

SSC Arduino

1.0

Generated by Doxygen 1.8.8

Fri Sep 12 2014 00:40:26

Contents

1	SSC-Arduino	1
2	Deprecated List	1
3	Data Structure Index	1
3.1	Data Structures	1
4	File Index	2
4.1	File List	2
5	Data Structure Documentation	2
5.1	Led Struct Reference	2
5.1.1	Detailed Description	2
5.2	LSR Struct Reference	2
5.2.1	Detailed Description	3
5.2.2	Field Documentation	3
5.3	Motor Struct Reference	3
5.3.1	Detailed Description	4
5.4	RLSR Struct Reference	4
5.4.1	Detailed Description	4
5.5	Sensor Struct Reference	5
5.5.1	Detailed Description	5
5.6	Switch Struct Reference	5
5.6.1	Detailed Description	6
6	File Documentation	6
6.1	Led.h File Reference	6
6.1.1	Detailed Description	6
6.1.2	Typedef Documentation	6
6.1.3	Function Documentation	7
6.2	LSR.h File Reference	7
6.2.1	Detailed Description	8
6.2.2	Typedef Documentation	8
6.2.3	Function Documentation	8
6.3	Motor.h File Reference	10
6.3.1	Detailed Description	10
6.3.2	Typedef Documentation	10
6.3.3	Function Documentation	11
6.4	RLSR.h File Reference	11
6.4.1	Detailed Description	12

6.4.2	Typedef Documentation	12
6.4.3	Function Documentation	12
6.5	Sensor.h File Reference	12
6.5.1	Detailed Description	13
6.5.2	Typedef Documentation	13
6.5.3	Function Documentation	13
6.6	Switch.h File Reference	14
6.6.1	Detailed Description	14
6.6.2	Typedef Documentation	14
6.6.3	Function Documentation	14
6.7	utils.h File Reference	15
6.7.1	Detailed Description	15
6.7.2	Function Documentation	15
	Index	17

1 SSC-Arduino

Arduino Code for the SSC Drill project.

2 Deprecated List

Global `getTrendLSR` (`LSR_t *target`)

Can be safely removed, but does not add any size to the code simple function to return the calculated trend.

Global `motorSetRpm` (`Motor_t *target`, `long rpm`)

The motor is not a class anymore. Direct change can be made and this code can be safely removed.

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Led	2
LSR	2
Motor	3
RLSR	4
Sensor	5
Switch	5

4 File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Led.h	6
LSR.h	7
Motor.h	10
RLSR.h	11
Sensor.h	12
Switch.h	14
utils.h	15

5 Data Structure Documentation

5.1 Led Struct Reference

```
#include <Led.h>
```

Data Fields

- byte [pin_RGB](#) [2]
maps each pin for acting.
- byte [RGB](#) [2]
maps each RGB color of the desired color in the LED.
- boolean [digital](#)
used to discern digital or analog LEDs.

5.1.1 Detailed Description

pseudo-class as a structure for LED abstraction.

constitutes of *pin_RGB* vector, giving the pins designed to make the led act in the desired RGB color, *RGB* vector containing the so-called numbers and *boolean* digital; which is used within the function of acting to discern between digital and analog LEDs.

The documentation for this struct was generated from the following file:

- [Led.h](#)

5.2 LSR Struct Reference

```
#include <LSR.h>
```

Data Fields

- unsigned long [n](#)
internal sampling size.
- unsigned long [cut_index](#)
- double [sx](#)
sum if all X values.
- double [sy](#)
sum of all Y values.
- double [sxx](#)
sum of all X values squared.
- double [sxy](#)
sum of the product of all the X and Y values.
- double [trend](#)
the trend calculated for the sample.
- double [constant](#)
the constant calculated for the sample.
- double [Den](#)
A denominator internal variable for the calculations.

5.2.1 Detailed Description

pseudo-class abstraction for the least squares algorithm.

The implementation does not use a vector, rather, it uses single variables to keep track of the necessary sums and sampling size; thus, keep in mind that the internal variables may overflow eventually for a huge sample.

5.2.2 Field Documentation

5.2.2.1 unsigned long [cut_index](#)

true sampling size.

When the internal sampling size reaches the index; the abstraction performs the necessary calculations to store the trend and constant into its internal variables, which can be accessed at any time.

The documentation for this struct was generated from the following file:

- [LSR.h](#)

5.3 Motor Struct Reference

```
#include <Motor.h>
```

Data Fields

- byte [pin_rpm](#)
Arduino pin for the PMW output.
- byte [pin_master_enable](#)
Arduino pin for the enable/disable control.
- long [max_rpm](#)
a variable representing the max RPM the motor can achieve.
- long [min_rpm](#)

a variable representing the min RPM the motor can achieve. This can be negative for orientation.

- long [current_rpm](#)

a variable representing the desired speed for the motor.

- boolean [master_enable](#)

internal variable controlling the enabled and disabled state.

5.3.1 Detailed Description

pseudo-class abstraction for motors.

The implementation consists the usual *pin_rpm* to indicate the PWM output of the arduino, another pin to control the enable/disable state and RPM control variables.

The *max_rpm* and *min_rpm* variables are necessary to make the arduino corretly process and act on the desired RPM, *current_rpm*. Note that the last variable is NOT the speed reading, but the reference speed. These variables and set in the initialization and can be changed by direct acess if needed later.

The documentation for this struct was generated from the following file:

- [Motor.h](#)

5.4 RLSR Struct Reference

```
#include <RLSR.h>
```

Data Fields

- unsigned int [n](#)
variable for the sample size.
- double [D1](#)
internal variable representing a denominator.
- double [D2](#)
internal variable representing a second denomiator.
- double [sx](#)
internal variable representing the X values summation.
- double [sy](#)
internal variable representing the Y values summation.
- double [sxx](#)
internal variable representing the X squared summation.
- double [sxy](#)
internal variable representing the X mutliplied by Y summation.
- double [trend](#)
variable representing the calculated trend.
- double [constant](#)
variable representing the calculated constant.

5.4.1 Detailed Description

Recursive least squares regression algorithm pseudo-class.

In the same fashion as the [LSR](#) abstraction, this pseudo-class contains all the internal calculations variables necessary to better performance and variables representing the trend and constant after each sampling.

See also

[LSR](#)

The documentation for this struct was generated from the following file:

- [RLSR.h](#)

5.5 Sensor Struct Reference

```
#include <Sensor.h>
```

Data Fields

- byte [pin](#)
variable to indicate the output PIN for the sensor.
- double [max_value](#)
internal variable to calculate corret measurements.
- double [min_value](#)
internal variable to calculate corret measurements.
- double [max_voltage](#)
internal variable to calculate corret measurements.
- double [min_voltage](#)
internal variable to calculate corret measurements.
- double [voltage_drift_linear](#)
internal variable to fine tune sensor readings.
- double [voltage_drift_const](#)
internal variable to fine tune sensor readings.
- double [reading](#)
internal variable to store the pure arduino reading, for later processing.

5.5.1 Detailed Description

[Sensor](#) pseudo-class abstraction.

The implementation consists the usual *pin* to indicate the input of the arduino, *reading* internal variable for the voltage output, and some maximum and minimum settings for better value intepretation.

There is also *voltage_drift_const* and *voltage_drift_linear* for fine tune of the readings; but experience showed that the arduino is not relaiable even with these in the best way tuned.

The documentation for this struct was generated from the following file:

- [Sensor.h](#)

5.6 Switch Struct Reference

```
#include <Switch.h>
```

Data Fields

- byte [pin](#)
variable to represent Arduino PIN.
- bool [reading](#)
digital reading made by the arduino of the switch.
- bool [pullup](#)
internal flag to control if the switch should be treated normally or as pullup.

5.6.1 Detailed Description

[Switch](#) pseudo-class abstraction.

implements the usual pin abstraction along with the boolean reading. The pullup works as a flag which should only be given at the initialization and not changed later.

The documentation for this struct was generated from the following file:

- [Switch.h](#)

6 File Documentation

6.1 Led.h File Reference

```
#include <Arduino.h>
```

Data Structures

- struct [Led](#)

Typedefs

- typedef struct [Led](#) [Led_t](#)

Functions

- [Led_t](#) * [ledInit](#) (byte pin_red, byte pin_green, byte pin_blue, boolean digital_)
- void [ledSetColor](#) ([Led_t](#) *target, byte red, byte green, byte blue)
- void [ledAct](#) ([Led_t](#) *target)

6.1.1 Detailed Description

LED abstraction definitions.

This files contains both the pseudo-class function definitions and the structure which acts kile the surmounted class.

6.1.2 Typedef Documentation

6.1.2.1 typedef struct [Led](#) [Led_t](#)

pseudo-class as a structure for LED abstraction.

constitutes of *pin_RGB* vector, giving the pins designed to make the led act in the desired RGB color, *RGB* vector containing the so-called numbers and *boolean* digital; which is used within the function of acting to discern between digital and analog LEDs.

6.1.3 Function Documentation

6.1.3.1 void ledAct (Led_t * target)

LED abstraction acting function.

If the LED is digital, check if each of the RGB's in the target abstraction are greater than 127; if so, this color is set to HIGH. Else, just write each RGB color set directly into each color's analog pin.

Parameters

in	<i>target</i>	The LED abstraction to be acted.
----	---------------	----------------------------------

6.1.3.2 Led_t* ledInit (byte pin_red, byte pin_green, byte pin_blue, boolean digital_)

ledInit LED abstraction initialization.

After giving the necessary pins and a flag to check if the LED is digital or analog; space for it is allocated internally and a Led_t type is returned to be used.

Parameters

in	<i>pin_red</i>	The pin for the red I/O.
in	<i>pin_blue</i>	The pin for the blue I/O.
in	<i>pin_green</i>	The pin for the green I/O.
in	<i>digital_</i>	Flag to be used internally for acting.

6.1.3.3 void ledSetColor (Led_t * target, byte red, byte green, byte blue)

Change the color settings of the target by giving new RGB.

Parameters

in	<i>target</i>	abstraction to me modified.
in	<i>red</i>	new red value.
in	<i>green</i>	new green value.
in	<i>blue</i>	new blue value.

6.2 LSR.h File Reference

Data Structures

- struct [LSR](#)

Typedefs

- typedef struct [LSR](#) [LSR_t](#)

Functions

- [LSR_t](#) * [initLSR](#) (unsigned long cut_index_)
- void [addValuePairLSR](#) ([LSR_t](#) *target, double x, double y)
- void [addValueLSR](#) ([LSR_t](#) *target, double y)
- double [getTrendLSR](#) ([LSR_t](#) *target)

6.2.1 Detailed Description

Least Squares Regression.

This header file contains the [LSR](#) algorithm abstraction and its corresponding functions; such as calculating the trend and axis intersection and adding a new value to the list.

6.2.2 Typedef Documentation

6.2.2.1 typedef struct LSR LSR_t

pseudo-class abstraction for the least squares algorithm.

The implementation does not use a vector, rather, it uses single variables to keep track of the necessary sums and smapling size; thus, keep in mind that the internal variables may overflow eventually for a huge sample.

6.2.3 Function Documentation

6.2.3.1 void addValueLSR (LSR_t * target, double y)

adds a new value pair to *target* assuming that X is the current index.

This functions behaves exactly in the same manner as [addValuePairLSR\(\)](#), except that the y corresponding X value is assumed to be the integer representing the y index in the list, e.g. if the sample size is 6, the next y index would be 7.

Parameters

in	<i>target</i>	abstraction to me modified.
----	---------------	-----------------------------

See also

[addValuePairLSR\(LSR_t *target, double x, double y\)](#)

Parameters

in	y	an Y value for the algorithm calculation.
----	---	---

6.2.3.2 void addValuePairLSR (LSR_t * target, double x, double y)

adds a new value pair to *target* of type [LSR](#).

By giving x and y values, all the internal calculations and necessary updates within the least squares algorithm are taken case of. If the sample size has reached the cutting index; this algorithm calculates the trend and the constant of the linear regression.

See also

[LSR](#)

Parameters

in	<i>target</i>	abstraction to me modified.
in	x	an X value for the algorithm.
in	y	an X corresponding Y value for the algorithm.

6.2.3.3 double getTrendLSR (LSR_t * target)

Deprecated Can be safely removed, but does not add any size to the code simple function to return the calculated trend.

Deprecated since the internal variables can be accessed directly; exists as a reminiscent of a previous class-driven implementation.

Parameters

<i>in</i>	<i>target</i>	abstraction to me modified.
-----------	---------------	-----------------------------

Returns

double

6.2.3.4 LSR_t* initLSR (unsigned long *cut_index_*)[LSR](#) pseudo-class initialization.

by giving *cut_index* to the function; an empty [LSR](#) is allocated within memory and returned. If later the cutting index need to be changed, this can be done by directly accessing the structure internal variables; no harm is done o the calculations in this matter.

Parameters

<i>in</i>	<i>cut_index_</i>	Initial cutting index; can be changed.
-----------	-------------------	--

Returns[LSR](#)**6.3 Motor.h File Reference**

```
#include <Arduino.h>
```

Data Structures

- struct [Motor](#)

Typedefs

- typedef struct [Motor](#) [Motor_t](#)

Functions

- [Motor_t](#) * [motorInit](#) (byte pin_rpm_, byte pin_master_enable_, long max_rpm_, long min_rpm_)
- void [motorSetRpm](#) ([Motor_t](#) *target, long rpm)
- void [motorAct](#) ([Motor_t](#) *target)

6.3.1 Detailed Description[Motor](#) abstraction header

This header file contains the motor abstraction and its corresponding functions; such as acting, switching and initialization. Since the abstraction is a data structure, there is no need for private access methods; these changes are made by directly accessing the structure variables.

6.3.2 Typedef Documentation**6.3.2.1 typedef struct [Motor](#) [Motor_t](#)**

pseudo-class abstraction for motors.

The implementation consists the usual *pin_rpm* to indicate the PWM output of the arduino, another pin to control the enable/disable state and RPM control variables.

The *max_rpm* and *min_rpm* variables are necessary to make the arduino corretly process and act on the desired RPM, *current_rpm*. Note that the last variable is NOT the speed reading, but the reference speed. These variables and set in the initialization and can be changed by direct access if needed later.

6.3.3 Function Documentation

6.3.3.1 void motorAct (Motor_t * target)

[Motor](#) abstraction standard acting

Takes care of the enable/disable state control and automatically sets the correct voltage PWM based upon the *min_rpm* and *max_rpm* pseudo-class variables.

However an important assumption is made: the minimum RPM occurs at 10% of the admissable voltage apply while the maximum occurs at 90%. If that is not the case, the internal implementation needs to be changed. This is true for the ESCON 50/5 controller.

6.3.3.2 Motor_t* motorInit (byte pin_rpm_, byte pin_master_enable_, long max_rpm_, long min_rpm_)

[Motor](#) abstraction initialization.

For initialization, control and internal calculations; along the necessary pins the maximum and minimum achievable RPMs should be supplied.

For example, in the SSC prototype; the motor could achieve 178 rpm CCW, hence being positive and 178 rpm CW, being then negative. This setup initialization would be the pin rpms with *min_rpm_=-178* and *max_rpm_=178*.

Parameters

in	<i>pin_rpm_</i>	Arduino PIN for the PWM output.
in	<i>pin_master_↔ enable_</i>	Arduino PIN for the enable/disable control.
in	<i>max_rpm_</i>	Maximum RPM achievable.
in	<i>min_rpm_</i>	Minimum RPM achievable.

6.3.3.3 void motorSetRpm (Motor_t * target, long rpm)

Deprecated The motor is not a class anymore. Direct change can be made and this code can be safely removed.

6.4 RLSR.h File Reference

Data Structures

- struct [RLSR](#)

Typedefs

- typedef struct [RLSR](#) [RLSR_t](#)

Functions

- [RLSR_t * initRLSR \(\)](#)
 - void [addValuePairRLSR \(RLSR_t *target, double x, double y\)](#)
 - void [addValueRLSR \(RLSR_t *target, double y\)](#)
 - double [getTrendRLSR \(RLSR_t *target\)](#)
- deprecated since the pseudo-class trend can be accessed directly.*

6.4.1 Detailed Description

Recursive least squares regression algorithm abstraction header/

Except for the underlying algorithms, All the organization is the same as the [LSR](#) abstraction. The difference is that for each new added variable, a new trend and constant is calculated immediatly.

6.4.2 Typedef Documentation

6.4.2.1 typedef struct **RLSR** **RLSR_t**

Recursive least squares regression algorithm pseudo-class.

In the same fashion as the [LSR](#) abstraction, this pseudo-class contains all the internal calculations variables necessary to better performance and variables representing the trend and constant after each sampling.

See also

[LSR](#)

6.4.3 Function Documentation

6.4.3.1 void addValuePairRLSR (**RLSR_t** * *target*, double *x*, double *y*)

add a value pair to the [RLSR](#) abstraction algorithm.

Works exactly like adding a pair to [LSR](#) abstraction, except that the new trend and constant are available right away.

See also

[addValuePairLSR\(\)](#)

6.4.3.2 void addValueRLSR (**RLSR_t** * *target*, double *y*)

add a value to the [RLSR](#) algorithm assuming X value to be the current index.

Works exactly like its [LSR](#) counterpart, except that the new trend and constant are available right away.

See also

[addValueLSR\(\)](#)

6.4.3.3 **RLSR_t*** initRLSR ()

[RLSR](#) initialization procedure.

Unlike [LSR](#), no cutting index is necessary as every pair addition results in an automatic calculation of the trend and constant without having to redo all the [LSR](#) calculations.

A note is needed in this respect: Just like the [LSR](#) suffers from miscalculation if the cutting index is really big, the recursive estimation eventually wears off and give wrong results as sample size increases without bounds. Of course, this can be corrected by manually resetting the sampling size to 0, Restarting the algorithm.

6.5 Sensor.h File Reference

```
#include <Arduino.h>
```

Data Structures

- struct [Sensor](#)

Typedefs

- typedef struct [Sensor](#) [Sensor_t](#)

Functions

- [Sensor_t](#) * [sensorInit](#) (byte *pin_*, double *min_value_*, double *max_value_*, double *min_voltage_*, double *max_vo↵_* *voltage_*)
- double [sensorGetReading](#) ([Sensor_t](#) **target*)
- void [sensorAct](#) ([Sensor_t](#) **target*)

The standard acting function for sensor, more of a general placeholder.

6.5.1 Detailed Description

[Sensor](#) abstraction header.

This header file contains the sensor abstraction and its corresponding functions; such as acting, initialization and reading calculation.

A function is necessary for the reading interpretation as the arduino does not output the its reading directly, but rather the read voltage.

6.5.2 Typedef Documentation

6.5.2.1 typedef struct [Sensor](#) [Sensor_t](#)

[Sensor](#) pseudo-class abstraction.

The implementation consists the usual *pin* to indicate the input of the arduino, *reading* internal variable for the voltage output, and some maximum and minimum settings for better value intepretation.

There is also *voltage_drift_const* and *voltage_drift_linear* for fine tune of the readings; but experience showed that the arduino is not reliable even with these in the best way tuned.

6.5.3 Function Documentation

6.5.3.1 double [sensorGetReading](#) ([Sensor_t](#) * *target*)

interpretable reading fuction.

based upon the structure's internal variables of maximu mand variables, return the expected (in ideal situations) reading of the sensor. This is the preferred way of getting information from the sensors, rather than directly acessing the reading internal variable and converting it to some value; this is a particular case of what this function does.

6.5.3.2 [Sensor_t](#)* [sensorInit](#) (byte *pin_*, double *min_value_*, double *max_value_*, double *min_voltage_*, double *max_vo↵_* *voltage_*)

[Sensor](#) abstraction initialization.

The arguments required for the abstraction allocation are essential for the true reading calculation. Note that the fine tune paramenters are set by direct acces; not given in one pseudo-class initialization.

Parameters

in	<i>pin_</i>	Arduino sensor PIN.
----	-------------	---------------------

in	<i>min_value_</i>	minimum value the readings can assume.
in	<i>max_value_</i>	maximum value the readings can assume.
in	<i>min_voltage_</i>	the voltage read by the Arduino when the value is at the minimum.
in	<i>max_voltage_</i>	the voltage read by the Arduino when the value is at the maximum.

6.6 Switch.h File Reference

```
#include <Arduino.h>
```

Data Structures

- struct [Switch](#)

Typedefs

- typedef struct [Switch](#) [Switch_t](#)

Functions

- [Switch_t](#) * [switchInit](#) (byte pin)
- [Switch_t](#) * [switchInitPullUp](#) (byte pin)
- void [switchAct](#) ([Switch_t](#) *target)

6.6.1 Detailed Description

[Switch](#) abstraction header.

This header file contains the switch abstraction and its corresponding functions; such as acting.

To change the switch state, the boolean variable which represents it must be directly changed to the desired state.

6.6.2 Typedef Documentation

6.6.2.1 typedef struct [Switch](#) [Switch_t](#)

[Switch](#) pseudo-class abstraction.

implements the usual pin abstraction along with the boolean reading. The pullup works as a flag which should only be given at the initialization and not changed later.

6.6.3 Function Documentation

6.6.3.1 void [switchAct](#) ([Switch_t](#) * *target*)

[Switch](#) acting function.

Takes note of the structure internal variables and acts accordingly. The internal behaviour is different if the switch is initialized normally or as pull-up, but the end result is given as the same by the function.

6.6.3.2 [Switch_t](#)* [switchInit](#) (byte *pin*)

[Switch](#) abstraction normal initialization.

This allocates memory for the pseudo-class and sets its necessary internal variables. along with the flags for further acting.

6.6.3.3 Switch_t* switchInitPullUp (byte pin)

[Switch](#) abstraction pull-up initialization.

This allocates memory for the pseudo-class and sets its necessary internal variables. along with the flags for further acting.

6.7 utils.h File Reference

Functions

- double [doubleMap](#) (double, double, double, double, double)
- double [vectorDot](#) (unsigned int, double *, double *)
- double [vectorSum](#) (unsigned int, double *)
- double [linearRegAngCoef](#) (unsigned int, double *, double *)
- double [linearRegLinCoef](#) (unsigned int, double *, double *)
- double * [linearReg](#) (unsigned int, double *, double *)
- double [vectorMean](#) (unsigned int, double *)
- void [printDouble](#) (double, unsigned int)

6.7.1 Detailed Description

header containing useful functions.

As the project progressed, many of these functions were not used anymore, but they were kept here in case they are needed later on. Most of them consists of useful mathematical operations.

6.7.2 Function Documentation

6.7.2.1 double doubleMap (double , double , double , double , double)

arduino built-in map for double.

Similiar to the arduino built-in function map, working in the same way but accepting double as inputs and return a double as result.

6.7.2.2 double* linearReg (unsigned int, double * , double *)

separate linear regression

Takes the length of two double vectors along themselves and calculate the linear regression of the data, returning a two-dimensional vector of the trend and coefficient as a result.

6.7.2.3 double linearRegAngCoef (unsigned int, double * , double *)

separate linear regression trend

Takes the length of two double vectors along themselves and calculate the linear regression trend of the data.

6.7.2.4 double linearRegLinCoef (unsigned int, double * , double *)

separate linear regression coefficient

Takes the length of two double vectors along themselves and calculate the linear regression coefficient of the data.

6.7.2.5 void printDouble (double , unsigned int)

print in Arduino in a double format.

meant to be more malleable than the standard Serial.print of the Arduino platform, by inputting the desired double to be printed and the decimal cases which should also be printed.

6.7.2.6 double vectorDot (unsigned *int*, double *, double *)

dot product for vector.

takes two double vectors and their length (they must be equal, hence only one is needed), and compute their euclidian dot product, return a double.

6.7.2.7 double vectorMean (unsigned *int*, double *)

arithmetic mean of a vector

Takes the lenght of a vector along the vector itself and return the mean of its elements.

6.7.2.8 double vectorSum (unsigned *int*, double *)

sum the elements of a vector.

Takes a double vector and its length and return the sum of its elements as a double.

Index

Led, [2](#)

Motor, [3](#)

Sensor, [5](#)

Switch, [5](#)