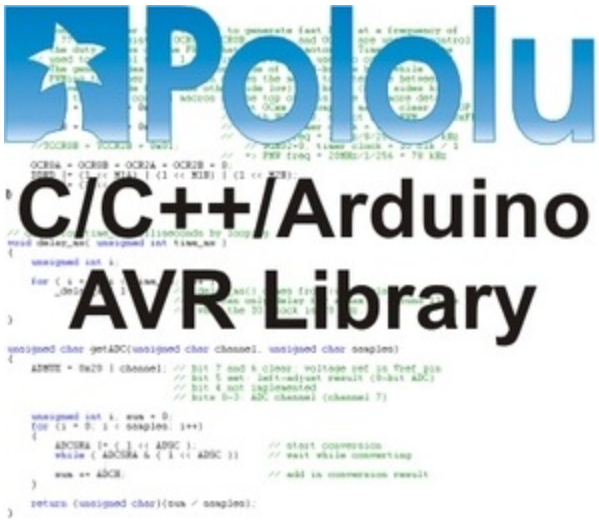
Pololu AVR Library Command Reference



1. Introduction

This document describes the **Pololu AVR C/C++ Library**, a programming library designed for use with Pololu products. The library is used to create programs that run on Atmel ATmega1284P, ATmega644P, ATmega324PA, ATmega328P, ATmega168 and ATmega48 processors, and it supports the following products:

Pololu 3pi robot [http://www.pololu.com/product/975]: a mega168/328-based programmable robot. The 3pi robot essentially contains an SV-328 and a 5-sensor version of the QTR-8RC, both of which are in the list below.

Pololu Orangutan SVP-1284 [http://www.pololu.com/product/1327]: based on the mega1284, the SVP-1284 is our newest Orangutan robot controller. It is a super-sized version of the SV-328, with a built-in AVR ISP programmer, more I/O lines, more regulated power, and more memory. It also features hardware that makes it easy to control up to eight servos with a single hardware PWM and almost no processor overhead.

Pololu Orangutan SVP-324 [http://www.pololu.com/product/1325]: based on the mega324, the SVP-324 is a version of the SVP-1284 with less memory. The two versions are completely code-compatible (the same code will run on both devices as long as it's small enough to fit on the ATmega324PA).

Pololu Orangutan X2 [http://www.pololu.com/product/738]: based on the mega1284, the X2 robot controller is the most powerful Orangutan. It features an auxiliary microcontroller devoted to controlling much of the integrated hardware (it also acts as a built-in AVR ISP programmer) and high-power motor drivers capable of delivering hundreds of watts.

Pololu Orangutan SV-328 [http://www.pololu.com/product/1227]: a full-featured, mega328-based robot controller that includes an LCD display. The SV-328 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. This library also supports the original **Orangutan SV-168 [http://www.pololu.com/product/1225]**, which was replaced by the SV-328.

Pololu Orangutan LV-168 [http://www.pololu.com/product/775]: 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 [http://www.pololu.com/product/1215]: 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-328 [http://www.pololu.com/product/1220]: a mega328 version of the above. The mega328 offers more memory for your programs (32 KB flash, 2 KB RAM). This library also supports the **Baby Orangutan B-168 [http://www.pololu.com/product/1216]**, which was replaced by the Baby B-328.

Pololu QTR-1A [http://www.pololu.com/product/958] and QTR-8A [http://www.pololu.com/product/960] 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 [http://www.pololu.com/product/959] and QTR-8RC [http://www.pololu.com/product/961] 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 [http://www.pololu.com/product/1217]: a wheel encoder solution that allows a robot to measure how far it has traveled.

The library is written in C++ with C wrapper functions of all of the methods and may be used in three different programming environments:

- **C++:** supported by the AVR-GCC/WinAVR project. See the **Pololu AVR Programming Quick Start Guide [http://www.pololu.com/docs/0J51]** to get started.

- **C / Atmel 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 Studio [http://www.atmel.com/microsite/atmel_studio6/]**. See the **Pololu AVR Programming Quick Start Guide [http://www.pololu.com/docs/0J51]** to get started.

- **Arduino [http://www.arduino.cc]:** a popular, beginner-friendly programming environment for the ATmega168/328, 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.

For more information, see the **Pololu AVR C/C++ Library User's Guide [http://www.pololu.com/docs/0J20]**.

2. Orangutan Analog-to-Digital Conversion

The OrangutanAnalog class and the C functions in this section make it easy to use to the analog inputs and integrated analog hardware on the Orangutan robot controllers (LV, SV, SVP, X2, and Baby Orangutan) and 3pi robot. These functions take care of configuring and running the analog-to-digital converter.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 3.a** of the **Pololu AVR C/C++ Library User's Guide [http://www.pololu.com/docs/0J20]**.

Channels

The tables below give a summary of the analog inputs available on your AVR. Some of these pins are hardwired or jumpered to the battery voltage or sensors or trimpots built in to your device, while other pins are available for you to connect your own sensors to. Please refer to the pin assignment table in the user's guide for your device for more information.

Analog Channels on the 3pi robot, SV, LV, and Baby Orangutan

Channel	Pin	Keyword	Note
0	PC0/ADC0		
1	PC1/ADC1		
2	PC2/ADC2		
3	PC3/ADC3		
4	PC4/ADC4		
5	PC5/ADC5		
6	ADC6	TEMP_SENSOR	LV: External temperature sensor SV: 1/3 VIN 3pi: 2/3 VIN
7	ADC7	TRIMPOT	User trimmer potentiometer
14	internal 1.1V		bandgap voltage (slow to stablize)
15	internal 0V		GND

Single-Ended Analog Channels on the Orangutan SVP

Channel	Pin	Keyword	Note
0	PA0/ADC0		
1	PA1/ADC1		
2	PA2/ADC2		
3	PA3/ADC3		
4	PA4/ADC4		
5	PA5/ADC5		
6	PA6/ADC6		
7	PA7/ADC7		
14	internal 1.1V		bandgap voltage (slow to stablize)
15	internal 0V		GND
128	ADC/$\overline{SS}$	TRIMPOT	Measured by auxiliary processor*
129	A	CHANNEL_A	Measured by auxiliary processor*
130	B	CHANNEL_B	Measured by auxiliary processor*
131	C	CHANNEL_C	Measured by auxiliary processor*
132	D/RX	CHANNEL_D	Measured by auxiliary processor*

*Reading the analog channels on the Orangutan SVP that are measured by the auxiliary processor requires the auxiliary processor to be in the correct mode. See the documentation for the Orangutan SVP **setMode** command in **Section 13** for more information.

Single-Ended Analog Channels on the Orangutan X2

Channel	Pin	Keyword	Note
0	PA0/ADC0		
1	PA1/ADC1		
2	PA2/ADC2		
3	PA3/ADC3		
4	PA4/ADC4		
5	PA5/ADC5		
6	PA6/ADC6		SMT jumper to 5/16 VIN*
7	PA7/ADC7	TRIMPOT	SMT jumper to trimpot*
14	internal 1.1V		bandgap voltage (slow to stablize)
15	internal 0V		GND

*Pins PA6 and PA7 on the Orangutan X2 are connected by default through SMT jumpers to the a battery voltage divider circuit and the user trimpot, respectively. You should not make external connections to these pins without first breaking the default SMT jumper connections, which are located on the underside of the main board and labeled in the silkscreen "ADC6=BATLEV" and "ADC7=TRIMPOT".

Differential Analog Channels on the Orangutan SVP and X2

Channel	Positive Differential Input	Negative Differential Input	Gain
9	PA1/ADC1	PA0/ADC0	10x
11	PA1/ADC1	PA0/ADC0	200x
13	PA3/ADC3	PA2/ADC2	10x
15	PA3/ADC3	PA2/ADC2	200x
16	PA0/ADC0	PA1/ADC1	1x
18	PA2/ADC2	PA1/ADC1	1x
19	PA3/ADC3	PA1/ADC1	1x
20	PA4/ADC4	PA1/ADC1	1x
21	PA5/ADC5	PA1/ADC1	1x
22	PA6/ADC6	PA1/ADC1	1x
23	PA7/ADC7	PA1/ADC1	1x
24	PA0/ADC0	PA2/ADC2	1x
25	PA1/ADC1	PA2/ADC2	1x
27	PA3/ADC3	PA2/ADC2	1x
28	PA4/ADC4	PA2/ADC2	1x
29	PA5/ADC5	PA2/ADC2	1x

Function 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 *mode* argument should be the either MODE_8_BIT or MODE_10_BIT. When the ADC is in 8-bit mode, conversion results will range from 0 to 255 for voltages ranging from 0 to 5 V. When the ADC is in 10-bit mode, conversion results will range from 0 to 1023 for voltages ranging from 0 to 5 V. The default mode setting is MODE_10_BIT.

Example:

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


// run the ADC in 10-bit conversion mode
set_analog_mode(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 to 255 for voltages from 0 to 5 V. In 10-bit mode, the result will range from 0 to 1023 for voltages from 0 to 5 V. The *channel* argument should be a channel number or keyword from the appropriate **table above**. This function will occupy program execution until the conversion is complete (approximately 100 us). This function does not alter the I/O state of the analog pin that corresponds to the specified ADC channel.

static unsigned int OrangutanAnalog::readMillivolts(unsigned char *channel*)
unsigned int analog_read_millivolts(unsigned char *channel*)

This function is just like analog_read() except the result is returned in millivolts. A return value of 5000 indicates a voltage 5 V. In most cases, this function is equivalent to `to_millivolts(analog_read(channel))`. However, on the Orangutan SVP, the channels measured by the auxiliary processor are reported to the AVR in millivolts, so calling analog_read_millivolts is more efficient for those channels.

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 to 255 for voltages from 0 to 5 . In 10-bit mode, the result will range from 0 to 1023 for voltages from 0 to 5 V. The *channel* argument should be a channel number or keyword from the appropriate **table above**. This function will occupy program execution until all of the requested conversions are complete (approximately 100 us per sample). This function does not alter the I/O state of the analog pin that corresponds to the specified ADC channel. On the Orangutan SVP, the channels measured by the auxiliary processor are averaged on the auxiliary processor, and the library does not support further averaging. For those channels, this function is equivalent to analog_read.

static unsigned int OrangutanAnalog::readAverageMillivolts(unsigned char *channel*, unsigned int *numSamples*)
unsigned int analog_read_average_millivolts(unsigned char *channel*, unsigned int *numSamples*)

This function is just like analog_read_average() except the result is returned in millivolts. A return value of 5000 indicates a voltage of 5 V. This function is equivalent to `to_millivolts(analog_read_average(channel, numSamples))`.

static unsigned int OrangutanAnalog::readVCCMillivolts()
unsigned int read_vcc_millivolts()

Performs ten 10-bit analog-to-digital conversions on the fixed internal 1.1V bandgap voltage and computes the analog reference voltage, VCC, from the results. On the Orangutans and 3pi, VCC is regulated to a steady 5 V, but VCC will fall below 5 V on all devices except the Orangutan LV if VIN drops to around 5 V or lower. On the LV, VCC will rise above 5 V if VIN exceeds 5 V. This function can be used to monitor VCC and catch when it is lower or higher than it should be. It can also be used as an argument to the set_millivolt_calibration() function to ensure that millivolt conversions of analog voltage readings stay accurate when the ADC reference voltage, VCC, cannot be regulated to 5 V. Note that this function performs 10-bit conversions regardless of the analog mode, but it restores the analog mode to its previous state before the function returns.

static void OrangutanAnalog::setMillivoltCalibration(unsigned int *referenceMillivolts*)
void set_millivolt_calibration(unsigned int *referenceMillivolts*)

This function updates the value of the reference voltage used by the to_millivolts() function (and all other functions that return conversion results in millivolts) to be *referenceMillivolts*. The default callibration value is 5000 mV, which is accurate when VCC is 5 V, as it is during normal operation for all devices. If VCC is not 5 V and the analog reference voltage value is not adjusted to the new VCC, however, millivolt conversions of analog readings performed by this library will be inaccurate. An easy way to ensure that your conversions remain accurate even when VCC falls below 5 V is to call `set_millivolt_calibration(read_vcc_millivolts())` in your main loop.

static unsigned int OrangutanAnalog::readTrimpot()
unsigned int read_trimpot()

Performs 20 analog-to-digital conversions on the output of the trimmer potentiometer on the Orangutan (including the SVP) or 3pi robot. In 8-bit mode, the result will range from 0 to 255 for voltages from 0 to 5 . In 10-bit mode, the result will range from 0 to 1023 for voltages from 0 to 5 V. This method is equivalent to `readAverage(TRIMPOT, 20)`.

static unsigned int OrangutanAnalog::readTrimpotMillivolts()
unsigned int read_trimpot_millivolts()

This function is just like read_trimpot() except the result is returned in millivolts. On the Orangutan SVP, this function is more efficient than `to_millivolts(read_trimpot())`. On all other devices the two are equivalent.

static int readBatteryMillivolts_3pi()
int read_battery_millivolts_3pi()

Performs ten 10-bit analog-to-digital conversions on the battery voltage sensing circuit (a 2/3 voltage divider of VIN) of the 3pi and returns the average measured battery voltage 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. This function will only return the correct result when the ADC6 shorting block is in place on the front-left side of the robot. The 3pi ships with this shorting block in place, so this function will return the correct result by default. Note that this function performs 10-bit conversions regardless of the analog mode, but it restores the analog mode to its previous state before the function returns.

static int readBatteryMillivolts_SV()
int read_battery_millivolts_sv()

Performs ten 10-bit analog-to-digital conversions on the battery voltage sensing circuit (a 1/3 voltage divider of VIN) of the Orangutan SV and returns the average measured battery voltage in millivolts. A result of 9153 would mean a battery voltage of 9.153 V. This function will only return the correct result when there is a short across the

"ADC6=VBAT/3" SMT jumper on the bottom side of the PCB. The SV ships with this SMT jumper shorted with a solder bridge, so this function will return the correct result by default. Note that this function performs 10-bit conversions regardless of the analog mode, but it restores the analog mode to its previous state before the function returns.

static int readBatteryMillivolts_SVP()
int read_battery_millivolts_svp()

Returns the measured Orangutan SVP battery voltage in millivolts. The battery voltage on the SVP is measured by the auxiliary microcontroller, so this function just asks the auxiliary MCU for the most recent measurement; it does not use the ADC on the user MCU.

static int readBatteryMillivolts_X2()
int read_battery_millivolts_x2()

Performs ten 10-bit analog-to-digital conversions on the battery voltage sensing circuit (a 5/16 voltage divider of VIN) of the Orangutan X2 and returns the average measured battery voltage in millivolts. A result of 9153 would mean a battery voltage of 9.153 V. This function will only return the correct result when there is a short across the "ADC6=BATLEV" SMT jumper on the bottom side of the PCB. The X2 ships with this SMT jumper shorted with a solder bridge, so this function will return the correct result by default. Note that this function performs 10-bit conversions regardless of the analog mode, but it restores the analog mode to its previous state before the function returns.

static int OrangutanAnalog::readTemperatureF()
int read_temperature_f()

Performs 20 10-bit analog-to-digital conversions on the output of the temperature sensor on the Orangutan LV and returns average result in tenths of a degree Farenheit, so a result of 827 would mean a temperature of 82.7 degrees F. The temperature sensor is on analog input 6 on the Orangutan LV, so this method is equivalent to `readAverage(TEMP_SENSOR, 20)` converted to tenths of a degree F. This function will only return the correct result when there is a short across the "ADC6=TEMP" SMT jumper on the bottom side of the Orangutan LV PCB. The LV ships with this SMT jumper shorted with a solder bridge, so this function will return the correct result by default. Note that this function performs 10-bit conversions regardless of the analog mode, but it restores the analog mode to its previous state before the function returns. Only the Orangutan LV has an external temperature sensor, so this function will only return correct results when used on an Orangutan LV.

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. This function will only return correct results when used on an Orangutan LV, and it requires a short across the "ADC6=TEMP" SMT jumper (this short is present by default).

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 *channel* argument should be a channel number or keyword from the appropriate **table above**. The procedure is to start a conversion on an analog input with this method, then poll is_converting() in your main loop. Once is_converting() returns a zero, the result can be obtained through a call to analog_conversion_result() or analog_conversion_result_millivolts() and this method can be used to start a new conversion. This function does not alter the I/O state of the analog pin that corresponds to the specified ADC channel.

static unsigned char OrangutanAnalog::isConverting()
unsigned char analog_is_converting()

Returns a 1 if the ADC is in the middle of performing an analog-to-digital conversion, otherwise it returns a 0. The AVR is only capable of performing one 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 to 255 for voltages from 0 to 5 V. In 10-bit mode, the result will range from 0 to 1023 for voltages from 0 to 5 V.

static unsigned int conversionResultMillivolts()
unsigned int analog_conversion_result_millivolts()
This function is just like analog_conversion_result() except the result is returned in millivolts. A return value of 5000 indicates a voltage 5 V. In most cases, this function is equivalent to `to_millivolts(analog_conversion_result(channel))`. However, on the Orangutan SVP, the channels measured by the auxiliary processor are reported to the AVR in millivolts, so calling analog_conversion_result_millivolts is more efficient for those channels.

static unsigned int toMillivolts(unsigned int *adcResult*)
unsigned int to_millivolts(unsigned int *adc_result*)
Converts the result of an analog-to-digital conversion to millivolts. By default, this assumes an analog reference voltage of exactly 5000 mV. The analog reference voltage used by this function can be changed using the set_millivolt_calibration() function.

Example:

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


to_millivolts(analog_read(0));
// e.g. returns 5000 if analog input 0 is at 5 V
```

3. 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 LV, SV, SVP, X2, and 3pi robot, from simple beeps to complex tunes. The buzzer is controlled using one of the Timer 1 PWM outputs, so it will conflict with any other uses of Timer 1. Note durations are timed using a Timer 1 overflow interrupt (**TIMER1_OVF**), which will briefly interrupt execution of your main program at the frequency of the sound being played. In most cases, the interrupt-handling routine is very short (several microseconds). However, when playing a sequence of notes in PLAY_AUTOMATIC mode (the default mode) with the **play()** command, this interrupt takes much longer than normal (perhaps several hundred microseconds) every time it starts a new note. It is important to take this into account when writing timing-critical code.

Note that the Orangutan X2 uses its auxiliary microcontroller to control the buzzer. The firmware on this microcontroller duplicates the play_note() and play_frequency() functionality of this library, so it is possible to play music on the Orangutan X2 without using Timer 1 (see the **Section 15** for Orangutan X2 functions). The OrangutanBuzzer library functions call the appropriate functions in the OrangutanX2 library, so they produce the same results on the X2 as they do on the other Orangutans, and they use Timer 1 to achieve the proper note durations.

This library can be used with the Baby Orangutan if one pin of an **external buzzer** **[http://www.pololu.com/product/1261]** is connected to pin PB2 and the other pin is connected to ground.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 3.b** of the **Pololu AVR C/C++ Library User's Guide** **[http://www.pololu.com/docs/0J20]**.

Note: The OrangutanServos and OrangutanBuzzer libraries both use Timer 1, so they cannot be used together simultaneously. It is possible to alternate use of OrangutanBuzzer and OrangutanServos routines, however. The `servos-and-buzzer` example program shows how this can be done and also provides functions for playing notes on the buzzer without using Timer 1 or the OrangutanBuzzer functions.

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). The *frequency* argument 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. The *volume* argument controls the buzzer volume, with 15 being the loudest and 0 being the quietest. A *volume* of 15 supplies the buzzer with a 50% duty cycle PWM at the specified *frequency*. Lowering *volume* by one halves the duty cycle (so 14 gives a 25% duty cycle, 13 gives a 12.5% duty cycle, etc). The volume control is somewhat crude and should be thought of as a bonus feature.

This function plays the note in the background while your program continues to execute. If you call another buzzer function while the note is playing, the new function call will overwrite the previous and take control of the buzzer. If you want to string notes together, you should either use the play() function or put an appropriate delay after you start a note playing. You can use the is_playing() function to figure out when the buzzer is through playing its note or melody.

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);


// play a 6 kHz note for 250 ms at half volume
play_frequency(6000, 250, 7);

// wait for buzzer to finish playing the note
while (is_playing());

// play a 44.5 Hz note for 1 s at full volume
play_frequency(DIV_BY_10 | 445, 1000, 15);
```

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). The *note* argument is an enumeration for the notes of the equal tempered scale (ETS). See the Note Macros section below for more information. The *volume* argument controls the buzzer volume, with 15 being the loudest and 0 being the quietest. A *volume* of 15 supplies the buzzer with a 50% duty cycle PWM at the specified *frequency*. Lowering *volume* by one halves the duty cycle (so 14 gives a 25% duty cycle, 13 gives a 12.5% duty cycle, etc). The volume control is somewhat crude and should be thought of as a bonus feature.

This function plays the note in the background while your program continues to execute. If you call another buzzer function while the note is playing, the new function call will overwrite the previous and take control of the buzzer. If you want to string notes together, you should either use the play() function or put an appropriate delay after you start a note playing. You can use the is_playing() function to figure out when the buzzer is through playing its note or melody.

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).

This function plays the string of notes in the background while your program continues to execute. If you call another buzzer function while the melody is playing, the new function call will overwrite the previous and take control of the buzzer. If you want to string melodies together, you should put an appropriate delay after you start a melody playing. You can use the is_playing() function to figure out when the buzzer is through playing the melody.

Examples:

```
// play a C major scale up and back down:
OrangutanBuzzer::play("!L16 V8 cdefgab>cbagfedc");
while (OrangutanBuzzer::isPlaying());
// the first few measures of Bach's fugue in D-minor
OrangutanBuzzer::play("!T240 L8 a gafaeada c+adaeafa >aa>bac#ada c#adaeaf4");
```

```
// play a C major scale up and back down:
play("!L16 V8 cdefgab>cbagfedc");
while (is_playing());
// the first few measures of Bach's fugue in D-minor
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);
}
```

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

void someFunction()
{
  play_from_program_space(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 play_check() method. If *mode* is PLAY_AUTOMATIC, the sequence will play automatically in the background, driven by the Timer 1 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 play_check() 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 OrangutanBuzzer::isPlaying()
unsigned char is_playing()

This method returns 1 (true) if the buzzer is currently playing a note/frequency or if it is still playing a sequence started by play(). 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.

4. Orangutan Digital I/O

This section of the library provides commands for using the AVR's pins as generic digital inputs and outputs. The code is all inline, which lets it compile to very small, fast, efficient assembly code if you use constants as your arguments. For example, the line:

```
set_digital_output(IO_D3, HIGH);  // set pin PD3 as driving high output
//in C++: OrangutanDigital::setOutput(IO_D3, HIGH);
```

compiles to the assembly:

```
sbi 0x0b, 3  ;i.e. PORTD |= 1 << 3; (PD3 set as output)
sbi 0x0a, 3  ;i.e. DDRD  |= 1 << 3; (PD3 drives high)
```

If your function arguments are constants, you can use this library in place of raw digital I/O register manipulation without worrying about any significantly increased overhead or processing time. Using variables as function arguments can increase overhead and processing time, but the functions in this library allow for simpler programmatic approaches to working with digital I/O, since you no longer have to deal with a multitude of pin-specific registers.

The digital pins on the AVR default to high-impedance inputs after a power-up or reset.

For a high-level explanation of what the AVR's digital I/O pins can do, and example programs using this section of the library, see **Section 3.c** of the **Pololu AVR C/C++ Library User's Guide [http://www.pololu.com/docs/0J20]**.

The *pin* argument

All of the functions in this section of the library take a pin number as their first argument. On the ATmegaxx8-based Orangutans (LV, SV, and Baby) and 3pi robot, these pin numbers are consistent with the Arduino pin numbering system. However, the library defines keywords that you can use instead of remembering the numbers. The keywords have the form IO_*LN* where *L* is the port letter and *N* is the pin number. For example, the keyword IO_D1 refers to pin PD1.

This library also defines:

```
#define INPUT            0
#define OUTPUT           1
#define LOW              0
#define HIGH             1
#define TOGGLE           0xFF
#define HIGH_IMPEDANCE   0
#define PULL_UP_ENABLED  1
```

Reference

C++ and Arduino methods are shown in red.
C functions are shown in green.

static void OrangutanDigital::setOutput(unsigned char pin, unsigned char outputState);
void set_digital_output(unsigned char pin, unsigned char output_state);

Sets the specified pin as an output. The *pin* argument should be one of the IO_* keywords (e.g. IO_D1 for pin PD1). The *output_state* argument should either be LOW (to drive the line low), HIGH (to drive the line high), or TOGGLE (to toggle between high and low).

static void OrangutanDigital::setInput(unsigned char pin, unsigned char inputState);
void set_digital_input(unsigned char pin, unsigned char input_state);

Sets the specified pin as an input. The *pin* argument should be one of the IO_* keywords (e.g. IO_D1 for pin PD1). The *input_state* argument should either be HIGH_IMPEDANCE (to disable the pull-up resistor) or PULL_UP_ENABLED (to enable the pull-up resistor).

static unsigned char OrangutanDigital::isInputHigh(unsigned char pin);
unsigned char is_digital_input_high(unsigned char pin);

Reads the input value of the specified pin. The *pin* argument should be one of the IO_* keywords (e.g. IO_D1 for pin PD1). If the reading is low (0 V), this method will return 0. If the reading is high (5 V), it will return a non-zero number that depends on the pin number. This function returns the value of the pin regardless of whether it is configured as an input or an output. If you want the pin to be an input, you must first call set_digital_input() to make the pin an input.

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 robot controller (LV, SV, SVP, and X2) and 3pi robot, providing an essential tool for user interfaces and debugging. The library implements the standard 4-bit HD44780 protocol (except on the Orangutan X2, for which the 8-bit protocol is used), and it uses the busy-wait-flag feature to avoid the unnecessarily long delays present in other 4-bit LCD Arduino libraries at the time of this library's release. It is designed to gracefully handle alternate use of the 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 LED drivers on the 3pi and Orangutans.

For a list of the standard characters available on the LCD, see page 17 of the **HD44780 interface datasheet [http://www.pololu.com/file/download/HD44780.pdf?file_id=0J72]** (330k pdf).

For C and C++ users, the standard C function **printf()** from stdio.h 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 3.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 OrangutanLCD::clear()
void clear()
Clears the display and returns the cursor to the upper-left corner (0, 0).

static 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 program and data memory. This function is intended to work with the LCD that came with your Orangutan; if you are using a different LCD, you can use lcd_init_printf_with_dimensions() to initialize printf() for the width and height of your LCD. This function only needs to be called once in your program, prior to any printf() calls.

Example

```
#include <stdio.h>
void someFunction(int x)
{
  lcd_init_printf();
  // in C++: OrangutanLCD::initPrintf();
  printf("x=%5d", x);
}
```

static void OrangutanLCD::initPrintf(unsigned char *lcd_width*, unsigned char *lcd_height*)
void lcd_init_printf_with_dimensions(unsigned char *lcd_width*, unsigned char *lcd_height*)
Initializes the display for use with the standard C function **printf()** and lets you specify the width and height of your LCD using the *lcd_width* and *lcd_height* arguments. 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 program and data memory. If you are using the LCD that came with your Orangutan, you can use the argument-free lcd_init_printf(), which automatically initializes printf() for the width and height of your Orangutan's (or 3pi's) LCD. This version of the function is useful if you are using a different LCD or for some reason only want to use a portion of your LCD. This function only needs to be called oncce in your program, prior to any printf() calls.

Example

```
#include <stdio.h>
void someFunction(int x)
{
  // initialize printf() for a 20x4 character LCD
  lcd_init_printf_with_dimensions(20, 4);
  // in C++: OrangutanLCD::initPrintf(20, 4);
  printf("x=%5d", x);
}
```

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:

```
print_character('A');
// in C++: 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:

```
print("Hello!");
// in C++: 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 hello[] PROGMEM = "Hello ";

void someFunction()
{
  print_from_program_space(hello);
  // in C++: OrangutanLCD::printFromProgramSpace(hello);
  print_from_program_space(PSTR("there!"));
  // in C++: OrangutanLCD::printFromProgramSpace(PSTR("there!"));
}
```

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);
// in C: print_long(-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 **un**signed 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 **un**signed 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(unsigned char *cursorType*)
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
  lcd_load_custom_character(smile, 3);
  // in C++: OrangutanLCD::loadCustomCharacter(smile, 3);

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

  // display the character
  print_character(3);
  // in C++: OrangutanLCD::print((char)3);
}
```

6. Orangutan LEDs

The OrangutanLEDs class and the C functions in this section are a very simple interface to the user LEDs included on Orangutan controllers and 3pi. The Orangutan X2 has five user LEDs (two red, two green, and one yellow), the Orangutan LV, SV, SVP, and 3pi have two user LEDs (one red and one green), and the Baby Orangutan has one user LED (red).

On all devices except the Orangutan X2, the red LED is on the same pin as the UART0 serial transmitter (PD1), so if you are using UART0 for serial transmission then the red LED functions will not work, and you will see the red LED blink briefly whenever data is transmitted on UART0.

On all devices except the Baby Orangutan, the green LED is on the same pin as an LCD data pin, and the green LED will blink briefly whenever data is sent to the LCD, but the two functions will otherwise not interfere with each other. On the Orangutan X2, all the user LEDs are LCD data pins, so they will briefly flash whenever data is sent to the LCD, and the LCD can sometimes drive these LEDs if the AVR pins they are on are configured as inputs (their default configuration). To stop the LCD from keeping the LEDs lit, you can use the functions in this library to explicitly turn them off at the start of your program, which will make the AVR pins driving-low outputs.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 3.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 OrangutanLEDs::red(unsigned char *state*)
void red_led(unsigned char *state*)

This method will turn the red user LED off if *state* is zero, it will toggle the LED state if *state* is 255, else it will turn the red user LED on. You can use the keyword HIGH as an argument to turn the LED on, and you can use the keyword LOW as an argument to turn the LED off. You can use the keyword TOGGLE to toggle the LED state.

Example:

```
red_led(HIGH);
// in C++: OrangutanLEDs::red(HIGH);  // turn the red LED on
```

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, it will toggle the LED state if *state* is 255, else it will turn the green user LED on. You can use the keyword HIGH as an argument to turn the LED on, and you can use the keyword LOW as an argument to turn the LED off. You can use the keyword TOGGLE to toggle the LED state. The Baby Orangutan does not have a green user LED, so this function will just affect the output state of user I/O pin PD7.

Example:

```
green_led(LOW);
// in C++: OrangutanLEDs::green(LOW);  // turn the green LED off
```

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

For the Orangutan LV, SV, X2, Baby Orangutan, and the 3pi, this method is an alternate version of red(). The red LED is on the left side of most of these boards. For the Orangutan SVP, this method is an alternate version of green().

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

For the Orangutan LV, SV, X2, Baby Orangutan, and the 3pi, this method is an alternate version of green(). The green LED is on the right side of most of these boards. For the Orangutan SVP, this method is an alternate version of red().

static void OrangutanLEDs::red2(unsigned char *state*)
void red_led2(unsigned char *state*)

This method controls the Orangutan X2's second red user LED and is only defined for the Orangutan X2 version of the library. You can use the keyword HIGH as an argument to turn the LED on, and you can use the keyword LOW as an argument to turn the LED off. You can use the keyword TOGGLE to toggle the LED state.

static void OrangutanLEDs::green2(unsigned char *state*)
void green_led2(unsigned char *state*)

This method controls the Orangutan X2's second green user LED and is only defined for the Orangutan X2 version of the library. You can use the keyword HIGH as an argument to turn the LED on, and you can use the keyword LOW as an argument to turn the LED off. You can use the keyword TOGGLE to toggle the LED state.

static void OrangutanLEDs::yellow(unsigned char *state*)
void yellow_led(unsigned char *state*)

This method controls the Orangutan X2's yellow user LED and is only defined for the Orangutan X2 version of the library. You can use the keyword HIGH as an argument to turn the LED on, and you can use the keyword LOW as an argument to turn the LED off. You can use the keyword TOGGLE to toggle the LED state.

7. Orangutan Motor Control

The OrangutanMotors class and the C functions in this section provide PWM-based speed (and direction) control of the two motor channels on the Orangutan controllers and 3pi. The motor control functions rely on the hardware PWM outputs from Timer 0 and Timer 2 for all Orangutans (and the 3pi) except for the Orangutan SVP, which just uses Timer 2 PWM outputs and leaves Timer 0 free, and the Orangutan X2, which uses its auxiliary microcontroller to interface with the motor drivers and leaves both Timer 0 and Timer 2 free.

General Note about Timers: The functions in Pololu AVR library will conflict with code that tries to reconfigure the hardware timers it is relying upon, so if you want to use a timer for some custom task, you cannot use the portion of the library that relies on that timer. The Pololu AVR library only uses Timer 0 for motor PWM generation, and it is only used for this purpose on the Orangutan LV, SV, Baby Orangutan, and 3pi robot, so the Orangutan SVP and X2 can safely use Timer 0 for any custom task. Timer 1 is used by OrangutanBuzzer for sound generation on all devices, and it is used by OrangutanServos for servo pulse generation on all devices. Timer 2 is used for motor PWM generation on all devices except Orangutan X2, and it is used by OrangutanTime to run the system timer on all devices. Additionally, the Orangutan SVP-**1284** has a second 16-bit timer, Timer 3, that can safely be used for any custom task (the Pololu AVR library does not use Timer 3 for anything).

Unfortunately, the Arduino environment relies on Timer 0 for its millis() and delay() functions, and it uses Timer 0 in a way that would conflict with this library. To fix the problem, this library disables the Arduino environment's Timer 0 interrupt and enables a special Timer 2 interrupt when used in the Arduino environment that restores the functionality of millis() and delay() to normal. This interrupt is not included in the C and C++ versions of the library. When not used in the Arduino environment, the millis() and delay() functions come from the OrangutanTime class, which is driven by its own Timer 2 overflow interrupt.

Since the Orangutan X2 uses its auxiliary MCU to control the motor drivers, the OrangutanMotors functions in the X2 version of this library just call the appropriate motor commands from the OrangutanX2 class. The OrangutanX2 class has a number of additional functions for higher-level motor control (e.g. variable braking, setting the PWM frequency, acceleration, current-limiting, paralleling the outputs for control of a single, more powerful motor, and more). Please see **Section 15** 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 3.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 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. A *speed* of 0 results in full brake (full coast on the Orangutan X2) while a *speed* of 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 *m1Speed*, int *m2Speed*)
This method will set the speeds and directions of motors 1 and 2.

8. Orangutan Pulse/PWM Inputs

This section of the library makes it easy to measure pulse inputs such as hobby servo RC pulses, PWM frequency and duty cycle, discharge time of a capacitor (measured digitally), etc. For example, with these functions, it becomes simple to program your Orangutan to respond to multiple channels from an RC receiver or to read data from the multitude of sensors that use pulse outputs. The pulses can be connected to any digital input, and there is no limit to the number of pulse inputs you can have (beyond the obvious I/O line and RAM limitations; each pulse channel requires 17 bytes of RAM). These functions are not available within the Arduino environment, which has its own pulse-reading functions.

The pulse-measuring occurs in the background and is entirely interrupt driven, requiring no further action from you once you have initiated it by calling **pulse_in_start()**. Required memory is allocated using **malloc()** when pulse-measuring is initiated, which conserves RAM. The **pulse_in_stop()** function frees this dynamically allocated memory. Once measuring is started, you can use these functions to get information about the most recently received high and low pulse, as well as the current pulse in progress. The system tick counter from OrangutanTime is used to time the pulses, which means pulse width measurements have a resolution of 0.4 µs, and the upper limit on pulse width is approximately 28 minutes.

This section of the library uses the AVR's pin-change interrupts: **PCINT0**, **PCINT1**, and **PCINT2** (and **PCINT3** on the Orangutan SVP and X2). It will conflict with any other code that uses pin-change interrupts (e.g. OrangutanWheelEncoders).

Pulse Information Variables

The OrangutanPulseIn code uses an array of 17-byte PulseInputStruct structs to keek track of the pulse data; each element of the array represents one pulse input channel. Struct data is automatically updated by the pin-change interrupt service routine (ISR) as pulses are received.

```
#define LOW_PULSE   1
#define HIGH_PULSE  2
#define ANY_PULSE   3

struct PulseInputStruct
{
  volatile unsigned char* pinRegister;
  unsigned char bitmask;
  volatile unsigned long lastPCTime;
  volatile unsigned char inputState;
  volatile unsigned long lastHighPulse;
  volatile unsigned long lastLowPulse;
  volatile unsigned char newPulse;
};
```

The bottom five struct variables are declared "volatile" because their values are changed by the pin-changed interrupt service routine (ISR), so we require the compiler to load them from RAM every time they are referenced. If a variable is not declared "volatile", the compiler will eliminate what it decides are superfluous reads of a variable from RAM and instead work with that variable in local registers, which saves time and code space, but this can cause situations where you fail to detect when an ISR has changed that variable's value.

- ***pinRegister & bitmask*** **:** Used by the pin-change ISR to determine the input state of the channel. You should not use these two struct variables in your programs because they might be removed in future versions of the library.
- ***lastPCTime*** **:** The system tick counter value when the last state change happened on this channel. You can use `(get_ticks()-lastPCTime)` to figure out how long ago (in units of 0.4 µs) the last pulse ended.
- ***inputState*** **:** This byte has a value of 1 (HIGH) if the channel is high, else it has a value of 0 (LOW).

- ***lastHighPulse*** **:** The width of the last detected high pulse in system ticks (units of 0.4 µs). A high pulse is one that starts with a channel transition from low to high and ends with a channel transition from high to low.
- ***lastLowPulse*** **:** The width of the last detected low pulse in system ticks (units of 0.4 µs). A low pulse is one that starts with a channel transition from high to low and ends with a channel transition from low to high.
- ***newPulse*** **:** The bits of this byte are set whenever a new pulse occurs, which lets you tell when new data has arrived. The LOW_PULSE bit is set when a low pulse finishes, and the HIGH_PULSE bit is set when a high pulse finishes. The functions in this section clear the *newPulse* flags after returning them. Once set, the flags stay set until they are read. Masking *newPulse* with the keyword ANY_PULSE with the code `(newPulse&ANY_PULSE)` will be non-zero if either new-pulse bit flag is set.

Reference

C++ methods are shown in red.
C/C++ functions are shown in green.

static unsigned char OrangutanPulseIn::start(const unsigned char *pulse_pins*[], unsigned char *num_pins*)
unsigned char pulse_in_start(const unsigned char *pulse_pins*[], unsigned char *num_pins*)

Configures the AVR's pin-change interrupts to measure pulses on the specified pins. The *num_pins* parameter should be the length of the *pulse_pins* array. A nonzero return value indicates that the needed memory could not be allocated.

The *pulse_pins* parameter should be the RAM address of an array of AVR I/O pin numbers defined using the IO_* keywords provided by the library (e.g. IO_D1 for a pulse input on pin PD1). This function does not configure the *pulse_pins* as digital inputs, which makes it possible to measure pulses on pins being used as outputs. AVR I/O pins default to digital inputs with the internal pull-up disabled after a reset or power-up.

Example

```
// configure pins PD0 and PD1 as pulse input channels
pulse_in_start((unsigned char[]) {IO_D0, IO_D1}, 2);


// configure pins PD0 and PD1 as pulse input channels
OrangutanPulseIn::start((unsigned char[]) {IO_D0, IO_D1}, 2);
```

static void OrangutanPulseIn::stop()
void pulse_in_stop()

Disables all pin-change interrupts and frees up the memory that was dynamically allocated by the pulse_in_start() function. This can be useful if you no longer want state changes of your pulse-measuring channels to interrupt program execution.

static void OrangutanPulseIn::getPulseInfo(unsigned char *channel*, struct PulseInputStruct* *pulse_info*)
void get_pulse_info(unsigned char *channel*, struct PulseInputStruct* *pulse_info*)

This function uses the *pulse_info* pointer to return a snapshot of the pulse state for the specified channel, *channel*. After get_pulse_info() returns, *pulse_info* will point to a **copy** of the PulseInputStruct that is automatically maintained by the the pin-change ISR for the specified channel. Additionally, after the copy is made, this function clears the *newPulse* flags (both high pulse and low pulse) in the original, ISR-maintained PulseInputStruct. Since *pulse_info* is a copy, the pin-change ISR will never change the *pulse_info* data. Working with *pulse_info* is especially useful if you need to be sure that all of your puse data for a single channel came from the same instant in time, since pin-change interrupts are disabled while the ISR-maintained PulseInputStruct is being copied to *pulse_info*.

The argument *channel* should be a number from 0 to one less than the total number of channels used (*num_pins*-1); the channel acts as an index to the *pulse_pins* array supplied to the pulse_in_start() function.

See the "Pulse Information Variables" section at the top of this page for documentation of the members of the PulseInputStruct *pulse_info,* and see the `pulsein1` sample program for an example of how to use get_pulse_info() as the basis of your pulse-reading code.

Example

```
// configure pins PD0 and PD1 as pulse input channels
pulse_in_start((unsigned char[]) {IO_D0, IO_C3}, 2);

struct PulseInputStruct pulseInfo;
get_pulse_info(1, &pulseInfo);  // get pulse info for pin PC3
```

```
if (pulseInfo.newPulse & HIGH_PULSE)
  someFunction(pulseInfo.lastHighPulse);


// configure pins PD0 and PD1 as pulse input channels
OrangutanPulseIn::start((unsigned char[]) {IO_D0, IO_C3}, 2);

struct PulseInputStruct pulseInfo;
OrangutanPulseIn::getPulseInfo(1, &pulseInfo);  // get pulse info for pin PC3
if (pulseInfo.newPulse & HIGH_PULSE)
  someFunction(pulseInfo.lastHighPulse);
```

static unsigned char OrangutanPulseIn::newPulse(unsigned char *channel*)
unsigned char new_pulse(unsigned char *channel*)

This function returns the *newPulse* flags for the specified pulse input channel and then clears them, so subsequent new_pulse() calls will return zero until the next new pulse is received. The return value of this fuction will be 0 only if no new pulses have been received. It will be non-zero if a new high or low pulse has been received. You can check the type of new pulse by looking at the HIGH_PULSE and LOW_PULSE bits of the return value.

Example

```
// check for new pulses on pulse input channel 0:
unsigned char newPulse = new_pulse(0);
if (newPulse)  // if there has been a new pulse of any kind
  doSomething();
if (newPulse & HIGH_PULSE)  // if there has been a new high pulse
  doSomethingElse();


// check for new pulses on pulse input channel 0:
unsigned char newPulse = OrangutanPulseIn::newPulse(0);
if (newPulse)  // if there has been a new pulse of any kind
  doSomething();
if (newPulse & HIGH_PULSE)  // if there has been a new high pulse
  doSomethingElse();
```

static unsigned char OrangutanPulseIn::newHighPulse(unsigned char *channel*)
unsigned char new_high_pulse(unsigned char *channel*)

This function returns the *newPulse* flags if there is a new high pulse on the specified pulse input channel (i.e. the return value is non-zero), else it returns zero. It also clears the HIGH_PULSE bit of the *newPulse* flag byte, so subsequent new_high_pulse() calls will return zero until the next new high pulse is received. The LOW_PULSE bit is not changed by this function.

static unsigned char OrangutanPulseIn::newLowPulse(unsigned char *channel*)
unsigned char new_low_pulse(unsigned char *channel*)

This function returns the *newPulse* flags if there is a new low pulse on the specified pulse input channel (i.e. the return value is non-zero), else it returns zero. It also clears the LOW_PULSE bit of the *newPulse* flag byte, so subsequent new_low_pulse() calls will return zero until the next new low pulse is received. The HIGH_PULSE bit is not changed by this function.

static unsigned long OrangutanPulseIn::getLastHighPulse(unsigned char *channel*)
unsigned long get_last_high_pulse(unsigned char *channel*)

This function returns the length in system ticks (units of 0.4 µs) of the last complete high pulse received on the specified pulse input channel. It gives you (interrupt-safe) access to the *lastHighPulse* member of the ISR-maintained PulseInputStruct for the specified channel. A high pulse is one that starts with a channel transition from low to high and ends with a channel transition from high to low. Note that if the last high “pulse” was longer than 28.6 minutes, the value returned by this function will have overflowed and the result will be incorrect. You should disregard the first high pulse after the pulse input line has been in a steady-high state for more than 28. minutes.

static unsigned long OrangutanPulseIn::getLastLowPulse(unsigned char *channel*)
unsigned long get_last_low_pulse(unsigned char *channel*)

This function returns the length in system ticks (units of 0.4 µs) of the last complete low pulse received on the specified pulse input channel. It gives you (interrupt-safe) access to the *lastLowPulse* member of the ISR-maintained PulseInputStruct for the specified channel. A low pulse is one that starts with a channel transition from high to low and ends with a channel transition from low to high. Note that if the last low "pulse" was longer than 28.6 minutes, the value returned by this function will have overflowed and the result will be incorrect. You should disregard the first low pulse after the pulse input line has been in a steady-low state for more than 28. minutes.

static void OrangutanPulseIn::getCurrentState(unsigned char *channel*, unsigned long* pulse_width, unsigned char* *state*)
void get_current_pulse_state(unsigned char *channel*, unsigned long* pulse_width, unsigned char* *state*)

This function gives you information about what is currently happening on the specified pulse input channel by using the argument *pulse_width* to return the number of system ticks that have elapsed since the last complete pulse ended (in units of 0.4 µs) and by using the argument *state* to return whether the voltage on the input channel is currently HIGH (1) or LOW (0). The *pulse_width* and *state* arguments are pointers to integer types whose values are respectively set (in an ISR-safe way) based on the *lastPCTime* and *inputState* members of the ISR-maintained PulseInputStruct for the specified channel. Specifically, **pulse_width* is set to: `get_ticks()-pulseInfo[channel].lastPCTime`

Example

```
unsigned long pulse_width;
unsigned char state;
// get info for pulse input channel channel 0:
get_current_pulse_state(0, &pulse_width, &state);
if (pulse_width > 250000 && state == HIGH)
{
  // if more than 100 ms have passed since last pulse ended
  // and the pulse input channel is currently high
  doSomething();
}
```

```
unsigned long pulse_width;
unsigned char state;
// get info for pulse input channel channel 0:
OrangutanPulseIn::getCurrentState(0, &pulse_width, &state);
if (pulse_width > 250000 && state == HIGH)
{
  // if more than 100 ms have passed since last pulse ended
  // and the pulse input channel is currently high
  doSomething();
}
```

unsigned long OrangutanPulseIn::toMicroseconds(unsigned long pulse_width)
unsigned long pulse_to_microseconds(unsigned long pulse_width)

Converts the provided *pulse_width* argument from system ticks to microseconds (this is the same as multiplying *pulse_width* by 0.4 µs, but without using floating-point math) and returns the result. The same result could be achieved by calling `ticks_to_microseconds(pulse_width)`.

Example

```
// if last high pulse was longer than 1500 microseconds
if (pulse_to_microseconds(get_last_high_pulse(0)) > 1500)
  doSomething();
```

```
// if last high pulse was longer than 1500 microseconds
if (OrangutanPulseIn::toMicroseconds(OrangutanPulseIn::getLastHighPulse(0)) > 1500)
  doSomething();
```

9. Orangutan Pushbuttons

The OrangutanPushbuttons class and the C functions in this section make it easy to use to the three pushbuttons on the Orangutan LV, SV, SVP, X2, and 3pi robot as user-interface control inputs to your program.

For a higher level overview of this library and programs that show how this library can be used, please see **Section 3.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 unsigned char OrangutanPushbuttons::getSingleDebouncedPress(unsigned char *buttons*)
unsigned char get_single_debounced_button_press(unsigned char *buttons*)

This is a non-blocking function that makes it very easy to perform button-triggered activities from within your main loop. It uses a four-step finite-state machine to detect the transition of a button press and returns the value of the pressed button once such a transition is detected. It requires the button to be up for at least 15 ms and then down for at least 15 ms before it reports the press, at which point it resets and will not report another button until the next press is detected. This process takes care of button debouncing, so a bouncy button press will still only result in one reported press. This function should be called repeatedly (and often) in a loop with the same button mask argument *buttons* (i.e. do not call this function multiple times in the same loop with different button mask arguments).

The argument *buttons* can be a combination of the keywords TOP_BUTTON, MIDDLE_BUTTON, and BOTTOM_BUTTON (intended for use with the Orangutans) or BUTTON_A, BUTTON_B, and BUTTON_C (intended for use with the 3pi) separated by the bitwise OR operator | or the addition operator +. You can use the keyword ANY_BUTTON to wait for any of the three buttons to be pressed. The returned value is the ID of the button (or buttons) that was pressed. Calls to this function can be combined with calls to get_single_debounced_button_release() in the same loop.

The `pushbuttons2` example shows how this function can be used to trigger events in your main loop.

Example

```
while (1)  // main loop (loop forever)
{
  // put main loop code here

  // the following line immediately returns 0 unless a button has just been pressed
  unsigned char button = get_single_debounced_button_press(ANY_BUTTON);
  // C++: unsigned char button = OrangutanPushbuttons::getSingleDebouncedPress(ANY_BUTTON);

  if (button & TOP_BUTTON)  // if top button pressed
    function1();
  if (button & MIDDLE_BUTTON)  // if middle button pressed
    function2();
  if (button & BOTTOM_BUTTON)  // if bottom button pressed
    function3();
}
```

static unsigned char OrangutanPushbuttons::getSingleDebouncedRelease(unsigned char *buttons*)
unsigned char get_single_debounced_button_release(unsigned char *buttons*)

This is a non-blocking function that makes it very easy to perform button-triggered activities from within your main loop. It uses a four-step finite-state machine to detect the transition of a button release and returns the value of the released button once such a transition is detected. It requires the button to be down for at least 15 ms and then up for at least 15 ms before it reports the release, at which point it resets and will not report another button until the next release is detected. This process takes care of button debouncing, so a bouncy button release will still only result in one reported release. This function should be called repeatedly (and often) in a loop with the same button mask argument *buttons* (i.e. do not call this function multiple times in the same loop with different button mask arguments). The returned value is the ID of the button (or buttons) that was pressed. Calls to this function can be combined with calls to get_single_debounced_button_press() in the same loop.

The `pushbuttons2` example shows how this function can be used to trigger events in your main loop.

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

This function 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 keywords TOP_BUTTON, MIDDLE_BUTTON, and BOTTOM_BUTTON (intended for use with the Orangutans) or BUTTON_A, BUTTON_B, and BUTTON_C (intended for use with the 3pi) separated by the bitwise OR operator | or the addition operator +. You can use the keyword ANY_BUTTON to wait for any of the three buttons to be pressed. 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 = wait_for_button_press(TOP_BUTTON | BOTTOM_BUTTON);


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 function 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. Since the default state of the user buttons is up, you will typically not want to supply this function with the ANY_BUTTON argument, as execution will return immediately if any one of the three buttons is up. Rather, a common argument to this function is the return value of a function that detects a button press.

Example

```
unsigned char button = wait_for_button_press(ANY_BUTTON);
someFunction();  // do something as soon as button is pressed
wait_for_button_release(button);  // wait for pressed button to be released


unsigned char button = OrangutanPushbuttons::waitForPress(ANY_BUTTON);
someFunction();  // do something as soon as button is pressed
OrangutanPushbuttons::waitForRelease(button);  // wait for pressed button to be released
```

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

This function 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 function will returns all of the buttons specified by *buttons* that are currently pressed. For example, if you call `button_is_pressed(ANY_BUTTON)` 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 a single button or multiple buttons (see the wait_for_button_press() function above). This function returns the immediate button input state and hence standard debouncing precautions should be taken by the user.

10. Orangutan Serial Port Communication

The OrangutanSerial class and the C functions in this section provide access to the serial port(s) on the Orangutan controllers and 3pi robot., enabling two-way TTL-level communication with another microcontroller, a serial device, or (through a USB-serial adapter or RS-232 level converter) a personal computer.

The Baby Orangutan, Orangutan SV, Orangutan LV, and 3pi robot are based on the ATmega48/168/328 line of AVR processors, which have a single UART that communicates on pins PD0 (RXD) and PD1 (TXD). Since there is only one UART on these devices, you must **omit** the *port* argument when using the commands below.

The Orangutan SVP and Orangutan X2 are based on the AVR ATmega324/644/1284 line of AVR processors, which have two UARTs. Port UART0 uses pins PD0 (RXD0) and PD1 (TXD0). Port UART1 uses pins PD2 (RXD1) and PD3 (TXD1). The SVP and X2 also have a port called USB_COMM, which lets you connect your Orangutan directly to a computer to send and receive bytes over USB. When using this port, you must call **serial_check()** regularly because this port does not support interrupts. See the Orangutan SVP User's Guide for more information about using this port on the Orangutan SVP. Since there are multiple serial ports, you must **include** the *port* argument when using the commands below, and it must be either UART0, UART1, or USB_COMM.

When sending data on a UART, a **UDRE** interrupt vector is called by default after each byte is sent, allowing the library to automatically start sending the next byte from the send buffer. When receiving data, an **RX** interrupt vector is called by default 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 **serial_set_mode()** and **serial_check()** functions below.

These functions are not available within the Arduino environment, which has its own serial functions.

For a higher level overview of this library and programs that show how this library can be used, please see **Section 3.h** 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 char *port*,] unsigned long *baud*)
unsigned char serial_set_baud_rate([unsigned char *port*,] 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. This function is not needed for (and has no effect on) the Orangutan SVP's USB_COMM port. It is required for setting the baud for all UART ports and for the Orangutan X2's USB_COMM port.

static void OrangutanSerial::receive([unsigned char *port*,] char * *buffer*, unsigned char *size*)
void serial_receive([unsigned char *port*,] 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([unsigned char *port*,] char * *buffer*, unsigned char *size*, unsigned int *timeout_ms*)
char serial_receive_blocking([unsigned char *port*,] 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. Returns 1 if the timeout occurred before the buffer filled up. Returns 0 if the buffer has been filled up. This function is useful for simple programs and for situations in which you know exactly how many bytes to expect.

static void OrangutanSerial::receiveRing([unsigned char *port*,] char * *buffer*, unsigned char *size*)
void serial_receive_ring([unsigned char *port*,] 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, serial_get_received_bytes() will reset to zero, and data will continue to be inserted at the beginning of the buffer.

static void OrangutanSerial::cancelReceive([unsigned char *port*])
void serial_cancel_receive([unsigned char *port*)
Stops background serial reception.

static inline unsigned char OrangutanSerial::getReceivedBytes([unsigned char *port*])
unsigned char serial_get_received_bytes([unsigned char *port*])
Returns 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([unsigned char *port*])
char serial_receive_buffer_full([unsigned char *port*])
Returns 1 (true) when the receive buffer has been filled with received bytes, so that serial reception is halted. Returns 0 (false) otherwise. This function should not be called when receiving data into a ring buffer.

static void OrangutanSerial::send([unsigned char *port*,] char * *buffer*, unsigned char *size*)
void serial_send([unsigned char *port*,] char * *buffer*, unsigned char *size*)
Sets up a buffer for background transmission. Data from this buffer will be transmitted until *size* bytes have been sent. If serial_send() is called before serial_send_buffer_empty() returns true (when transmission of the last byte has started), the old buffer will be discarded and tramission will be cut short. This means that you should almost always wait until the data has been sent before calling this function again. See serial_send_blocking(), below, for an easy way to do this.

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

void serial_send_blocking([unsigned char *port*,] char * *buffer*, unsigned char *size*)

Same as serial_send(), but waits until transmission of the last byte has started before returning. When this function returns, it is safe to call serial_send() or serial_send_blocking() again.

Warning: When using the Orangutan SVP's **USB_COMM** port, serial_send_blocking() might never return because the rate at which bytes can be sent to the computer is dictated by how often the computer requests to read bytes from the Orangutan. If the Orangutan's USB port is not connected, or for some reason the computer has stopped reading bytes from the Orangutan's USB Communications port, then this function might never return. This is not a problem for UART-based serial ports or the Orangutan X2's **USB_COMM** port because the rate of transmission is dictated only by the AVR's code and the baud rate, so all the bytes will finish transmitting in a relatively predictable amount of time.

static inline unsigned char OrangutanSerial::getSentBytes([unsigned char *port*])
unsigned char serial_get_sent_bytes([unsigned char *port*])

Returns the number of bytes that have been sent since serial_send() was called.

static char OrangutanSerial::sendBufferEmpty([unsigned char *port*])
char serial_send_buffer_empty([unsigned char *port*])

Returns 1 (true) when the send buffer is empty; when there are no more bytes to send. Returns 0 (false) otherwise.

static void OrangutanSerial::setMode([unsigned char *port*,] unsigned char *mode*)
void serial_set_mode([unsigned char *port*,] unsigned char *mode*)

Sets the serial library to use either interrupts (with the argument SERIAL_AUTOMATIC) or polling (SERIAL_CHECK). If SERIAL_CHECK is selected, your code must call serial_check() often to ensure reliable reception and timely transmission of data. The default mode for all UART-based ports is SERIAL_AUTOMATIC. The default and only allowed mode for the Orangutan SVP's and Orangutan X2's USB_COMM port is SERIAL_CHECK.

static char OrangutanSerial::getMode([unsigned char *port*,] unsigned char *mode*)
char serial_get_mode([unsigned char *port*])

Returns the current serial mode.

static void OrangutanSerial::check()
void serial_check()

Checks for any bytes to be received or transmitted (on all available ports) and performs the required action. You only need to use this function if one of your ports is in SERIAL_CHECK mode. If all of your ports are in SERIAL_AUTOMATIC mode, you will not need to use this function. The default and only allowed mode for the Orangutan SVP's and Orangutan X2's USB_COMM port is SERIAL_CHECK, so you should call this function often if you want to use that port.

11. Orangutan Servos

This section of the library provides the ability to control up to 16 servos by generating digital pulses directly from your Orangutan without the need for a separate servo controller. These servo control functions are non-blocking; pulses are generated in the background by an interrupt while the rest of your code executes. Required memory is allocated by the **servos_start()** or **servos_start_extended()** functions using **malloc()**, which conserves RAM. The **servos_stop()** function frees this dynamically allocated memory.

This section of the library uses the AVR's Timer 1 and several interrupts: **TIMER1_CAPT**, **TIMER1_COMPA** (not used on the Orangutan SVP), and **TIMER1_COMPB**.

For a higher-level overview of how servo control works and how the library works on the different Orangutan models, see **Section 3.i** of the **Pololu AVR C/C++ Library User's Guide [http://www.pololu.com/docs/0J20]**.

Note: The OrangutanServos and OrangutanBuzzer libraries both use Timer 1, so they cannot be used together simultaneously. It is possible to alternate use of OrangutanBuzzer and OrangutanServos routines, however. The `servos-and-buzzer` example program shows how this can be done and also provides functions for playing notes on the buzzer without using Timer 1 or the OrangutanBuzzer functions. The `servo-control-using-delays` example shows how to generate servo pulses using delays rather than Timer 1 and the OrangutanServos functions.

Reference

C++ and Arduino methods are shown in red.
C functions are shown in green.

static unsigned char OrangutanServos::start(const unsigned char *servo_pins*[], unsigned char *num_pins*)
unsigned char servos_start(const unsigned char *servo_pins*[], unsigned char *num_pins*)

Configures the AVR's Timer 1 module to generate pulses for up to 8 servos. The *num_pins* parameter should be the length of the *servo_pins* array. A nonzero return value indicates that the needed memory could not be allocated.

The *servo_pins* parameter should be the RAM address of an array of AVR I/O pin numbers defined using the IO_* keywords provided by the library (e.g. IO_D1 for a servo output on pin PD1).

On the Orangutan SVP: this function takes an array of AVR pins that you have wired to the demux selection pins. The length of the array should be 0–3. The number of servos you can control is 2^{num_pins}. The servo pulses are generated on pin PD5, which is a hardware PWM output connected to the input of the demux, and servos should be connected to demux (servo port) outputs 0 – 2^{num_pins} on the SVP. See the "Servo Demultiplexer" portion of **Section 4** of the Orangutan SVP user's guide for more information.

On the other Orangutans: this function takes an array of AVR pins that you have connected to the signal pins on your servos. The length of the array should be 0–8. Each pin controls one servo.

This function was previously called **servos_init()**. To maintain compatibility with older versions of the library, servos_init() still exists, but use of servos_start() is encouraged.

Example

```
// Orangutan SVP: configure PD0 and PD1 to be demux inputs SA and SB,
//   which generates servo pulses on demux pins 0 - 3
// All else: configure for two servo outputs on pins PD0 and PD1
servos_start((unsigned char[]) {IO_D0, IO_D1}, 2);
// C++: OrangutanServos::start((unsigned char[]) {IO_D0, IO_D1}, 2);
```

static unsigned char OrangutanServos::start(const unsigned char *servo_pins*[], unsigned char *num_pins*, const unsigned char *servo_pins_b*[], unsigned char *num_pins_b*)
unsigned char servos_start_extended(const unsigned char *servo_pins*[], unsigned char *num_pins*, const unsigned char *servo_pins_b*[], unsigned char *num_pins_b*)

Configures the AVR's Timer 1 module to generate pulses for up to 16 servos. A nonzero return value indicates that the needed memory could not be allocated. The *servo_pins* and *num_pins* parameters serve the same purpose here as they do in the function above. The *num_pins_b* parameter should be the length of the *servo_pins_b* array. The *servo_pins_b* parameter should be the RAM address of an array of AVR pin numbers defined using the IO_* keywords provided by the library (e.g. IO_D1 for servo pulses on pin PD1). The pins should be connected to the signal pins on your servos. If you don't want this second set of servos, use a *num_pins_b* value of 0 (and a *servo_pins_b* value of 0) or use the servos_start() function above. Similarly, if you want to only use the second set of servos, you can use a *num_pins* value of 0 (and a *servo_pins* value of 0).

This function was previously called **servos_init_extended()**. To maintain compatibility with older versions of the library, servos_init_extended() still exists, but use of servos_start_extended() is encouraged.

static void OrangutanServos::stop()
void servos_stop()

Stops Timer 1, sets all used servo pins as driving-low outputs, and frees up the memory that was dynamically allocated by the servos_start() or servos_start_extended() function. If you want to use Timer 1 for some other

purpose, such as playing sounds using OrangutanBuzzer functions, you should call servos_stop() first. Servos cannot be used after calling servos_stop() without first calling servos_start() or servos_start_extended() again.

static void OrangutanServos::setServoTarget(unsigned char *servo_num*, unsigned int *pos_us*)
static unsigned int OrangutanServos::getServoTarget(unsigned char *servo_num*)
static void OrangutanServos::setServoTargetB(unsigned char *servo_num*, unsigned int *pos_us*)
static unsigned int OrangutanServos::getServoTargetB(unsigned char *servo_num*)
void set_servo_target(unsigned char *servo_num*, unsigned int *pos_us*)
unsigned int get_servo_target(unsigned char *servo_num*)
void set_servo_target_b(unsigned char *servo_num*, unsigned int *pos_us*)
unsigned int get_servo_target_b(unsigned char *servo_num*)

These functions set and get a servo's target position in units of microseconds. This is the pulse width that will be transmitted eventually, but this pulse width may not be transmitted immediately if there is a speed limit set for the servo, or if the target was changed within the last 20 ms. A position value of 0 turns off the specified servo (this is the default). Otherwise, valid target positions are between 400 and 2450 us. The *servo_num* parameter should be a servo number between 0 and 7 and should be less than the associated *num_servos* parameter used in the servos_start() or servos_start_extended() function.

static void OrangutanServos::setServoSpeed(unsigned char *servo_num*, unsigned int *speed*)
static unsigned int OrangutanServos::getServoSpeed(unsigned char *servo_num*)
static void OrangutanServos::setServoSpeedB(unsigned char *servo_num*, unsigned int *speed*)
static unsigned int OrangutanServos::getServoSpeedB(unsigned char *servo_num*)
void set_servo_speed(unsigned char *servo_num*, unsigned int *speed*)
unsigned int get_servo_speed(unsigned char *servo_num*)
void set_servo_speed_b(unsigned char *servo_num*, unsigned int *speed*)
unsigned int get_servo_speed_b(unsigned char *servo_num*)

These functions set and get a servo's speed limit in units of tenths of a microsecond per 20 ms. A speed value of 0 means there is no speed limit. A non-zero speed value means that each time the library sends a servo pulse, that pulse's width will be within *speed/10* µs of the previous pulse width sent to that servo. For example, with a speed value of 200, the pulse width will change by at most 20 µs each time the library sends a pulse. The library sends a pulse every 20 ms, so it will take 1000 ms (50 pulses) to change from a pulse width of 1000 µs to a pulse width of 2000 µs. The *servo_num* parameter should be a servo number between 0 and 7 and should be less than the associated *num_servos* parameter used in the servos_start() or servos_start_extended() function.

static unsigned int getServoPosition(unsigned char *servo_num*)
static unsigned int getServoPositionB(unsigned char *servo_num*)
unsigned int get_servo_position(unsigned char *servo_num*)
unsigned int get_servo_position_b(unsigned char *servo_num*)

These functions get a servo's current position in units of microseconds. This is the last pulse width that was transmitted to the servo, but this pulse width might not be equal to the target if there is a speed limit set for the servo, or if the target has changed with the last 20 ms. This method does **not** rely on feedback from the servo, so if the servo is being restrained or overly torqued or simply cannot physically move as quickly as the pulse widths are changing, it might not return the actual position of the servo. If there is a speed limit set for this servo, you can use this method to determine when the servo pulse signal has reached its desired target width. The *servo_num* parameter should be a servo number between 0 and 7 and should be less than the associated *num_servos* parameter used in the servos_start() or servos_start_extended() function.

12. Orangutan SPI Master Functions

This section of the library provides commands for using the AVR's Serial Peripheral Interface (SPI) module in master mode to communicate with slave SPI devices.

SPI is a synchronous communication protocol where the basic transaction consists of the master pulsing the SPI clock (SCK) line 8 times while bits are simultaneously exchanged between the (selected) slave and the master on the Master-in/Slave-out (MISO) and Master-out/Slave-in (MOSI) lines. There is no way for the master to send a byte without receiving one, and there is no way for the master to receive a byte without sending one.

The functions in this section will automatically configure the MOSI and SCK lines as outputs and the MISO line as an input.

The AVR's SPI module is designed so that if the $\overline{SS}$ pin is an input and it reads low (0 V), then the SPI module will automatically go in to slave mode (the MSTR bit in SPCR will become zero) and all SPI transmission functions in this library will return a result of zero. Therefore, it is recommended to make $\overline{SS}$ an output before doing SPI master communication. If $\overline{SS}$ is an input, then the SPI initialization routine in this library will enable the pull-up resistor on that line.

The Orangutan LV, Orangutan SV, Baby Orangutan, and 3pi robot are based on the ATmega48/168/328 line of AVR processors, so $\overline{SS}$ is pin PB2, MOSI is PB3, MISO is PB4, and SCK is PB5.

The Orangutan SVP and Orangutan X2 are based on the AVR ATmega324/644/1284 line of AVR processors, so $\overline{SS}$ is pin PB4, MOSI is PB5, MISO is PB6, and SCK is PB7. The Orangutan SVP and X2 are both aided by auxiliary microcontrollers that are slave SPI peripherals. The functions in this class are used by the OrangutanSVP (see **Section 13**) and OrangutanX2 (see **Section 15**) classes to communicate with these auxiliary microcontrollers.

Reference

C++ and Arduino methods are shown in red.
C functions are shown in green.

static void OrangutanSPIMaster::init(unsigned char *speed_divider*, unsigned char *options*)
void spi_master_init(unsigned char *speed_divider*, unsigned char *options*)

Initializes the AVR's hardware SPI module in master mode. This command makes the MOSI and SCK pins outputs so that the AVR can send data to the slave device. This command makes MISO an input so that the AVR can receive data from the slave device. If $\overline{SS}$ is an input, this function enables its pull-up resistor so that it is less likely that $\overline{SS}$ goes low and knocks the SPI module out of master mode (see note above).

The *speed_divider* parameter specifies the ratio of the AVR's clock frequency to the SPI frequency. The library defines several keywords of the form **SPI_SPEED_DIVIDER_xxx** for use as the *speed_divider* argument. These keywords are shown in the table below:

Valid values for *speed_divider*

speed_divider	SPI Frequency (assuming 20 MHz clock)
SPI_SPEED_DIVIDER_2	10 MHz
SPI_SPEED_DIVIDER_4	5 MHz
SPI_SPEED_DIVIDER_8	2.5 MHz
SPI_SPEED_DIVIDER_16	1.25 MHz
SPI_SPEED_DIVIDER_32	625 kHz
SPI_SPEED_DIVIDER_64	313 kHz
SPI_SPEED_DIVIDER_128	156 kHz

The *options* argument controls three important configuration options for the SPI module. The *options* argument can be 0 to select all the default options. To over-ride the defaults, the options argument should be a combination of some of the following keywords, combined using the inclusive-or operator "|".

- SPI_SCK_IDLE_LOW **(default)**: The idle state of SCK will be low; the leading edge will be rising and the trailing edge will be falling.
- SPI_SCK_IDLE_HIGH: The idle state of SCK will be high; the leading edge will be falling and the trailing edge will be rising.
- SPI_MSB_FIRST **(default)**: Bytes will be transmitted/received starting with the most-significant bit first.
- SPI_LSB_FIRST: Bytes will be transmitted/received starting with the least-significant bit first.
- SPI_EDGE_LEADING **(default)**: The AVR will sample data on MISO on the leading edge of SCK.
- SPI_EDGE_TRAILING: The AVR will sample data on MISO on the trailing edge of SCK.

Example

```
// Initialize the SPI module in master mode at 20/2 = 10 MHz, sample on the trailing edge,
// LSB first, SCK idle state low.
spi_master_init(SPI_SPEED_DIVIDER_2, SPI_EDGE_TRAILING | SPI_LSB_FIRST);
// C++: OrangutanSPIMaster::init(SPI_SPEED_DIVIDER_2, SPI_EDGE_TRAILING | SPI_LSB_FIRST);
```

static unsigned char OrangutanSPIMaster::transmit(unsigned char *data*)
unsigned char spi_master_transmit(unsigned char *data*)

Transmits the given byte of data to the SPI slave device, and returns the byte that the slave simultaneously sent back.

static unsigned char OrangutanSPIMaster::transmitAndDelay(unsigned char *data*, unsigned char *post_delay_us*)
unsigned char spi_master_transmit_and_delay(unsigned char *data*, unsigned char *post_delay_us*)

This command is just like spi_master_transmit() except that after the transmission has finished it delays for the specified number of microseconds before returning. This added delay is useful if you are communicating with any slave device that requires some time between SPI transmissions to give it time to process the byte it has received (e.g. the auxiliary processors on the Orangutan X2 and Orangutan SVP). *post_delay_us* should be a number between 0 and 255.

13. Orangutan SVP Functions

This section of the library provides commands that are only available on the **Orangutan SVP [http://www.pololu.com/product/1325]**.

For a higher level overview of how the library works on the Orangutan SVP, please see **Section 3.j** 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 OrangutanSVP::setMode(unsigned char *mode*)
void svp_set_mode(unsigned char *mode*)

This command sets the mode of the Orangutan SVP's auxiliary processor. The *mode* parameter determines the functions of the auxiliary lines A, B, C, and D.

- If *mode* is SVP_MODE_RX, then lines A, B, and C will be analog inputs (**Section 2**) and D/RX will be used as the serial receive line. TTL-level serial bytes received on D/RX will be sent to the computer on the Pololu Orangutan SVP TTL Serial Port. The D/RX line, along with the TX line (which is always the serial transmit line), let you use the Orangutan as a two-way USB-to-serial adapter. This is the default mode; the auxiliary processor will be in this mode whenever it starts running, and will revert to this mode whenever the AVR is reset for any reason.
- If *mode* is SVP_MODE_ANALOG, then lines A, B, C, and D will be analog inputs.
- If *mode* is SVP_MODE_ENCODERS, then lines A and B together are inputs for a quadrature encoder (called encoder AB), and lines C and D together are inputs for a quadrature encoder (called encoder CD). The functions below whose names end in AB or CD provide access to the output from these encoders.

Additionally, the *mode* parameter can be inclusively ored with the keyword SVP_SLAVE_SELECT_ON to enable the SPI slave select feature.

- If SPI slave select is not enabled (default), then the ADC/$\overline{SS}$ line will be an analog input. This line is hardwired to a user trimpot, so the name of the analog channel is TRIMPOT. However, if you want to use this analog input for something else, you can cut the labeled trace between POT and ADC/$\overline{SS}$ on the bottom of the board. This is the default mode; the auxiliary processor will be in this mode whenever it starts running, and will revert to this mode whenever the AVR is reset for any reason.
- If SPI slave select is enabled, then the line will be used as the SPI slave select line, $\overline{SS}$. When $\overline{SS}$ is high, the auxiliary processor will ignore all bytes received on SPI and will not drive the MISO line. This allows the AVR to communicate with other SPI devices using the hardware SPI module. See **Section 12**. When $\overline{SS}$ is driven low, then the AVR can communicate with the auxiliary processor over SPI as usual. The $\overline{SS}$ line is pulled high through a 100 kilo-ohm pull-up resistor.

static unsigned char OrangutanSVP::usbPowerPresent()
unsigned char usb_power_present()

Returns 1 if the voltage on the power line of the USB connector is high. This indicates that the device is plugged in to a computer or USB power supply. Returns 0 otherwise. This function is useful if you want your Orangutan to behave differently when it is plugged in to USB (for example, by not running its motors).

static unsigned char OrangutanSVP::usbConfigured()
unsigned char usb_configured()

Returns 1 if the device has reached the USB Configured State as defined in the USB 2.0 Specification. This indicates that the device is plugged in to a computer and the right drivers might be installed. Returns 0 otherwise.

static unsigned char OrangutanSVP::usbSuspend()
unsigned char usb_suspend()

Returns 1 if the device is in the USB Suspend State. If the device is connected to a computer, this usually indicates that the computer has gone to sleep, but on some computers the Suspend State occurs several times whenever the device is plugged in.

static unsigned char OrangutanSVP::dtrEnabled()
unsigned char dtr_enabled()
Returns either 0 or 1, indicating the value of the DTR virtual handshaking line on the Pololu Orangutan SVP USB Communications Port. This line is controlled by the computer's virtual serial port driver and terminal software. This line corresponds to the DtrEnable member of the .NET System.IO.Ports.SerialPort class. This line defaults to 0. Several standard terminal programs set this line to 1 when they connect to a serial port, and then set it to 0 when they disconnect. Therefore this command can be used to determine whether a terminal program is connected to the Orangutan, and make the Orangutan's behavior dependent on that.

static unsigned char OrangutanSVP::rtsEnabled()
unsigned char rts_enabled()
Returns either 0 or 1, indicating the value of the RTS virtual handshaking line on the Pololu Orangutan SVP USB Communications Port. This line is controlled by the computer's virtual serial port driver and terminal software. This line corresponds to the RtsEnable member of the .NET System.IO.Ports.SerialPort class. This line defaults to 0.

static int OrangutanSVP::getCountsAB()
static int OrangutanSVP::getCountsCD()
int svp_get_counts_ab()
int svp_get_counts_cd()
Returns the number of counts measured on encoder AB or CD. For the Pololu wheel encoders, the resolution is about 3 mm/count, so this allows a maximum distance of 32767 × 3 mm or about 100 m. For longer distances, you will need to occasionally reset the counts using the functions below. The counts will be increase if A/C changes before B/D, and decrease if B/D changes before A/C.

static int OrangutanSVP::getCountsAndResetAB()
static int OrangutanSVP::getCountsAndResetCD()
int svp_get_counts_and_reset_ab()
int svp_get_counts_and_reset_cd()
Returns the number of counts measured on encoder AB or CD, and resets the stored value to zero.

static int OrangutanSVP::checkErrorAB()
static int OrangutanSVP::checkErrorCD()
int svp_check_error_ab()
int svp_check_error_cd()
These commands check whether there has been an error on AB or CD; that is, if both A/B or C/D changed simultaneously. They return 1 if there was an error, then reset the error flag to zero.

static unsigned char OrangutanSVP::getFirmwareVersion()
unsigned char svp_get_firmware_version()
This command asks the Orangutan SVP's auxiliary processor what version of its firmware is running. The return value of this function for all Orangutans released so far should be 1. This command can be useful for testing or debugging the SPI connection to the auxiliary processor.

14. Orangutan System Resources

This section of the library is intended offers general AVR resources. 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 diffucult-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 get_free_ram() 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 get_free_ram() 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.

15. Orangutan X2 Functions

These functions are all commented in the library source code **`libpololu-avr/src/OrangutanX2/OrangutanX2.cpp` [https://github.com/pololu/libpololu-avr/blob/master/src/OrangutanX2/OrangutanX2.cpp]**.

16. 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/product/961]** together with the Orangutan. 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 19** 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 an 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.

Note: This library is implemented differently for use on Arduinos. Please install the **Arduino-specific version [http://www.pololu.com/docs/0J19]** if you want to use QTR sensors with the Arduino.

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 3.k** of the **Pololu AVR C/C++ Library User's Guide [http://www.pololu.com/docs/0J20]**.

Reference

C++ and 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)
unsigned int 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 2, etc. Intermediate values indicate that the line is between two sensors. The formula is:

```
 0*value0 + 1000*value1 + 2000*value2 + ...
-------------------------------------------
     value0  +  value1  +  value2 + ...
```

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 minumum 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* should contain the pin numbers for each sensor, defined using the IO_* keywords provided by the library. For example, if *pins* is {IO_D3, IO_D6, IO_C1}, sensor 0 is on PD3, sensor 1 is on PD6, and sensor 2 is on PC1.

For ATmegaxx8-based controllers, the pin numbers are Arduino-compatible so you can define the *pins* array using Arduino pin numbers. 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 correpsond 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 16.

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 PololuQTRSensorsAnalog::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 ATmegaxx8 has 8 total analog input channels (ADC0 – ADC7) that correspond to port C pins PC0 – PC5 and dedicated analog inputs ADC6 and ADC7. The Orangutan SVP and X2 also have 8 analog input channels (ADC0 – ADC7) but they correspond to port A pins PA0 – PA7. The analog channels provided by the Orangutan SVP's auxiliary processor (channels A, B, C, and D) are *not* supported by this library.

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.

17. Timing and Delays

The following timing and delay functions are designed for the Orangutans and 3pi robot, 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.

The timing functions use a Timer 2 overflow interrupt (**TIMER2_OVF**), and Timer 2 is automatically configured when any OrangutanTime function is called. This means that the timing code will conflict with other code that uses Timer 2 in an incompatible way. However, the functions here are compatible with the other uses of Timer 2 within the Pololu library.

The Timer 2 overflow ISR is written in assembly to make it as short as possible. When the Timer 2 overflow interrupt occurs, your code will be interrupted for a total span of 2.65 µs (this includes the time it takes to jump into the ISR and the time it takes to return from it) once every 102.4 µs. Once every millisecond, the Timer 2 overflow ISR will take a little bit longer: 3.85 µs instead of the usual 2.65 µs. So in all, maintaining the system timers takes up approximately 2.5% of your available processing time.

General Note about Timers: The functions in Pololu AVR library will conflict with code that tries to reconfigure the hardware timers it is relying upon, so if you want to use a timer for some custom task, you cannot use the portion of the library that relies on that timer. The Pololu AVR library only uses Timer 0 for motor PWM generation, and it is only used for this purpose on the Orangutan LV, SV, Baby Orangutan, and 3pi robot, so the Orangutan SVP and X2 can safely use Timer 0 for any custom task. Timer 1 is used by OrangutanBuzzer for sound generation on all devices, and it is used by OrangutanServos for servo pulse generation on all devices. Timer 2 is used for motor PWM generation on all devices except Orangutan X2, and it is used by OrangutanTime to run the system timer on all devices. Additionally, the Orangutan SVP-**1284** has a second 16-bit timer, Timer 3, that can safely be used for any custom task (the Pololu AVR library does not use Timer 3 for anything).

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 of 65536 µs). Because this is a loop delay, the presence of interrupts will extend the delay period. For example, the Timer 2 overflow interrupt used by OrangutanTime to maintain the system timers will extend this delay period by approximately 2.5%, so if your system clock is 100% accurate, delay_ms(1000) will actually delay for approximately 1025 ms.

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). Because this is a loop delay, the presence of interrupts will extend the delay period. For example, the Timer 2 overflow interrupt used by OrangutanTime to maintain the system timers will extend this delay period by approximately 2.5%, so if your system clock is 100% accurate, delay_us(1000) will actually delay for approximately 1025 µs.

static void OrangutanTime::reset()
void time_reset()

Resets the system millisecond timer, but does not reset the system tick counter.

static unsigned long OrangutanTime::ms()
unsigned long get_ms()
unsigned long millis()

Returns the number of elapsed milliseconds since the first time an OrangutanTime function was called or the last time_reset() call, whichever is shorter. 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. Because the millisecond counter overflows on an even data boundary, differential millisecond calculations will produce correct results across the millisecond counter overflow.

static unsigned long OrangutanTime::ticks()
unsigned long get_ticks()

Returns the number of elapsed ticks (in units of 0.4 µs) since the first time an OrangutanTime function was called. The tick counter is not reset by time_reset(). get_ticks() can be used as a high-resolution system timer that will overflow back to zero after approximately 28.6 minutes of counting. Because the tick counter overflows on an even data boundary, differential tick calculations will produce correct results across the tick counter overflow. In order to take advantage of this property, it is important to always perform tick-to-microsecond conversions of the differential result, not the individual arguments of the differential calculation.

Example

```
// if the tick counter has overflowed once since we called prev_ticks = get_ticks(),
// the following call will result in a CORRECT measure of the elapsed time in us
unsigned long elapsed_time_us = ticks_to_microseconds(get_ticks() - prev_ticks);

// and the following call will result in an INCORRECT measure of the elapsed time in us
elapsed_time_us = ticks_to_microseconds(get_ticks()) - ticks_to_microseconds(prev_ticks);
```

static unsigned long OrangutanTime::ticksToMicroseconds(unsigned long *num_ticks*)

unsigned long ticks_to_microseconds(unsigned long *num_ticks*)

Returns the number of microseconds corresponding to the supplied number of ticks, *num_ticks*. One tick equals 0.4 µs, so 2.5 ticks equals one microsecond.

18. Wheel Encoders

The PololuWheelEncoders class and the associated C functions provide an easy interface for using the **Pololu Wheel Encoders [http://www.pololu.com/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. Specifically, it uses the AVR's **PCINT0**, **PCINT1**, and **PCINT2** (and **PCINT3** on the Orangutan SVP and X2) interrupts, and it will conflict with any other code that uses pin-change interrupts (e.g. OrangutanPulseIn).

For a higher level overview of this library and example programs that show how this library can be used, please see **Section 3.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.

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. Each pin is specified using the IO_* keywords provided by the library (e.g. IO_D1 for an encoder input on pin PD1). 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 × 3mm 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.

19. 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 16** 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 16** (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 *disable_emitter_pin* = 0)
unsigned char pololu_3pi_init(unsigned int *line_sensor_timeout*)
unsigned char pololu_3pi_init_disable_emitter_pin(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). Setting *disable_emitter_pin* to 1 (C++) or calling the function pololu_3pi_init_disable_emitter_pin() causes pin PC5 to not be used at all by the library, so that you can use it for something else.

void Pololu3pi::readLineSensors(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)
unsigned int read_line(unsigned int **sensorValues*, unsigned char *readMode*)
unsigned int 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:

```
 0*value0 + 1000*value1 + 2000*value2 + ...
-------------------------------------------
     value0  +  value1  +  value2 + ...
```

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, its 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 minumum 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.