

Pololu AVR Library Command Reference



1. Introduction	2
2. Timing and Delays	3
3. Orangutan Analog-to-Digital Conversion	5
4. Orangutan Buzzer: Beeps and Music	8
5. Orangutan LCD	12
6. Orangutan LEDs	16
7. Orangutan Motor Control	18
8. Orangutan Pushbuttons	20
9. Orangutan Serial Port Communication	22
10. Orangutan System Resources	25
11. QTR Reflectance Sensors	26
12. 3pi Robot Functions	31
13. Wheel Encoders	34

1. Introduction

This document describes a programming library designed for use with Pololu products. The library is used to create programs that run on Atmel ATmega168 and ATmega48 processors, and it supports the following 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. This version has been replaced by the Baby Orangutan B-328.



Pololu Baby Orangutan B-328: a mega328 replacement of the above Baby Orangutan B-168. The mega328 offers even more memory for your programs (32 KB flash, 2 KB RAM).



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.



Encoder for Pololu Wheel 42×19 mm: a wheel encoder solution that allows a robot to measure how far it has traveled.

The library is written in C++ and may be used in three different programming environments:

- **Arduino** [<http://www.arduino.cc>]: a popular, beginner-friendly programming environment for the mega168, using simplified C++ code. We have written a **guide to using Arduino with Orangutan controllers** [<http://www.pololu.com/docs/0J17>] to help you get started.
- **C++**: supported by the AVR-GCC/WinAVR project. See the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>] to get started.
- **C / AVR Studio**: bindings to the C language are included in the library so that you can write programs entirely in C, which is the standard for Atmel's **AVR Studio** [<http://www.atmel.com/avrstudio/>]. See the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>] to get started.

2. Timing and Delays

The following timing and delay functions are designed for the Orangutans and 3pi, which run at 20 MHz. They will give different results at other processor frequencies. **These functions are not available within the Arduino environment, which has its own delay functions.** For the functions in this section only, the C functions are also available from C++.

The timing functions use an interrupt on Timer2, which is configured when `time_reset()`, `get_ms()`, or an equivalent function is called. This means that the timing code will conflict with other code that uses Timer2. However, the functions here are compatible with the other uses of Timer2 within the Pololu library.

Reference

C++ methods are shown in red.

C/C++ functions are shown in green.

static void OrangutanTime::delayMilliseconds(unsigned int *milliseconds*)

void delay_ms(unsigned int *milliseconds*)

void delay(unsigned int *milliseconds*)

Delays for the specified number of milliseconds. Note that if the supplied argument *milliseconds* has a value of zero, this function will return execution immediately (unlike `delayMicroseconds(0)`, which will delay for the maximum time possible).

static void OrangutanTime::delayMicroseconds(unsigned int *microseconds*)

void delayMicroseconds(unsigned int *microseconds*)

void delay_us(unsigned int *microseconds*)

Delays for the specified number of microseconds. Note that if the supplied argument *microseconds* has a value of zero, this function will delay for 65536 us (unlike `delayMilliseconds(0)`, which produces no delay at all).

static void OrangutanTime::reset()

void time_reset()

Starts/resets the system timer. This begins using an interrupt on Timer2 to record the elapsed time since reset.

static unsigned long OrangutanTime::ms()

unsigned long get_ms()

unsigned long millis()

Returns the number of elapsed milliseconds since reset. The value can be as high as the maximum value stored in an unsigned long, 4,294,967,295 ms, which corresponds to a little more than 49 days, after which it starts over at 0.

3. Orangutan Analog-to-Digital Conversion

The `OrangutanAnalog` class and the C functions in this section allow easy access to the analog inputs on the Orangutan controllers and 3pi. These functions take care of configuring and running the analog-to-digital converters, but they do not automatically set the ports to be inputs or enable/disable the pull-up resistors. By default, all ports on the Orangutan are set to inputs, and the pull-ups are turned off, so no additional configuration is needed for most applications.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 5.a** of the guide to **Programming Orangutans and the 3pi Robot from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] or **Section 6.c** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ and Arduino methods are shown in red.

C functions are shown in green.

static void OrangutanAnalog::setMode(unsigned char mode)

void set_analog_mode(unsigned char mode)

Used to set the ADC for either 8-bit or 10-bit conversions. The library defines the keywords **MODE_8_BIT** and **MODE_10_BIT**, which can be used as the argument to this method. When the ADC is in 8-bit mode, conversion results will range from 0 – 255 for voltages ranging from 0 – 5 V. When the ADC is in 10-bit mode, conversion results will range from 0 – 1023 for voltages ranging from 0 – 5 V. The default mode setting is **MODE_10_BIT**.

Example:

```
// run the ADC in 10-bit conversion mode
OrangutanAnalog::setMode(MODE_10_BIT);
```

static unsigned char OrangutanAnalog::getMode()

unsigned char get_analog_mode()

Returns the current ADC mode. The return value will be **MODE_8_BIT** (1) if the ADC is in 8-bit conversion mode, otherwise it will be **MODE_10_BIT** (0). The default mode setting is **MODE_10_BIT**.

static unsigned int OrangutanAnalog::read(unsigned char channel)

unsigned int analog_read(unsigned char channel)

Performs a single analog-to-digital conversion on the specified analog input channel and returns the result. In 8-bit mode, the result will range from 0 – 255 for voltages from 0 – 5 V. In 10-bit mode, the result will range from 0 – 1023 for voltages from 0 – 5 V. The *channel* argument should be 0 – 7. This function will occupy program execution until the conversion is complete (approximately 100 us). The specified pin is configured as an input and its internal pull-up resistor is automatically disabled during the conversion; after the conversion, the pin is restored to its previous state.

static unsigned int OrangutanAnalog::readAverage(unsigned char channel, unsigned int numSamples)

unsigned int analog_read_average(unsigned char channel, unsigned int numSamples)

Performs *numSamples* analog-to-digital conversions on the specified analog input channel and returns the average value of the readings. In 8-bit mode, the result will range from 0 – 255 for voltages from 0 – 5 V. In 10-bit mode, the result will range from 0 – 1023 for voltages from 0 – 5 V. The *channel* argument should be 0 – 7. This function will occupy program execution until all of the requested conversions are complete (approximately 100 us per sample). Pull-up resistors are automatically disabled during the conversion, then restored to their previous state.

static unsigned int OrangutanAnalog::readTrimpot()

unsigned int read_trimpot()

Performs 20 analog-to-digital conversions on the output of the trimmer potentiometer on the 3pi, Orangutan SV-168, Orangutan LV-168, or Baby Orangutan B and returns the average result. In 8-bit mode, the result will range from 0 – 255 for voltages from 0 – 5 V. In 10-bit mode, the result will range from 0 – 1023 for voltages from 0 – 5 V. The trimpot is on analog input 7, so this method is equivalent to `readAverage(TRIMPOT, 20)`.

static int OrangutanAnalog::readTemperatureF()

int read_temperature_f()

Performs 20 analog-to-digital conversions on the output of the temperature sensor on the Orangutan LV-168 and returns average result in tenths of a degree Fahrenheit, so a result of 827 would mean a temperature of 82.7 degrees F. The temperature sensor is on analog input 6, so this method is equivalent to `readAverage(TEMP_SENSOR, 20)` converted to tenths of a degree F.

static int readTemperatureC()

int read_temperature_c()

This method is the same as readTemperatureF() above, except that it returns the temperature in tenths of a degree Celcius.

static int readBatteryMillivolts_3pi()

int read_battery_millivolts_3pi()

Performs 10 analog-to-digital conversions on the battery voltage sensing circuit of the 3pi and returns the average result in millivolts. A result of 5234 would mean a battery voltage of 5.234 V. For rechargeable NiMH batteries, the voltage usually starts at a value above 5 V and drops to around 4 V before the robot shuts off, so monitoring this number can be helpful in determining when to recharge batteries.

static int readBatteryMillivolts_SV168()

int read_battery_millivolts_sv168()

Just like read_battery_millivolts_3pi() above, but uses the correct voltage factor for the Orangutan SV-168. The minimum operating voltage for the SV-168 is 6 V, but if the batteries drop significantly below their rated value, whatever it is, they should be recharged. For example, a six-cell NiMH pack should be recharged if it drops below about 7 V.

static void OrangutanAnalog::startConversion(unsigned char channel)

void start_analog_conversion(unsigned char channel)

Initiates an ADC conversion that runs in the background, allowing the CPU to perform other tasks while the conversion is in progress. The procedure is to start a conversion on an analog input with this method, then poll isConverting() in your main loop. Once isConverting() returns a zero, the result can be obtained through a call to conversionResult() and this method can be used to start a new conversion. Note that this function automatically configures the specified pin as an input and disables its internal pull-up resistor. Since this function only initiates the conversion, it cannot restore the pin to its previous state once the conversion is finished like the read() method does.

static unsigned char OrangutanAnalog::isConverting()

unsigned char analog_is_converting()

Returns a 1 if the ADC is in the middle of performing a conversion, otherwise it returns a 0. The ATmega168 is only capable of performing one analog-to-digital conversion at a time.

static unsigned int conversionResult()

unsigned int analog_conversion_result()

Returns the result of the previous analog-to-digital conversion. In 8-bit mode, the result will range from 0 – 255 for voltages from 0 – 5 V. In 10-bit mode, the result will range from 0 – 1023 for voltages from 0 – 5 V.

static unsigned int toMillivolts(unsigned int adcResult)

static unsigned int to_millivolts(unsigned int adc_result)

Converts the result of an analog-to-digital conversion to millivolts. This assumes a board power level of exactly 5000 mV.

Example:

```
OrangutanAnalog::toMillivolts(OrangutanAnalog::read(0));  
// e.g. will return 5000 if analog input 0 is at 5 V
```

4. Orangutan Buzzer: Beeps and Music

The OrangutanBuzzer class and the C functions in this section allow various sounds to be played on the buzzer of the Orangutan SV-168, Orangutan LV-168, and 3pi, from simple beeps to complex tunes. The buzzer is controlled using one of the Timer1 PWM outputs, so it will conflict with any other uses of Timer1. Note durations are timed using a Timer1 overflow interrupt, which will briefly pause your main program at the frequency of the sound. In most cases, the interrupt-handling routine is very short. However, when playing a sequence of notes in PLAY_AUTOMATIC mode (the default mode) with the **play()** command, longer interrupts occur between notes. It is important to take this into account when writing timing-critical code.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 5.b** of the guide to **Programming Orangutans from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] or **Section 6.d** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ and Arduino methods are shown in red.

C functions are shown in green.

static void OrangutanBuzzer::playFrequency(unsigned int *frequency*, unsigned int *duration*, unsigned char *volume*)

void play_frequency(unsigned int *freq*, unsigned int *duration*, unsigned char *volume*)

This method will play the specified frequency (in Hz or 0.1 Hz) for the specified duration (in ms). *frequency* must be between 40 Hz and 10 kHz. If the most significant bit of *frequency* is set, the frequency played is the value of the lower 15 bits of *frequency* in units of 0.1 Hz. Therefore, you can play a frequency of 44.5 Hz by using a *frequency* of **DIV_BY_10** | 445. If the most significant bit of *frequency* is not set, the units for *frequency* are Hz. *volume* controls the buzzer volume, with 15 being the loudest and 0 being the quietest.

Example

```
// play a 6 kHz note for 250 ms at half volume
OrangutanBuzzer::playFrequency(6000, 250, 7);
// wait for buzzer to finish playing the note
while (OrangutanBuzzer::isPlaying());

// play a 44.5 Hz note for 1 s at full volume
OrangutanBuzzer::playFrequency(DIV_BY_10 | 445, 1000, 15);
// wait for buzzer to finish playing the note
// (or just use a 1000 ms delay)
while (OrangutanBuzzer::isPlaying());
```

Caution: *frequency*duration/1000* must be no greater than 0xFFFF (65535). This means you can't use a max duration of 65535 ms for frequencies greater than 1 kHz. For example, the maximum duration you can use for a frequency of 10 kHz is 6553 ms. If you use a duration longer than this, you will produce an integer overflow that can result in unexpected behavior.

static void OrangutanBuzzer::playNote(unsigned char *note*, unsigned int *duration*, unsigned char *volume*)

void play_note(unsigned char *note*, unsigned int *duration*, unsigned char *volume*);

This method will play the specified note for the specified duration (in ms). *volume* controls the buzzer volume, with 15 being the loudest and 0 being the quietest. The *note* argument is an enumeration for the notes of the equal tempered scale (ETS):

Note Macros

To make it easier for you to specify notes in your code, this library defines the following macros:

```
// x will determine the octave of the note
#define C( x )      ( 0 + x*12 )
#define C_SHARP( x ) ( 1 + x*12 )
#define D_FLAT( x ) ( 1 + x*12 )
#define D( x )      ( 2 + x*12 )
#define D_SHARP( x ) ( 3 + x*12 )
#define E_FLAT( x ) ( 3 + x*12 )
#define E( x )      ( 4 + x*12 )
#define F( x )      ( 5 + x*12 )
#define F_SHARP( x ) ( 6 + x*12 )
#define G_FLAT( x ) ( 6 + x*12 )
#define G( x )      ( 7 + x*12 )
#define G_SHARP( x ) ( 8 + x*12 )
#define A_FLAT( x ) ( 8 + x*12 )
#define A( x )      ( 9 + x*12 )
#define A_SHARP( x ) ( 10 + x*12 )
#define B_FLAT( x ) ( 10 + x*12 )
#define B( x )      ( 11 + x*12 )
```

```
// 255 (silences buzzer for the note duration)
#define SILENT_NOTE 0xFF

// e.g. frequency = 445 | DIV_BY_10
// gives a frequency of 44.5 Hz
#define DIV_BY_10 (1 << 15)
```

static void OrangutanBuzzer::play(const char* sequence)
void play(const char* sequence)

This method plays the specified sequence of notes. If the play mode is **PLAY_AUTOMATIC** (default), the sequence of notes will play with no further action required by the user. If the play mode is **PLAY_CHECK**, the user will need to call `playCheck()` in the main loop to initiate the playing of each new note in the sequence. The play mode can be changed while the sequence is playing. The sequence syntax is modeled after the **PLAY** commands in GW-BASIC, with just a few differences.

The notes are specified by the characters **C**, **D**, **E**, **F**, **G**, **A**, and **B**, and they are played by default as “quarter notes” with a length of 500 ms. This corresponds to a tempo of 120 beats/min. Other durations can be specified by putting a number immediately after the note. For example, **C8** specifies C played as an eighth note, with half the duration of a quarter note. The special note **R** plays a rest (no sound). The sequence parser is case-insensitive and ignore spaces, which may be used to format your music nicely.

Various control characters alter the sound:

- ‘**A**’ – ‘**G**’: used to specify the notes that will be played
- ‘**R**’: used to specify a rest (no sound for the duration of the note)
- ‘>’ plays the next note one octave higher
- ‘<’ plays the next note one octave lower
- ‘+’ or ‘#’ after a note raises any note one half-step
- ‘-’ after a note lowers any note one half-step
- ‘.’ after a note “dots” it, increasing the length by 50%. Each additional dot adds half as much as the previous dot, so that “A..” is 1.75 times the length of “A”.
- ‘**O**’ followed by a number sets the octave (default: **O4**).
- ‘**T**’ followed by a number sets the tempo in beats/min (default: **T120**).
- ‘**L**’ followed by a number sets the default note duration to the type specified by the number: 4 for quarter notes, 8 for eighth notes, 16 for sixteenth notes, etc. (default: **L4**).
- ‘**V**’ followed by a number from 0-15 sets the music volume (default: **V15**).
- ‘**MS**’ sets all subsequent notes to play play staccato – each note is played for 1/2 of its allotted time, followed by an equal period of silence.
- ‘**ML**’ sets all subsequent notes to play legato – each note is played for full length. This is the default setting.
- ‘!’ resets the octave, tempo, duration, volume, and staccato setting to their default values. These settings persist from one `play()` to the next, which allows you to more conveniently break up your music into reusable sections.
- ‘**1**’ – “**2000**”: when immediately following a note, a number determines the duration of the note. For example, **C16** specifies C played as a sixteenth note (1/16th the length of a whole note).

Examples:

```
// play a C major scale up and back down:
OrangutanBuzzer::play("!L16 V8 cdefgab>cbagfedc");
```

```
// the first few measures of Bach's fugue in D-minor
OrangutanBuzzer::play("!T240 L8 a gafaeada c+adaeafa >aa>bac#ada c#adaeaf4");
```

static void `playFromProgramSpace`(const char* *sequence*)

void `play_from_program_space`(const char* *sequence*)

A version of `play()` that takes a pointer to program space instead of RAM. This is desirable since RAM is limited and the string must be stored in program space anyway.

Example:

```
#include <avr/pgmspace.h>
const char melody[] PROGMEM = "!L16 V8 cdefgab>cbagfedc";

void someFunction()
{
    OrangutanBuzzer::playFromProgramSpace(melody);
}
```

static void `OrangutanBuzzer::playMode`(unsigned char *mode*)

void `play_mode`(char *mode*)

This method lets you determine whether the notes of the `play()` sequence are played automatically in the background or are driven by the `playCheck()` method. If *mode* is **PLAY_AUTOMATIC**, the sequence will play automatically in the background, driven by the Timer1 overflow interrupt. The interrupt will take a considerable amount of time to execute when it starts the next note in the sequence playing, so it is recommended that you do not use automatic-play if you cannot tolerate being interrupted for more than a few microseconds. If *mode* is **PLAY_CHECK**, you can control when the next note in the sequence is played by calling the `playCheck()` method at acceptable points in your main loop. If your main loop has substantial delays, it is recommended that you use automatic-play mode rather than play-check mode. Note that the play mode can be changed while the sequence is being played. The mode is set to **PLAY_AUTOMATIC** by default.

static unsigned char `OrangutanBuzzer::playCheck`()

unsigned char `play_check`()

This method only needs to be called if you are in **PLAY_CHECK** mode. It checks to see whether it is time to start another note in the sequence initiated by `play()`, and starts it if so. If it is not yet time to start the next note, this method returns without doing anything. Call this as often as possible in your main loop to avoid delays between notes in the sequence. This method returns 0 (false) if the melody to be played is complete, otherwise it returns 1 (true).

static unsigned char `isPlaying`()

unsigned char `is_playing`()

This method returns 1 (true) if the buzzer is currently playing a note/frequency. Otherwise, it returns 0 (false). You can poll this method to determine when it's time to play the next note in a sequence, or you can use it as the argument to a delay loop to wait while the buzzer is busy.

static void `OrangutanBuzzer::stopPlaying`()

void `stop_playing`()

This method will immediately silence the buzzer and terminate any note/frequency/melody that is currently playing.

5. Orangutan LCD

The OrangutanLCD class and the C functions in this section provide a variety of ways of displaying data to the LCD screen of an Orangutan SV-168, Orangutan LV-168, and 3pi, providing an essential tool for user interfaces and debugging. The library 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 Arduino libraries. It 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 function, for example, as pushbutton inputs and an LED driver on the 3pi and Orangutans.

For C and C++ users, the standard C function **printf()** is made available. See below for more information.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 5.c** of the guide to **Programming Orangutans from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] or **Section 6.e** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ and Arduino methods are shown in red.

C functions are shown in green.

static void OrangutanLCD::clear()

void clear()

Clears the display and returns the cursor to the upper-left corner (0, 0).

void OrangutanLCD::initPrintf()

void lcd_init_printf()

Initializes the display for use with the standard C function **printf()**. This is not available in the Arduino environment. See **the avr-libc manual** [http://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html] for more information on how to use printf with an AVR, and please note that using **printf()** will consume a significant amount of your Orangutan's resources.

static void OrangutanLCD::print(unsigned char character)

Prints a single ASCII character to the display at the current cursor position.

static void OrangutanLCD::print(char character)

void print_character(char character)

Prints a single ASCII character to the display at the current cursor position. This is the same as the *unsigned char* version above.

Example:

```
OrangutanLCD::print('A');
```

static void OrangutanLCD::print(const char *str)

void print(const char *str)

Prints a zero-terminated string of ASCII characters to the display starting at the current cursor position. The string will not wrap or otherwise span lines.

Example:

```
OrangutanLCD::print("Hello!");
```

static void OrangutanLCD::printFromProgramSpace(const char *str)

void print_from_program_space(const char *str)

Prints a string stored in program memory. This can help save a few bytes of RAM for each message that your program prints. Even if you use the normal `print()` function, the strings will be initially stored in program space anyway, so it should never hurt you to use this function.

Example:

```
#include <avr/pgmspace.h>
const char message[] PROGMEM = "Hello!";

void someFunction()
{
    OrangutanLCD::printFromProgramSpace(message);
}
```

static void OrangutanLCD::print(int value)

Prints the specified signed integer (2-byte) value to the display at the current cursor position. It will not wrap or otherwise span lines. There is no C version of this method, but `print_long(value)` should be sufficient.

Example:

```
OrangutanLCD::print(-25);
```

static void OrangutanLCD::print(long value)

void print_long(long value)

Prints the specified signed long (4-byte) value to the display at the current cursor position. It will not wrap or otherwise span lines.

static void OrangutanLCD::print(unsigned int value)

Prints the specified unsigned integer (2-byte) value to the display at the current cursor position. The value will not wrap or otherwise span lines and will always be positive.

static void OrangutanLCD::print(unsigned long value)

void print_unsigned_long(unsigned long value)

Prints the specified unsigned long (4-byte) value to the display at the current cursor position. The value will not wrap or otherwise span lines and will always be positive.

static void OrangutanLCD::printHex(unsigned int value)

void print_hex(unsigned int value)

Prints the specified two-byte value in hex to the display at the current cursor position. The value will not wrap or otherwise span lines.

static void OrangutanLCD::printHex(unsigned char value)

void print_hex_byte(unsigned char value)

Prints the specified byte value in hex to the display at the current cursor position. The value will not wrap or otherwise span lines.

static void OrangutanLCD::printBinary(unsigned char value)

void print_binary(unsigned char value)

Prints the specified byte in binary to the display at the current cursor position. The value will not wrap or otherwise span lines.

static void OrangutanLCD::gotoXY(unsigned char x, unsigned char y)

void lcd_goto_xy(int col, int row)

Moves the cursor to the specified (x, y) location on the LCD. The top line is y = 0 and the leftmost character column is x = 0, so you can return to the upper-left home position by calling lcd.gotoXY(0, 0), and you can go to the start of the second LCD line by calling lcd.gotoXY(0, 1);

static void OrangutanLCD::showCursor(unsigned char cursorType)

void lcd_show_cursor()

Displays the cursor as either a blinking or solid block. This library defines literals **CURSOR_BLINKING** and **CURSOR_SOLID** for use as an argument to this method.

static void OrangutanLCD::hideCursor()

void lcd_hide_cursor()

Hides the cursor.

static void OrangutanLCD::moveCursor(unsigned char direction, unsigned char distance)

void lcd_move_cursor(unsigned char direction, unsigned char num)

Moves the cursor left or right by *distance* spaces. This library defines literals **LCD_LEFT** and **LCD_RIGHT** for use as a *direction* argument to this method.

static void OrangutanLCD::scroll(unsigned char direction, unsigned char distance, unsigned int delay_time)

void `lcd_scroll`(unsigned char *direction*, unsigned char *num*, unsigned int *delay_time*)

Shifts the display left or right by *distance* spaces, delaying for *delay_time* milliseconds between each shift. This library defines literals `LCD_LEFT` and `LCD_RIGHT` for use as a *direction* argument to this method. Execution does not return from this method until the shift is complete.

static void `OrangutanLCD::loadCustomCharacter`(const char **picture_ptr*, unsigned char *number*)

void `lcd_load_custom_character`(const char **picture_ptr*, unsigned char *number*)

Loads a custom character drawing into the memory of the LCD. The parameter ‘number’ is a character value between 0 and 7, which represents the character that will be customized. That is, `lcd.print((char)number)` or `print_character(number)` will display this drawing in the future.

Note: the `clear()` method must be called before these characters are used.

The pointer *picture_ptr* must be a pointer to an 8 byte array in **program space** containing the picture data. Bit 0 of byte 0 is the upper-right pixel of the 5×8 character, and bit 4 of byte 7 is the lower-left pixel. The example below demonstrates how to construct this kind of array.

Example:

```
#include <avr/pgmspace.h>
// the PROGMEM macro comes from the pgmspace.h header file
// and causes the smile pointer to point to program memory instead
// of RAM
const char smile[] PROGMEM = {
  0b00000,
  0b01010,
  0b01010,
  0b01010,
  0b00000,
  0b10001,
  0b01110,
  0b00000
};

void setup()
{
  // set character 3 to a smiley face
  OrangutanLCD::loadCustomCharacter(smile, 3);

  // clear the lcd (this must be done before we can use the above character)
  OrangutanLCD::clear();

  // display the character
  OrangutanLCD::print((char)3);
}
```

6. Orangutan LEDs

The OrangutanLEDs class and the C functions in this section are a very simple interface to the two user LEDs included on Orangutan controllers and 3pi. Note that the green/right LED is on the same pin as an LCD control pin; this LED will blink briefly whenever data is sent to the LCD, but the two functions will otherwise not interfere with each other.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 5.d** of the guide to **Programming Orangutans from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] or **Section 6.f** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ and Arduino methods are shown in red.

C functions are shown in green.

static void OrangutanLEDs::red(unsigned char *state*)

void red_led(unsigned char *state*)

This method will turn the red user LED off if *state* is zero, otherwise it will turn the red user LED on. You can use the Arduino keyword **HIGH** as an argument to turn the LED on, and you can use the Arduino keyword **LOW** as an argument to turn the LED off.

Example:

```
OrangutanLEDs::red(0); // turn the red LED on
```

static void OrangutanLEDs::left(unsigned char *state*)

void left_led(unsigned char *state*)

This method is an alternate version of red(). The red LED is on the left side of the 3pi robot.

static void OrangutanLEDs::green(unsigned char *state*)

void green_led(unsigned char *state*)

This method will turn the green user LED off if *state* is zero, otherwise it will turn the green user LED on. Within the Arduino environment, you can use the Arduino keyword **HIGH** as an argument to turn the LED on, and you can use the Arduino keyword **LOW** as an argument to turn the LED off. This method will **not** work on the Baby Orangutan as it does not have a green user LED.

Example:

```
OrangutanLEDs::green(1); // turn the green LED on
```

static void OrangutanLEDs::right(unsigned char *state*)

void right_led(unsigned char *state*)

This method is an alternate version of green(). The green LED is on the right side of the 3pi robot.

7. Orangutan Motor Control

The OrangutanMotors class and the C functions in this section allow PWM speed control of the two motor channels on the Orangutan controllers and 3pi. The motor control functions rely on PWM outputs from Timer0 and Timer2, so they will conflict with other code using these timers. Unfortunately the Arduino environment relies on Timer0 for its **millis()** and **delay()** functions, but this library enables a Timer2 interrupt that restores the functionality of **millis()** and **delay()** to normal. This interrupt is not included in the C and C++ versions of the library.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 5.e** of the guide to **Programming Orangutans from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] or **Section 6.g** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ and Arduino methods are shown in red.

C functions are shown in green.

static void OrangutanMotors::setM1Speed(int speed)

void set_m1_speed(int speed)

This method will set the speed and direction of motor 1. Speed is a value between -255 and +255. The sign of *speed* determines the direction of the motor and the magnitude determines the speed. *speed* = 0 results in full brake while *speed* = 255 or -255 results in maximum speed forward or backward. If a speed greater than 255 is supplied, the motor speed will be set to 255. If a speed less than -255 is supplied, the motor speed will be set to -255.

static void OrangutanMotors::setM2Speed(int speed)

void set_m2_speed(int speed)

This method will set the speed and direction of motor 2.

static void OrangutanMotors::setSpeeds(int m1Speed, int m2Speed)

void set_motors(int m1, int m2)

This method will set the speeds and directions of motors 1 and 2.

8. Orangutan Pushbuttons

The OrangutanPushbuttons class and the C functions in this section provide access to the three pushbuttons on the Orangutan SV-168, Orangutan LV-168, and 3pi. Various methods are provided for accessing button presses, which will be useful in different situations.

For a higher level overview of this library and programs that show how this library can be used, please see **Section 5.f** of the guide to **Programming Orangutans from the Arduino Environment** [<http://www.pololu.com/docs/0J17>] or **Section 6.h** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ and Arduino methods are shown in red.

C functions are shown in green.

static unsigned char OrangutanPushbuttons::waitForPress(unsigned char *buttons*)

unsigned char wait_for_button_press(unsigned char *buttons*)

This method will wait for any of the buttons specified by *buttons* to be pressed, at which point execution will return. The argument *buttons* can be a combination of the literals **TOP_BUTTON**, **MIDDLE_BUTTON**, and **BOTTOM_BUTTON** (for the Orangutans) or **BUTTON_A**, **BUTTON_B**, and **BUTTON_C** (for the 3pi) separated by the bitwise OR operator `|`. The returned value is the ID of the button that was pressed. Note that this method takes care of button debouncing.

Example:

```
unsigned char button = OrangutanPushbuttons::waitForPress(TOP_BUTTON | BOTTOM_BUTTON);
```

static unsigned char OrangutanPushbuttons::waitForRelease(unsigned char *buttons*)

unsigned char wait_for_button_release(unsigned char *buttons*)

This method will wait for any of the buttons specified by *buttons* to be released, at which point execution will return. The returned value is the ID of the button that was released. Note that this method takes care of button debouncing.

static unsigned char OrangutanPushbuttons::waitForButton(unsigned char *buttons*)

unsigned char wait_for_button(unsigned char *buttons*)

This method will wait for any of the buttons specified by *buttons* to be pressed, and then it will wait for the pressed button to be released, at which point execution will return. The returned value is the ID of the button that was pressed and released. Note that this method takes care of button debouncing.

static unsigned char OrangutanPushbuttons::isPressed(unsigned char *buttons*)

unsigned char button_is_pressed(unsigned char *buttons*)

This method will return the value of any of the buttons specified by *buttons* that is currently pressed. For example, if you call `buttons.isPressed(ALL_BUTTONS)` and both the top and middle buttons are pressed, the return value will be `(TOP_BUTTON | MIDDLE_BUTTON)`. If none of the specified buttons is pressed, the returned value will be 0. The argument *buttons* can refer to multiple buttons (see the `waitForPress()` method above).

9. Orangutan Serial Port Communication

The `OrangutanSerial` class and the C functions in this section provide access to the serial port on the Baby Orangutan B, Orangutan SV-168, Orangutan LV-168, and 3pi. This allows two-way, TTL-level communication on pins PD0 (RX) and PD1 (TX) with another microcontroller, a serial device, or (through a USB-serial adapter or RS-232 level converter) a laptop or desktop computer. **These functions are not available within the Arduino environment, which has its own serial functions.**

When sending data, a `USART_UDRE_vect` interrupt vector is called after each byte is sent, allowing the library to automatically start sending the next byte from the send buffer. When receiving data, a `USART_RX_vect` interrupt vector is called after each byte is received, allowing the library to automatically store the byte in the receive buffer. To use a polling method instead of interrupts, see the `setMode()` and `check()` functions below.

For a higher level overview of this library and programs that show how this library can be used, please see **Section 6.j** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ methods are shown in red.

C functions are shown in green.

static void OrangutanSerial::setBaudRate(unsigned long *baud*)

unsigned char serial_set_baud_rate(unsigned long *baud*)

Sets the baud rate on the serial port. Standard values up to 115200 should work fine; for higher speeds, please consult the AVR documentation.

static void OrangutanSerial::receive(char **buffer*, unsigned char *size*)

void serial_receive(char **buffer*, unsigned char *size*)

Sets up a buffer for background reception. This function returns immediately, but data arriving at the serial port will be copied into this buffer until *size* bytes have been stored.

static char OrangutanSerial::receiveBlocking(char **buffer*, unsigned char *size*, unsigned int *timeout_ms*)

char serial_receive_blocking(char **buffer*, unsigned char *size*, unsigned int *timeout_ms*)

Receives data, not returning until the buffer is full or the timeout (specified in milliseconds) has expired. This function is useful for simple programs and for situations in which you know exactly how many bytes to expect.

static void OrangutanSerial::receiveRing(char **buffer*, unsigned char *size*)

void serial_receive_ring(char *)*buffer*, unsigned char *size*)

Sets up a ring buffer for background reception. This is a more advanced version of `serial_receive` that is useful when you need to read in data continuously without pauses. When the buffer is filled, `getReceivedBytes()` will reset to zero, and data will continue to be inserted at the beginning of the buffer.

static void OrangutanSerial::cancelReceive()

void serial_cancel_receive()

Stops background serial reception.

static inline unsigned char OrangutanSerial::getReceivedBytes()

unsigned char serial_get_received_bytes()

Gets the number of bytes that have been read into the buffer; this is also the index of the location at which the next received byte will be added.

static inline char OrangutanSerial::receiveBufferFull()

char serial_receive_buffer_full()

Returns true when the receive buffer has been filled with received bytes, so that serial reception is halted. This function should not be called when receiving data into a ring buffer.

static void OrangutanSerial::send(char **buffer*, unsigned char *size*)

void serial_send(char **buffer*, unsigned char *size*)

Sets up a buffer for background transmit. Data from this buffer will be transmitted until *size* bytes have been sent. If `send()` is called before `sendBufferEmpty()` returns true (when transmission of the last byte has started), the old buffer will be discarded and transmission will be cut short. This means that you should almost always wait until the data has been sent before calling this function again. See `sendBlocking()`, below, for an easy way to do this.

static void OrangutanSerial::sendBlocking(char **buffer*, unsigned char *size*)

void serial_send_blocking(char **buffer*, unsigned char *size*)

Same as `send()`, but waits until transmission of the last byte has started. When this function returns, it is safe to call `send()` or `sendBlocking()` again.

static inline unsigned char OrangutanSerial::getSentBytes()

unsigned char serial_get_sent_bytes()

Gets the number of bytes that have been sent since **send()** was called.

static char OrangutanSerial::sendBufferEmpty()

char serial_send_buffer_empty()

True when the send buffer is empty; when there are no more bytes to send.

static void OrangutanSerial::setMode(unsigned char mode)

void serial_set_mode(unsigned char mode)

Sets the serial library to use either interrupts (with the argument **SERIAL_AUTOMATIC**; the default) or polling (**SERIAL_CHECK**). If **SERIAL_CHECK** is selected, your code must call **check()** often to ensure reliable reception and timely transmission of data.

static char OrangutanSerial::getMode(unsigned char mode)

char serial_get_mode()

Returns the current serial mode,

static void OrangutanSerial::check()

void serial_check()

Checks for any bytes to be received or transmitted and performs the required action. Call this function often when in **SERIAL_CHECK** mode. In **SERIAL_AUTOMATIC** mode, you will not need to use this function.

10. Orangutan System Resources

This section of the library is intended to provide access to information about resources that are available on the Orangutan board. Currently, it only provides information about the amount of free RAM on the AVR.

static unsigned char OrangutanResources::getFreeRAM()
unsigned char get_free_ram()

Returns an estimate of the available free RAM on the AVR, in bytes. This is computed as the difference between the bottom of the stack and the top of the static variable space or the top of the **malloc()** heap. This function is very useful for avoiding disastrous and difficult-to-debug problems that can occur at any time due to the nature of the C and C++ programming languages. Local variables and the location of function calls are stored on the *stack* in RAM, global and data variables take up additional RAM, and some programs dynamically allocate RAM with the **malloc()** set of functions. While **malloc()** will refuse to allocate memory that has already been used for another purpose, if the stack grows large enough it will silently overwrite other regions of RAM. This kind of problem, called a *stack overflow*, can have unexpected and seemingly random effects, such as:

- a program restart, as if the board was reset,
- sudden jumps to arbitrary locations in the program,
- behavior that seems logically impossible, and
- data corruption.

Small stack overflows that happen rarely might cause bugs that are subtle and hard to detect. We recommend that you use **getFreeRAM()** within your main loop and also at some points within function calls, especially any recursive or highly nested calls, and cause your robot to display an error indicator or a warning of some type if memory gets tight. If your Orangutan is controlling a system that might damage itself or cause danger to an operator it should go into a safe shutdown mode immediately upon detection of a low memory error. For example, a BattleBot could shut down all motors, and a robotic aircraft could deploy its parachute.

By checking available memory at various levels within your code, you can get an idea of how much memory each function call consumes, and think about redesigning the code to use memory more efficiently. The **getFreeRam()** function itself should not take a noticeable amount of time and use just 6 bytes of RAM itself, so you can use it freely throughout your code.

See the **avr-libc malloc page** [<http://www.nongnu.org/avr-libc/user-manual/malloc.html>] for more information about memory organization in C on the AVR.

11. QTR Reflectance Sensors

The `PololuQTRSensors` class and the C functions in this section provide an interface for using Pololu's **QTR reflectance sensors** [<http://www.pololu.com/catalog/product/961>] together with the Orangutan, Arduino, or other mega168-based boards. The library provides access to the raw sensors values as well as to high level functions including calibration and line-tracking.

We recommend not using this part of the library directly on the 3pi. Instead, we have provided an initialization function and convenient access functions through the `Pololu3pi` class. See **Section 12** for details.

This section of the library defines an object for each of the two QTR sensor types, with the **`PololuQTRSensorsAnalog`** class intended for use with QTR-xA sensors and the **`PololuQTRSensorsRC`** class intended for use with QTR-xRC sensors. 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. This is achieved by having both of these classes derive from the abstract base class **`PololuQTRSensors`**. This base class cannot be instantiated.

The `PololuQTRSensorsAnalog` and `PololuQTRSensorsRC` classes are the only classes in the Pololu AVR library that must be instantiated before they are used. This allows multiple QTR sensor arrays to be controlled independently as separate `PololuQTRSensors` objects. The multiple independent array support is not available within the C environment, but multiple arrays can still be configured as a single array, as long as the total number of sensors does not exceed 8.

For calibration, memory is allocated using the `malloc()` command. This conserves RAM: if all eight sensors are calibrated with the emitters both on and off, a total of 64 bytes would be dedicated to storing calibration values. However, for an application where only three sensors are used, and the emitters are always on during reads, only 6 bytes are required.

Note that the `PololuQTRSensorsRC` class uses `Timer2` during sensor reads to time the sensor pulses, so it might not work with code that uses `Timer2` for other purposes. Once the sensor read is complete, `Timer2` is restored to its original state; there are no restrictions on its use between sensor reads. The `PololuQTRSensorsAnalog` class does not use `Timer2` at all, and all of the `PololuQTRSensors` code is compatible with the other Pololu AVR libraries.

For a higher level overview of this library and example programs that show how this library can be used, please see the guide **Arduino Libraries for the Pololu QTR Reflectance Sensors** [<http://www.pololu.com/docs/0J19>] or **Section 6.i** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ and Arduino methods are shown in red.

C functions are shown in green.

void PololuQTRSensors::read(unsigned int *sensorValues, unsigned char readMode = QTR_EMITTERS_ON)

void qtr_read(unsigned int *sensorValues, unsigned char readMode)

Reads the raw sensor values into an array. There **MUST** be space for as many values as there were sensors specified in the constructor. The values returned are a measure of the reflectance in units that depend on the type of sensor being used, with higher values corresponding to lower reflectance (a black surface or a void). QTR-xA sensors will return a raw value between 0 and 1023. QTR-xRC sensors will return a raw value between 0 and the *timeout* argument provided in the constructor (which defaults to 4000). The units will be in Timer2 counts, where Timer2 is running at the CPU clock divided by 8 (i.e. 2 MHz on a 16 MHz processor, or 2.5 MHz on a 20 MHz processor).

The functions that read values from the sensors all take an argument *readMode*, which specifies the kind of read that will be performed. Several options are defined: **QTR_EMITTERS_OFF** specifies that the reading should be made without turning on the infrared (IR) emitters, in which case the reading represents ambient light levels near the sensor; **QTR_EMITTERS_ON** specifies that the emitters should be turned on for the reading, which results in a measure of reflectance; and **QTR_EMITTERS_ON_AND_OFF** specifies that a reading should be made in both the on and off states. The values returned when the **QTR_EMITTERS_ON_AND_OFF** option is used are given by **on + max – off**, where **on** is the reading with the emitters on, **off** is the reading with the emitters off, and **max** is the maximum sensor reading. This option can reduce the amount of interference from uneven ambient lighting. Note that emitter control will only work if you specify a valid emitter pin in the constructor.

Example usage:

```
unsigned int sensor_values[8];
sensors.read(sensor_values);
```

void PololuQTRSensors::emittersOn()

void qtr_emitters_on()

Turn the IR LEDs on. This is mainly for use by the read method, and calling these functions before or after the reading the sensors will have no effect on the readings, but you may wish to use these for testing purposes. This method will only do something if a valid emitter pin was specified in the constructor.

void PololuQTRSensors::emittersOff()

void qtr_emitters_off()

Turn the IR LEDs off. This is mainly for use by the read method, and calling these functions before or after the reading the sensors will have no effect on the readings, but you may wish to use these for testing purposes.

void PololuQTRSensors::calibrate(unsigned char readMode = QTR_EMITTERS_ON)

void qtr_calibrate(unsigned char readMode)

Reads the sensors for calibration. The sensor values are not returned; instead, the maximum and minimum values found over time are stored internally and used for the **readCalibrated()** method. You can access the calibration (i.e raw max and min sensor readings) through the public member pointers **calibratedMinimumOn**, **calibratedMaximumOn**, **calibratedMinimumOff**, and **calibratedMaximumOff**. Note that these pointers will point to arrays of length *numSensors*, as specified in the constructor, and they will only be allocated after **calibrate()** has been called. If you only calibrate with the emitters on, the calibration arrays that hold the off values will not be allocated.

void PololuQTRSensors::readCalibrated(unsigned int *sensorValues, unsigned char readMode = QTR_EMITTERS_ON)

void `qtr_read_calibrated`(unsigned int **sensorValues*, unsigned char *readMode*)

Returns sensor readings calibrated to a value between 0 and 1000, where 0 corresponds to a reading that is less than or equal to the minimum value read by `calibrate()` and 1000 corresponds to a reading that is greater than or equal to the maximum value. Calibration values are stored separately for each sensor, so that differences in the sensors are accounted for automatically.

unsigned int `PololuQTRSensors::readLine`(unsigned int **sensorValues*, unsigned char *readMode* = `QTR_EMITTERS_ON`, unsigned char *whiteLine* = 0)

void `qtr_read_line`(unsigned int **sensorValues*, unsigned char *readMode*)

Operates the same as `read calibrated`, but with a feature designed for line following: this function returns an estimated position of the line. The estimate is made using a weighted average of the sensor indices multiplied by 1000, so that a return value of 0 indicates that the line is directly below sensor 0, a return value of 1000 indicates that the line is directly below sensor 1, 2000 indicates that it's below sensor 2000, etc. Intermediate values indicate that the line is between two sensors. The formula is:

$$\frac{0 * \text{value0} + 1000 * \text{value1} + 2000 * \text{value2} + \dots}{\text{value0} + \text{value1} + \text{value2} + \dots}$$

As long as your sensors aren't spaced too far apart relative to the line, this returned value is designed to be monotonic, which makes it great for use in closed-loop PID control. Additionally, this method remembers where it last saw the line, so if you ever lose the line to the left or the right, it's line position will continue to indicate the direction you need to go to reacquire the line. For example, if sensor 4 is your rightmost sensor and you end up completely off the line to the left, this function will continue to return 4000.

By default, this function assumes a dark line (high values) surrounded by white (low values). If your line is light on black, set the optional second argument *whiteLine* to true. In this case, each sensor value will be replaced by the maximum possible value minus its actual value before the averaging.

unsigned int* `PololuQTRSensors::calibratedMinimumOn`

unsigned int* `qtr_calibrated_minimum_on`()

The calibrated minimum values measured for each sensor, with emitters on. The pointers are unallocated and set to 0 until `calibrate()` is called, and then allocated to exactly the size required. Depending on the *readMode* argument to `calibrate()`, only the On or Off values may be allocated, as required. This and the following variables are made public so that you can use them for your own calculations and do things like saving the values to EEPROM, performing sanity checking, etc. The calibration values are available through function calls from C.

unsigned int* `PololuQTRSensors::calibratedMaximumOn`

unsigned int* `qtr_calibrated_maximum_on`()

The calibrated maximum values measured for each sensor, with emitters on.

unsigned int* `PololuQTRSensors::calibratedMinimumOff`

unsigned int* `qtr_calibrated_minimum_off`()

The calibrated minimum values measured for each sensor, with emitters off.

unsigned int* `PololuQTRSensors::calibratedMaximumOff`

unsigned int* `qtr_calibrated_maximum_off`()

The calibrated maximum values measured for each sensor, with emitters off.

`PololuQTRSensors::~~PololuQTRSensors`()

The destructor for the `PololuQTRSensors` class frees up memory allocated for the calibration arrays. This feature is not available in C.

`PololuQTRSensorsRC::PololuQTRSensorsRC`()

This constructor performs no initialization. If it is used, the user must call **init()** before using the methods in this class.

PololuQTRSensorsRC::PololuQTRSensorsRC(unsigned char* pins, unsigned char numSensors, unsigned int timeout = 4000, unsigned char emitterPin = 255);

This constructor just calls **init()**, below.

```
void PololuQTRSensorsRC::init(unsigned char* pins, unsigned char numSensors, unsigned int timeout = 4000, unsigned char emitterPin = 255)
void qtr_rc_init(unsigned char* pins, unsigned char numSensors, unsigned int timeout, unsigned char emitterPin)
```

Initializes a QTR-RC (digital) sensor array.

The array *pins* contains the (Arduino) pin numbers for each sensor. For example, if *pins* is {3, 6, 15}, sensor 0 is on digital pin 3 or PD3, sensor 1 is on digital pin 6 or PD6, and sensor 2 is on digital pin 15 or PC1 (Arduino analog input 1). Digital pins 0 – 7 correspond to port D pins PD0 – PD7, respectively. Digital pins 8 – 13 correspond to port B pins PB0 – PB5. Digital pins 14 – 19 correspond to port C pins PC0 – PC5, which are referred to in the Arduino environment as analog inputs 0 – 5.

numSensors specifies the length of the ‘pins’ array (the number of QTR-RC sensors you are using). *numSensors* must be no greater than 8.

timeout specifies the length of time in Timer2 counts beyond which you consider the sensor reading completely black. That is to say, if the pulse length for a pin exceeds *timeout*, pulse timing will stop and the reading for that pin will be considered full black. It is recommended that you set *timeout* to be between 1000 and 3000 us, depending on factors like the height of your sensors and ambient lighting. This allows you to shorten the duration of a sensor-reading cycle while maintaining useful measurements of reflectance. On a 16 MHz microcontroller, you can convert Timer2 counts to microseconds by dividing by 2 (2000 us = 4000 Timer2 counts = *timeout* of 4000). On a 20 MHz microcontroller, you can convert Timer2 counts to microseconds by dividing by 2.5 or multiplying by 0.4 (2000 us = 5000 Timer2 counts = *timeout* of 5000).

emitterPin is the Arduino digital pin that controls whether the IR LEDs are on or off. This pin is optional and only exists on the 8A and 8RC QTR sensor arrays. If a valid pin is specified, the emitters will only be turned on during a reading. If an invalid pin is specified (e.g. 255), the IR emitters will always be on.

PololuQTRSensorsAnalog::PololuQTRSensorsAnalog()

This constructor performs no initialization. If this constructor is used, the user must call **init()** before using the methods in this class.

PololuQTRSensorsAnalog::PololuQTRSensorsAnalog(unsigned char* analogPins, unsigned char numSensors, unsigned char numSamplesPerSensor = 4, unsigned char emitterPin = 255)

This constructor just calls **init()**, below.

```
void OrangutanAnalog::init(unsigned char* analogPins, unsigned char numSensors, unsigned char numSamplesPerSensor = 4, unsigned char emitterPin = 255)
void qtr_analog_init(unsigned char* analogPins, unsigned char numSensors, unsigned char numSamplesPerSensor, unsigned char emitterPin)
```

Initializes a QTR-A (analog) sensor array.

The array *pins* contains the analog pin assignment for each sensor. For example, if *pins* is {0, 1, 7}, sensor 1 is on analog input 0, sensor 2 is on analog input 1, and sensor 3 is on analog input 7. The ATmega168 has 8 total

analog input channels (ADC0 – ADC7) that correspond to port C pins PC0 – PC5 and dedicated analog inputs ADC6 and ADC7.

numSensors specifies the length of the *analogPins* array (the number of QTR-A sensors you are using). *numSensors* must be no greater than 8.

numSamplesPerSensor indicates the number of 10-bit analog samples to average per channel (per sensor) for each reading. The total number of analog-to-digital conversions performed will be equal to *numSensors* times *numSamplesPerSensor*. Increasing this parameter increases noise suppression at the cost of sample rate. Recommended value: 4.

emitterPin is the digital pin (see **qtr_rc_init()**, above) that controls whether the IR LEDs are on or off. This pin is optional and only exists on the 8A and 8RC QTR sensor arrays. If a valid pin is specified, the emitters will only be turned on during a reading. If an invalid pin is specified (e.g. 255), the IR emitters will always be on.

12. 3pi Robot Functions

This section of the library provides convenient access for 3pi-specific hardware. Currently, it only provides access for the 5 QTR-based line sensors that are included in the 3pi. That is, the QTR functions described in **Section 11** do not need to be used for the 3pi. The functions described below are enabled by including one of the 3pi files:

```
#include <pololu/3pi.h>           // use this line for C
#include <pololu/Pololu3pi.h>     // use this line for C++
```

The necessary Orangutan include files will be included automatically.

Using this library will automatically configure Timer2, which will cause it to conflict with other libraries that use Timer2. See **Section 7** (Motors) and **Section 11** (Sensors) for more information.

For a higher level overview of this library and programs that show how this library can be used, please see the **Pololu 3pi Robot User's Guide** [<http://www.pololu.com/docs/0J21>].

static unsigned char Pololu3pi::init(unsigned int line_sensor_timeout = 1000)

unsigned char pololu_3pi_init(unsigned int line_sensor_timeout)

Initializes the 3pi robot. This sets up the line sensors, turns the IR emitters off to save power, and resets the system timer (except within the Arduino environment). The parameter *line_sensor_timeout* specifies the timeout in Timer2 counts. This number should be the length of time in Timer2 counts beyond which you consider the sensor reading completely black. It is recommended that you set timeout to be between 500 and 3000 us, depending on factors like the ambient lighting. This allows you to shorten the duration of a sensor-reading cycle while maintaining useful measurements of reflectance. For the 3pi, you can convert Timer2 counts to microseconds by dividing by 2.5 or multiplying by 0.4 (2000 us = 5000 Timer2 counts = *line_sensor_timeout* of 5000).

void Pololu3pi::read(unsigned int *sensorValues, unsigned char readMode = IR_EMITTERS_ON)

void read_line_sensors(unsigned int *sensorValues, unsigned char readMode)

Reads the raw sensor values into an array. There **MUST** be space for five unsigned int values in the array. The values returned are a measure of the reflectance, between 0 and the *line_sensor_timeout* argument provided in to the init() function.

The functions that read values from the sensors all take an argument *readMode*, which specifies the kind of read that will be performed. Several options are defined: **IR_EMITTERS_OFF** specifies that the reading should be made without turning on the infrared (IR) emitters, in which case the reading represents ambient light levels near the sensor; **IR_EMITTERS_ON** specifies that the emitters should be turned on for the reading, which results in a measure of reflectance; and **IR_EMITTERS_ON_AND_OFF** specifies that a reading should be made in both the on and off states. The values returned when the **IR_EMITTERS_ON_AND_OFF** option is used are given by **on + max – off**, where **on** is the reading with the emitters on, **off** is the reading with the emitters off, and **max** is the maximum sensor reading. This option can reduce the amount of interference from uneven ambient lighting. Note that emitter control will only work if you specify a valid emitter pin in the constructor.

Example usage:

```
unsigned int sensor_values[5];
read_line_sensors(sensor_values);
```

void Pololu3pi::emittersOn()

void emitters_on()

Turn the IR LEDs on. This is mainly for use by **read_line_sensors()**, and calling this function before or after the reading the sensors will have no effect on the readings, but you may wish to use it for testing purposes.

void Pololu3pi::emittersOff()

void emitters_off()

Turn the IR LEDs off. This is mainly for use by `read_line_sensors()`, and calling this function before or after the reading the sensors will have no effect on the readings, but you may wish to use it for testing purposes.

void Pololu3pi::calibrate(unsigned char readMode = IR_EMITTERS_ON)

void calibrate_line_sensors(unsigned char readMode)

Reads the sensors for calibration. The sensor values are not returned; instead, the maximum and minimum values found over time are stored internally and used for the `readLineSensorsCalibrated()` method.

void Pololu3pi::readLineSensorsCalibrated(unsigned int *sensorValues, unsigned char readMode = IR_EMITTERS_ON)

void read_line_sensors_calibrated(unsigned int *sensorValues, unsigned char readMode)

Returns sensor readings calibrated to a value between 0 and 1000, where 0 corresponds to a reading that is less than or equal to the minimum value read by `calibrate()` and 1000 corresponds to a reading that is greater than or equal to the maximum value. Calibration values are stored separately for each sensor, so that differences in the sensors are accounted for automatically.

unsigned int Pololu3pi::readLine(unsigned int *sensorValues, unsigned char readMode = IR_EMITTERS_ON, unsigned char whiteLine = 0)

void read_line(unsigned int *sensorValues, unsigned char readMode)

void read_line_white(unsigned int *sensorValues, unsigned char readMode)

Operates the same as `read calibrated`, but with a feature designed for line following: this function returns an estimated position of the line. The estimate is made using a weighted average of the sensor indices multiplied by 1000, so that a return value of 0 indicates that the line is directly below sensor 0, a return value of 1000 indicates that the line is directly below sensor 1, 2000 indicates that it's below sensor 2000, etc. Intermediate values indicate that the line is between two sensors. The formula is:

$$\frac{0 * \text{value0} + 1000 * \text{value1} + 2000 * \text{value2} + \dots}{\text{value0} + \text{value1} + \text{value2} + \dots}$$

As long as your sensors aren't spaced too far apart relative to the line, this returned value will be monotonic, which makes it great for use in closed-loop PID control. Additionally, this method remembers where it last saw the line, so if you ever lose the line to the left or the right, it's line position will continue to indicate the direction you need to go to reacquire the line. For example, since sensor 4 is your rightmost sensor, if you end up completely off the line to the left, this function will continue to return 4000.

By default, this function assumes a dark line (high values) surrounded by white (low values). If your line is light on black, set the optional second argument `whiteLine` to true or call `read_line_white()`. In this case, each sensor value will be replaced by the maximum possible value minus its actual value before the averaging.

unsigned int* Pololu3pi::getLineSensorsCalibratedMinimumOn()

unsigned int* get_line_sensors_calibrated_minimum_on()

The calibrated minimum values measured for each sensor, with emitters on. The pointers are unallocated and set to 0 until `calibrate()` is called, and then allocated to exactly the size required. Depending on the `readMode` argument to `calibrate()`, only the On or Off values may be allocated, as required. You can use them for your own calculations and do things like saving the values to EEPROM, performing sanity checking, etc.

unsigned int* PololuQTRSensors::getLineSensorsCalibratedMaximumOn()

unsigned int* get_line_sensors_calibrated_maximum_on()

The calibrated maximum values measured for each sensor, with emitters on.

unsigned int* PololuQTRSensors::getLineSensorsCalibratedMinimumOff()

unsigned int* `get_line_sensors_calibrated_minimum_off()`

The calibrated minimum values measured for each sensor, with emitters off.

unsigned int* `PololuQTRSensors::getLineSensorsCalibratedMaximumOff()`

unsigned int* `get_line_sensors_calibrated_maximum_off()`

The calibrated maximum values measured for each sensor, with emitters off.

13. Wheel Encoders

The `PololuWheelEncoders` class and the associated C functions provide an easy interface for using the **Pololu Wheel Encoders** [<http://www.pololu.com/catalog/product/1217>], which allow a robot to know exactly how far its motors have turned at any point in time.

This section of the library makes uses of pin-change interrupts to quickly detect and record each transition on the encoder.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 6.k** of the **Pololu AVR C/C++ Library User's Guide** [<http://www.pololu.com/docs/0J20>].

Reference

C++ and Arduino methods are shown in red.

C functions are shown in green.

static void PololuWheelEncoders::init(unsigned char *m1a*, unsigned char *m1b*, unsigned char *m2a*, unsigned char *m2b*)

void encoders_init(unsigned char *m1a*, unsigned char *m1b*, unsigned char *m2a*, unsigned char *m2b*)

Initializes the wheel encoders. The four arguments are the four pins that the wheel encoders are connected to, according to the Arduino numbering: Arduino digital pins 0 – 7 correspond to port D pins PD0 – PD7, respectively. Arduino digital pins 8 – 13 correspond to port B pins PB0 – PB5. Arduino analog inputs 0 – 5 are referred to as digital pins 14 – 19 (these are the enumerations you should use for this library) and correspond to port C pins PC0 – PC5. The arguments are named *m1a*, *m1b*, etc. with the intention that when motor M1 is spinning forward, pin *m1a* will change before pin *m1b*. However, it is difficult to get them all correct on the first try, and you might have to experiment.

static int getCountsM1()

static int getCountsM2()

int encoders_get_counts_m1()

int encoders_get_counts_m2()

Returns the number of counts measured on M1 or M2. For the Pololu wheel encoders, the resolution is about 3mm/count, so this allows a maximum distance of $32767 \times 3\text{mm}$ or about 100m. For longer distances, you will need to occasionally reset the counts using the functions below.

static int getCountsAndResetM1()

static int getCountsAndResetM2()

int encoders_get_counts_and_reset_m1()

int encoders_get_counts_and_reset_m2()

Returns the number of counts measured on M1 or M2, and resets the stored value to zero.

static unsigned char checkErrorM1()

static unsigned char checkErrorM2()

unsigned char encoders_check_error_m1()

unsigned char encoders_check_error_m2()

These functions check whether there has been an error on M1 or M2; that is, if both *m1a*/*m2b* or *m2a*/*m2b* changed simultaneously. They return 1 if there was an error, then reset the error flag automatically.