is almost identical to C++.

Complete documentation of these functions can be found in **Section 11 of the Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>].

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 <pololu/orangutan.h>

int main()
{
    // initialize your QTR sensors
    int qtr_rc_pins[] = {14, 15, 16};
    qtr_rc_init(qtr_rc_pins, 3, 2000, 255); // 800 us timeout, no emitter pin
    // int_qtr_analog_pins[] = {0, 1, 2};
    // qtr_analog_init(qtr_analog_pins, 3, 10, 14); // 10 samples, emitter pin is PC0

    // 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

    while (1)
    {
        // main body of program goes here
    }
}
```



```
    return 0;
}
```

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 **qtr_read_calibrated()** function, 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 **qtr_read_line()** functions, 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. **qtr_read_line()** 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() // call this routine repeatedly from your main program
{
    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_read_line(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 **qtr_read_line()** 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() // call this routine repeatedly from your main program
{
  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_read_line(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
}
```

6.j. Orangutan Serial Port Communication Functions

Overview

This section of the library provides routines for accessing the serial port (USART) on the **3pi Robot** [<http://www.pololu.com/catalog/product/975>], **Orangutan SV-168** [<http://www.pololu.com/catalog/product/1225>], **Orangutan LV-168** [<http://www.pololu.com/catalog/product/775>], and **Baby Orangutan B** [<http://www.pololu.com/catalog/product/1216>]. The functions were designed for use in master/slave control of the 3pi, but they can be used in many other applications.

The serial port routines normally use the `USART_UDRE_vect` and `USART_RX_vect` interrupts, so they will conflict with any code that also uses these interrupts.

Complete documentation of this library's methods can be found in **Section 9** of the **Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>].

Usage Examples

Example code for making the 3pi Robot into a serial slave, controlled by another microcontroller, is given in **Section 10.a** of the **Pololu 3pi Robot User's Guide** [<http://www.pololu.com/docs/0J21>].

7. Using the Pololu AVR Library for your own projects

After getting one of the simple examples to work with an Orangutan controller or 3pi robot, you can start working on more complicated programs of your own. The library provides easy access to all of the features of the Orangutans and 3pi, including the LCD screen, buttons, LEDs, motors, and buzzer. There are also functions in the library that make it easy for you to do more general-purpose operations with the AVR, such as timing and analog-to-digital conversion. The library also provides support for the Pololu QTR sensors, which should work even if you are not using an Orangutan controller. For a complete list of functions provided by the library, see the **command reference** [<http://www.pololu.com/docs/0J18>].

Usually, the easiest way to adapt this code to your own projects will be to start with a working example and gradually add the things that you need, one step at a time. However, if you want to start from scratch, there are just a few things that you need to know. First, to use the library with C, you must place one of the following lines

```
#include <pololu/orangutan.h>
#include <pololu/3pi.h>
#include <pololu/qtr.h>
```

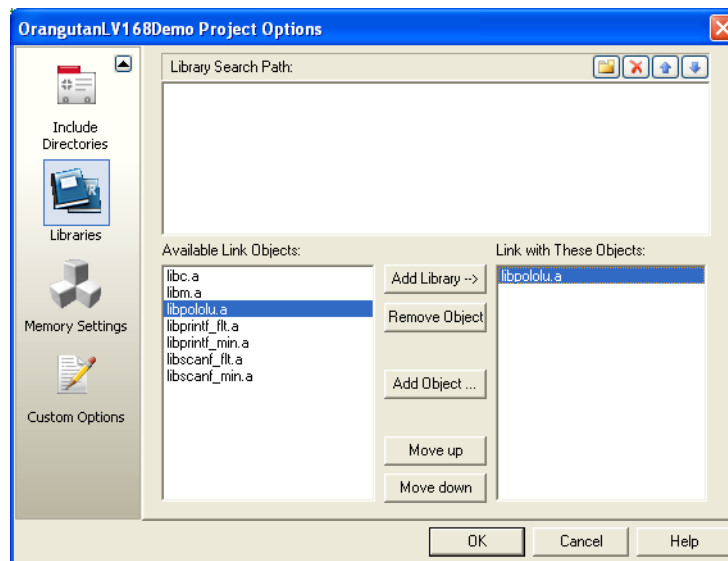
at the top of any C file that uses functions provided by the library. To use the library with C++, the equivalent lines are

```
#include <pololu/orangutan>
#include <pololu/Pololu3pi.h>
#include <pololu/PololuQTRsensors.h>
```

The line or lines that you include depend on which product you are using with the library.

Second, when compiling, you must link your object files with `libpololu.a`. This is accomplished by passing the `-lpololu` option to `avr-gcc` during the linking step.

To add the `-lpololu` option within AVR studio, select **Project > Configuration Options > Libraries**. You should see `libpololu.a` listed as an option on the left column. Select this file and click “add library” to add it to your project.

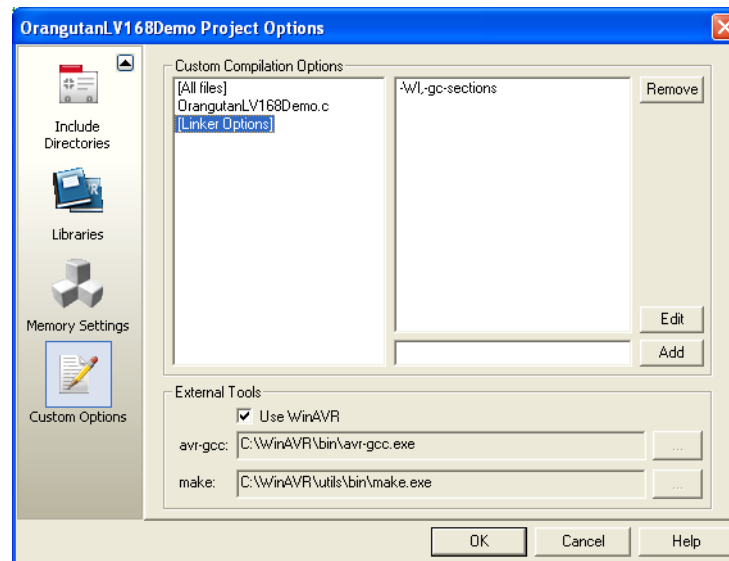


AVR Studio library settings for using the Pololu AVR library (`libpololu`).

Finally, we also strongly recommend the linker option `-Wl,-gc-sections`. This causes unused library functions to not be included, resulting in a much smaller code size. To include this in AVR Studio, select **Project > Configuration Options > Custom Options**. Click on [Linker options] and add:

```
-Wl,-gc-sections
```

to the list. This linker option is included in both the AVR Studio and Linux-based example programs described earlier.



Recommended AVR Studio linker options for projects using the Pololu AVR Library.

8. Additional resources

To learn more about programming AVRs and using the Pololu AVR Library, see the following list of resources:

- **Pololu AVR Library Command Reference** [<http://www.pololu.com/docs/0J18>]: detailed information about every function in the library.
- **Programming Orangutans and the 3pi Robot from the Arduino Environment** [<http://www.pololu.com/docs/0J17>]: information about using this library to program Orangutans and the 3pi robot from within the Arduino environment.
- **Arduino Library for the Pololu QTR Reflectance Sensors** [<http://www.pololu.com/docs/0J19>]: information about using the QTR sensor portion of this library from within the Arduino environment.
- **Pololu 3pi Robot User's Guide** [<http://www.pololu.com/docs/0J21>]
- **Pololu Orangutan SV-168 and LV-168 User's Guide** [<http://www.pololu.com/docs/0J27>]
- **Pololu Baby Orangutan B User's Guide** [<http://www.pololu.com/docs/0J14>]
- **Pololu Orangutan USB Programmer User's Guide** [<http://www.pololu.com/docs/0J6>]
- **WinAVR** [<http://winavr.sourceforge.net/>]
- **AVR Studio** [<http://www.atmel.com/avrstudio/>]
- **AVR Libc Home Page** [<http://www.nongnu.org/avr-libc/>]
- **ATmega168 documentation** [http://www.atmel.com/dyn/products/product_card.asp?PN=ATmega168]
- **Tutorial: AVR Programming on the Mac** [<http://bot-thoughts.blogspot.com/2008/02/avr-programming-on-mac.html>]

Finally, we would like to hear your comments and questions over at the **Pololu Robotics Forum** [<http://forum.pololu.com/>]!